Vrije Universiteit Brussel

Faculteit van de Wetenschappen
Vakgroep Computerwetenschappen
Laboratorium voor Programmeerkunde

Ambient References: Object Designation in Mobile Ad Hoc Networks

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Tom Van Cutsem

Promotoren: Prof. Dr. Wolfgang De Meuter en Prof. Dr. Theo D'Hondt

Mei 2008

Print: DCL Print & Sign, Zelzate

2008 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. ++32 (0)2 289 26 50
Fax ++32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5487 487 4
NUR 989
Legal deposit D/2008/11.161/031

“Civilization advances by extending the number of important operations which we can perform without thinking of them.”

– Alfred North Whitehead, Introduction to Mathematics (1911)
English mathematician & philosopher (1861 - 1947)

Samenvatting

Twee decennia geleden stelde Mark Weiser een visie voor van de toekomst van onze computertechnologie. In zijn visie van *ubiquitous computing* wordt de computer onzichtbaar. Deze zal verwerkt zitten in alledaagse objecten, alomtegenwoordig in onze dagdagelijkse, fysieke omgeving. Naarmate zijn visie momentum kreeg in de onderzoekswereld werd het duidelijk dat de grote moeilijkheid in de verwezenlijking van zijn visie niet het ontwikkelen van geminiaturiseerde, ingebedde, draadloze hardware betrof (deze doelen hebben we vandaag reeds bereikt) dan wel het ontwikkelen van de software die op dit alomtegenwoordige netwerk van onzichtbare computers moet draaien.

Deze verhandeling onderzoekt schaalbare software-ingenieurstechnieken voor ubiquitous computing. Bekeken als een gedistribueerd systeem zijn ubiquitous computing applicaties verbonden via een zogenaamd *mobiel ad hoc netwerk*: een computernetwerk van mobiele toestellen die draadloos met mekaar kunnen communiceren. De karakteristieke eigenschappen van zulke computernetwerken zorgen er echter voor dat de aannames waarop hedendaagse software-abstractietechnieken steunen niet meer gelden. Deze observatie vormde de onmiddelijke aanleiding voor het ontwerpen van een nieuwe generatie programmeertalen gebaseerd op het *ambient-georiënteerd programmeerparadigma*.

Het doel van ambient-georiënteerd programmeren is om gedistribueerde object-georiënteerde softwaretechnologie zodanig te kneden dat deze kan omgaan met de gewijzigde karakteristieke eigenschappen van mobiele ad hoc netwerken. Binnen dit paradigma lichten we bestaande abstracties door voor het coördineren van gedistribueerde softwareprocessen en bestuderen we hoe deze omgaan met de karakteristieke eigenschappen van mobiele ad hoc netwerken. We zullen vaststellen dat hedendaagse object-georiënteerde gedistribueerde software-abstractietechnieken ongeschikt zijn voor deze netwerken. Event-gestuurde communicatie daarentegen lijkt vooralsnog het meest schaalbare communicatieparadigma voor deze netwerken. Dientengevolge moeten object-geörienteerde applicaties hun traditionele communicatie-abstractietechnieken inruilen voor event-gestuurde technieken. Deze integratie van objecten met events verloopt echter niet zonder slag of stoot, een fenomeen dat we de *object-event impedance mismatch* zullen dopen.

Om deze "impedance mismatch" te vermijden stellen we *ambient references* voor, een nieuwe programmeertaalconstructie voor het *aanwijzen van* en *communiceren met* objecten verspreid over een mobiel ad hoc netwerk. Ambient references vermijden de "impedance mismatch" door de *event broker* – de centrale component van een event-gedreven architectuur – voor te stellen als een gedistribueerde objectverwijzing. In tegenstelling tot klassieke objectverwijzingen dewelke een enkel uniek object aanwijzen, verwijst een ambient reference naar een vluchtige groep van nabije objecten. We beschrijven het ontwerp en de implementatie van ambient references in de context van *AmbientTalk/2*, een nieuwe ambient-georiënteerde programmeertaal.

Abstract

Two decades ago, Mark Weiser advanced a vision of the future of computing technology. In his vision of *ubiquitous computing*, the computer of the future will be invisible. It will be integrated into everyday objects, pervasive throughout our physical environment. As his vision gained momentum in computer science research, it became clear that the big hurdle to turn his vision into reality would not be the development of miniaturised, embedded, wireless hardware (which is a goal we have already reached today), but rather the development of the software that needs to run on this ubiquitous network of invisible computers.

This dissertation investigates scalable software engineering techniques for ubiquitous computing. From a distributed systems perspective, ubiquitous computing applications are distributed across a so-called *mobile ad hoc network*: a computer network of mobile devices that communicate via wireless technology. The idiosyncratic properties of such networks drastically change the assumptions underlying our contemporary software engineering abstractions. This observation has formed the direct motivation for a new breed of programming languages based on the *ambient-oriented programming* paradigm.

The goal of ambient-oriented programming is to mould distributed object-oriented software technology as to make it fit the changing hardware characteristics of mobile ad hoc networks. Within the framework provided by this paradigm, we survey existing abstractions to coordinate distributed processes and examine how they cope with the hardware characteristics of mobile ad hoc networks. As we will point out, contemporary object-oriented distribution abstractions are unsuitable for these networks. Event-based communication, on the other hand, has proven to be the most scalable communication paradigm to date. As a consequence, a state-of-the-art object-oriented application must abandon the classic distributed object-oriented abstractions in favour of an event-based communication paradigm. However, integrating objects with events is not without issues, leading us to define the *object-event impedance mismatch*.

In order to resolve this impedance mismatch, we propose *ambient references*, a novel language abstraction for *designating* and *communicating with* objects distributed across a mobile ad hoc network. Ambient references unify the concepts underlying both object-oriented and event-driven remote communication by turning the event broker – which lies at the heart of an event-driven, publish/subscribe architecture – into a remote object reference. Unlike a classic remote object reference, which designates a single unique object, an ambient reference designates a volatile group of proximate objects. We contribute the design and implementation of ambient references in the context of *AmbientTalk/2*, a novel ambient-oriented programming language.

Acknowledgements

First of all, I would like to heartily thank both of my promotors, Wolfgang De Meuter and Theo D'Hondt, for genuinely supporting my work. I am greatly indebted to Wolfgang in particular, who has supported me throughout every step of this research project, all the way from motivating me to formulate my nascent ideas in a workshop paper up to scrupulously proofreading this dissertation. I also want to explicitly thank him for inspiring me to do research on language design for mobile and ubiquitous computing. It was his "keukenpaper" (a paper more formally known by the name of "Wild abstraction ideas for highly dynamic software") that drew my attention to this topic when I was a graduate student.

I am indebted to my "reading committee" which, next to my promotors, included Elisa Gonzalez Boix and Jessie Dedecker. A special thank you goes to Elisa, who has had the courage to meticulously proofread every single sentence of my text. Proficiat! My thanks also go to Jessie, whose extensive knowledge on distributed programming has been of great help to me. I am also grateful to all of the members of my jury: Prof. Jean-Pierre Briot, Prof. Viviane Jonckers, Dr. Mark Miller, Prof. Kris Steenhaut and Prof. Dirk Vermeir for taking their time to be able to scrutinise this work. I would especially like to thank Mark Miller for his significant efforts to improve the quality and understanding of the text.

I am equally indebted to Stijn Mostinckx, who deserves a special attention as my inseparable partner in crime. Ever since our graduation thesis we have worked on a large part of our research in tandem, and I believe we have proven that the proverbial sum can indeed be greater than its parts. Stijn and myself are infamous for the heated discussions in our office. These discussions, however, have acted as the very engine of our research. The AmbientTalk/2 language discussed in this dissertation is our joint brainchild: evidently, every baby needs two parents.

The work described in this dissertation could not have been achieved without the proper "ambient" for it to grow in. Therefore, a big thank you goes to all of my past and present colleagues of the Programming Technology Lab. I would also like to thank the secretaries of the department for helping out with administrative issues on countless occasions.

Last but not least, I would like to thank my friends and family. In the first place, I owe not one but several debts of gratitude to my mother and father. They have supported my studies and provided me with the perfect place to write my text (that is, home). I am grateful also to my girlfriend Natalie and to my friends for providing me with the occasional moments of respite from research.

This work is funded entirely by a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

Contents

1 Introduction **1**
- 1.1 Research Context . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Research Goals . . . 4
- 1.4 A Language-oriented Approach . . . 5
  - 1.4.1 Domain-specific Languages . . . 5
  - 1.4.2 Language Integration and Impedance Mismatch . . . 5
  - 1.4.3 The Myth of Distribution Transparency . . . 7
- 1.5 Contributions . . . 8
- 1.6 Dissertation Roadmap . . . 9
- 1.7 Summary . . . 11

2 Ambient-Oriented Programming Revisited **13**
- 2.1 Motivation . . . 13
- 2.2 Mobile Ad Hoc Networks . . . 14
  - 2.2.1 Hardware Characteristics . . . 14
  - 2.2.2 Ad Hoc Networks versus Ad Hoc Applications . . . 16
- 2.3 Ambient-Oriented Programming . . . 17
  - 2.3.1 Classless Object Models . . . 17
  - 2.3.2 Non-Blocking Communication Primitives . . . 17
  - 2.3.3 Reified Communication Traces . . . 18
  - 2.3.4 Ambient Acquaintance Management . . . 18
  - 2.3.5 Summary . . . 19
- 2.4 AmbientTalk/1 . . . 19
  - 2.4.1 The AmbientTalk/1 Kernel Language . . . 19
  - 2.4.2 First-class Mailboxes . . . 20
  - 2.4.3 Ambient Acquaintance Management . . . 21
  - 2.4.4 AmbientTalk as a Language Laboratory . . . 21
- 2.5 Limitations of the AmbientTalk/1 Kernel . . . 23
  - 2.5.1 Limitations of the Object Model . . . 23
  - 2.5.2 Limitations of the Ambient Actor Model . . . 24
  - 2.5.3 Limitations of the Language Laboratory . . . 25
- 2.6 Conclusion . . . 26

3 Coordination in Mobile Ad hoc Networks **27**
- 3.1 Coordination . . . 27
- 3.2 Criteria for Coordination in MANETs . . . 29
  - 3.2.1 Decentralised Discovery . . . 29

3.2.2 Decoupled Communication . . . 30
3.2.3 Connection-independent Failure Handling . . . 33
3.2.4 Relation to Ambient-oriented Programming . . . 33
3.3 Survey of Related Work . . . 34
3.3.1 Languages for Local Area Networks . . . 35
3.3.2 Languages for Wide Area Networks . . . 35
3.3.3 Languages for Wireless Sensor Networks . . . 36
3.3.4 Models and Calculi for Wide Area Networks . . . 37
3.3.5 Tuple Space Middleware for Ad Hoc Networks . . . 39
3.3.6 Middleware for Nomadic Networks . . . 40
3.3.7 Publish-subscribe Middleware for Ad Hoc Networks . . . 42
3.3.8 Synthesis and Discussion . . . 45
3.4 The Object-Event Impedance Mismatch . . . 50
3.4.1 Specific versus Generic Communication . . . 51
3.4.2 Connection-oriented versus Connectionless Designation . . . 52
3.4.3 Bidirectional versus Unidirectional Communication . . . 53
3.4.4 Threads versus Event Loops . . . 53
3.4.5 Reconciling Objects with Events . . . 54
3.5 Conclusion . . . 55

4 AmbientTalk 57
4.1 History and Design Rationale . . . 57
4.2 AmbientTalk: an Object-oriented Language . . . 58
4.2.1 Objects, Instantiation and Delegation . . . 58
4.2.2 Block Closures . . . 60
4.2.3 Scoping, Nesting and Encapsulation . . . 61
4.2.4 Traits . . . 63
4.2.5 Type Tags . . . 65
4.2.6 Summary . . . 66
4.3 AmbientTalk: a Concurrent Language . . . 66
4.3.1 Event Loop Concurrency . . . 66
4.3.2 AmbientTalk Actors . . . 67
4.3.3 Message Passing Semantics . . . 68
4.3.4 Futures . . . 69
4.4 AmbientTalk: a Distributed Language . . . 74
4.4.1 Far References and Partial Failures . . . 75
4.4.2 Exporting Objects . . . 76
4.4.3 Service Discovery . . . 76
4.4.4 Partial Failures Revisited . . . 78
4.5 Discussion . . . 78
4.5.1 Event-driven Object-oriented Programming . . . 79
4.5.2 Suitability for Mobile Ad Hoc Networks . . . 80
4.6 Case Study: the Musical Match Maker . . . 81
4.6.1 Data Abstractions . . . 82
4.6.2 Exporting and Discovering Service Objects . . . 82
4.6.3 The Library Transmission Protocol . . . 83
4.6.4 Failure Handling . . . 85
4.7 Limitations and Future Work . . . 86
4.8 Notes on Implementation Status . . . 90
4.9 Previous and Related Work . . . 91

4.9.1 AmbientTalk/1 versus AmbientTalk/2 . . . 91
4.9.2 Notes on Related Work . . . 93
4.10 Conclusion . . . 95

5 Metalevel Engineering in AmbientTalk 97
5.1 First-class Messages and Methods . . . 97
5.1.1 First-class Messages . . . 97
5.1.2 First-class Methods . . . 98
5.2 Reflection . . . 99
5.2.1 Mirror-based Reflection . . . 99
5.2.2 Mirages: Mirror-based Intercession . . . 102
5.2.3 First-class References as Mirages . . . 103
5.2.4 Stratified Object References . . . 107
5.3 Linguistic Symbiosis with the JVM . . . 108
5.3.1 Linguistic Symbiosis . . . 109
5.3.2 Composing Threads with Actors . . . 111
5.3.3 Embedding AmbientTalk in Java . . . 113
5.4 Conclusion . . . 115

6 Ambient References 117
6.1 Motivation . . . 117
6.1.1 Roaming . . . 117
6.1.2 One-to-many Communication . . . 120
6.1.3 Provisional Services . . . 123
6.1.4 Summary . . . 126
6.2 Ambient References in a Nutshell . . . 126
6.2.1 Example: Broadcasting Stock Quote Updates . . . 126
6.2.2 Space-decoupled Object References . . . 128
6.3 Decomposing Ambient References . . . 129
6.4 Ambient references in AmbientTalk . . . 132
6.4.1 Scope . . . 133
6.4.2 Service Objects . . . 136
6.4.3 Arity . . . 138
6.4.4 Communication Lifetime . . . 140
6.4.5 Discovery Lifetime . . . 144
6.4.6 Relating Discovery Lifetime and Communication Lifetime . . . 147
6.4.7 Summary . . . 150
6.4.8 Interactions between Delivery Policies . . . 150
6.5 Delivery Guarantees . . . 152
6.5.1 Point-to-point Messages . . . 152
6.5.2 One-to-many Messages . . . 153
6.5.3 Delivery Order . . . 154
6.6 Reintroducing Connection-oriented Designation . . . 155
6.6.1 Anonymous Far References . . . 155
6.6.2 Snapshots . . . 158
6.6.3 Multireferences . . . 159
6.6.4 Summary . . . 160
6.7 On the Scale of Time and Space . . . 160
6.8 Conclusion . . . 161

7 Ambient References in Context **163**
7.1 Evaluation . . . 163
7.1.1 Decentralised Discovery . . . 163
7.1.2 Loosely-coupled Communication . . . 164
7.1.3 Connection-Independent Failure Handling . . . 167
7.1.4 Summary . . . 168
7.2 The Object-Event Impedance Mismatch Revisited . . . 168
7.2.1 Specific versus Generic Communication . . . 168
7.2.2 Connection-oriented versus Connectionless Designation . . . 169
7.2.3 Bidirectional versus Unidirectional Communication . . . 170
7.2.4 Threads versus Event Loops . . . 170
7.2.5 Reconciling Objects with Events . . . 171
7.3 Relation to Prior Work . . . 172
7.3.1 Elasticity . . . 172
7.3.2 Cardinality . . . 173
7.3.3 Reference-centric versus Message-centric View . . . 173
7.4 Limitations and Future Work . . . 175
7.5 Notes on Related Work . . . 175
7.5.1 Tuple Spaces . . . 176
7.5.2 Actors and Far References . . . 177
7.5.3 M2MI Handles . . . 178
7.5.4 ActorSpace . . . 179
7.5.5 One.world . . . 180
7.5.6 Distributed Asynchronous Collections . . . 183
7.5.7 Joule Channels . . . 184
7.6 Conclusion . . . 184

8 Implementing Ambient References **185**
8.1 Implementation Strategies . . . 186
8.2 Implementation Outline . . . 186
8.3 Extensional Reach . . . 188
8.3.1 Representing Reach . . . 188
8.3.2 Delivery Policies as Traits . . . 189
8.3.3 Representing Discovery Lifetime . . . 190
8.3.4 Representing Arity . . . 192
8.3.5 Representing Communication Lifetime . . . 194
8.3.6 Representing Expirable Messages . . . 195
8.3.7 Representing Exported Objects . . . 196
8.3.8 Snapshots . . . 196
8.3.9 Summary . . . 197
8.4 Intensional Reach . . . 198
8.4.1 Representing Reach . . . 198
8.4.2 Delivery Policies as Traits . . . 199
8.4.3 Representing Arity . . . 199
8.4.4 Representing Discovery Lifetime . . . 202
8.4.5 Communication Lifetime and Expirable Messages . . . 203
8.4.6 Representing Exported Objects . . . 203
8.4.7 Snapshots . . . 205
8.4.8 Summary . . . 205
8.5 Evaluation . . . 205

8.6 Ambient References as Custom Eventual References 207
8.7 Many to Many Invocations 209
8.7.1 Motivation 209
8.7.2 Applying M2MI 210
8.7.3 Using M2MI 210
8.8 Implementing Connection-oriented References 213
8.8.1 Anonymous Far References 213
8.8.2 Multireferences 214
8.9 Marshalling Ambient References 214
8.10 Garbage Collection 215
8.11 Conclusion 216

9 Ambient References in Action **217**
9.1 Collaborative Chat 218
9.1.1 Implementation via ambient references 218
9.1.2 Evaluation 221
9.2 Collaborative Slideshow 222
9.2.1 Implementation via Ambient References 223
9.2.2 Evaluation 226
9.3 Comparing Ambient References with M2MI 227
9.3.1 Roaming 227
9.3.2 One-to-many Communication 229
9.3.3 Provisional Services 231
9.4 Conclusion 232

10 Conclusion **235**
10.1 Research Goals 235
10.2 Restating the Contributions 236
10.3 Limitations of our Approach 238
10.3.1 Language Integration versus Language Separation 238
10.3.2 Custom Message Delivery Policies 238
10.4 Work influenced by Ambient References 239
10.5 Avenues for Future Research 241
10.5.1 Aspect-oriented Programming 241
10.5.2 Service Selection 242
10.5.3 Session Types 242
10.6 Concluding Remarks 243

A Ambient References Source Code **245**
A.1 Ambient References Language Module 245
A.2 Custom Eventual References Module 251
A.3 Extensional Implementation Module 253
A.4 Intensional Implementation Module 257

Bibliography **262**

Index **277**

List of Figures

3.1 Contrasting object-oriented with event-driven communication. 51

4.1 AmbientTalk actors as communicating event loops. 68
4.2 The library transmission protocol. 84

5.1 Reflective view on an AmbientTalk actor. 101
5.2 Entities in the linguistic symbiosis between AmbientTalk and the JVM. 110
5.3 Mediating between JVM invocations and AmbientTalk messages. . . 112

6.1 Scope, communication range and reach of an ambient reference. . . . 130
6.2 Point-to-point ambient message delivery. 138
6.3 One-to-many ambient message delivery. 139
6.4 Delivery of an expirable one-to-many ambient message. 149
6.5 Taxonomy of Ambient Message Delivery Policies. 150
6.6 Impact of an ambient reference's recall period on message delivery. . 154
6.7 An unbound anonymous far reference. 157
6.8 A bound anonymous far reference. 157
6.9 Flow chart classifying referencing abstractions according to designation. 160

8.1 Overview of the implementation of ambient references. 187
8.2 Delivery policy traits used by a message handler (extensional impl.). . 190
8.3 Multiple dispatch implementing ambient message delivery. 197
8.4 Delivery policy traits used by a message handler (intensional impl.). . 199
8.5 Anycast protocol to select a single receiver. 201

9.1 Structure of the collaborative chat application in AmbientTalk. 218
9.2 Structure of the collaborative slideshow application. 223
9.3 Protocol to transmit a slideshow from a projector to a group of screens. 223

10.1 Abstraction afforded by space-decoupled communication. 241

List of Tables

3.1 Service Lookup versus Service Discovery. 30
3.2 Survey of Related Work. 46
3.3 Degrees of coupling and failure handling. 47

4.1 Overview of AmbientTalk's units of operation. 74

6.1 Combining communication (rows) and discovery (columns) lifetimes. 148

7.1 Comparing Ambient References to Event notification systems. 168

Chapter 1

Introduction

This dissertation is about programming language abstractions for ubiquitous computing. Ubiquitous computing is a research vision (proposed by Mark Weiser [Wei91]) in which computer technology becomes pervasive throughout our daily lives. Weiser predicts a "disappearing computer": in his vision, the technology of the future no longer demands the focus and active attention of the user. Rather, computer technology will become an inadvertent part of our physical environment. This also brings about a change in scale: whereas the era of the Personal Computer is characterised by a one-to-one mapping between computer and user, in the era of ubiquitous computing, thousands of microprocessors will be embedded in the environment to assist users. While ubiquitous computing is not yet happening in the large, we can see the first signs of it happening in the small. In the automotive industry, for example, most of today's modern cars are already equipped with dozens of microprocessors which each try to assist the user in an invisible or unobtrusive way. Many people today already have a "personal area network" of mobile devices such as PDAs, cellular phones or music players. It will not be long before our clothing will be equipped with sensors and microprocessors that communicate with washing machines and wardrobes. Even today it is already possible to buy running shoes equipped with sensors which wirelessly communicate with a music player.

A key enabler for ubiquitous computing to happen is the networking technology by which the different devices in our environment will collaborate with one another. Because ubiquitous computing deals with dynamic networks where nodes spontaneously and continuously come and go (because people move about) and moreover because it stresses the unobtrusiveness of computer technology, it is clear that ubiquitous computing will rely heavily on so-called *mobile ad hoc* computer networks (MANETs). Such networks enable computers to communicate with one another invisibly (they employ *wireless* communication) and unobtrusively (they do not or should not require laborious configuration or administration to set up, they form spontaneously simply by collocating devices).

While we see rapid developments at the level of hardware (increased accuracy of sensors, reliability of wireless communication, etc.), the developments at the software engineering level are nowhere near as groundbreaking. Today, developers find themselves working with tools (in the broadest sense of the word, including analysis and design techniques, middleware, programming languages, etc.) which have come of age in the era of the personal computer, *not* that of the ubiquitous computer. As a result, there is a huge gap between the novel ubiquitous hardware waiting to be pro-

grammed and the software tools available to do so. A testament to this is the launch of a plethora of research initiatives over the past few years that target "software engineering for ubiquitous computing" in one way or another. Examples include the EU IST Advisory Group's vision of "Ambient Intelligence" [Gro03], where the emphasis is on smart homes and domotics, IBM Research's "Autonomic Computing" initiative [KC03], where the emphasis is on self-configurable computer networks and their services, and the International Telecommunication Union's vision of the "Internet of Things" [Int05], envisioning a world wide web of everyday material objects that can communicate with one another.

The vision behind our research is to ultimately close the gap between ubiquitous computing and today's software technology. Needless to say, this dissertation represents but a small step towards such an ambitious goal. In particular, we will focus on how to reconcile traditional software abstractions for expressing communication between distributed processes with the novel type of mobile ad hoc networks. More concretely, we propose a novel object-oriented programming language abstraction which we term "ambient references". It is the explicit design goal of ambient references to be "remote object references" that are suitable for use in mobile ad hoc networks. Like a classic remote object reference, an ambient reference designates remote objects and can carry messages to those objects. Unlike a classic remote object reference, which designates a single unique object, an ambient reference designates a *volatile group of proximate objects*. We extensively explore the effects of this new form of object-oriented designation on message passing.

In the remainder of this introductory chapter, we first sketch the research context of this work. We highlight the problems to be tackled and how we intend to solve them. Furthermore, we extensively discuss our programming language-driven approach to solving the problem. We conclude the chapter with a preliminary overview of this dissertation's contributions and a roadmap to assist the reader in browsing the text.

1.1 Research Context

Our research lies within the intersection of four research domains. We describe each of the domains, sketching their relationship with one another as we go along.

Ubiquitous computing has already been briefly discussed in the introduction. The vision has spawned a tremendous amount of research in different areas of computer science: researchers investigate sociological impacts, new ways of human-computer interaction, privacy and security issues, low-level sensor technology, etc. We are looking at ubiquitous computing from a software engineering perspective. Concretely, we focus on the distributed aspects of such software. Many scenarios that extoll the vision of ubiquitous computing are built upon ad hoc networking technology, which brings us to the next point.

Mobile ad hoc networks are computer networks composed of mobile devices whose topology dynamically changes as devices move about. Often, devices communicate with one another through wireless communication technology (e.g. Bluetooth, WiFi, ZigBee,...) to guarantee maximum mobility of the devices. Because of the use of wireless communication, devices can communicate and cooperate with other devices in their proximity unobtrusively. This makes such computer networks a key technological enabler of the vision of ubiquitous computing. However, the research area of mobile ad hoc networks itself remains vast,

with many technological issues left to improve. We are particularly interested in software development for mobile ad hoc networks. More specifically, we want to investigate scalable coordination models for the software running on the mobile nodes of an ad hoc network.

Coordination abstractions Because mobile networks engender an extremely dynamic network topology, traditional coordination models to express communication between distributed processes fail to provide the right abstraction boundaries for application developers. New abstractions are required which are designed from the ground up to be used in mobile ad hoc networks.

Programming language design Coordination abstractions can take on numerous guises. They can be abstract calculi, design patterns, operating system constructs, the APIs of frameworks or libraries, or new programming language constructs. In this dissertation, we pursue the development of coordination abstractions as programming language constructs. We extensively motivate this choice in section 1.4.

1.2 Problem Statement

Mobile ad hoc networks exhibit different properties than traditional, fixed computer networks. While we defer an in-depth discussion of these properties until section 2.2, it is clear that fundamental changes in the underlying network technology have an impact on the way distributed software running on top of them is conceived. Abstraction layers for distributed computing are always built with certain assumptions in mind. Example assumptions could be "communication is for the most part reliable" or "the network is mostly totally connected" or "device failures are rare" (which are all valid assumptions in a contemporary local area network). However, when the properties of the underlying network change, some of these assumptions fail to hold any longer. As a result, contemporary abstractions for distributed computing fail to provide adequate abstraction boundaries in mobile ad hoc networks.

We are not the first to note this discrepancy between traditional distributed computing abstractions and ad hoc networking technology. As we will describe in chapter 3, many new abstractions have been proposed which *do* scale in mobile ad hoc networks. What gap are we trying to close, then? As we will extensively discuss later, it turns out that none of the emerging coordination abstractions for mobile ad hoc networks are object-oriented but rather event-driven, while the lion's share of today's software is written in object-oriented programming languages. This leads us to postulate the *object-event impedance mismatch*: a characterisation of the difficulties in combining the object paradigm with the event-driven communication paradigm for the purposes of distributed computing.

The observation that the idiosyncratic properties of mobile ad hoc networks require changes at the level of distributed programming abstractions also forms the original motivation behind the ambient-oriented programming paradigm, as proposed by Jessie Dedecker [Ded06]. The goal of ambient-oriented programming (AmOP) is to mould distributed object technology as to make it suitable for the next generation of computer networks. Within the context demarcated by the AmOP paradigm, this work specifically addresses the issue of *designating* and *communicating with* objects in a mobile ad hoc network:

Designation In a mobile ad hoc network, applications cannot rely on centralised servers that are accessible to all peers in the network. This has a large impact on the way objects in an object-oriented distributed program can designate one another. The traditional method of resolving symbolic names into object references via centralised directories or name servers ceases to work in mobile ad hoc networks, requiring us to rethink the way objects address remote peers.

Communication The wireless technology employed in mobile ad hoc networks undermines the scalability of the traditional "remote method invocation", which is the de facto standard object-oriented distributed communication abstraction. The pillars on which it builds – reliable communication links with a relatively low network latency – are essentially nonexistent in mobile ad hoc networks. It requires us to rethink the way objects communicate with remote peers.

1.3 Research Goals

The goals of the research described in this dissertation are threefold:

- We want to investigate which coordination abstractions are appropriate for use in a mobile ad hoc network and which are not. By studying both classic and state of the art abstractions documented in the literature, we want to come up with a number of criteria that capture the essence of coordinating concurrent processes in a mobile ad hoc network. As described previously, the outcome of our investigation reveals that classic object-oriented abstractions for distributed computing fail to adhere to the postulated criteria. Therefore,

- We want to uncover why scalable coordination abstractions for mobile ad hoc networks are not based on (or not well integrated with) object-oriented principles and the message passing metaphor to represent distributed communication. In other words, we want to lay bare the causes of the aforementioned object-event impedance mismatch. Based on this,

- We want to resolve the object-event impedance mismatch by developing a coordination abstraction that is both object-oriented and event-driven. As such, we want to show (by means of a "proof by construction") that it is possible to make distributed object technology scalable in mobile ad hoc networks.

We achieve our proof by construction by developing a novel language construct in a novel programming language. Together, they form the chief contributions of this dissertation:

AmbientTalk/2 In chapters 4 and 5, we extensively describe AmbientTalk/2, a novel ambient-oriented programming language which will serve as the core technological platform in which we can experiment with novel coordination abstractions for mobile ad hoc networks. Importantly, AmbientTalk/2 is an object-oriented language whose execution model is entirely and inherently event-driven. As we will see, this enables distributed language abstractions to be built on top of the language which can be naturally integrated with the event-driven communication paradigm.

Ambient References Chapters 6,7,8 and 9 respectively describe the design, evaluation, implementation and application of ambient references as a novel object-oriented event-driven coordination abstraction for mobile ad hoc networks. Ambient references enable objects to intensionally *designate* proximate objects without the need to resolve names into references. They enable objects to *communicate* with proximate objects using the event paradigm without giving up on the object referencing or message passing metaphors of the object paradigm.

In the following section, we motivate *why* we developed ambient references as a novel language construct in a novel programming language.

1.4 A Language-oriented Approach

The work described in this dissertation puts a strong emphasis on programming language design. This choice is motivated both by cultural as well as practical considerations. Language research at the Programming Technology Lab has a long history of experiments performed in so-called *little languages* [Ben86]. Examples include Pico [D'H96], Agora [Ste94b], ChitChat [De 04, DTM+05] and AmbientTalk/1 [Ded06].

1.4.1 Domain-specific Languages

While Bentley uses the term "little language" in his 1986 CACM column, contemporary computer scientists are more familiar with the term "domain-specific languages" (DSLs) [MHS05]. What is it that makes such languages attractive from a language designer's perspective? The major asset of a DSL is that it has the potential of augmenting the expressive power to solve a specific class of problems. Common to all DSLs is that they do this by making the data structures and operations from their problem domain the basic building blocks of the language. Examples of DSLs truly abound: YACC is a language for describing grammars, so its basic building blocks are grammatical rules and tokens; LaTeX is a language for typesetting documents, to its basic building blocks include sections, paragraphs and references; Make and ANT are languages for build automation, so their basic building blocks include files and dependencies; Linda is a process coordination language so its basic building blocks include primitives for process communication; AmbientTalk/1 is a language for coordinating asynchronous distributed objects, so its basic building blocks include message queues and service discovery; and so forth.

While DSLs certainly have their merits, they are not a panacea. It should not be forgotten that their expressive power is reciprocal to their generality. This explains why the creation of a DSL is often a painstaking process of balancing between domain-specific features and general-purpose language constructs.

1.4.2 Language Integration and Impedance Mismatch

One advantage of domain-specific languages over conventional libraries or middleware is their ability to enforce certain global rules or a certain programming style by design. It is this advantage which we will exploit throughout the rest of this dissertation. Being able to enforce certain properties becomes particularly important when considering the *composition* of two or more systems. It is well-known in computer science, and other engineering disciplines, that the composition of two systems that foster their own

style sometimes leads to an "impedance mismatch". Quoting Lämmel and Meijer, who themselves derive their definition from Wikipedia [LM07]:

> 'Impedance mismatch' is derived from the usage of impedance as a measurement of the ability of one system to efficiently accommodate the output (energy, information, etc.) of another. [...] Although the term originated in the field of electrical engineering, it has been generalized and used as a term of art in systems analysis, electronics, computer science, informatics, and physics. In particular, the term is also used "to refer to the difficulties encountered when attempting to connect two systems which have very different conceptual bases".

In computer science, the term most often refers to the *object-relational* impedance mismatch, describing the difficulties of mapping objects onto relational tables and vice versa [CD96]. Similar problems exist in mapping between objects and XML documents [LM07]. Indeed, the term is often used to describe the difficulties of converting between different data formats. In this dissertation, we study impedance mismatch in a different sense and consider the difficulties that arise when composing *language features*. There are many examples in computer science where different language features fail to compose well:

- Functional programming languages exhibit a property known as *referential transparency*, which implies that any expression can always be substituted for its value without any effect on the semantics of the program (in other words: invoking a function with the same input always yields the same output). However, such a functional system cannot be composed safely with a system that allows assignment: the composite system can no longer guarantee referential transparency.

- Logic programming languages with a goal-driven derivation procedure exhibit a property known as *backtracking*, which occurs when the search algorithm conducting a proof derivation considers alternatives after descending down a failing branch of the search tree. When backtracking, substitutions for logic variables made in the failing subtree are automatically "undone", enabling the programmer to abstract over failed proof derivations. Again, if such a system is composed with a system allowing assignment (whether via features like Prolog's `assert` or via linguistic symbiosis with a non-functional language [GWDD06]), then the composite system loses the abstraction barrier, as side-effects are not undone via backtracking. Gybels et al. have named this impedance mismatch a *paradigm leak*, because the effects of one paradigm become visible to the other paradigm which cannot cope with them appropriately.

- In an object-capability system, objects are capability secure because they can only be affected by objects that have a reference to them and because there exists a well-defined operator for controlling the spread of references in the system (the "Granovetter" operator [MMF00]). However, composing such a system with one in which references are forgeable (e.g. by casting an integer address into a pointer in C or C++) leads to a system that is not capability secure: the rules that previously allowed the system to enforce security can now be circumvented thanks to operators beyond its control.

- In a system with event loop concurrency (cf. section 4.3.1), concurrency within one event loop is restricted by processing events sequentially. Hence, all events

> run in mutual exclusion and it is not necessary to introduce additional synchronisation constructs within the event loop itself. However, composing an event loop system with a multithreaded system leads to a composite where the concurrency invariants of event loops are easily violated by threads accessing an event loop's objects [VMD07].

In all of the above cases, the problem is that a global property of one system can no longer be a property of the composite due to a conflicting feature of the other system, creating an impedance mismatch. How is it that a language-oriented approach can avoid or mitigate the impedance mismatch? When designing a library or middleware as a separate system, with its own rules and programming style, one *necessarily* needs to compose that system with its "host" language, the (mostly general purpose) language in which it is implemented. As a result, some properties of the library/middleware may not be globally enforced. Whether or not this is an actual problem depends on the nature of the property: while the backtracking/assignment problem can be seen as a nuisance, a security leak in a capability-secure system is a serious issue. Nevertheless, by building a system as a well-designed language, the impedance mismatch can be avoided because there is no need to compose that language with an implementation system. The language interpreter has total control over every aspect of the system's execution.

Because a programming language makes it possible to truly *enforce* a certain style of programming to tackle a certain type of problem, a good language design can enforce the programmer to use a programming style which has been carefully thought out to better match the problem at hand. To restate this with a perlisism: "a language that doesn't affect the way you think about programming, is not worth knowing" [Per82].

1.4.3 The Myth of Distribution Transparency

Throughout the history of distributed computing, there has been a consistent desire to abstract from the network layer as much as possible. This desire is justified because programming a local application is easier than programming a distributed one, so the more distributed software development can resemble traditional software development, the better. Or so it seems. Following the widespread use of the remote procedure call [BN84], researchers and practitioners alike came to see that distributed programming is inherently more complex than local programming, and that distributed computing cannot be subsumed by traditional software engineering techniques. Rather than becoming the holy grail, distribution transparency turned into myth [GF99]. However, this has not kept researchers from developing novel abstractions for aiding in the development of distributed applications. On the contrary, researchers are now aware of the fallacies of the myth and design their abstractions accordingly. Quoting Eugster et al. on the goals behind their Distributed Asynchronous Collections abstraction (cf. section 7.5.6) [EGS00], p. 274:

> "[...] distribution transparency is a myth that is both misleading and dangerous. Distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable to that of a local interaction. The possibility of partial failures can fundamentally change the semantics of an invocation. High availability and masking of partial failures involves distributed protocols that are usually expensive and hard, if not impossible to implement in the presence of network failures (*partitions*).

> We have been considering an alternative approach where the programmer would be very aware of distribution but where the ugly and complicated aspects of distribution would be encapsulated inside specific abstractions with a well-defined interface."

In this dissertation, we will follow exactly such an approach. While AmbientTalk/2 and ambient references strive to enable the programmer to abstract from the underlying network as much as is practically possible, it is never their goal to provide distribution transparency. A developer using them must be very aware of the issues engendered by distributed computing, but *only* when they are fundamental in nature.

1.5 Contributions

It is always difficult to provide an overview of contributions in advance, with the problem statement only vaguely introduced and without the technical foundations required to support them. However, listing the contributions early on helps to sketch the context and subject domain of the dissertation. In short, this dissertation makes the following conceptual and technical contributions in the intersecting domains of mobile ad hoc networks, programming language research and coordination languages and models:

- We fine-tune the definition of the original ambient-oriented programming paradigm based on the fundamental properties that discriminate mobile ad hoc networks from traditional networks. We revisit the first ambient-oriented language (AmbientTalk/1) and highlight its strengths and weaknesses (cf. chapter 2).

- We describe a set of criteria which highlight what properties a programming abstraction for coordinating distributed processes must posses in order for it to scale in a mobile ad hoc network (cf. section 3.2).

- We provide an extensive survey of related work on coordination abstractions for mobile ad hoc networks (cf. section 3.3). As will be described, the outcome of the survey is that event-based abstractions are the most scalable in a MANET.

- We describe the difficulties of integrating object-oriented programming with event-driven programming for the purposes of distributed computing and name this phenomenon the object-event impedance mismatch (cf. sections 3.4 and 7.2).

- We provide a second embodiment of the ambient-oriented programming paradigm. More specifically, we introduce the AmbientTalk/2 programming language, the successor to the original AmbientTalk language developed by Dedecker [Ded06] (cf. chapter 4). The design and implementation of this language are not solely this author's contribution: they can be attributed to multiple researchers of the Programming Technology Lab.

- We discuss metalevel engineering techniques in AmbientTalk (cf. chapter 5). Most importantly, we discuss a mirror-based reflective architecture that allows custom object references to be introduced reflectively in AmbientTalk. We also contribute a technique to compose AmbientTalk's event-driven actor model with the threads of the Java Virtual Machine using linguistic symbiosis.

- We contribute a novel kind of distributed language abstraction which we name ambient references (cf. chapter 6). Ambient references are designed specifically for designating and communicating with remote objects across a mobile ad hoc network. They define a taxonomy of different message passing semantics which allows the expression of many different kinds of interaction patterns within a single unified referencing framework.

- We show how ambient references unify object-oriented with event-driven communication. We also relate them to existing referencing abstractions described in the literature by reformulating these abstractions in terms of ambient references (cf. chapter 7).

- We show how AmbientTalk/2, extended with the referencing abstractions introduced in chapter 6 forms a suitable platform for expressing coordination between objects distributed across a MANET, by showing how it satisfies the different coordination criteria introduced earlier in chapter 3 (cf. section 7.1).

- We describe a concrete implementation of ambient references (in AmbientTalk) which uses state of the art object-oriented composition techniques to modularise the different kinds of message passing semantics (cf. chapter 8). The implementation depends on a complex yet robust combination of computational reflection and linguistic symbiosis. Computational reflection is used to represent ambient references as a special kind of "object references" in the language, while linguistic symbiosis enables ambient references to be built on top of the M2MI framework, a Java library for performing many-to-many communication in ad hoc networks [KB02].

- We validate ambient references by employing them to implement two representative collaborative applications designed to run in a mobile ad hoc network. Subsequently, we show how the communication patterns which ambient references can readily express would be implemented in Java using M2MI. Based on these implementations, we can show how ambient references improve upon the state of the art (cf. chapter 9).

1.6 Dissertation Roadmap

Ambient references form the main scientific contribution of this work. However, before going into details on ambient references, we first sketch the research context in which they have been developed. To ground the technical discussion, we also need to introduce AmbientTalk, the programming language in which ambient references have been incorporated and implemented. Once the core concepts comprising ambient references have been introduced, ambient references themselves can be placed in context. Below, we summarise each subsequent chapter in the dissertation.

Chapter 2: Ambient-oriented Programming Revisited provides the reader with an introduction to the ambient-oriented programming paradigm. We describe the paradigm as originally proposed by Dedecker [Ded06] but also adjust its characteristics where appropriate. In particular, this chapter clearly highlights what properties are fundamental to mobile ad hoc networks and require a rethink of

the classic programming abstractions for distributed applications. These properties form the fundamental assumptions underlying a set of criteria defined in the next chapter.

Chapter 3: Coordination in Mobile Ad hoc Networks puts forward a set of criteria which characterise whether or not a certain coordination abstraction is suitable for use in a mobile ad hoc network. In the subsequent survey of related work, these criteria serve as a discriminant for determining the most suitable approaches to coordination in mobile ad hoc networks. The outcome of this survey is that event-based approaches scale much better in a MANET than the traditional object-oriented approaches. However, composing object-oriented programs with event systems for the purposes of distributed communication is not without problems, leading us to define the *object-event impedance mismatch*. The main motivation of this dissertation is how to resolve this impedance mismatch by means of appropriately designed language constructs, to wit ambient references.

Chapter 4: AmbientTalk introduces the AmbientTalk/2 programming language. This language is the successor of the original AmbientTalk language developed by Dedecker [Ded06]. We discuss those language features required to understand the technical details of ambient references. The focus is on the language's object-oriented, concurrent and subsequently distributed language features. These features are then put to work in a concrete case study (a "musical match maker" ad hoc application). We also highlight how AmbientTalk in itself already partly reconciles objects with events. Finally, the AmbientTalk language design is put in perspective by relating it to the original AmbientTalk language and other prior work.

Chapter 5: Metalevel Engineering in AmbientTalk discusses AmbientTalk's reflective architecture, how custom object references can be added to the language reflectively and how AmbientTalk programs can interoperate with the Java Virtual Machine. Because embedding an actor-based language like AmbientTalk into a thread-based language like Java is not without conceptual problems, we devote a section to explain the embedding process in more detail. Reflection and linguistic symbiosis are subsequently employed in chapter 8 to implement ambient references in AmbientTalk.

Chapter 6: Ambient References describes ambient references from the designer's point of view. It is an exposition of the principal contribution of the dissertation. Even though AmbientTalk already features high-level language abstractions, we will show in an example-driven manner that they are insufficient to express many collaborations in mobile ad hoc networks. Subsequently, we define ambient references and describe a taxonomy of the different kinds of communication patterns which they support.

Chapter 7: Ambient References in Context describes ambient references from the computer scientist's point of view. It abstracts from the technical details of ambient references and relates them to previous work. Also, we evaluate ambient references by means of the criteria set forth in chapter 3. Finally, with both AmbientTalk and ambient references introduced, we describe how their combined properties facilitate the combination of objects with events to coordinate distributed processes, in part mitigating the object-event impedance mismatch.

Chapter 8: Implementing Ambient References describes ambient references from the implementor's point of view and once again exposes the reader to the technicalities of AmbientTalk and ambient references. We describe and contrast two strategies for implementing ambient references. We discuss how the metalevel engineering techniques introduced in chapter 5 can be employed to implement ambient references reflectively in AmbientTalk.

Chapter 9: Ambient References in Action describes ambient references from the application programmer's point of view. We describe two small but realistic mobile ad hoc networking applications and compare an implementation using ambient references with an implementation using M2MI [KB02].

Chapter 10: Conclusion summarises the contributions made in this dissertation. At that point we are able to evaluate the contributions of the dissertation with hindsight, naturally leading to a discussion on the limitations of this work and on possible directions for future research.

1.7 Summary

This dissertation is an exposition about novel object-oriented referencing and communication abstractions for mobile ad hoc networks. We focus on such networks because they form an important technology underlying the vision of ubiquitous computing. We will describe why and how scalable coordination abstractions for mobile ad hoc networks are not based upon object-oriented principles. To resolve this mismatch, we propose a novel language construct (ambient references) in a novel programming language (AmbientTalk/2) which reconciles object-oriented distributed computing with mobile ad hoc networks.

The next chapter provides the reader with the necessary background information on mobile ad hoc networks and their relation to ambient-oriented programming. Afterwards, in chapter 3, this background information will prove to be crucial to understand which coordination abstractions scale in mobile ad hoc networks. Chapter 2 also discusses AmbientTalk/1, the first ambient-oriented language. This language's limitations form the motivation behind AmbientTalk/2, which is introduced in chapter 4.

Chapter 2

Ambient-Oriented Programming Revisited

This dissertation's contribution lies in the domain of *ambient-oriented programming*, a novel programming language paradigm proposed by Dedecker et. al [DVM+05, DVM+06a, Ded06]. As the work described in this dissertation builds upon this prior work, this chapter provides a summary of ambient-oriented programming to serve as a frame of reference when discussing object designation using ambient references in later chapters. We will repeat in full the *motivation* behind ambient-oriented programming, which are the idiosyncratic properties of mobile ad hoc networks. We also briefly introduce AmbientTalk/1, the first ambient-oriented programming language. We describe what key features of this language make it suitable for use in mobile ad hoc networks. We also describe issues and experiences with the language, motivating the need for its successor language, AmbientTalk/2.

2.1 Motivation

As discussed in the introduction, mobile ad hoc networks constitute the network technology of ubiquitous computing. The salient properties of such networks are explained in the following section. In a nutshell, the most difficult problems are that network connections between devices are unstable due to the limited wireless communication range of participating devices, that the network is open – (new) devices frequently join and leave the network – and has little or no infrastructure (e.g. to support globally accessible servers).

Contemporary distributed programming languages, middleware and libraries offer abstractions that are built with different assumptions of the properties of the underlying communications network. For example, in a conventional distributed system, a network partition is regarded as a failure, i.e. an exceptional event. In a mobile network, disconnections between devices become the norm, rather than the exception; this change in the physical nature of the network percolates all the way up to the application layer [MCE02]. However, the intermediary abstractions between the application and network layer cannot optimally cope with these changes, resulting in seemingly ill-defined abstractions for the application programmer. For example, handling failing communication using the classic programming language abstractions of exception handling results in application code that is literally polluted with failure handling. An

ambient-oriented language foregoes such issues by changing the intermediary abstraction barriers in order to fit the changed hardware phenomena of the underlying network itself [DVM+06a].

As our work builds on the ambient-oriented programming paradigm, we recapitulate the properties of mobile ad hoc networks in the following section.

2.2 Mobile Ad Hoc Networks

A good definition of mobile ad hoc networks given by Murphy et al. [MRV98] and also quoted in [GR03] is the following:

Definition 1 (Mobile Ad Hoc Network) *A transitory association of mobile nodes which do not depend upon any fixed support infrastructure. [...] An ad-hoc network can be visualized as a continuously changing graph. Connection and disconnection is controlled by the distance among nodes and by willingness to collaborate in the formation of a cohesive, albeit transitory community.*

The two key points in the above definition are the fact that devices are *mobile* and that there is no *fixed* infrastructure, resulting in *wireless* peer-to-peer interaction among the nodes.

Note that this definition allows for quite a broad range of possible networks, as the type of mobile device and wireless communication medium can vary, catering to a diverse set of envisaged applications. Devices might be as small as coins, embedded in material objects such as wrist watches, door handles, lamp posts, cars, etc. They may even be as lightweight as sensor nodes or they may be material objects "digitized" via an RFID tag[1]. Devices may also be as "heavyweight" as a cellular phone, a PDA or a car's on-board computer. All of these devices can in turn be interconnected by a diverse range of wireless networking technology, with ranges as wide as those of WiFi technology or as limited as those of IrDA (infrared).

2.2.1 Hardware Characteristics

Mobile ad hoc networks composed of mobile devices and wireless communication links exhibit a number of phenomena which are rare in their traditional, fixed counterparts. In his original work, Dedecker identified four discriminating "hardware phenomena" that are inherent to mobile networks [DVM+06a]. In this work, we claim that only two of the four phenomena are fundamental and we argue how the other two can be derived from the fundamental ones. The two fundamental characteristics of a MANET are:

Volatile Connections (originally named **Connection Volatility**). Mobile devices equipped with wireless communication media only have a limited communication range. This implies that communicating devices may move out of earshot at any time because of the happenstance of node mobility. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Such network connections are also said to be intermittent [MCE02]. Quite often, transient network partitions should not abort

[1] Such tags can be regarded as tiny computers with an extremely small memory, able to respond to read and write requests.

a distributed interaction. Rather, communicating parties in an ad hoc network are more interested in continuing their collaboration when the connection is restored – they expect communication to work in the presence of volatile connections. Dealing with partial failures is not a new ingredient of distributed systems, but these more frequent transient disconnections do expose applications to a much higher rate of partial failure than that which most distributed languages or middleware have been designed for. In mobile networks, disconnections become so omnipresent that they should be considered the rule, rather than an exceptional case.

Zero Infrastructure (originally named **Ambient Resources**). Mobile ad hoc networks have no or very little fixed infrastructure [MCE02]. Networks are formed by the mere colocation of mobile nodes. In such networks, the services available to an application thus depend on the host device's *location*. As devices move, services may spontaneously appear and disappear as their hosts join with and disjoin from the ad hoc network. Moreover, a mobile ad hoc network is often not globally administered by a central authority. In stationary networks, applications usually expect services to be available and accessible via an a-priori known URL or similar designator. In a mobile ad hoc network applications have to find their required services dynamically in the environment. Services have to be discovered on proximate devices, possibly without the help of shared infrastructure.

In the original motivation for AmOP, Dedecker states that services, and more generally resources, are said to be "ambient" because they are only available to an application when they are physically proximate.

Additionally, Dedecker identifies the following two characteristics [DVM+06a]:

Autonomy. Most distributed applications today are developed using the client-server approach. The server often plays the role of a "higher authority" which coordinates interactions between the clients. In mobile networks a connection to such a "higher authority" is not always available. Every device should be able to act as an autonomous computing unit.

Natural Concurrency. In theory, distribution and concurrency are two different phenomena. For instance in a client-server setup, a client device might explicitly wait for the results of a request to a serving device.

We argue that autonomy is a corollary of the lack of shared infrastructure. Without a shared infrastructure, devices (or more accurately their applications) necessarily need to be autonomous. The fact that they should be autonomous follows from the fact that they cannot *depend* on infrastructure, because there is (mostly) none available. Ad hoc applications have to be structured such that they can cope with necessary resources or services being unavailable for an extended period of time.

Natural concurrency already follows from the fact that applications are deployed on a distributed computer network. Hence, any ad hoc networking application is essentially distributed and concurrent. Granted, if synchronous client-server communication is used, the degree of concurrency in the system is largely diminished, but not entirely. The client and the server are still separate, distributed processes – they may fail independently. It is not possible to hide their distributed nature completely from the applications built on top [WWWK96].

Any application that is to be deployed on a mobile ad hoc network has to deal with the above phenomena. What is important to note is that the above phenomena are *universal*, in the sense that they do not depend on the particularities of a specific application. Exactly because of this universality, it is worth investing in a computational model that facilitates distributed programming for mobile ad hoc networks by taking these phenomena into account *from the ground up*. Such a computational model could be implemented as a new language, library or middleware platform. However, because the effects engendered by partial failures and the absence of remote services tend to affect and pervade the entire application, the above phenomena are not easily hidden behind traditional library abstractions. Therefore, distribution is often dealt with in dedicated middleware or programming languages. Together with the argumentation given in section 1.4, this forms the motivation behind our language-oriented approach to object designation for MANETs in later chapters.

2.2.2 Ad Hoc Networks versus Ad Hoc Applications

In a short but insightful paper, Garbinato and Rupp discuss the differences between ad hoc *networks* and ad hoc *applications* [GR03]. They argue that ad hoc networks necessarily imply ad hoc applications, but not necessarily the other way around. In other words, it is possible to envision ad hoc applications in other network infrastructures as well. To Garbinato and Rupp, an ad hoc application exhibits [GR03]:

- Mobility: a user should be able to use the ad hoc application anywhere, at any time. Hence, they argue, the application should not be limited by the communication range of the mobile device (e.g. Bluetooth technology). The communication range should depend on application logic, not on the underlying physical communication medium.
- Peer-to-peer interactions: devices should be able to logically communicate directly, without reference to a central server. However, Garbinato and Rupp make the point that physically, communication may still be routed through a shared infrastructure, as long as the communication is logically peer-to-peer.
- Collocation: logical interactions of ad hoc applications are consequences of the physical interactions of their users. Hence, ad hoc applications are "location-based".

It is indeed possible to construct applications fulfilling the above requirements without resorting to an ad hoc network. Garbinato and Rupp describe an application named the *ubiquitous flea market* [GR03] which is intended to be run on cellular phones. Using the application, users can advertise items they wish to sell or place a demand for items they wish to buy (e.g. concert tickets). What makes this an ad hoc application is that matching buyers and sellers are only notified of one another when they are proximate (e.g. when both are at a venue). While this proximity could be detected by means of real ad hoc networking technology (for example, using the phones' Bluetooth protocol), Garbinato and Rupp discuss this application in the context of a nomadic network (which is a network consisting of both mobile devices and fixed support infrastructure, e.g. base stations). Mobile phones communicate via a shared web service (using the GSM network), which keeps track of all phones' locations and performs the matching on behalf of the phones. This allows proximity to be defined independently of any ad hoc networking technology.

Hence, the claims made in the previous section that ad hoc applications necessarily need to deal with the hardware phenomena of MANETs only holds for that subset of ad hoc applications physically deployed on a MANET. In the rest of this dissertation, we will for the most part keep the focus on ad hoc applications for ad hoc networks. While such networks limit the kind of applications one may build, they nevertheless form an important subset. Infrastructure may not always be available (e.g. in disaster relief scenarios) or it may be inappropriate for the task at hand (e.g. prohibitively expensive to use). Ad hoc applications deployed in MANETs introduce more challenges from a systems perspective because of the inescapable hardware characteristics explained previously. Developing ad hoc applications without reference to a real MANET is inherently simpler: communication can be made much more reliable and one may exploit existing infrastructure.

2.3 Ambient-Oriented Programming

In his dissertation, Dedecker extensively motivates why previously developed programming languages and middleware are not readily suitable for MANETs, because they do not directly address the hardware phenomena described above [Ded06]. In reaction, he defines the ambient-oriented programming paradigm to clearly identify what characteristics of a system make it suitable for deployment in a mobile ad hoc network. A language is ambient-oriented if it exhibits a specific set of characteristics. Hence, the definition of the ambient-oriented *paradigm* is exactly this set of language characteristics. They are extensively discussed elsewhere [DVM$^+$05, DVM$^+$06a, Ded06]. We summarise each of them below such that they can be evaluated in light of coordination abstractions for mobile ad hoc networks in the following chapter.

2.3.1 Classless Object Models

An ambient-oriented language disallows the use of classes as they are employed in traditional class-based languages like Java and Smalltalk. In such languages, when objects are copied between devices (to share information, e.g. using parameter-passing), the class has to be copied as well. Hence, a single class can become duplicated across several devices in the network. In a volatile network like a MANET, it becomes impossible to implicitly ensure that all of these duplicate classes are kept in synchronisation. However, this impossibility breaks the abstraction barrier that all instances of conceptually the same class should behave the same way: two instances of conceptually the same class may behave differently on different machines.

An ambient-oriented language avoids these problems by requiring objects to be entirely self-sufficient (containing their own code and data). When objects are copied between hosts, they are self-descriptive and need no shared external entity (the class) to be duplicated. Of course, this solution is not a “silver bullet” either: if a class-like sharing relationship between objects is required, it must now be explicitly encoded by the programmer, who is himself responsible for the consistency of this relationship in the face of volatile connections.

2.3.2 Non-Blocking Communication Primitives

All distributed object-oriented programming languages have primitives for sending and receiving messages across the network. An ambient-oriented language requires these

primitives to be "non-blocking": a process or thread of control should not be suspended if the operation cannot be completed immediately. This requirement is based on the fact that in a wireless ad hoc network, communicating parties can often be unavailable, and making a communication operation block until the communicating party is available may lead to unacceptable delays. Furthermore, blocking primitives increase the risk of distributed deadlocks which are hard to detect in an ad hoc network, because the network lacks a global coordinator to detect or break the cycle.

Non-blocking message sending is better known under the term *asynchronous* message sending. More specifically, it is the form of asynchronous message sending where the sender does not even have to wait for the message to be transmitted to the receiver[2]. Non-blocking message reception is better known as *event-driven* computation. A non-blocking receive operation enables a process to register its interest in a certain type of message without blocking the control flow. In an event-driven application, the locus of control lies *outside* of the application; the control flow is determined by external events, rather than being encoded within the application itself.

2.3.3 Reified Communication Traces

The above requirement of making all communication between processes non-blocking implies that devices are no longer implicitly synchronised while communicating. If communication primitives no longer synchronise processes implicitly, other means must be provided to do so for some distributed computations to make sense. The ambient-oriented programming paradigm re-introduces synchronisation by requiring information about the communication traces of processes to be reified (i.e. made explicit). By communication traces, we mean e.g. the messages that a process successfully received in the past or the messages that are still pending to be sent, etc. Systems such as Virtual Time [Jef85] have proven that this information suffices to implement a variety of synchronisation mechanisms, such as optimistic transactions which do not use locks to synchronise processes but rather rollback computations should conflicts occur.

2.3.4 Ambient Acquaintance Management

As mentioned in section 2.2, mobile ad hoc networks have no infrastructure, which requires devices to detect services ("ambient resources") dynamically as they are roaming. In addition, services may be "anonymous": they have no a priori known address or URL by which they can be addressed. An ambient-oriented language should therefore include language features to communicate with anonymous services.

Ambient acquaintance management also entails a mechanism that allows a program to keep track of which services become available and unavailable as devices roam. An ambient-oriented language requires this mechanism to be peer-to-peer because of the lack of infrastructure: devices must be able to advertise their own services directly, without reference to third party servers. Note that this does not imply that an ambient-oriented program must be peer-to-peer as a whole: it is always possible to structure applications according to a client-server pattern. Ambient-oriented programming only

[2] Some asynchronous message passing models do not require a sender to wait for the message to be processed, but *do* require the sender to wait until the message was successfully transmitted across the network. For example, Caromel distinguishes between *synchronous* transmission but *asynchronous* servicing of requests [Car93].

states that applications should not be *forced* to use a client-server setup in order to interact.

2.3.5 Summary

The goal of an ambient-oriented programming language is to ease the construction of software deployed in a mobile ad hoc network because its distributed language features are carefully constrained to fit the hardware characteristics inherent to a MANET. Communicating over volatile connections and communicating with anonymous discovered services form the essence of the paradigm.

2.4 AmbientTalk/1

AmbientTalk/1 is the first concrete ambient-oriented programming language. It was originally developed in the context of the dissertation of Dedecker [Ded06] as the programming language incarnation of an abstract calculus, known as the ambient actor model [DV04]. AmbientTalk/1 is the direct precursor to the AmbientTalk/2 programming language presented in this dissertation. In this section, we highlight how AmbientTalk/1 succeeded in converting the AmOP requirements into usable programming language abstractions. However, we also highlight the limitations of the language. These limitations form the direct motivation for the introduction of the AmbientTalk/2 programming language, in chapter 4.

2.4.1 The AmbientTalk/1 Kernel Language

AmbientTalk/1 is a reflectively extensible kernel designed as a language laboratory to facilitate experimentation with AmOP language features. At the heart of the language is a distinction between so-called *active* and *passive* objects, resulting in a "double-layered" object model [DVM+06a]. AmbientTalk/1's active objects are based on the model of ABCL/1 [YBS86]. This model features active objects which consist of a perpetually running thread, updatable state, methods and a message queue. These concurrently running active objects communicate by asynchronous message passing. Upon reception, messages are scheduled in the active object's message queue and are processed one by one by the active object's thread. By enforcing an active object to process messages sequentially, race conditions on the updatable state are avoided. The merit of the model is that it unifies imperative object-oriented programming and concurrent programming without suffering from omnipresent race conditions.

Because it would be rather heavyweight to equip *every* object in an application with these (relatively heavyweight) concurrency control provisions, AmbientTalk/1 distinguishes between active and passive objects. Passive objects are normal objects without any built-in support for concurrency control. Because of this, the language must impose very strict rules on how passive objects are shared and manipulated by concurrently executing active objects. AmbientTalk/1's solution to this problem is to disallow active objects to share the same passive objects. It enforces this restriction by means of two rules [DVM+06a]:

- *Containment:* Upon creation, every passive object is contained within exactly one active object. The only thread that can enter the passive object is the thread of this active object that created and contains it.

- *Parameter Passing Rules:* When an asynchronous message is sent to an active object, objects may be sent along as arguments. In order not to violate the containment principle, a passive object that is about to cross the boundaries of its active container in this way is passed by copy. Any instance variables of the passive object are transitively deep-copied up to the level of references to active objects. Active objects process messages one by one and can therefore be safely shared by two different active objects. Hence, they are passed by reference.

A concurrent AmbientTalk application is structured as a set of communicating active objects. Active objects are also AmbientTalk/1's unit of distribution and are the only kind of object that can be referred to across device boundaries. Several active objects can run on a device and every active object contains an object graph of passive objects. Objects in this graph can refer to active objects that may reside on any device. In other words, AmbientTalk/1's remote object references are always references to active objects. Remote references to passive objects are disallowed: upholding the traditional method invocation semantics of passive objects would imply the use of synchronous remote method invocations, thereby violating the requirement for all distributed communication to be non-blocking (cf. section 2.3.2).

2.4.2 First-class Mailboxes

AmbientTalk/1's novelty with respect to similar active object models based on the actor model (such as ABCL/1 [YBS86]) lies in the treatment of the queues or mailboxes in which the asynchronous messages reside. AmbientTalk/1 introduces a mailbox for each stage in the life-cycle of an asynchronous message:

- The `inbox` stores the messages which an actor has already received and acknowledged, but not yet processed.
- The `outbox` stores the messages which an actor has already sent, but which are not yet known to be received (i.e. their acknowledgment is pending).
- The `rcvbox` stores the messages which an actor has processed.
- The `sentbox` stores the messages which have been successfully sent (i.e. they were acknowledged to be received).

The notion of an `inbox` and an `outbox` are well-known in other active object models and in the original actor model as defined by Hewitt and Agha [Hew77, AH87]. However, the introduction of mailboxes that capture the *history* of messages is far less common. The real novelty of AmbientTalk/1 is the fact that these mailboxes are reified as first-class passive objects that can be manipulated by the programmer[3]. AmbientTalk/1's mailboxes embody the AmOP characteristic of reified communication traces (cf. section 2.3.3). In section 2.4.4, we briefly discuss how they have been used to develop ambient-oriented synchronisation constructs.

[3]Reification is a term stemming from the domain of computational reflection, cf. section 5.2.2. A "reification" of an abstract entity differs from a "representation" of that entity in the sense that the former must remain *causally connected* [Mae87] to the entity while the latter must not.

2.4.3 Ambient Acquaintance Management

The final novelty of AmbientTalk/1 lies in its provisions for the discovery of active objects in the ad hoc network. In the original actor model, an actor `a` can only get introduced to another actor `b` when it is passed a reference to `b` (by construction or by message passing). There is no way for it to construct a reference to `b` by means of an abstract name. AmbientTalk/1 enables active objects to get acquainted through a system of four additional mailboxes that reflect the object's environment:

- The `providedbox` is a queue of tags (textual descriptions) denoting the abstract names under which the active object publishes itself in the ad hoc network.
- The `requiredbox` is a queue of tags denoting the names of remote active objects with which the active object wants to get acquainted.
- The `joinedbox` is a queue of resolutions – tuples of the form `(tag,activeobj)` – denoting all discovered remote active objects that are currently available for communication.
- The `disjoinedbox` is a queue of resolutions denoting previously discovered remote active objects which are currently disconnected (e.g. due to a network partition).

The AmbientTalk/1 interpreter tries to match the tags in the `providedbox` and the `requiredbox` of distributed active objects. If two active objects running in separate AmbientTalk/1 interpreters enter one another's communication range while having an identical descriptive tag in their `providedbox` resp. `requiredbox`, the `joinedbox` of the active object that *required* the collaboration is updated with a resolution containing the corresponding descriptive tag and a remote reference to the active object that *provided* that tag. Conversely, when a network partition occurs, the resolution is moved from the `joinedbox` to the `disjoinedbox`. If the network partition is healed, the resolution is moved back to the `disjoinedbox`. This mechanism allows an active object to detect new acquaintances in its ambient and to detect when these have disappeared or reappeared. It provides AmbientTalk/1 with a way to perform ambient acquaintance management, as explained in section 2.3.4.

2.4.4 AmbientTalk as a Language Laboratory

AmbientTalk/1 has primarily been designed as a "language laboratory" for experimenting with novel concurrent and distributed language constructs. It can be regarded as a research vehicle for exploring the ambient-oriented programming paradigm. To support this exploratory research, AmbientTalk/1 relies on the following features:

- The mailboxes are the fundamental semantic building blocks for implementing new language constructs because they enable access to an active object's past and future communication traces. These data-structures allow a programmer to e.g. re-send previously sent but unacknowledged messages, to retract messages from a mailbox to prevent them from being delivered or processed, etc.
- AmbientTalk/1 has a concise metaobject protocol (MOP) which enables the programmer to intervene whenever the interpreter moves messages between an active object's mailboxes. The MOP allows language constructs developed by

metalevel programmers to react to changes in an active object's computational context.

- Extensions to the metaobject protocol can be bundled in so-called *language mixins*, which act as a module system for language constructs [DVM+06a]. It enables language constructs to be applied locally to one active object without modifying the metaobject protocol of other active objects.

- AmbientTalk/1 inherits call-by-name parameter passing from its predecessor Pico [D'H96, DDD04]. This parameter passing semantics is often used to extend the language with new "forms" (e.g. control constructs). Hence, the language can be enriched with new primitives that can delay the evaluation of code until the appropriate moment.

The above language features have been put to work in order to illustrate how AmbientTalk/1 can be augmented with novel ambient-oriented language constructs. A comprehensive overview of these experiments can be found in Dedecker's dissertation [Ded06]. Below, we summarise the most important experiments and highlight the role of the mailboxes in each of them.

Non-blocking Futures In the AmbientTalk/1 kernel language, an asynchronous message send does not return any value. Futures are a well-known abstraction that can represent the return value of asynchronous invocations [Hal85, LS88, YBS86]. Using the first-class mailboxes, futures have been reflectively added to the kernel language. In short, a future object is attached to any message added to the `outbox`. The future is itself an active object representing the "reply address" of the message: the receiver will send the return value to this object. While the return value is not yet available, the future buffers any incoming asynchronous messages in its own `inbox`. When the return value becomes available, the future transparently starts forwarding messages to the return value by moving all messages in its `inbox` to its `outbox`. We will extensively revisit futures in sections 4.3.4 and 4.9.2.

Ambient References While this dissertation discusses ambient references and their incarnation in AmbientTalk/2, preliminary versions of this language construct have been implemented in AmbientTalk/1 [Van06]. In AmbientTalk/1, ambient references are represented as active objects that transparently store and forward any incoming message to a discovered remote active object. An ambient reference is initialised with a required tag. By means of the `requiredbox` and the `joinedbox`, matching remote active objects are discovered. When a remote active object disconnects, the ambient reference makes use of the first-class access to its `outbox` to retract all messages whose transmission is pending and to re-insert them in its own `inbox`. This enables the ambient reference to resend these messages to another matching active object discovered at a later point in time.

Weak Replication Dedecker shows how the mailboxes can be used to synchronise weakly replicated active objects across an ad hoc network without global coordination [Ded06]. The protocol used to synchronise the active objects is based on a combination of the Bayou [TPST98] and TimeWarp [Jef85] protocols. In TimeWarp, synchronisation between abstract processes is achieved by the careful logging and replaying of messages. To this end, messages are also kept in dedicated message queues at

each step in their life-cycle. These message queues almost directly correspond to AmbientTalk/1's mailboxes, making it an ideal language to experiment with the protocol.

2.5 Limitations of the AmbientTalk/1 Kernel

We motivate the need for a revised ambient-oriented programming language based on the limitations of AmbientTalk/1. We distinguish between shortcomings of the object model, the ambient actor model and the features that make up the language laboratory.

2.5.1 Limitations of the Object Model

Double-Layered Object Model Recall that AmbientTalk/1 distinguishes between active and passive objects and that each passive object is fully contained within exactly one active object. This containment forbids direct references to passive objects from objects contained within another active object. As a result, active objects are the only entities addressable across device boundaries, resulting in a relatively coarse-grained addressing scheme. If an individual passive object needs to be addressed from within another active object, its containing active object has to play the role of "facade": the identity of the passive object must somehow be explicitly encoded, and the active object must forward incoming messages to the contained passive object based on that encoding. This is clumsy as it boils down to the programmer writing his own object referencing abstraction. The only alternative is to decompose the active object into multiple active objects. However, this introduces the overhead of allocating additional queues and threads. Moreover, the decomposition requires refactoring code that previously had synchronous access to other contained objects to employ asynchronous message passing instead. In short, the impossibility to remotely address individual passive objects leads to coarse-grained interfaces at the level of active objects.

Inter-Actor Message Passing Semantics To avoid concurrent access to passive objects, AmbientTalk/1 prevents these objects to be shared by different active objects. As described in section 2.4.1, this is enforced by parameter-passing passive objects *by deep copy* in asynchronous message sends. However, this parameter passing semantics introduces a number of issues. First, creating a deep copy of a complicated data-structure can be a costly operation. Hence, programmers must carefully design their applications in order to prevent excessive copying of passive objects.

A deep copy of an object always copies all passive objects transitively reachable via the object graph. The only way in which the programmer can influence the transitive closure of this graph is by partitioning the passive objects among multiple active objects (because a deep copy stops at the level of references to active objects). This partitioning leads to problems similar to those described previously: it may introduce unnecessary concurrency and it requires code to be refactored. Here, the distinction between active and passive objects is too coarse-grained to express certain serialisation concerns.

Next to the runtime cost of the deep-copy semantics, creating copies of objects often introduces object identity issues. Within the source code of the passive object, there is no indication as to whether the object will ever by copied because it is parameter-passed to another active object. Copying objects may break existing code because of e.g. object comparisons based on object identity: a copy of the object no longer has the same identity as the original. This is especially problematic if passive objects are

passed back and forth across two active objects: rather than resulting in the original object, the object passed back will be a copy of a copy of the original.

We will show how AmbientTalk/2 mitigates the above issues by means of a redesigned object model in section 4.3.2.

2.5.2 Limitations of the Ambient Actor Model

Service Discovery Even though AmbientTalk/1 has support for ambient acquaintance management through four dedicated mailboxes, it does not introduce any high-level service discovery abstractions (e.g. in the form of a publish/subscribe API) for direct use by application programmers. All service discovery abstractions must thus be handcrafted via the metaobject protocol. This limitation is due to the fact that there is very little prior work in the literature on exposing service discovery as a *language construct*. Service discovery is almost always provided by means of external libraries or middleware.

A second limitation of AmbientTalk/1's service discovery is the lack of control over the matching process of the tags stored in the `requiredbox` and the `providedbox` of the active objects. AmbientTalk/1 matches these tags based on a simple string comparison. As a result, it is impossible to directly represent structured or hierarchical information using tags. This disallows service discovery from being polymorphic. For example, an active object whose provided tag equals "colorprinter" will not match a required tag named "printer".

Finally, we argue that the ambient actor model's representation of service discovery via message queues is arbitrary. For example, the `requiredbox` and `providedbox` are represented as queues, while it is not clear what meaning to attribute to the order of the tags or to duplicate tags in the mailbox. It is more appropriate to regard them as sets, because the order of their content does not matter and because it avoids duplicates.

Garbage Collection An issue left partially unaddressed by AmbientTalk/1 is the garbage collection of messages in the various mailboxes. Because the AmbientTalk/1 interpreter stores processed and sent messages in the `rcvbox` and `sentbox` respectively, it becomes necessary for an active object to manage the lifetime of these messages. If there are no language constructs that need past information, past messages should obviously be deleted at some point in time. A more space-preserving strategy would be to throw away message histories by default, unless some language construct has explicitly expressed its interest in them.

With respect to the `outbox`, garbage collection policies have already been investigated in the context of AmbientTalk/1 [DVM+06a]. `due`-blocks are similar in use to the well-known `try-catch`-blocks in exception handling. A `due`-block is associated with a time period and an exception handler. Every outgoing message sent in the dynamic scope of the `due`-block is associated with a time to live (the `due`-block's time period). Messages whose time to live expires are automatically removed from the `outbox` and trigger the `due`-block's exception handler. This prevents messages from remaining queued in the `outbox` forever.

Finally, on a more technical note, AmbientTalk/1 does not define any distributed garbage collection semantics for its remote references to active objects. Once an active object is referred to remotely, it is no longer reclaimed automatically.

2.5.3 Limitations of the Language Laboratory

Metalevel Engineering issues: Language Mixins The metalevel architecture of AmbientTalk/1 has a number of drawbacks preventing the construction of scalable systems. The drawbacks of this architecture have already been summarised [Mar06]; we repeat the arguments here for completeness.

First, AmbientTalk/1's reflective architecture is not *stratified*. A stratified architecture assigns base- and metalevel code to separate layers (cf. section 5.2.1). Lacking such stratification, AmbientTalk/1's metaobject protocol is prone to name clashes between base and metalevel code. For example, a base level method may accidentally be regarded as overriding a metalevel method simply because its name matches that of a metalevel operation, thereby changing the language's semantics unintentionally. Second, metalevel code is not *encapsulated*. The details of AmbientTalk/1's reflective language constructs are not hidden. As such, base level code may circumvent a language construct's public interface and is exposed to internal implementation details.

In AmbientTalk/1, modifications to the metaobject protocol are bundled in language mixins, which are then "mixed-in" with the base-level behaviour of an active object. In order to install multiple language constructs simultaneously, language mixins have to be composed. In AmbientTalk/1, this composition is done *ad hoc*. There are no language constructs that aid the developer in checking for e.g. name clashes or to express the composition of language mixins. As a result, the composition of different language constructs may lead to erroneous behaviour that can be hard to debug at runtime [Mar06].

We show how AmbientTalk/2 foregoes the above issues by means of a stratified metalevel architecture in section 5.2. Also, while AmbientTalk/1's reflective architecture is confined to active objects, AmbientTalk/2's architecture consistently reifies both the sequential and concurrent parts of the language.

Base-level Engineering issues: Dynamic Scoping Variables in AmbientTalk/1 are dynamically scoped. This implies that free variables are looked up in the dynamic runtime environment (the call stack) rather than in the lexical environment (the environment of definition). This form of unrestricted dynamic scoping has issues that have been well-documented in the literature, such as accidental name capture (in early LISP dialects, this was known as the `funarg` problem [SS78]). Furthermore, because of dynamic rather than lexical scoping, nested objects cannot access lexically enclosing variables, which defeats one of the most beneficial properties of nesting (cf. section 4.2.3).

Linguistic Symbiosis AmbientTalk/1 lacks a mechanism to interoperate with foreign languages. Such a mechanism is useful for an experimental language such as AmbientTalk/1, as it allows the language to remain minimal while all of the functionality necessary for programming in the large can be drawn from foreign languages' libraries. Lacking such a mechanism, programmers must manually re-implement many abstractions which are readily accessible via libraries in other languages. While this is no fundamental problem of AmbientTalk/1, it is nevertheless a very pragmatic issue which causes AmbientTalk/1's usability to remain limited in practice.

2.6 Conclusion

Ambient-oriented programming is a paradigm the feature set of which is developed specifically to accomodate the idiosyncratic properties of mobile ad hoc networks. Such networks, composed of mobile devices with wireless communication links exhibit two discriminating characteristics: connections between devices are volatile and infrastructure is scarce or nonexistent. Networks may be formed *ad hoc* simply by collocating devices.

AmbientTalk/1 is the ambient-oriented programming paradigm incarnate. While it has proven to be a successful research vehicle to explore the paradigm, it remains in many ways a fledgling language. The AmbientTalk/2 language, while built around the same guiding principles, is our attempt at combining the requirements of an ambient-oriented language with a more scalable object model. We postpone a discussion of AmbientTalk/2 until chapter 4.

With the AmOP frame of reference in place, we first focus on coordination abstractions for mobile ad hoc networks. In the next chapter, we describe how the distinguishing characteristics of MANETs lead to a set of criteria that discriminate suitable coordination abstractions for this type of networks. Armed with these criteria, we subsequently survey related work in search of suitable existing abstractions. Some of these abstractions will then re-surface as language constructs in AmbientTalk/2, in chapter 4 while others will be embedded in the ambient reference language abstraction, in chapter 6.

Chapter 3

Coordination in Mobile Ad hoc Networks

In the previous chapter, we discussed which hardware characteristics are inherent to MANETs and how they relate to AmOP. In this chapter, we use these hardware characteristics to define a set of criteria which circumscribe suitable coordination abstractions for MANETs. Next, we survey related work and gauge the suitability of each of the discussed approaches for MANETs by means of the postulated criteria.

3.1 Coordination

Before defining criteria for coordination abstractions in MANETs, we explain our use of the term "coordination abstraction". Processes coordinate to reach a common goal which they cannot achieve by themselves. In this dissertation, we focus on applications that are distributed by their nature (e.g. an instant messenger, a collaborative text editor, ...) rather than centralised applications which are distributed to exploit parallelism. It is our explicit assumption that coordinating distributed (and hence concurrent) processes involves at least:

Discovery Distributed processes can only coordinate if they are somehow introduced to one another in the network. Discovery can be through centralised directories, e.g. the Domain Name Service of the WWW, or it could be entirely decentralised by multicasting advertisements in a local area network. Note that discovery does not necessarily imply that processes have to discover *one another* explicitly. It may well be that they both discover a middle man, which implicitly connects the processes (good examples of such mediators are tuple spaces and event notification services, both discussed later in this chapter).

Communication Coordination involves communication: processes have to be able to transmit and receive information. This may happen very explicitly, e.g. via a point-to-point communication channel with explicit `send` and `receive` operations, or it may happen very implicitly, e.g. by registering a callback function to be invoked whenever a certain event occurs.

Synchronisation To coordinate their actions, processes require mechanisms to put temporal constraints on their actions (e.g. waiting for a reply to come in). Syn-

chronisation may be completely aligned with communication (as in the case of a remote procedure call), or it can be made completely independent of communication (e.g. a purely asynchronous event notification).

Failure Handling Processes can end up in interactions that do not progress because of network or process failures. Proper coordination abstractions must allow processes to recover from such situations. Such abstractions can range from very simple (e.g. timeouts) to very complex (e.g. distributed transactions).

We will use the term coordination abstraction to denote any software abstraction that addresses one of the above aspects. Note how, for each of these aspects, widely different approaches exist to address them. To be able to determine which approaches are suitable for mobile ad hoc networks, the following section presents a set of criteria to evaluate the applicability of an approach to coordinate processes in a MANET. First, we concretise the above aspects of coordination by means of an example.

We ground our following discussion on coordination in MANETs in a concrete ad hoc networking application. The *Musical Match Maker* (♫uMaMa) is an application intended to be deployed on mobile devices such as PDAs or cellular phones. ♫uMaMa contains a library of the user's songs. When two people running ♫uMaMa enter one another's personal area network (demarcated by e.g. the bluetooth communication range of their cellular phones), their ♫uMaMa applications exchange one another's music library's index (not necessarily the songs themselves). After the exchange, ♫uMaMa can calculate the percentage of songs both users have in common. If this percentage exceeds a certain (perhaps user-defined) threshold, ♫uMaMa can notify the user that someone with a similar taste in music is nearby. As a side effect of the exchange, the user can also inspect what songs are in the other user's library that are not in his or her own library, allowing the user to learn about new artists he or she is likely to enjoy.

What makes ♫uMaMa an ad hoc application is that the musical taste of users is only matched when they are proximate, e.g. if they have joined the same ad hoc network. The advantage over similar internet-based applications is that the mobility of the application provides for an additional social context: the "musical match" is made at a time when the users can physically interact. Although the ♫uMaMa application is relatively small, it is a representative application because it embodies all characteristics of a typical mobile ad hoc networking application. In particular, it has to deal with the different aspects of coordination introduced above:

Discovery The processes that represent the ♫uMaMa application have to discover one another without the help of any predefined infrastructure (cf. the zero infrastructure hardware characteristic from section 2.2.1). Once the processes have discovered one another, they need to set up a *session* to transmit their music libraries.

Communication Once the processes have established a session, they need to transmit their library index. This transmission can be easily disrupted by the unpredictable movement of the users (cf. the volatile connections hardware characteristic from section 2.2.1). ♫uMaMa is designed such that transient disconnections do not cause the transmission to fail immediately, increasing the chances that it can eventually be completed and a match can be performed (and thereby also avoiding angry users).

Synchronisation When the ♫uMaMa applications transmit their library index, they

need to synchronise with one another to know when they can send subsequent song information, and when the library transmission has terminated.

Failure handling If a network partition does persist, the session between both processes should eventually be terminated such that any resources associated with that session (e.g. the partially downloaded library index of the remote party) can be reclaimed.

In the following section, we introduce criteria for coordination in MANETs. Where possible, we relate the introduced criteria to the above example. Later, in chapter 4, we will revisit this example and discuss its implementation in AmbientTalk/2.

3.2 Criteria for Coordination in MANETs

We describe criteria that define which coordination abstractions are suitable for use in a MANET. Similar to the way Dedecker motivates the AmOP paradigm based on the hardware characteristics listed in section 2.2 [Ded06], we motivate criteria for coordination abstractions based on the same hardware characteristics. One question we should address first is whether these criteria are minimal and/or complete. Lacking a formal model, we can prove neither property. We can only argue that no criterion is redundant by showing how it addresses specific issues of coordination in a MANET. We argue that they are complete to the extent that they cover at least the aspects of coordination discussed in the previous section.

3.2.1 Decentralised Discovery

In traditional, stationary networks a centralised lookup service or name server is often used for the purposes of discovering remote services. In a mobile ad hoc network, such lookup services are too inflexible:

- A lookup service is a form of infrastructure (e.g. a server hosting a shared directory service). A key aspect of coordination in a MANET is that two mobile devices "out in the field" (without access to shared infrastructure) should be able to communicate with one another directly. For example, in the ♫uMaMa application, two people should be able to exchange and match their music libraries anywhere, as long as they are proximate to one another.
- Lookup services often make use of synchronous request/response communication to resolve names into references (e.g. querying the registry in Java RMI [Sun98]). This style of communication corresponds to a client "polling" the lookup service for the availability of a service. In a MANET, this is inappropriate as the unavailability of a service is commonplace. It is thus more suitable to have a service "push" its availability to clients.
- As we have discussed in section 2.2.1, the limited communication range in combination with the lack of any centralised network infrastructure causes the availability of services in a MANET to change frequently. Traditional lookup services do not notify clients of any changes in a service's registration.

To deal with the above issues, different service discovery protocols have been devised. Such protocols forego a synchronous request/response interaction model in

favour of an asynchronous publish/subscribe interaction model [McG00]. Such a discovery protocol allows clients to express their interest in a particular kind of service and notifies them asynchronously when a matching service becomes available in the network, usually passing along a reference to the remote service. In a survey on service discovery protocols for ubiquitous computing, McGrath explicitly distinguishes "lookup" from "discovery", lookup corresponding to the request/response interaction model of traditional lookup services and discovery corresponding to the publish/subscribe interaction model of service discovery protocols [McG00]. Table 3.1 summarises his view on the most important differences between both.

Lookup	**Discovery**
Uses static databases of information.	"Spontaneous" discovery.
Maintained by privileged administrators.	Low or no human administration.
Limited support for searching (lookup by means of a unique identifier).	Extensive searching abilities (e.g. selecting a specific *type* of service).
Does not generate events when resources register and unregister.	Automatic adaptation to mobile and sporadic availability of services.

Table 3.1: Service Lookup versus Service Discovery.

Service discovery protocols tackle all of the aforementioned issues save the first. While a service discovery protocol surely does not *need* to rely on infrastructure, it also does not *rule out* the use of infrastructure per se. The Jini architecture, discussed later in section 3.3.6.2 exemplifies this. Given our particular MANET setting, we explicitly state the importance of a *decentralised* service discovery abstraction as our first criterion:

Criterion 1 (Decentralised Discovery) *Processes require a decentralised service discovery protocol that enables them to* autonomously *act upon the (un)availability of nearby services.*

3.2.2 Decoupled Communication

Considering communication in a MANET, we discern two key issues. The first is how processes can abstract from the intermittent connectivity of the underlying network. The second is how processes can abstract over the *total number* of entities with which they interact. We discuss each of these issues below.

3.2.2.1 Abstracting From Volatile Connections

We describe three criteria that allow communicating parties to abstract from the underlying state of the (ad hoc) network. The motivation behind this abstraction is that, in mobile ad hoc networks, the state of the network is in constant flux because devices move about in unpredictable ways. A coordination abstraction for a MANET should allow applications to abstract from this underlying physical state when communicating because it can make communication more resilient in the face of temporary disconnections, as explained later.

The following three criteria are well-known in the literature, especially in the context of publish-subscribe architectures [EFGK03]. They pertain to *decoupling* the communicating parties along three dimensions.

Decoupling in Time The volatile connections in MANETs lead us to consider communication models that can abstract from the network connectivity between communicating processes. It should be possible for two processes to express communication independently of their connectivity. This significantly reduces the case-analysis for the programmer, which can reason in terms of a fully connected network by default, and can deal with border cases in an orthogonal way (as explained later).

Criterion 2 (Decoupling in Time) *Communicating processes do not necessarily need to be online at the same time.*

Decoupling in time is achieved either by synchronising processes until a connection is available (which, as we argue below, is not a very scalable solution in a MANET) or by storing sent messages in an intermediary data-structure. This makes it possible for communicating parties to interact across volatile connections, because the logical act of information *sending* is decoupled from the physical act of information *transmission*, allowing for the information to be saved and transmitted at a later point in time, when the connection between both parties is restored. For example, in the ♫uMaMa application, song information sent to a disconnected peer can be buffered and then transmitted when the users are in close proximity once more.

Decoupling in Space Decoupling in space implies that communicating processes do not necessarily need to know one another's exact address or location a priori in order to collaborate. It is directly motivated by the scarcity of infrastructure in a mobile ad hoc network, making the reliance on servers to mediate collaborations impractical. A second motivation for space-decoupled communication is that it enables applications to adapt more gracefully to changes in their physical environment. In mobile networks, equivalent services may be hosted by different devices. As a device roams, it may use different instances of conceptually the same service. For example, in a city tour application, the handheld device of a tourist may connect to the same tourist information service via different access points physically dispersed throughout the city. A communication model that is decoupled in space supports such transitory relationships, because it allows one to make abstraction from specific service instances.

Criterion 3 (Decoupling in Space) *Communicating processes do not necessarily need to know each other beforehand.*

Decoupling in space implies a form of *anonymous* communication, often implemented by a form of communication where senders and receivers of data are matched on the content of the data itself. A prototypical example of this is communication via tuples in tuple spaces. Gelernter refers to space decoupling in the context of tuple spaces as "distributed naming" [Gel85].

Finally, decoupling in space is closely related to decentralised discovery, in the sense that decentralised discovery is usually *implemented* in terms of communication which is decoupled in space. However, decoupling in space does not imply a decentralised form of discovery. For example, in the Linda coordination language [Gel85], processes are decoupled in space but there is no form of discovery to connect decentralised distributed processes.

Synchronisation Decoupling In a mobile ad hoc network, an application may find itself deprived of access to a certain service or resource for extensive periods of time. In fact, *not* having access to a remote service should be considered the default in a MANET. For example, in a collaborative meeting application, the application only has access to the calendars of other people when they are physically nearby. In the ♫uMaMa application, two people that are in the process of exchanging their song libraries while commuting to work by train may only meet hours later (on the way home) to finish the library exchange. Such extensive periods of time between potential periods of interaction suggest that synchronisation between different parties should be performed without blocking their control flow (i.e. without suspending their thread of control). Blocking synchronisation can lead to applications which remain unresponsive for extensive periods of time. A reactive synchronisation style is more appropriate in a MANET, as it leaves processes responsive to other events while waiting for (information provided by) another process.

Criterion 4 (Synchronisation Decoupling) *The control flow of communicating processes is not blocked upon sending or receiving.*

Synchronisation decoupling implies that a sender can employ a form of *asynchronous* message passing, such that the act of message *sending* becomes decoupled from the act of message *transmission*. Likewise, allowing recipients to process messages asynchronously decouples the act of message *reception* from the act of message *processing*. Message transmission and reception require a connection between sender and receiver, but message sending and processing can be decoupled, allowing communicating process to abstract from the fact that the other process is online or not.

3.2.2.2 Abstracting From Arity

In mobile networks groups of devices are often not statically determined, but are rather formed *ad hoc* as devices roam. One is often interested in communicating with only the *proximate* devices. The number of such proximate devices is not a priori known to the application. It is therefore important that interactions can be expressed without explicit reference to the *number* of participants. A good coordination abstraction should enable the programmer to express interactions with a varying number of participants, e.g. one-to-one, one-to-many or many-to-many interactions.

Criterion 5 (Arity Decoupling) *Processes do not necessarily need to know the total number of processes communicated with.*

We use the term arity decoupling to attribute forms of communication that can target more than one recipient, without explicitly specifying the total number of recipients. Arity decoupling is a special case of space decoupling. Space decoupling can be regarded as abstracting from *both* the identity of receivers *and* the total number of receivers [EFGK03]. We choose to address these two aspects separately because both concepts are orthogonal in principle. A communication abstraction can be anonymous yet specify a maximum number of receivers.

Finally, note that the arity of the interaction influences the synchronisation of the participating processes. In a point-to-point interaction, a process often needs to await a reply from a previous request before being able to continue. However, in a one-to-many interaction, different kinds of synchronisation abstractions are called for.

3.2.3 Connection-independent Failure Handling

In most distributed systems, network failures are represented as exceptional events that must be dealt with by applications at every point of interaction between processes. For example, failures are often represented as exceptions raised by communication primitives. We argue that failure handling between processes in a MANET should be independent of any network failures between those processes. We justify this claim by means of two observations:

1. Connection-independent failure handling enables processes to tolerate network failures, because their failure handling code is not written in terms of those failures directly. Tolerating network failures by default is a useful property in a MANET because – as previously discussed in section 2.2.1 – disconnections as a result of device mobility can be transient. It often makes sense to be able to abstract from such transient failures, resuming computation upon reconnection. Again, in the ♫uMaMa application, the transmission of the users' music libraries should not be aborted but rather paused upon disconnection. Treating disconnections as a normal mode of operation is an optimistic form of partial failure handling. Reacting to enduring network partitions must thus be done by means of other mechanisms, in order not to lose the benefit of being able to abstract from failures by default.

2. Connection-independent failure handling potentially makes processes more robust, because failure handling code can equally be triggered even if there is no physical network failure. For example, a process may be killed, garbage collected or become otherwise unresponsive (intended or unintended). Such failures are logical rather than physical, but equally require potential cleanup code to be ran.

Criterion 6 (Connection-independent Failure Handling) *Processes should be able to perform failure handling independent of any network failures.*

Note that we are not arguing against abstractions that enable processes to *react* to changes in their underlying network connection with a remote process. On the contrary, being aware of the state of the connection is often extremely useful information that may well percolate up to the graphical user interface of an ad hoc application. We are only arguing that the event of a process disconnecting from the network should not be represented to the application as a failure, but as an ordinary event. Simply put, in a MANET, network failures should not be modelled as exceptions because they are not exceptional.

3.2.4 Relation to Ambient-oriented Programming

The previous sections have introduced a set of criteria that specify which kind of abstractions are suitable for coordinating processes in mobile ad hoc networks. Here, we relate them to the four requirements of an ambient-oriented programming language introduced in section 2.3. These requirements have previously been used by Dedecker to discriminate appropriate from inappropriate language constructs for MANETs [Ded06]. As we shall see, some of our criteria are refinements of Dedecker's AmOP requirements, while others are new with respect to the AmOP requirements. We relate each criterion to the AmOP requirements below:

Decentralised Discovery This criterion is implicit in the *ambient acquaintance management* requirement of the AmOP paradigm (cf. section 2.3.4). Ambient acquaintance management implies that programs should be able to construct an up-to-date view of proximate devices and their hosted services.

Time-decoupled Communication The implicit assumption underlying the "non-blocking communication" requirement of an AmOP language (cf. section 2.3.2) is that it decouples processes in time *and* in synchronisation. We merely make this distinction explicit. Making the difference between time and synchronisation-decoupling is important however, because neither implies the other. As we shall discuss in the following section, there exist systems that possess one, but not the other.

Space-decoupled Communication The requirement for communication to be decoupled in space is implicitly present in the *ambient acquaintance management* requirement of the AmOP paradigm. While this requirement focuses on enabling programs to have an up-to-date view on the availability of nearby services, the direct consequence is that it allows programs to get acquainted with nearby services without knowing about them a priori. Hence, in an AmOP language it should be possible to perform space-decoupled communication using the "ambient acquaintance management" properties of that language.

Synchronisation-decoupled Communication As mentioned above, the non-blocking communication requirement implicitly includes both time and synchronisation-decoupling.

Arity-decoupled Communication The AmOP paradigm never prescribes that the language should support one-to-many interactions. Hence, this is a novel criterion for coordination abstractions with respect to the AmOP requirements.

Connection-independent Failure Handling No AmOP requirement specifies explicitly how failure handling should be dealt with. The non-blocking communication requirement does implicitly assume that disconnections should not be regarded as failures. It is assumed that failure handling can be implemented by means of the reified communication traces (cf. section 2.3.3). Again, our criterion is more abstract in that it does not consider a particular implementation of failure handling.

The above analysis uncovers that our coordination criteria are closely related to the original AmOP requirements of Dedecker [Ded06]. Some of the criteria are merely *refinements* of earlier AmOP requirements, while others are novel or focus on more specific aspects of coordination. As we shall see later, all of these criteria are relevant when discussing object designation for MANETs.

3.3 Survey of Related Work

In this section, we analyse a broad range of systems (languages, calculi, middleware, ...) with the shared goal of raising the level of abstraction for concurrent and distributed programming for different kinds of computer networks. We will not only focus on systems developed specifically for mobile ad hoc networks, both because systems outside of this domain provide interesting ideas and because this brings to light the

differences between systems developed for traditional versus ad hoc networks. In the subsequent sections, when discussing systems that were not designed a priori for mobile ad hoc networks, it has to be kept in mind that our evaluation makes no claims as to the applicability of the discussed work in general. We only judge their suitability for mobile ad hoc networks.

3.3.1 Languages for Local Area Networks

Before the rise of the Internet, research in distributed programming languages mainly focused on local area networks (LANs). Two exemplar languages are Emerald [BHJL86, JLHB88] and Obliq [Car95]. Emerald is mainly known for its introduction of *mobile* objects: objects that can migrate from one node in the network to another (e.g. to avoid latency or partial failure, or to balance the load of the system). Obliq's most distinguishing language feature is its *distributed lexical scope*: the ability for programs to access lexically visible variables that effectively reside on a different machine.

Most languages for LANs, including the above two, are based on the abstraction of a remote procedure call (RPC) [BN84]. An RPC is a distributed adaptation of a regular procedure call, making the caller block and wait for the callee to compute a return value. The attractiveness of the RPC model is that it can make a procedure call seemingly network-transparent (modulo latency and the possibility of partial failure). However, RPC scales very poorly in ad hoc or even wide-area networks [SG01].

RPC neither decouples communicating processes in time, space nor synchronisation [EFGK03]. Processes can only communicate when they are both online, the caller needs to know the callee, and the caller remains blocked until the callee has produced a result. Despite the shortcomings of the RPC model, it remains popular to this day, albeit in an object-oriented guise. This can be witnessed by the widespread use of distributed object systems like Sun Microsystems' Java RMI [Sun98] and the OMG's Object Request Broker Architecture [Obj02].

Not all languages developed for LANs are based on RPC. Most notably, the family of actor-based concurrent languages (ABCL) [YBS86, Yon90] features asynchronous message passing as the main communication abstraction, although it still permits RPC and introduces a `select` statement that allows active objects to block until a certain message is received. Note that we classify ABCL as a language for LANs because it was introduced in the context of parallel programming on clusters of computers. However, the language's actor legacy would actually make this language quite scalable in a WAN as well, as shall quickly become clear in the following section.

In a classic LAN, the network topology is relatively stable. Ethernet connections are very reliable, and the nodes in the network are for the most part not mobile. This explains why, in most languages for LANs, failure of the communications network is considered as an exceptional case. This motivates e.g. the design of failure handling as exception handling in Emerald. It also explains why these languages offer no abstractions to monitor changes in the network. Furthermore, because the network is often globally administered and new nodes do not appear frequently, languages for LANs feature no dynamic service discovery abstractions. Processes are mostly introduced by means of a priori known URLs or via a central name server.

3.3.2 Languages for Wide Area Networks

There exist a large number of distributed languages that have been developed for both LANs and WANs. Some are based on variations of the RPC model, such as

Argus [Lis88]. Others are distributed versions of constraint logic programming languages, like Distributed Oz [HRBS98] and Janus [KS90], while still others are based on the message passing model of actors, like Erlang [AVWW96], Salsa [VA01] and E [MTS05].

The main difference between languages for WANs and LANs lies in the synchronisation decoupling of their communication primitives. Many WAN-oriented languages introduce asynchronous message passing to e.g. cope with the higher latency of communication. Argus and E feature asynchronous messaging with *promises* as return values [LS88] (see also section 4.9.2). Salsa introduces asynchronous message passing with explicit continuations to synchronise actors on return values. Erlang features asynchronous message passing without direct support for return values. However, Argus, Erlang, Oz and Janus do not feature full synchronisation decoupling because they introduce language constructs that enable processes to suspend until certain conditions are met (i.e. claiming a promise in Argus, receiving a message in Erlang, reading an unbound logic variable in Oz, asking a constraint in Janus). Salsa and E allow processes to react to incoming messages purely asynchronously and feature full synchronisation decoupling.

Because WANs – like LANs – are still supported by an infrastructure, none of the above languages feature any form of service discovery or anonymous, space-decoupled communication. Rather, processes are introduced to one another via explicit URLs or similar names (e.g. a universal actor name (UAN) in Salsa) that require a universally accessible "introducer" in order to be resolved into a useful reference or communication channel. This solution is feasible in a WAN because such a network is supported by the necessary infrastructure to make the introducer easily accessible (e.g. via one or more well-known servers). The Internet's Domain Name System (DNS) is an exemplar of such an introducer.

Languages for WANs differ widely in their support for failure handling, but in general provide more elaborate failure handling features than their LAN-oriented siblings. Argus features built-in support for atomic transactions to cleanly deal with unfinished computations resulting from partial failures. In Erlang, processes can be linked in a tree-structure to perform failure detection and recovery. Erlang also provides primitives to *monitor* the connection with a remote process. E similarly allows observers to be registered on remote references that trigger upon failure. Distributed Oz features *handlers* and *watchers*: handlers are associated with an operation (e.g. a remote message send) and trigger when their associated operation fails; watchers are associated with e.g. remote objects and trigger when the system detects that the object has become disconnected. One might say that handlers perform failure detection lazily (when their associated operation is performed), while watchers perform eager failure detection, triggering failure handling even when no explicit operation is performed.

To the best of our knowledge, none of the above languages feature service discovery: they all assume that infrastructure is available to introduce distributed computations to one another explicitly. Most languages also do not feature time decoupling, assuming that communicating processes are mostly online at the same time. Languages for WANs also do not cater to arity decoupling directly – references are point-to-point.

3.3.3 Languages for Wireless Sensor Networks

Recently, programming languages have been designed specifically for wireless sensor networks (WSNs). Such networks are mobile ad hoc networks in which the nodes are often very resource-scarce computers (sometimes called "motes") equipped with sen-

sors, communicating via radio technology (e.g. ZigBee) with their neighbours. Most applications devised for WSNs are "divide-and-conquer" programs that request a measurement from all nodes in the network, or in a certain area of the network. Those measurements are subsequently aggregated along intermediary nodes and finally delivered to a wireless base station. Both because of this application structure, and because reconfiguring thousands of motes is impossible to do by hand, languages for WSNs strongly exploit and embrace code mobility.

ActorNet [KSMA06] is a Scheme dialect with support for migratable actors. Actors can be dynamically created and can communicate via asynchronous message sending. Actors can migrate to neighbouring nodes by grabbing their current continuation (via Scheme's `call-with-current-continuation` primitive [ADH+98]) and by parameter-passing it in a message. This allows actors to migrate, execute code on a sensor node and then migrate back to the base station, where sensor readings can be processed. It is not clear exactly how actors on different motes can get acquainted with one another. It seems they can implicitly communicate by means of broadcasted messages. ActorNet actors seem to enjoy the time decoupling properties of the original actor model, although this is not explicitly mentioned by Kwon et al. [KSMA06].

SpatialViews [NKSI05] is an extension of Java with support for the unique concepts of *spatial views* and *spatial view iterators*. A spatial view declaratively denotes a virtual overlay network consisting of all nodes in the ad hoc network providing a certain service, within certain geometrical boundaries. A spatial view iterator replicates a piece of code across all members of a spatial view for a bounded period of time. Interestingly, discovery and migration are transparently handled by the language runtime.

SpatialViews is hard to compare with traditional coordination abstractions because programs are not organised as a set of communicating entities, but rather as migrating pieces of code. However, it provides a form of indirect, time decoupled communication between programs via *service variables*. These are variables allocated in nodes for a limited amount of time, and can be read and written by iterators visiting the node. A form of space decoupling is also provided, because a spatial view specifies the nodes to visit in an anonymous way, by means of a Java interface type. Nodes that contain objects of that type are implicitly part of the spatial view. Communication is performed via aforementioned service variables and can be regarded as synchronisation-decoupled. Service discovery is not provided directly. To get a "view" on which services are available, one can execute an iterator whose sole purpose is to return the list of nodes it visited. SpatialViews is clearly a domain-specific language whose domain is the set of "divide-and-conquer" sensing applications typical of WSNs. While it excels in its domain, it is hard to imagine using SpatialViews to structure applications that fall outside of this domain.

3.3.4 Models and Calculi for Wide Area Networks

In the following two sections, we survey models and calculi, rather than concrete programming languages, designed for representing distributed computations in WANs.

3.3.4.1 Actors and ActorSpace

Actors form a model for distributed computation in open networks [Agh86, Agh90]. An actor has identity (a mail address), a message queue (a mailbox) and a behaviour. The behaviour provides the "script" to process incoming messages. Messages may

be processed concurrently from the message queue by providing a "replacement behaviour" that can start processing the next message in the queue. This does not introduce data races if the original and the replacement behaviour do not share state.

When an actor sends a message to an acquaintance, the message is placed in a mail queue and is guaranteed to be eventually delivered by the actor system. Although the actor's `send` primitive decouples actors in time and in synchronisation (actors communicate strictly asynchronously), it does not decouple them in space. A mail address represents a unique actor and does not allow actors to discover one another by means of an abstract description. Furthermore, the model does not specify any failure detection abstractions.

The inability of mail addresses to represent unknown, undiscovered actors have been addressed in the ActorSpace model [AC93, CA94]. This model is a unification of concepts from both the actor model and the tuple space model of Linda (see section 3.3.5). Callsen and Agha note that, on the one hand, the actor model provides a secure model of communication as an actor may only communicate with actors whose mail address it has been explicitly given via message passing. On the other hand, this disallows actors from getting acquainted with other actors in a time- and space-decoupled manner, as is the case in Linda via tuple spaces.

The ActorSpace model augments the actor model with *patterns*, denoting an abstract specification of a group of actors. The actor model's `send` primitive, which normally takes a receiver mail address and a message as arguments and sends the message to the corresponding mail address, is changed such the receiver of the message can also be denoted by means of a pattern rather than a mail address. For example, an invocation of `send("Stereo","playSong")` sends the `playSong` message to any actor whose own name matches the pattern `"Stereo"` within the context of a so-called *actorspace*.

The `send` primitive delivers the message to a non-deterministically chosen matching actor. Although this behaviour is good when it does not matter to the sender which specific actor receives the message (e.g. when the receiver is a replicated file server), it is not similar to a remote object reference in the sense that multiple messages sent to the same pattern may be received by several different actors. When there are no matching actors, the message send is suspended until at least one matching actor appears. The ActorSpace model explicitly caters to arity decoupling by means of a `broadcast` primitive, which sends a message to all available actors matching a given pattern.

An actorspace is a container for actors and nested actorspaces and acts as a hierarchical scoping mechanism for the resolution of patterns to mail addresses. In order for actors to be subject to pattern matching, they have to be made explicitly visible in an actorspace. Controlling this visibility of actors and actorspaces is done by means of capabilities [CA94].

The ActorSpace model adheres to almost all of the criteria postulated in section 3.2. However, it does not provide any explicit means of performing service discovery or failure handling, a direct heritage from the underlying actor model. Finally, the implementation of actorspaces as discussed by Callsen and Agha [CA94] relies on centralised "coordinators", making the implementation (but not necessarily the model) unsuitable for mobile ad hoc networks.

3.3.4.2 Mobile Ambients

Mobile ambients [CG98, Car99] form a process calculus developed with the goal of unifying the fields of mobile *computing* (i.e. physically mobile computers) and mobile

computation (i.e. logically mobile processes). In the mobile ambient calculus, processes or agents are contained within *ambients*. An ambient is a bounded place where computation happens [CG98]. Processes can implicitly navigate between ambients by making their ambient move into neighbouring and out of enclosing ambients. Processes communicate by asynchronously posting and by synchronously reading messages in their enclosing ambient.

Mobile ambients are an interesting abstraction for modelling ad hoc networks because these networks are naturally composed of mobile devices which can be considered as ambients. In the ambient calculus, operations like entering or opening an ambient automatically suspend a process until a matching ambient is available. By aligning network connectivity with the presence of neighbouring ambients, communication between physically mobile computations can be succinctly expressed. However, one may only interact with an ambient if one knows its name. In order for anonymous ambients to interact, they must somehow be given well-known names with network-wide scope.

Because the ambient acts as a container for the messages posted by its contained processes, mobile ambients feature time and space-decoupled communication: messages do not carry a receiver, they are simply posted within the ambient and can be read at a later point in time by any other process in the same ambient. Processes are not entirely decoupled in synchronisation because message reception is synchronous. The calculus does not feature built-in operations to become aware of neighbouring ambients. However, if one knows the name of an ambient, it is easy to construct a process that tests whether the ambient is available by trying to move into it. However, failure detecting has to be implemented entirely on top of the calculus. Failure of an ambient is represented in the calculus implicitly (but realistically) as that ambient becoming unreachable forever [CG98].

Attempts have been made to convert the ideas of mobile ambients into programming language features. N# [WBB06] is a C#-like object-oriented language with support for ambients that can communicate via *ports*. Interestingly, the language runtime automatically creates and terminates ambients as entities come and go (e.g. people entering or leaving a meeting room). However, for this to work, the language runtime depends on a central "context server", making it unsuitable for ad hoc networks.

3.3.5 Tuple Space Middleware for Ad Hoc Networks

Tuple spaces as originally introduced in the coordination language Linda [Gel85] have received renewed interest by researchers in the field of mobile computing. In the tuple space model, processes communicate by inserting and removing tuples from a shared tuple space, which acts like a globally shared memory. Because tuples are anonymous, they are extracted by means of pattern matching on their content. Tuple space communication is decoupled in time because processes can insert and retract tuples independently. It is decoupled in space because the publisher of a tuple does not necessarily specify, or even know, which process will extract the tuple. This makes Linda ideal for coordinating loosely-coupled processes.

Naturally, a globally shared, centralised, tuple space does not fit the hardware characteristics of mobile ad hoc networks. Adaptations of tuple spaces for mobile computing, such as Linda in a Mobile Environment (LIME) [MPR01], introduce *agents* which have their own, local *interface tuple space* (ITS). Whenever their host device encounters proximate devices, the ITS of the different agents is merged into a *federated transiently shared tuple space*, making tuples in a remote agent's tuple space accessible while the connection lasts.

In the original tuple space model, synchronisation decoupling is violated because there exist synchronous (blocking) operations to extract tuples from the tuple space. However, as the need for total synchronisation decoupling became apparent for mobile networks, mobile computing middleware such as LIME extends the basic model with *reactions* which are callbacks that trigger asynchronously when a matching tuple becomes available in the tuple space.

LIME enables (decentralised) discovery and failure handling by introducing a read-only, system-maintained tuple space whose tuples represent metadata, such as the hosts that are currently connected. Registering reactions on such tuples allows processes to observe their connection with other processes. Even though tuple spaces do not offer explicit arity-decoupling, one-to-many communication can be encoded implicitly either by having a producer post multiple (duplicate) tuples in the space, or by having consumer processes read but not retract tuples, leaving the original tuple for other processes to consume [EFGK03].

Interestingly, like mobile ambients, LIME unifies mobile computing ("physical mobility") with mobile computations ("logical mobility") by allowing agents to migrate between hosts. Using agent migration, agents can be co-located to ensure a stable connection between their ITS. Mobile Agent Reactive Spaces (MARS) [CLZ00] employs this mobile agent paradigm for similar reasons, although it does not define a transiently shared tuple space. In MARS, each device hosts a tuple space and agents can only access that local tuple space. To access another tuple space, agents can migrate between hosts. MARS features a metalevel tuple space that allows programs to register reactions: callbacks that trigger whenever agents perform a read and/or write operation on the baselevel tuple space. However, it is not clear whether this tuple space also records the mobility of agents, which is necessary if one wants to get an overview of all currently connected agents.

LIME extends Linda's operations with context parameters, like explicit location. This trades anonymous communication for more control over the tuple space in which the tuple should be placed. A tuple that was given an explicit tuple space location will independently migrate to that tuple space once it is connected. The net effect is that the tuple will still be accessible to the owner of that tuple space if it is no longer connected to the emitter of the tuple. Tuples on the Air (TOTA) [MZ04] is based on a similar mechanism: rather than merging local tuple spaces upon network connection, tuples are equipped with a *propagation rule* that determines how the tuple migrates from one tuple space to another. Hence, in TOTA, agents can access one another's tuples because it are the tuples themselves that propagate through the network as connections are established. Like LIME, it augments the tuple space model with a form of event notification to notify agents when certain tuples arrive in their tuple space.

Tuple spaces act as a middle man between different processes. As a result, there is no notion of a reference to any particular process. Tuple space-based communication is necessarily global to all processes sharing the tuple space, which may lead to unexpected interactions between concurrently communicating processes.

3.3.6 Middleware for Nomadic Networks

We now describe middleware platforms that have been designed for nomadic networks, which assume a mix of mobile and stationary nodes and hence rely on infrastructure.

3.3.6.1 Rover

The Rover toolkit [JdT+95, JTK97] is designed to support both *mobile-transparent* and *mobile-aware* applications. The desire to hide mobility from applications enables the reuse of existing techniques and applications in a mobile setting. However, Rover recognises the fact that a mobile network is too dynamic to be able to abstract away entirely. Hence, it introduces additional features to make applications aware of mobility at key points. Rover is based on a client/server architecture. Rover applications are partitioned into clients, which are usually deployed on mobile devices, and servers, which mostly reside on a stationary host. There is no support for direct communication between the mobile clients themselves.

Rover offers support for *disconnected operation*: the ability of the mobile client to continue performing useful work while being disconnected from the server. It does this by means of two concepts: *relocatable dynamic objects* (RDOs) and *queued remote procedure call* (QRPC). An RDO is essentially a mobile object, whose code and data can be migrated from client to server or vice versa. Hence, code and data can be co-located on the mobile host, avoiding network traffic and enabling disconnected operation. Every RDO has a "home" server, which maintains a primary copy of the RDO's data. Clients essentially download replicas from the server and can modify and interact with their local copy.

Clients communicate updates to their local RDOs to the server through queued RPCs. QRPC decouples communication between client and server in time: Rover queues the RPCs in a log, and performs the RPCs when a connection between client and server is available. At the application level, clients either block while the QRPC is pending, or get notified of the result asynchronously via a callback method (enabling a QRPC to additionally introduce synchronisation-decoupling). When the update of the client is made to the master copy, conflicts resulting from concurrent modifications may occur. Rover provides hooks that allow applications to provide custom conflict resolution strategies.

Rover has a notification mechanism to enable applications to react to changes in their execution context. The application can either poll for changes in the network, or register callbacks to get notified. As Rover is not designed for ad hoc networks, it provides no support for service discovery.

3.3.6.2 Java Intelligent Network Infrastructure

Sun Microsystem's Jini architecture for network-centric computing [Wal99, Arn99] is a platform for service-oriented computing built on top of Java. Jini introduces the notion of lookup services[1]. Services may advertise themselves by uploading a proxy to the lookup service. Clients search the network for lookup services and may launch queries for services they are interested in. The fully qualified names of Java interface types are used as a common ontology to describe and discover services. Clients can download the advertised proxy of a remote service and may interact with the remote service through the proxy. Jini's lookup services do not enforce decentralised discovery as they may be hosted on a remote device, separate from that of either client or service [Arn99]. Discovery can be made decentralised if the devices hosting the clients or services themselves also directly host a lookup service. However, judging from its

[1] With respect to the terminology introduced in section 3.2.1, a Jini lookup service corresponds to a service discovery mechanism, not to a traditional "lookup service".

bootstrapping discovery protocols, Jini is designed for its lookup services to be hosted on separate devices [Wal01].

Jini provides direct support to deal with the two most apparent phenomena of ad hoc networks: the fact that clients and service providers may join with and disjoin from the network at any time, without any prior warning. To deal with unannounced disconnections, Jini employs leasing (see also section 4.9.2): when services advertise themselves in a lookup service, they are given a lease. They must then explicitly renew their lease with the lookup service; if they cannot (e.g. because of a network partition), the lookup service removes the service advertisement such that it doesn't provide stale information. Likewise, clients should interact with services on the basis of a lease such that a service may reclaim any resources allocated for the client session whenever either one disjoins from the network. Leasing is essentially a form of connection-independent failure handling.

Jini is primarily a framework for bringing clients and services together in a network with minimal administrative infrastructure. Once a client has downloaded a service proxy, the proxy is the communication channel to the service. This proxy may encapsulate different implementations to communicate with its service [Wal01]. For example, the proxy may buffer messages when the remote service is disconnected to achieve decoupling in time. Hence, Jini's *architecture* is flexible enough to accomodate time, synchronisation or even arity-decoupled references. However, to the best of our knowledge, Jini does not offer this functionality. By default, Jini relies on Java RMI and the proxies advertised by services communicate synchronously with their service over point-to-point protocols.

3.3.7 Publish-subscribe Middleware for Ad Hoc Networks

The publish/subscribe paradigm has been proposed by many researchers as a suitable abstraction for mobile ad hoc networks [Mei02, CJ02, EGH05] precisely because it inherently provides support for loosely coupled interaction between publishers and subscribers and because it naturally supports arity decoupling (events may be delivered to any number of subscribers).

3.3.7.1 Location-based Publish-Subscribe

Location-based Publish/Subscribe (LPS) [EGH05, EGH06] is a publish/subscribe architecture designed specifically for ad hoc applications, but not necessarily for mobile ad hoc networks. The authors of LPS regard ad hoc applications as independent of the underlying physical network infrastructure. Hence, LPS is not necessarily confined to real ad hoc networks. In fact, their current implementation makes use of infrastructure in the form of a centralised web service that manages the publications and subscriptions.

In essence, LPS is a content-based publish/subscribe architecture, with publishers publishing events and subscribers filtering events based on the event's content. LPS is designed for nomadic networks and it is assumed that all mobile devices can communicate implicitly via a shared infrastructure, even if they are not in communication range. However, in order to scope interactions between devices, event dissemination and reception is bounded in physical space: a publisher defines a *publication range* and a subscriber defines a *subscription range*. Both are independent of the devices' communication range. Only when the publication range of the publisher and the subscription range of the subscriber physically overlap is an event disseminated to the subscriber.

This introduces a form of spatial scoping on top of the content-based publish/subscribe paradigm.

LPS decouples publishers and subscribers in time, space and synchronisation. Decoupling in time is bounded by an event's *time-to-live*: after this timeout period has expired, an event is no longer published. To the best of our knowledge, LPS provides no explicit discovery or failure handling mechanisms. It is not possible to react to connecting or disconnecting publishers or subscribers.

3.3.7.2 Scalable Timed Events and Mobility

Scalable Timed Events and Mobility (STEAM) is an event-based middleware designed for supporting collaborative applications in mobile ad hoc networks [MC03, MCNC05]. It shuns the use of centralised components such as lookup and naming services to avoid any dependencies of mobile devices on a common infrastructure. STEAM is essentially a publish/subscribe middleware where events can be filtered according to event type, event content and physical proximity.

STEAM builds upon the observation that the physically closer an event consumer is located to an event producer, the more interested it may be in that producer's events. For example, in a Vehicular Ad hoc Network (VAN), cars can notify one another of accidents further down the road, traffic lights can automatically signal their status to cars near a road intersection or ambulances could signal their right of way to cars in front of them. To this end, STEAM allows events disseminated by producers to be filtered based on geographical location using *proximities*. Proximities are first-class representations of a physical range, which may be absolute or relative (i.e. a relative proximity denotes an area surrounding a mobile node, changing as the node moves). In order to enforce this scoping of events using proximities, STEAM uses a location service that uses sensor data or GPS coordinates to determine the geographical location of nodes in the network.

STEAM decouples publishers and subscribers in space and synchronisation. It does not decouple them in time: published events are disseminated using multi-hop routing throughout their proximity, after which they disappear. Hence, if a subscriber is not in range at the time the event is disseminated, it will miss the event. Persistent events should thus be published repeatedly. Failure handling because of publishers or subscribers moving out of communication range also has to be encoded on top of the basic event service.

3.3.7.3 Epidemic Messaging Middleware for Ad Hoc Networks

The Epidemic Messaging Middleware for Ad Hoc Networks (EMMA) [MMH05] is an adaptation of the Java Message Service (JMS) API [MHS02] for mobile ad hoc networks. In JMS, Java components interact asynchronously by posting messages to and reading messages from message queues. This can be used both for point-to-point and publish/subscribe interaction. In JMS, queues are often managed by central servers. EMMA replaces such central servers by a discovery mechanism that allows queues to be discovered in the local ad hoc network. It uses leasing to remove stale advertisements, similar to Jini.

EMMA, like JMS, distinguishes between durable and non-durable subscriptions to message queues. A durable subscription remains valid upon disconnection. Non-durable subscriptions are cancelled upon disconnection and the subscription must be

made anew upon reconnection. In JMS, the server buffers events while a durable subscriber is disconnected. In EMMA, events for disconnected subscribers are not buffered but rather sent using an *asynchronous epidemic routing protocol*. Using this protocol, messages are broadcast to each host in range, which in turn sends them to all hosts in its range, and so on. Using epidemic routing, delivery is not guaranteed (i.e. the message may still be lost if there is no path between sender and receiver at the time of routing). Naturally, the delivery ratio increases as the number of nodes in the ad hoc network increases. The protocol does weed out duplicate messages, to ensure that the application receives messages at most once. If a message is flagged as *persistent*, the sender is notified of successful delivery via an acknowledgement. If no such acknowledgement is received within a given timeout, the delivery status is unknown.

Communication in EMMA is naturally synchronisation-decoupled using message queues. It is space decoupled thanks to the use of topics to describe publish/subscribe queues. Thanks to its automatic discovery management of queues, it is suitable for use in pure ad hoc networks.

3.3.7.4 Many-to-Many Invocation

Many-to-many invocation (M2MI) [KB02] is a paradigm for building collaborative systems deployed in MANETs. A middleware framework for Java exists where objects can refer to other objects in the mobile ad hoc network by means of *handles*. A handle is a "remote reference" that identifies objects in the proximity that implement a certain Java interface.

M2MI distinguishes between *unihandles*, *multihandles* and *omnihandles*. A unihandle represents one, a multihandle a specific group and an omnihandle *all* objects within communication range that implement the handle's Java interface. A message sent to an omnihandle means "every object out there that implements this interface, call this method" [KB02].

M2MI handles have asynchronous message passing semantics and hence decouple communicating objects in synchronisation. However, M2MI invocations require that asynchronous messages do not return a value or throw an exception: all methods of a handle's associated interface must have a `void` return type and cannot throw any checked exceptions. Request/response interaction or a future-type style of message passing has to be built on top of the basic abstractions.

Communication by means of handles is decoupled in space: the actual identity or location of the objects communicated with remains anonymous (the objects are only implicitly designated by means of the Java interface through which they are exported). However, M2MI handles do not decouple participants in time: if a message is sent to an object which is not in communication range at that time, the message is lost. Any message delivery guarantees have to be programmed on top of the weak delivery semantics offered by M2MI.

In short, M2MI's handlers are a suitable remote referencing abstraction for mobile ad hoc networks, but they are situated at a relatively low level of abstraction. While they abstract away low-level protocol and serialisation issues, aspects such as service discovery, stronger message delivery semantics, message ordering, return values and dealing with disconnections all have to be encoded on top of the basic abstractions.

We will revisit M2MI on several occasions throughout the rest of this dissertation. In section 7.5.3, we describe the differences and similarities between ambient references and M2MI. In chapter 8, we highlight the role of the M2MI Java library in the implementation of ambient references. Finally, in chapter 9, we extensively compare

two concrete ad hoc networking applications – one written using M2MI handles and the other using ambient references.

3.3.7.5 One.world

One.world is a system architecture developed on top of Java [GDL+04]. It can be regarded more as an "operating system" for pervasive computing rather than as middleware, providing a common execution platform for pervasive computing applications. In one.world, applications consist of a tree of *environments*. An environment consists of components (Java applications) and *tuples*, self-describing records used to encode an application's persistent data (one.world provides support for checkpointing). Additionally, one.world supports migration of components between different environments.

All components interact though asynchronous event notifications only. Again, an asynchronous publish/subscribe style of interaction is promoted because of its loose coupling, making communication decoupled in time, space and synchronisation. Events may be delivered to multiple components, enabling arity decoupling. One.world uses leasing to cater to connection-independent failure handling and it provides explicit support for service discovery. We will revisit One.world in more detail in section 7.5.5 when discussing work related to ambient references.

3.3.8 Synthesis and Discussion

Table 3.2 summarises our survey of related work. It indicates, for each system under scrutiny, whether and how it adheres to the criteria postulated in section 3.2. The keys used to indicate the degree of coupling between processes and the means of failure handling are explained in table 3.3. We also indicate for each system whether it supports communication by means of message passing. The reasons for including this will become clear in section 3.4.

Recall that the results presented in table 3.2 are naturally biased towards approaches designed *a priori* for mobile ad hoc networks. However, the fact that systems developed for MANETs satisfy more of our criteria than those not developed for MANETs does confirm the effectiveness of those criteria in classifying different approaches to coordination.

3.3.8.1 Interpretation by type of system

Focusing on the different kinds of systems (the table rows) first, we summarise the observations already made earlier in this section:

Languages for LANs introduce very tight coupling between processes at all levels. This is justified by the closed and administered topology of the network.

Languages for WANs mostly relax synchronisation between processes, to deal with failures and latency. Characteristic of all these languages is that processes are introduced via some form of explicit addressing. They rely on the network's infrastructure to reliably resolve such addresses into communication channels.

	Decoupling in				Failure	Decentr.	Message
	Time	Space	Synchronisation	Arity	Handling	Discovery	Passing
Languages for Local Area Networks							
Emerald	N (Error)	N (Address)	N (RPC)	N	Exc.	N	Y
Obliq	N (Error)	N (Address)	N (RPC)	N	Exc.	N	Y
ABCL	Y (Buffer)	N (Address)	Sender only	N	N	N	Y
Languages for Wide Area Networks							
Erlang	Y (Buffer)	N (Address)	Sender only	N	React	N	Y
Argus	N (Error)	N (Address)	N(RPC)	N	Exc.	N	Y
Janus	Y (Blocks)	N (Address)	N (Logic Var)	N	N	N	N
Salsa	Y (Buffer)	N (Address)	Y (Async Msg)	N	N	N	Y
E	N (Error)	N (Address)	Y (Async Msg)	N	React	N	Y
Distributed Oz	Y (Blocks)	N (Address)	N (Logic Var)	N	React	N	Y
Languages for Wireless Sensor Networks							
ActorNet	Y (Buffer)	N (Address)	Y (Async Msg)	Y	N	N	Y
SpatialViews	Y (Mediator)	Y (Discovery)	Y (Mediator)	Y	N	N	N
Models and Calculi for Wide Area Networks							
Actors	Y (Buffer)	N (URI)	Y (Async Msg)	N	N	N	Y
ActorSpace	Y (Buffer)	Y (Mediator)	Y (Async Msg)	Y	N	N	Y
Mobile Ambients	Y (Mediator)	Y (Mediator)	Sender only	N	N	N	N
Tuple Space Middleware for Ad Hoc Networks							
LIME	Y (Mediator)	Y (Mediator)	Y (Mediator)	Y	React	Y	N
TOTA	Y (Mediator)	Y (Mediator)	Y (Mediator)	Y	React	Y	N
MARS	Y (Mediator)	Y (Mediator)	Y (Mediator)	Y	Lease	N	N
Middleware for Nomadic Networks							
Rover	Y (Buffer)	N (Address)	Y (Async Msg)	N	React	N	Y
JINI	N (Error)	Y (Discovery)	N (RPC)	N	Lease	Y	Y
Publish-Subscribe Middleware for Ad Hoc Networks							
EMMA	N (Lost)	Y (Mediator)	Y (Mediator)	Y	Lease	Y	N
LPS	Y (Mediator)	Y (Mediator)	Y (Mediator)	Y	N	N	N
STEAM	N (Lost)	Y (Mediator)	Y (Mediator)	Y	N	N	N
M2MI	N (Lost)	Y (Broadcast)	Y (Async Msg)	Y	N	N	Y
one.world	N (Lost)	Y (Discovery)	Y (Mediator)	Y	Lease	Y	N

Table 3.2: Survey of Related Work.

Temporal Coupling	
Error	Communicating with an offline process raises an error.
Lost	Information communicated to an offline process is lost.
Blocks	Communication with offline process suspends until available.
Buffer	Communication with offline process is buffered (e.g. in a mailbox).
Mediator	Communication occurs via an intermediate store. Unlike buffers, mediators are not necessarily local to the communicating processes.
Spatial Coupling	
URI	Processes are introduced by explicit address.
Discovery	Service discovery is used to introduce processes.
Mediator	Processes communicate indirectly via a mediator (e.g. tuple space, ambient, event service).
Coupling in Synchronisation	
RPC	Synchronous communication by RPC (Argus allows asynchronous RPC but its promises require synchronisation by blocking).
Logic Var	Process suspends until logic variable is bound.
Sender only	Asynchronous send but synchronous (blocking) receive.
Async Msg	Asynchronous send and receive by message passing.
Mediator	Asynchronous communication via a mediator (for tuple spaces: we assume the use of reactions to read tuples, not the blocking `read` operation of Linda).
Failure Handling	
No	No explicit means of failure handling is provided.
Exc.	Failure handling is represented by means of exceptions, often as a result of performing a remote call.
React	It is possible to register callbacks that can react to failures.
Lease	Connections or exchanged data are leased. Failure can be represented as lease expiration.

Table 3.3: Degrees of coupling and failure handling.

Models and Calculi for WANs naturally share the same properties as languages for WANs, given that most languages are based on the models in one way or another.

Languages for WSNs are strongly biased towards a specific type of applications. Here, the emphasis is not on *communicating* processes but on *migrating* processes, enabling as much work as possible to be done local to a mote, without further communication.

Tuple Space-based Middleware for Ad Hoc Networks successfully decouples communicating processes, even across a volatile ad hoc network. However, one may question in how far these approaches still resemble the original tuple space model, given the profound modifications necessary to adapt the model for MANETs. Interestingly, Murphy et al. note that – to their own surprise – the features that turned out to be most useful in LIME were the reactions (especially reactions that fired *each time* a certain tuple was inserted in the tuple space) and the use of the system-maintained tuple space that keeps track of metadata, enabling applications to keep track of which hosts are connected or not [MPR01]. This reinforces the idea that reactive, event-driven programming seems to better match the ad hoc networking setting than the original tuple space abstractions. In effect, the metaphor of a globally shared memory space becomes inappropriate due to the fact that the computational context changes so frequently.

Middleware for Nomadic Networks naturally assumes some form of infrastructure. In the Rover system, this shows in that processes are tightly coupled in space. Rover is also based on the explicit assumption that clients can frequently synchronise with their server. Jini has very much the opposite properties: it provides good service discovery abstractions to enable applications to interact spontaneously, without prior coordination or administration. However, once processes have been introduced, Jini relies on standard Java RMI for communication, which is not designed for ad hoc networks. Jini introduces leasing to manage the lifetime of service advertisements.

Publish/Subscribe Middleware for Ad Hoc Networks successfully decouples processes in space and synchronisation. Many systems do not support true decoupling in time in the sense that, when an event is broadcast, it is lost to any subscriber that is temporarily disconnected. Nevertheless, they naturally cater to arity decoupling and sometimes do not even require an explicit form of service discovery (events are implicitly delivered to all subscribers in range). While anonymous, connectionless communication is a publish/subscribe architecture's major strength, it is at the same time its major weakness. Sometimes, processes will want to set up a stateful, connection-oriented collaboration. In a pure publish/subscribe architecture lacking any form of point-to-point communication, it is very hard to ensure that certain events are only delivered to certain subscribers. Moreover, publish/subscribe architectures offer no built-in support for correlating events (e.g. a request with a reply).

3.3.8.2 Interpretation by Criterion

In the following paragraphs, we evaluate the results of table 3.2 by column, i.e. which systems do or do not satisfy the coordination criteria from section 3.2.

Decentralised Discovery is the ability to *autonomously* react to services becoming (un)available in the local ad hoc network. Relatively few systems implement a form of service discovery. Jini is well-known for its service discovery abstraction, which enables applications to track new services and to deal with disappearing services through leasing, although discovery is not necessarily decentralised. In tuple space-based middleware like LIME and TOTA, the middleware performs service discovery implicitly. At the application-level, the effects of discovery become visible by merging tuple spaces or by enabling tuples to propagate to a new tuple space.

Decoupling in Time enables processes to communicate while being disconnected. It is achieved in many systems by either buffering messages sent to offline parties, or by communicating via a *mediator*. Often, the mediator takes responsibility for storing messages until a potential receiver becomes available. For example, in mobile ambients, the ambients act as stores for messages released by their contained processes. In tuple space-based middleware, the tuple space acts as a logically shared store of tuples. In publish/subscribe systems, the event notification service can buffer events when subscribers are disconnected. However, most pub/sub systems for mobile ad hoc networks do not buffer events for disconnected subscribers. This is motivated by the fact that most events are relevant only for a limited period of time, after which they become obsolete anyway.

Decoupling in Space enables processes to coordinate anonymously. In systems like Jini, which are layered on top of an object-oriented system, service discovery is used to acquire a reference to a service without knowing its exact address. After service discovery, communication becomes coupled in space. In tuple space or pub/sub systems, communication is always anonymous: published tuples or events do not carry a receiver. M2MI, interestingly, provides the properties of a pub/sub system with an object-oriented twist: messages are directed at a handle, but handles deliver the messages to objects anonymously based on a Java interface description.

Decoupling in Synchronisation ensures that the control flow of processes is not suspended upon communication. In most object or actor-based systems it is achieved by means of pure asynchronous message passing. In other systems, it is achieved by the use of a mediator that manages the actual transmission of the message or tuple. Note that tuple spaces provide full synchronisation decoupling only when using the reactive programming style advocated by adaptations for MANETs. These systems also allow processes to retract tuples by means of Linda's `read` operation, which blocks the process until a matching tuple becomes available.

Arity Decoupling enables processes to abstract from the number of processes communicated with. Evidently, point-to-point abstractions (offered by most object-oriented systems) do not decouple processes in arity. In a tuple space or publish/subscribe system, all communication is inherently one-to-many. In tuple spaces, processes operating on the same tuple space are all potential "receivers" of the tuple. In publish/subscribe systems, all subscribers that match the published event are eligible receivers. Note that the proximities of STEAM and the publication space of LPS allow for an additional form of scoping on the event delivery process.

Connection-Independent Failure Handling enables processes to detect and deal with failures independent of the underlying state of the network connection. In most languages for LANs, failure handling is represented by means of exception handling. Some languages for WANs, like Erlang, Oz and E introduce dedicated failure detection primitives thus providing first-class support for partial failure handling. While these primitives allow the programmer to react to disconnections, they do not immediately provide support for a more high-level description of failures (e.g. only considering a disconnection a failure if it outlasts a certain time period). LIME enables processes to monitor the connection status of available agents by means of a special system-maintained tuple space whose tuples describe information about the system configuration. TOTA similarly represents connection and disconnection events using dedicated tuples. Some systems, most notably Jini, use leasing to introduce failure handling. Lease expiration is treated implicitly as a failure. Since lease expiration is not directly related to network disconnection, leases form a good connection-independent failure handling mechanism. Finally, most publish/subscribe approaches offer no abstractions for monitoring connectivity with other processes. Indeed, in these systems, often one process is not even aware of the processes communicated with or even if a published event was successfully received by another party.

3.4 The Object-Event Impedance Mismatch

From our analysis of related work, we draw two conclusions:

1. Object-oriented abstractions based on message passing fail to provide a total decoupling of processes in a mobile ad hoc network.

2. Publish/Subscribe systems and Tuple Spaces specifically designed for mobile ad hoc networks enable the best decoupling of processes.

Combining these two facts, it appears that one is forced to abandon the object-oriented message passing metaphor if one wants to express scalable coordination between processes in a MANET. However, if the non-distributed part of an application is written in an object-oriented language, the overall application is then forced to combine two different paradigms: the object-oriented paradigm must interact with an alien communication paradigm (tuples or events).

A multi-paradigm approach is not necessarily a bad approach. However, its suitability depends on the difficulty of combining the paradigms involved. In this section, we argue that combining objects with events is far from trivial. Why is this the case? One of the design goals of most coordination languages and middleware is to provide abstractions which are *orthogonal to* (i.e. independent of) the language with which they are combined, follwing the original motivations of the Linda language [GC92]. As a result of this orthogonality, no unification is sought between elements of the application layer and elements of the communication layer. We claim that this lack of integration leads to what we call the *object-event impedance mismatch*, analogous to the way object persistence suffers from the infamous *object-relational* impedance mismatch [CD96].

The object-relational impedance mismatch is caused by the fundamental differences between modelling data as objects and modelling data as tuples which are part of relations. For example, objects encapsulate their state, enabling operations to be polymorphic. Tuples expose state, enabling efficient and expressive filtering, querying

and aggregation of data. Objects refer to one another via references, while tuples are associated with one another via foreign keys. Identity is fundamental to objects, while tuples lack any inherent form of identity, and so on. We discuss similar such differences, but rather than contrasting objects with the relational model, we will contrast objects with event-driven communication models. Figure 3.1 contrasts object-oriented with event-driven communication. The highlighted differences are discussed in each of the following sections.

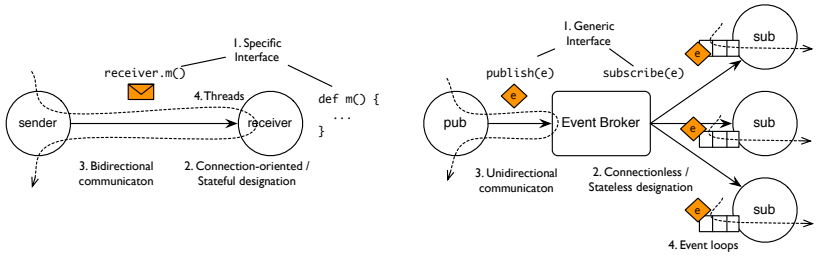

Figure 3.1: Contrasting object-oriented with event-driven communication.

3.4.1 Specific versus Generic Communication

In object-oriented programming languages, objects communicate by means of message passing. The interface of an object usually corresponds to the set of visible methods it (or its class) defines. Messages encapsulate a "selector", which usually corresponds to the name of the method to be invoked on the object to which the message is sent. In object-oriented programming, therefore, communication between objects is expressed in terms of very *specific* operations. The generic acts of *sending* and *receiving* these messages are entirely hidden by the language.

In a distributed object-oriented language, sending and receiving messages to and from *remote* objects is most often done implicitly by means of the regular message passing semantics already provided by the language. In an object-oriented language, remote communication can be expressed as `receiver.selector(arg)` rather than `send(receiver,selector,arg)`. Likewise, it is often not necessary to introduce an explicit `receive` statement: message reception is represented in terms of method invocation.

In event-driven systems and in tuple spaces, communication between processes is not in terms of specific operations, but rather in terms of very *generic* operations, whose arguments are the events or tuples to be communicated. This is very obvious in tuple spaces, where communication is performed in terms of explicit `in(tuple)` and `out(tuple)` operations. In an event-driven system, events are often dispatched to the event broker (the infrastructure between event producers and consumers) by means of a generic `broker.publish(event)` invocation. Likewise, event reception is often an explicit operation:

```
broker.subscribe(new Listener() {
  public void reactTo(Event e) {
    // handle incoming event
  }
})
```

One advantage of a generic communication interface is that it is much easier to express general *patterns* of communication. This is because such an interface already automatically abstracts from the details of the data being communicated. To express generic patterns via an object-oriented interface, one generally requires a special meta-level construct to be able to intercept messages without reference to their specific selector and arguments (e.g. `doesNotUnderstand:` in Smalltalk).

In short, if objects are to communicate with one another via events, they must abandon their otherwise specific communication interface in favour of the generic interface promoted by the event system.

3.4.2 Connection-oriented versus Connectionless Designation

In a distributed object-oriented program, objects communicate via point-to-point channels known as (remote) object references. As can be witnessed from our survey on related work, this abstraction provides limited or no support for space-decoupled (anonymous and one-to-many) communication (cf. the properties of the object-oriented languages for LANs and WANs). However, it remains a useful communication abstraction with two important properties. First, a remote object reference is a *connection-oriented* communication channel, which allows the sender to know the identity of the object to which it sends messages. Second, it is a *stateful* communication channel ensuring that multiple messages sent via the same reference are processed by the *same* receiver. Later messages are sent via the reference in the understanding that messages sent earlier are processed first.

Publish/subscribe and tuple space systems feature stateless one-to-many communication by default. It enables anonymous and arity-decoupled communication, which is required in a mobile ad hoc network. At the same time it becomes difficult to express connection-oriented communication, as an event or tuple becomes accessible to all participating processes, or stateful communication, because multiple consecutive events may be received by a different set of subscribers. These limitations of tuple spaces, which also apply to publish/subscribe systems, have been observed earlier by Agha and Callsen [AC93]:

> "[Linda] does not provide any locality, i.e. communication cannot be declared local to two or more processes that want to exchange information without allowing other processes to potentially interfere."

There exist ways of building connection-oriented communication on top of such systems (e.g. private tuple spaces or event topics encoding a connection), but these abstractions are second-class, just like connectionless one-to-many communication is a second-class abstraction in an object-oriented programming language.

Note that the notion of a connection-oriented communication channel does not necessarily imply a *secure* communication channel. The above discussion deals with scoping and coupling issues on a software engineering level, not with issues such as one process intentionally "eavesdropping" on a communication channel between other processes.

In short, if objects are to communicate with one another via events, they must abandon the remote referencing abstraction in favour of connectionless communication via the event broker. This makes the receiver(s) of a message anonymous, which enables space-decoupled communication, but at the same time may lead to interference between communicating objects.

3.4.3 Bidirectional versus Unidirectional Communication

Object-oriented programs communicate via message passing which fosters a request/reply style of communication. Even though this request/reply style is often implemented by means of synchronous (remote) method invocation, much research in concurrent object-oriented programming has been devoted to maintain the request/reply interaction pattern while relaxing the synchronisation constraints (cf. background on futures in section 4.9.2).

Publish/subscribe systems and tuple spaces decouple processes by essentially introducing pure asynchronous one-way operations (e.g. publishing an event, writing a tuple). Request/response interaction can of course be built on top of such systems, e.g. by manually correlating request and response events/tuples by means of an identifier. Again, these are second-class abstractions.

Conversely – and perhaps less obviously – an asynchronous, unidirectional event notification also has to be represented by means of second-class abstractions in an object-oriented language. Signalling an asynchronous event is often performed by *synchronously* invoking a method that returns no result. The first-class asynchronous notification of an event system is thus represented as a second-class synchronous method invocation in an OO system.

In short, if objects are to communicate with one another via events, they must abandon the bidirectional request/response message passing abstraction in favour of unidirectional event notification.

3.4.4 Threads versus Event Loops

In an event-driven system, events are usually delivered to an application by an *event loop* which is the process representing the event notification engine (cf. section 4.3.1 for a more in-depth discussion on event loops). However, the integration of event delivery with multithreaded object-oriented languages often leaves much to be desired. The archetypical integration represents event handlers as "listener" or "observer" objects. The methods of these objects are intended to be invoked by event loops. Because of the synchronous method invocation semantics predominant in object-orientation, the method is executed by the thread of control of the event loop itself. To make the situation more concrete, here is a canonical example of event notification in Java:

```
// code executed by application thread
publisher.subscribe(new Subscriber() {
  public void reactTo(Event e) {
    // code executed by event notification engine
  }
});
```

This style of event notification has two important consequences:

- Event handler objects must be made multiple thread-safe, i.e. they require synchronisation constructs to prevent data races when they concurrently access state manipulated by application-specific threads. Because thread management lies outside the control of the application, it is entirely implicit in the code whether multiple events are signalled to registered event handlers concurrently or sequentially.

- If the event handler object uses the event loop's thread of control to perform application-level computation, it can make the event loop unresponsive. A testament to this is the documentation of the event-driven Java GUI construction framework Swing which advises developers to structure their applications such that listener methods relinquish control to the framework as soon as possible.

Event loops provide the programmer with the ability to consider the handling of a single event as the unit of concurrent interleaving. The major strength of this model is that it significantly raises the level of abstraction for the developer, as the level of granularity at which the concurrent behaviour of the system can be studied is much more manageable. Rather than having to consider the possible interleaving of each *basic instruction* in each method body, the programmer need only consider the interleaving of the different methods themselves[2]. It has been argued even in the context of thread-based object-oriented concurrency models that complete mutual exclusion of each method of a concurrently accessible object ought to be the norm [Mey93].

Event loops also have their drawbacks. Event delivery is an asynchronous process, and most event-driven systems cannot succinctly express the overall control flow of an application. Rather, the control flow is dispersed across many different event handlers, a phenomenon known as *inversion of control* in the literature [HO06]. We further discuss this phenomenon and how to resolve it in section 4.5.1.

Choosing between thread-based or event-based concurrency is the topic of long-standing debates in the literature, and we are certainly not the first to contrast these two systems. A well-known talk by Ousterhout provides a more general discussion comparing the assets and drawbacks of threads and events [Ous96]. Miller, in his dissertation, studies the differences between both in more detail to support the use of event loops in the E programming language [Mil06].

In short, if objects are to communicate with one another via events, the event broker becomes an additional source of concurrency in the object-oriented program. In multithreaded languages, dealing with this additional source of concurrency is non-trivial. This raises the question of whether threads should – despite their predominance – remain the model of choice for representing concurrent activities between objects.

3.4.5 Reconciling Objects with Events

In this section, we have contrasted the communication properties of object-oriented and event-based publish/subscribe models. The simplest solution to resolve the object-event impedance mismatch is to discard either objects or events and to resort to a single-paradigm solution where only one of both is used. However, it should be clear from the above discussion that both paradigms have their merits. Neither does one solve the object-relational impedance mismatch by discarding objects or relational databases.

Eugster et al. propose an object-oriented language extension with support for events [EGD01]. However, this language extension does not attempt to integrate the publish/subscribe style with object-oriented concepts. Rather, their system provides *both* object-oriented and event-driven concepts, with little integration between the two. For example, the language extension still distinguishes event notification from message passing and objects maintain their thread-based execution model.

[2] In most pre-emptive thread systems, threads may be interrupted at the level of the instructions of the underlying (virtual) machine. This makes it even harder for a developer to consider the effects of concurrent interleaving because the exact points where control is yielded to other threads are completely hidden. Not even the execution of a single statement in a method body can be considered atomic.

The approach to resolving the object-event impedance mismatch which we will pursue in this dissertation is to find a *unification* of object-oriented and event-driven programming such that objects can use event-driven communication in a mobile ad hoc network without having to adapt to an alien communication paradigm. In particular, we want our unification to combine the aforementioned properties of objects and events as follows:

- Objects should be able to communicate via events by means of the familiar, *specific* communication interface afforded by message passing. This makes communication more concise because there is no need for explicit `publish` and `subscribe` operations.
- Objects should be able to communicate via events by means of the familiar object referencing mechanism. While sometimes stateful, connection-oriented communication is still required, object references must be augmented such that they can directly express stateless, connectionless communication.
- Objects should be able to communicate via events and still retain the ability to perform the *bidirectional* interactions afforded by message passing.
- Objects should be equipped with an *event loop* concurrency model that can adequately cope with the additional sources of concurrency introduced by event brokers.

We will revisit the object-event impedance mismatch and how ambient references resolve it by satisfying the above requirements in section 7.2.

3.5 Conclusion

Coordination abstractions allow programmers to describe how distributed processes discover one another, how they communicate and synchronise, and how they can gracefully recover from disrupted operations due to failures. The volatility and infrastructure-scarcity of mobile ad hoc networks forces an evaluation of existing coordination abstractions in light of this new type of networks. We do so by means of six criteria. Decentralised discovery and space decoupling allow processes to abstract from the lack of infrastructure in a MANET. Time and synchronisation-decoupling and connection-independent failure handling enable processes to better abstract over volatile connections. Finally, arity decoupling enables processes to abstract from the total number of proximate processes communicated with.

When using these criteria to interpret the suitability of existing coordination abstractions for use in mobile ad hoc networks, it becomes apparent that extensions of object-oriented coordination abstractions (i.e. abstractions using message passing) are unable to satisfy all of the postulated criteria. Approaches that introduce a mediator, like a tuple space or an event broker, scale much better in MANETs. However, these coordination abstractions do not compose gracefully with the object-oriented paradigm. In chapter 7, we will return to this issue after having introduced ambient references. As we shall describe, ambient references mitigate this impedance mismatch, by pulling the event broker into the language and by unifying concepts external to the paradigm with object-orientation. Before we can go into further detail, we first have to introduce ambient references and the AmbientTalk/2 language. The latter serves as our language

laboratory to study the unification of object-orientation with event-driven coordination abstractions, and is the topic of the following chapter.

Chapter 4

AmbientTalk

In this chapter, we introduce the AmbientTalk/2 programming language. It is the experimental research vehicle in which ambient references have both been designed and implemented. In particular, it offers an event-driven model of computation that aligns well with the requirements of ambient references and the event-driven nature of distributed computations. Next to explaining this event-driven computation model, we focus on AmbientTalk/2's domain-specific features for orchestrating service discovery and composition in mobile ad hoc networks. We also describe how the language enables an ambient-oriented programming style in order to treat network partitions as a normal mode of operation. We describe AmbientTalk/2's object model, concurrency model and distributed communication model in detail. We also highlight related work that has had a major influence on the design of the language.

A note on terminology before discussing AmbientTalk/2 in more detail: throughout this dissertation, we will use the term AmbientTalk to refer to AmbientTalk/2. Where confusion may arise between predecessor and successor, the languages' full names are used to clarify the text.

4.1 History and Design Rationale

AmbientTalk has been designed to fill a gap in the field of distributed programming languages. Even though distributed programming languages are rare, they form a suitable development tool for encapsulating many of the complex issues engendered by distribution [BST89]. Nevertheless, as can be witnessed by our survey of related work in section 3.3, most research on mobile computing has been performed in the area of middleware systems [MCE02]. Barring the languages developed for the specific class of wireless sensor networks (e.g. ActorNet [KSMA06] and SpatialViews [NKSI05]), the area of programming language research has seen little innovation with respect to tackling the issues raised by mobile ad hoc networks in general.

In early 2005, Jessie Dedecker developed AmbientTalk/1 in the context of his doctoral dissertation [Ded06]. The language is developed as an operational incarnation of the ambient actor model [DV04]. AmbientTalk/1's limitations, previously discussed in section 2.5, have formed the direct motivation for the creation of a successor language. AmbientTalk/2 is, however, not the brainchild of a single person. Its design and implementation have been influenced by different researchers at the Programming Technology lab over the past two years. Nevertheless, its overall design and implemen-

tation can be attributed to the author together with Stijn Mostinckx. Leaving the core language design in the hands of only two architects has enabled us to ensure that the language retains "conceptual integrity", as Frederick Brooks calls it in his landmark book "The Mythical Man-Month" [Bro95].

The differences between AmbientTalk/1 and AmbientTalk/2 are extensively discussed in section 4.9.1. Let us first focus on the commonalities instead. At their core, both AmbientTalk/1 and AmbientTalk/2 are small, dynamically typed, object-oriented programming languages. Both languages have an actor-based, event-driven model of concurrency and feature built-in abstractions for discovering remote objects. They both remain faithful to the four characteristics defining the ambient-oriented programming paradigm, previously explained in section 2.3. However, as will become clear later, they differ in the way in which they satisfy the characteristics.

Rather than employing a classic multithreaded concurrency model, AmbientTalk/2's model is founded on the actor model of computation [Agh86] and its many incarnations in languages such as Act1 [Lie87], ABCL [YBS86] and Actalk [Bri88]. However, AmbientTalk/2's closest relative is the E programming language [MTS05] (further described in section 4.9.2). E combines actors and objects into a unified model called *communicating event loops*, which is based on event loop concurrency, described in section 4.3.1.

4.2 AmbientTalk: an Object-oriented Language

AmbientTalk remains, first and foremost, a language to compose objects (services) across a mobile ad hoc network. Despite the domain-specific nature of its abstractions for distributed programming, AmbientTalk remains a full-fledged object-oriented programming language in its own right. It inherits most of its standard language features from Pico [D'H96], Self [US87], Scheme [ADH+98] and Smalltalk [GR89]. From Pico, it inherits its syntax for tables (arrays), assignment, infix operators and function invocation. From Scheme, it inherits lexically scoped anonymous closures. From Self and Smalltalk, it inherits an expressive block closure syntax, the representation of closures as objects and the combined use of block closures and message passing for the definition of control structures. AmbientTalk's object model is derived from Self: classless, slot-based objects using delegation [Lie86] as a reuse mechanism. We first focus on AmbientTalk's object-oriented abstractions.

4.2.1 Objects, Instantiation and Delegation

AmbientTalk is a dynamically typed, object-based language. Computation is expressed in terms of objects sending messages to one another. Objects are not instantiated from classes. Rather, they are either created *ex-nihilo* or by cloning and adapting existing objects. The following code illustrates standard object-oriented programming in AmbientTalk.

```
def Account := object: {
  def balance := 0;
  def init(amount) { balance := amount };
  def deposit(amnt) { balance := balance + amnt };
  def withdraw(amnt) { balance := balance - amnt };
};
def LimitAccount := object: {
```

```
  super := Account;
  def limit := 0;
  def init(lowest, amount) {
    super :=  Account^new(amount);
    limit := lowest;
  };
  def withdraw(amnt) {
    (self.balance - amnt < limit).ifTrue: {
      raise: TransactionException.new(self, amnt);
    } ifFalse: {
      super^withdraw(amnt);
    }
  };
};
def account := LimitAccount.new(-500, 1000);
account.deposit(20);
account.balance; // returns 1020
```

Two prototypes are defined, one for `Account` objects and one for `LimitAccount` child objects, which set a limit to the amount of money that can be withdrawn from the account. Objects can be created ex-nihilo, by cloning or by instantiating objects. In the above example code, two objects are created ex-nihilo (i.e. they are "anonymous objects") and bound to the variables `Account` and `LimitAccount` respectively. AmbientTalk objects consist of fields and methods, although methods can subsume fields because AmbientTalk unifies field access with nullary method application[1].

Instantiating an object is done by sending it the message **`new`**, which creates a shallow copy of that object and initialises the copy using its `init` method, which plays the role attributed to "constructors" in class-based languages. Every AmbientTalk object understands this **`new`** message. Any arguments passed to **`new`** are passed on to the copy's `init` method such that the copy can be reinitialised with new values. AmbientTalk's object instantiation protocol closely corresponds to class instantiation in class-based languages, except that the new object is a clone of an existing object, rather than an empty object allocated from a class.

By convention, when an object receives a message which it does not understand, it *delegates* the message to the object bound to its slot named **`super`**. The object stored in the **`super`** slot is called the *parent object* of the object storing it. In the above example, the parent of an `Account` object is **`nil`** while the parent of a `LimitAccount` object is an `Account` object. The semantics for delegating messages between objects follows that of Self [UCCH91] and Act1 [Lie86]: a delegated message is a message that is forwarded to another object, but for which the **`self`** pseudo-variable remains bound to the delegating object. Hence, AmbientTalk supports object-based (single) inheritance. The **`super`** slot is assignable, such that the parent of an object may change. This enables *dynamic inheritance* which is useful for implementing objects with state-dependent behaviour [UCCH91].

Next to AmbientTalk's built-in "implicit" delegation, which occurs when an object receives a message that it does not understand, AmbientTalk also allows objects to explicitly delegate a request to another object. The expression `obj^m()` delegates the message `m` to `obj`, leaving **`self`** bound to the sender as well. A traditional "super send"

[1] This property allows clients of an object to abstract over the fact whether data is stored in a field or calculated by means of a method. Two notable languages from which this feature originates include Self [US87] and Eiffel. Bertrand Meyer calls this property the *uniform access principle* [Mey00].

in AmbientTalk is then a message that is delegated to the object stored in an object's **super** slot, as shown in the `withdraw` method of the `LimitAccount` object. Note that super-sends and explicit delegation are two separate concepts in AmbientTalk: an object has the ability to delegate messages to objects other than its parent. A useful example thereof will be given in section 4.2.4. While the separation of delegation and super-sends has the benefit of abolishing ad hoc evaluation rules for super-sends, it does introduce the subtle difference between writing **super**^`m()` and **super.**`m()`. While the former is a super-send, the latter is an ordinary message send which happens to be directed at the object's parent object.

In the example, the delegation relationship between the account objects is implicitly represented by assigning an `Account` object to the **super** field of a `LimitAccount` object. Note that each new instance of a `LimitAccount` receives its own new instance of the `Account` prototype as its parent. AmbientTalk features a more declarative syntax for specifying this kind of delegation where copies of the child do not share the same parent object. Consider the definition of a prototypical planar point object:

```
def Point := object: {
  def x := 0;
  def y := 0;
  def init(newx, newy) {
    x := newx;
    y := newy;
  };
  def +(other) {
    self.new(x+other.x, y+other.y)
  };
  def distanceToOrigin() {
    (x*x + y*y).sqrt();
  };
}
```

Now consider a spatial point that consists of three coordinates, specified as an extension of a planar point:

```
def SpatialPoint := extend: Point with: {
  def z := 0;
  ...
}
```

In the above example, `SpatialPoint` and `Point` remain separate objects in their own right. The relationship between a child and a parent object defined by **extend: with:** implies that the child's **super** field is initialised to the parent object and that when a child is cloned, the clone's **super** field is bound to a clone of the parent object. Hence, when a `SpatialPoint` is cloned, the clone has its own `Point` parent object with its own copies of the `x` and `y` fields. Thus, **extend:with:** is the object-based equivalent of class-based inheritance.

4.2.2 Block Closures

AmbientTalk provides support for block closures reminiscent of those in Self and Smalltalk. A block closure is an anonymous function object that encapsulates a piece of code and the bindings of lexically free variables and **self**. Block closures are constructed by means of the syntax `{ |args| body }`, where the arguments can be omit-

ted if the block takes no arguments. The code excerpt below illustrates a typical use of blocks to map a function over a table of numbers[2]:

```
[1,2,3].map: { |i| i + 1 }
// result: [2,3,4]
```

The following code excerpt shows another typical usage of blocks to remove all elements from a collection that fail to satisfy a predicate:

```
def from: collection retain: predicate {
  result := clone: collection; // shallow copy
  collection.each: { |elt|
    predicate(elt).ifFalse: {
      result.remove(elt)
    }
  };
  result;
};
from: [1,-2,3] retain: { |e| e > 0 }
// result: [1,3]
```

Note that block closures can be applied using a familiar canonical function call syntax. Alternatively, they may be treated as objects: a block closure is an object with an `apply` method. In the above example, the call `predicate(elt)` can be equivalently expressed as `predicate.apply([elt])`. Block closures are frequently used in AmbientTalk to represent *delayed* computations, e.g. for implementing control structures but also for implementing nested event handlers, as will be described in section 4.5.1.

AmbientTalk supports both traditional canonical syntax (e.g. `dict.put(key,value)`) as well as keyworded syntax (e.g. `dict.at: key put: value`) for method definitions, message sends and function invocations. As a general rule, keyworded syntax is used for control structures (e.g. **`while:do:`**) or language constructs (e.g. **`object:`**). The canonical syntax is used for expressing application-level behaviour.

4.2.3 Scoping, Nesting and Encapsulation

This section describes AmbientTalk's semantics for the resolution of names (scoping), how nested objects behave, and how scoping can be used to encapsulate an object's state.

Lexical versus Object Scope AmbientTalk is a lexically scoped language, meaning that free variables in a function or method are looked up in their environment of definition. However, AmbientTalk is also an object-based language with delegation, which introduces a second scope in which to resolve names: the object scope. The object scope of an object is the set of all names defined in the object plus the object scope of its parent object (the object referenced by its `super` field). The rules for distinguishing which scope to use when resolving a name are straightforward:

- An unqualified identifier (e.g. `x`) is resolved in the lexical scope.
- A qualified identifier (e.g. `o.x`), is resolved in the receiver's object scope.

[2]Tables are arrays whose elements are indexed starting from index 1, not 0. The terminology stems from Pico [D'H96].

These rules have a large effect on programs: lexical variable access can be statically determined, while qualified access is subject to late binding (enabling object-oriented polymorphism). The interplay between object inheritance and lexical scoping is particularly subtle. Consider the following example:

```
def obj := object: {
  def x := 0;
  def staticAccess() { x };
  def dynamicAccess() { self.x };
}
```

In the code above, `obj` defines two accessors for its `x` field. The first accessor performs an unqualified access and hence looks up `x` in the lexical scope. The second accessor performs a self-send, looking up `x` in `obj`'s object scope. Both will access the same field. The difference between both only becomes apparent in the context of object-based delegation. Consider the following code:

```
def child := extend: obj with: {
  def x := 42;
}
```

Invoking `child.dynamicAccess()` yields `42`, because **`self.`**`x` is late-bound and starts the lookup in `child`. However, invoking `child.staticAccess()` returns `0`: the `x` identifier referred to within the `staticAccess` method is the lexically visible one, and no object can change its resolution. Hence, the resolution of `x` is not the same as that of **`self.`**`x`.

Nesting and Encapsulation In AmbientTalk, objects may be arbitrarily nested within other objects, functions or methods. Because of lexical scoping rules, this enables nested objects to access the lexically visible state and behaviour of enclosing objects. Nesting objects is crucial to achieve encapsulation because AmbientTalk has no notion of visibility modifiers for fields or methods. All fields and methods of an object are considered "public". Nevertheless, a field or method can be made "private" to a scope by means of lexical scoping. The following code shows the definition of an object inside the definition of a function.

```
def makeBankAccount(balance) {
  object: {
    def deposit(amnt) {
      balance := balance + amnt;
      "ok"
    };
  }
}
```

Because the bank account object encapsulates the `balance` variable in its private, lexical scope, it cannot be selected from within its object scope, i.e. `makeBankAccount(100).balance` would result in an exception indicating that the name `balance` could not be resolved.

This form of expressing encapsulation is by no means novel or specific to AmbientTalk. It is a routinely used pattern in languages with support for first-class functions and lexical scoping rules (such as E [MMF00] or Scheme [ADH+98]).

4.2.4 Traits

Next to object delegation, AmbientTalk also has support for composing objects explicitly using traits. Traits [SDNB03] are composable units of object behaviour[3]. They form a general reuse mechanism that subsumes other styles of object composition such as class-based (multiple) inheritance, mixins [BC90] and object-based delegation. In its pure form, a trait is a module that both *provides* a set of methods and *requires* a set of methods. The trait can be *used* by a so-called *composite* class or object. The operational effect is that the trait's provided methods are added to the composite, as if they were copy-pasted. The composite must ensure that all of the trait's required methods are provided.

Trait-based composition is complementary to delegation-based composition. The advantage of traits over object-based delegation is that they enable an object to reuse code from *multiple* sources, not just from a single parent object. Trait composition enables possible name clashes to be resolved using an expressive "little language" that allows for the exclusion or aliasing of names in the composite. This avoids the accidental overriding of methods to which delegation-based composition is prone. Furthermore, if it is checked whether the composite indeed provides all of the required methods of the imported trait, it can be verified whether the composite is complete, avoiding latent "method not found" errors. The advantage of delegation-based over trait-based composition is flexibility: while a delegate is easily changed at runtime (dynamic inheritance), traits are composed once at composition time into a single composite object.

AmbientTalk introduces trait composition by means of explicit delegation. Consider the following definition of an `Enumerable` trait, which specifies reusable behaviour on enumerable collections of objects:

```
def Enumerable := object: {
  // map a closure over the collection
  def collect: clo {
    def result := [];
    self.each: { |e| result := result + [clo(e)] };
    result
  };
  // return an element for which pred holds
  def detect: pred { ... };
  // return elements for which pred does not hold
  def reject: pred { ... };
  ...
}
```

Even though the body of `detect:` and `reject:` is not shown for brevity, what all of these methods have in common is that they can be implemented for arbitrary collections, as long as the collection implements the method `each:`. Hence, `Enumerable` is a trait providing `collect:`, `detect:` and `reject:` and requiring `each:`. The trait's required methods are not specified explicitly, and AmbientTalk will not enforce their implementation by a composite[4]. The above trait can be used by e.g. a `Range` abstraction representing intervals of the form `[start,end[` as follows:

[3] Note that traits as described here are not to be confused with the traits of the Self programming language [UCCH91].

[4] Library extensions exist which allow developers to make required methods explicit and to check whether a composite using the trait implements them.

```
def Range := object: {
  import Enumerable;
  def start := 0;
  def end := 0;
  def init(s,e) { start := s; end := e };
  def each: clo {
    start.to: end do: clo
  };
};
Range.new(0,10).reject: { i (i%3 != 0) }
// result: [0,3,6,9]
```

By importing the `Enumerable` trait, `Range` implicitly also provides all methods of the trait. It implements `each:` by applying a closure to all consecutive integers in that interval[5]. Note how trait composition is implemented by means of the **`import`** statement. The example code shows how `reject:` can be invoked as a method of `Range`.

When the **`import`** statement is executed at runtime, an exception is raised if a name clash occurs between the importing and the imported object. This requires the programmer to resolve name clashes at object composition time explicitly. To achieve this, the **`import`** statement provides **`alias`** and **`exclude`** clauses that allow the programmer to locally rename a method provided by a trait or to exclude the import of certain methods.

The operational effect of an **`import`** statement is best understood in terms of explicit delegation. The **`import`** statement is implemented by defining local *delegate* methods in the composite object for each method provided by the trait. The task of these methods is to delegate a message from the composite to the trait object from which they were imported. For example, the trait composition in `Range` can be rewritten without **`import`** as follows:

```
def Range := object: {
  def collect: clo { Enumerable^collect: clo };
  def detect: pred { Enumerable^detect: pred };
  def reject: pred { Enumerable^reject: pred };
  def start := 0;
  ... // as above
};
```

The use of delegation rather than message forwarding is crucial here: within the trait's methods, calls to required methods are expressed as self-sends to the composite object using the trait. Due to AmbientTalk's support for explicit delegation, `Range` can delegate requests to `Enumerable` even though that object is not its parent object. `Range` is free to inherit from an object other than `Enumerable`, and can also import more than one trait. Moreover, a single trait object can be imported by multiple composite objects. This makes traits a highly flexible yet robust composition mechanism.

One important difference between AmbientTalk traits and the original proposal of Schärli et al. [SDNB03] is that trait import in AmbientTalk cannot be thought of as "copy-pasting" the trait methods into the composite. Rather, the methods provided by a trait are shared (via delegation) by all objects using the trait. By not copying

[5]The primitive `to:do:` method defined on AmbientTalk integers applies a closure to each integer value between the receiver and the first argument.

the method implementations into composite objects, it can be ensured that lexically free variables within a trait's method remain bound in the correct scope, which is the lexical scope of the trait and not that of the composite.

In the original proposal of Schärli et al. traits are described as stateless entities (i.e. as a collection of methods only). AmbientTalk allows arbitrary objects to act as traits. If such objects define state via fields, these fields are copied into any composite importing such objects. This ensures that clones of the composite object each host their own copy of any state. This semantics does require a trait to be careful in the way it addresses its declared fields. To modify or access a field of the composite, a trait should use an explicit self-send to refer to the field. Otherwise, it refers to the lexically visible field of the trait itself (cf. section 4.2.3).

4.2.5 Type Tags

AmbientTalk is both a dynamically typed as well as a classless, prototype-based language. This introduces the problem that, in AmbientTalk, objects cannot be easily *classified*. In statically typed languages, the static type of the variable holding an object is often used for these purposes. In class-based languages, the class naturally plays the role of classifier. Object classification is useful for a diverse number of reasons. For example, in an exception handler, it is often useful to specify the type of objects that the handler can catch. Section 4.4.2 showcases the use of classification for the purposes of service discovery.

To recover the ability of classification, AmbientTalk introduces *type tags*. A type tag is identified by name (i.e. it is a nominal type) and it can be a subtype of zero or more other type tags. Objects, in turn, can be tagged with zero or more type tags. Type tags are not associated with a set of methods and are not used for static type checking. They are perhaps best compared with empty Java interfaces, like the typical "marker" interfaces used in Java libraries to merely tag objects (prominent examples are `java.io.Serializable` and `java.lang.Cloneable`). The following code illustrates the use of type tags:

```
deftype IndexableT;
deftype EnumerableT;
deftype OrderedT;
deftype SortableT <: EnumerableT, OrderedT;

def Array := object: {
  ...
} taggedAs: [ IndexableT, SortableT ];
```

Objects can only be tagged with type tags when they are created (via **`object: taggedAs:`**), and their set of type tags remains constant. The rationale behind this design decision will become clear when considering objects partitioned across multiple actors, explained in section 4.3.

A primitive function allows the programmer to perform a type test on objects, e.g. **`is:`** `Array` **`taggedAs:`** `Enumerable`. The type test determines whether an object *or one of its parents* is tagged with the given type tag *or a subtype* of the type tag. This is very reminiscent of the behaviour of the `instanceof` operator of Java.

Type tags are first-class objects. Thus, they can be parameter-passed as arguments, bound to variables, etc. However, type tags do not follow standard object identity semantics. Type tag equality is by their name rather than by their object identity.

4.2.6 Summary

Throughout this section, we have briefly touched upon those parts of the AmbientTalk language that are required to understand the technical contributions described in later chapters. A full account on all of AmbientTalk's language features is outside the scope of this chapter. However, the most important language features omitted from the above discussion – AmbientTalk's support for metaprogramming and reflection and its interoperability with the Java Virtual Machine – will be extensively discussed in the following chapter. A comprehensive introduction to all aspects of the AmbientTalk language can be found online [DGM+07]. The following sections shift the focus of attention to the concurrent (respectively distributed) features of the language.

4.3 AmbientTalk: a Concurrent Language

In AmbientTalk, concurrency is spawned by creating actors: one AmbientTalk virtual machine may host multiple actors which execute concurrently. AmbientTalk's concurrency model is based on the communicating event loops model of the E language [MTS05], which is itself an adaptation of the well-known actor model [Agh86]. E combines actors and objects into a unified concurrency model. Unlike previous actor languages such as Act1 [Lie87], ABCL [YBS86] and Actalk [Bri88], actors are not represented as "active objects", but rather as *vats* (containers) of regular objects, shielding them from harmful concurrent modifications. AmbientTalk actors are like E vats. Within the confines of a vat, computation happens sequentially. Incoming messages from objects living in other vats are processed in a serial manner in order to ensure that no race conditions can occur on the internal state of the objects within the vat.

4.3.1 Event Loop Concurrency

AmbientTalk's concurrency model is based on communicating event loops [MTS05]. This is an event-driven concurrency model that differs in many respects from traditional multithreaded concurrency. In this model, an actor is represented as an event loop. An *event loop* is a thread of execution that perpetually processes *events* from its *event queue* by invoking a corresponding *event handler*. Communicating event loops enforce three fundamental concurrency control properties:

Property 1 (Serial Execution) *An event loop processes incoming events from its event queue one by one, i.e. in a strictly serial order.*

As a consequence of serial execution, the handling of a single event happens in mutual exclusion with respect to other events. Hence, race conditions on an event handler's state caused by concurrent processing of events cannot occur.

Property 2 (Non-blocking Communication) *An event loop* never *suspends its execution to wait for another event loop to finish a computation. Rather, all communication between event loops occurs strictly by means of asynchronous event notifications.*

As a consequence of non-blocking communication, event loops can never deadlock one another. However, in order to guarantee progress, an event handler should not execute e.g. infinite `while` loops. Rather, long-running actions should be performed piecemeal by scheduling events recursively, such that an event loop always gets the

chance to respond to other incoming events. The only situation where an event loop can be suspended is when its event queue is empty.

Property 3 (Exclusive State Access) *Event handlers and their associated state belong to a single event loop. In other words, an event loop has* exclusive *access to its mutable state.*

As a consequence, two or more event loops never share synchronously accessible mutable state. Because event handlers are not shared between event loops, they never have to lock mutable state. Mutating another event loop's state has to be performed indirectly, by asking the event loop to mutate its own state via an event notification.

Event loop concurrency avoids deadlocks and certain race conditions *by design*. The nondeterminism of the system is confined to the order in which events are processed. In standard preemptive thread-based systems, the nondeterminism is more substantial because threads may be pre-empted upon each single instruction. Even though deadlock cannot occur, event loops cannot guarantee progress. For example, an event handler may simply never be invoked, stopping the progress of the application.

In the following section, we describe how the abstract event loop model is incorporated into the AmbientTalk language.

4.3.2 AmbientTalk Actors

As mentioned in the introduction, AmbientTalk actors are not represented as active objects, but rather as event loops: the event queue is represented by an actor's message queue, events are represented as messages, event notifications as asynchronous message sends, and event handlers are represented as (the methods of) regular objects. The actor's event loop thread perpetually takes a message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message. Messages are processed serially to avoid race conditions on the state of regular objects.

Each AmbientTalk object is said to be *owned* by exactly one actor. This ownership relation is established upon object creation and cannot change during the lifetime of the object. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor may communicate using standard, sequential message passing or using asynchronous message passing. AmbientTalk borrows from E the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). It is possible for objects owned by an actor to refer to objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [MTS05]) and only allow asynchronous access to the referenced object. Synchronous access to an object via a far reference raises a runtime exception. Any messages sent via a far reference to an object are enqueued in the message queue of the actor owning the object and processed by the owner itself.

Figure 4.1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent an actor's event loop which perpetually takes messages from its message queue and synchronously executes the corresponding methods on its owned objects. The control flow of an actor's event loop never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

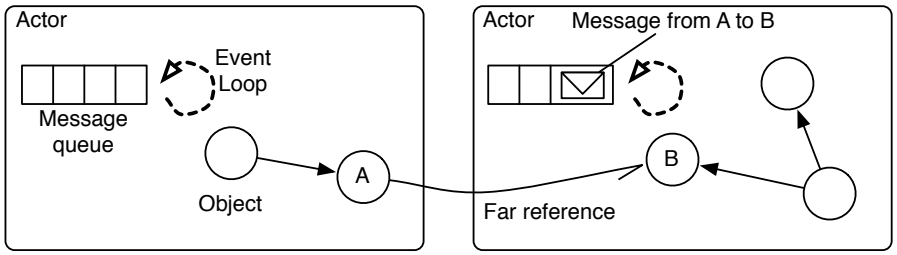

Figure 4.1: AmbientTalk actors as communicating event loops.

A far reference encapsulates a copy of the set of type tags with which its target object is tagged. This implies that a type test can be performed locally on a far reference to an object. As will be discussed in section 4.4, far references may refer to remote objects, which explains why the set of type tags of an object must remain constant: the far reference maintains a copy of that set on a potentially remote device. If the set could change, it would have to be kept in synchronisation across a volatile network, the effects of which are impossible to hide from the programmer.

Every AmbientTalk interpreter starts its execution with a single actor. An actor can spawn new actors by invoking **`actor:`** `{ ... }`. When a new, empty, actor is created it evaluates the code passed to the **`actor:`** primitive to construct the first object it will own. The return value of the **`actor:`** primitive is a far reference to this object, thus allowing the creating actor to communicate with the new object owned by the created actor.

4.3.3 Message Passing Semantics

In AmbientTalk, asynchronous messages can be sent between objects owned by the same or by different actors. In the case where both sender and receiver are owned by the same actor, the message is simply added to the owner's message queue and parameters are passed *by reference*, exactly as is the case with synchronous message sending. For inter-actor message sends, where an object sends an asynchronous message via a far reference to an object owned by another actor, objects are parameter-passed *by far reference*: the parameter of the invoked method is bound to a far reference to the object. In either case, messages are guaranteed to be delivered to an object in the same order as they were sent. Consider the following example, assuming that the code on the left-hand and the right-hand side is executed in two different actors:

```
def arg := object: { ... };
obj<-m(arg);
```

```
def obj := object: {
  def m(par) { ... }
};
```

In the method body of `m`, `par` will be bound to a far reference to `arg`. In some cases, the remotely invoked method may want to access its argument synchronously. To this end, AmbientTalk introduces the notion of an *isolate* object. Isolates are objects that are passed *by copy* over a far reference. This allows the recipient actor to operate on the copy synchronously, without additional inter-actor communication and

without violating the exclusive state access property. When an isolate is copied during parameter-passing, all objects it directly refers to are recursively parameter-passed (according to their own semantics). The following code provides an example of an isolate:

```
def ComplexNumber := isolate: {
  def re := 0;
  def im := 0;
  def init(r,i) { re := r; im := i; };
  ...
}
```

Isolates are regular objects, with two notable differences. First, as already remarked, they are passed by copy across far references. Second, isolates cannot use any free lexically visible names. Isolates are thus completely isolated pieces of code, without any implicit dependencies on the surrounding scope – hence their name. Isolates are best thought of as if they were lexically defined at top-level. AmbientTalk does provide a mechanism for isolates to explicitly *copy* a lexically visible variable into the object scope of the isolate, allowing a nested isolate to be parameterised explicitly with lexically visible variables.

Isolates are disallowed access to their lexical scope because they are copied during parameter passing. The restriction avoids having to *implicitly* copy any lexically visible variables referred to by the isolate. An alternative to this restriction would be a form of distributed lexical scoping, as done in Obliq [Car95]. However, in the case of AmbientTalk, the distributed name lookup would somehow have to be done asynchronously (in order not to violate the non-blocking communication property of both AmOP, cf. section 2.3.2, and Event Loop concurrency). Reconciling these is a topic of future research.

Because isolates are copied during parameter-passing, two further issues associated with them are identity and mutable state. First, each copy of an isolate has its own object identity. However, it is considered good programming practice to redefine object equality for isolates in terms of intrinsic properties such as their state. Second, each copy of an isolate has its own state. Again, it is considered good programming practice to treat isolates as "constant" objects, moving all mutable state into a separate non-isolate object that can be shared by all copies.

We conclude the discussion of AmbientTalk's message passing semantics with a note on message ordering. Successive asynchronous messages sent by one object to the same receiver object are delivered in FIFO order. However, it should be kept in mind that an arbitrary number of other asynchronous messages (sent by other objects) may be processed by the receiver in between handling these successive messages.

4.3.4 Futures

Up to this point, we have not yet discussed how to deal with return values from asynchronous message sends. By default, an asynchronous message send does not return a meaningful value (i.e. it returns `nil`). However, this can make the processing of return values quite cumbersome, as illustrated by the following example:

```
calculator<-add(x,y, object: {
  def reply(n) {
    system.println("result: "+n)
  }
})
```

```
def calculator := object: {
  def add(a,b,customer) {
    customer<-reply(a+b);
  }
}
```

In the above code, the anonymous object passed as the third argument to `add` serves as a "continuation" of the asynchronous send. The `reply` method serves as a "callback". However, the code is cumbersome to read because the general idea of a "return value" remains implicit. Moreover, it is a pattern that would need to be repeated each time a return value is required.

In order to reconcile return values with asynchronous message sends, AmbientTalk employs the notion of a *future* (also known as a *promise*). This is a frequently recurring language abstraction in concurrent programming languages (cf. section 4.9). In AmbientTalk, a future is a placeholder for the return value of an asynchronous message send. Once the return value is computed, it "replaces" the future object; the future is said to be *resolved* with the value. Using futures, the example above can be rewritten as:

```
def sumFuture := calculator<-add(x,y);
```

This change at least makes the handling of return values syntactically identical to that of the familiar synchronous message send. However, we have yet to explain how objects can synchronise on the actual value represented by the future. There are two styles for doing so, each of which is discussed below.

4.3.4.1 Control-Flow synchronisation

In many programming languages, futures act as synchronisation barriers while they are unresolved: if code tries to access the future's value before the future is resolved, the thread of control is suspended until the value has been computed. This principle is known as *wait by necessity* [Car93]. In a language with communicating event loops, however, such a semantics would ruin the non-blocking communication property (cf. section 4.3.1). It would imply that an event loop can suspend in a state other than when its event queue is empty. Hence, the event loop would become unresponsive to other events, and the entire system becomes prone to deadlock once again.

AmbientTalk avoids the wait by necessity semantics and instead employs the semantics first introduced by promises in E [MTS05]. An actor cannot suspend on an unresolved future. What it can do is to register its interest in the resolved value of the future by registering an observer – a closure, to be precise – that will be invoked later, when the future has become resolved. The following code illustrates how this observation mechanism can be used to print the sum from the previous example:

```
def sumFuture := calculator<-add(x,y);
when: sumFuture becomes: { |sum|
  system.println("result: " + sum);
}
```

The **when:becomes:** function takes a future and a block closure as its arguments, and registers the closure as an observer on the future. If the future is resolved to a

proper value[6], the closure is applied to the resolved value. Multiple observer closures may be registered on the same future. Note that the **when:becomes:** function itself returns immediately. The code specified in the block closure is always delayed, i.e. it is executed after the code following the call to **when:becomes:**, even if `sumFuture` is already resolved at the time the observer is registered. It is also guaranteed to be executed by the same actor that performed the asynchronous message send. Hence, the execution of the observer is always serialised with respect to other activities within the same actor (cf. section 4.4.1 for an explanation on how this serialisation is achieved).

The block closure passed to **when:becomes:** acts as an in-line event handler. It effectively enables the sender of an asynchronous message to synchronise on and process the result of that message in the scope where it was sent. Because block closures close over their lexical scope, all variables in scope at the time the message was sent are still available when handling the return value at a later point in time. The programmer does have to be aware of the fact that the *values* of the variables in scope may have changed since the time the message was sent, as other code may have run within the actor while the code was delayed.

A final aspect of synchronising on a future is dealing with exceptions. AmbientTalk features a standard exception model that allows objects to be raised and caught as exceptions. When an exception is raised in an asynchronously invoked method, the exception propagates up to the level of the asynchronous invocation. At this point, the only available continuation is the future attached to the message. In order to signal the exception to the sender, the future is *ruined* with the exception. When a future is ruined, regular observers on the future are not triggered. However, a separate exception handler can be specified as follows:

```
def quotientFuture := calculator<-divide(x,y);
when: quotientFuture becomes: { |quotient|
  system.println("result: " + quotient);
} catch: DivisionByZero using: { |e|
  system.println("error: divided "+ x +" by zero");
}
```

In AmbientTalk, exception types are modelled using type tags (cf. section 4.2.5). Hence, it is assumed that `DivisionByZero` is a type tag. Should `quotientFuture` become ruined with (a subtype of) `DivisionByZero`, the second block closure is applied to the exception. This enables the handling of asynchronously raised exceptions much in the same way as the well-known **try–catch** construct is used for regular exception handling.

The return value of a call to **when:becomes:catch:using:** is itself a future. The future is resolved with the return value of an observer block closure, or ruined if an exception occurs during the execution of that closure. In effect, the future that is returned by **when:becomes:** is dependent on the future on which it operates: resolving or ruining the latter eventually leads to resolving or ruining the former.

4.3.4.2 Data-Flow synchronisation

From the discussion in the previous section, it appears that the programmer can only perform one operation on a future, which is to wait for its resolved value by means of **when:becomes:**. However, futures also specify a meaningful semantics for message passing. It is always legal to send an asynchronous message to a future, regardless of

[6] A *proper value* is any value that is not itself a future, cf. the following section.

the fact whether the future is resolved or not. If the future is unresolved, the message is simply stored in a private message queue by the future. If the future is resolved, the message is forwarded to the resolved value instead, as if the sender had sent it to the actual value in the first place. At the moment the future becomes resolved, all previously accumulated asynchronous messages are flushed and forwarded to the resolved value.

An asynchronous message sent via a future `f1` may itself have an attached future `f2`. This leads to a situation similar to the one described in the previous section: the resolution of `f2` becomes dependent on the resolution of `f1`. We refer to this phenomenon as *future chaining*. The constructed chains of futures effectively encode a data-flow graph. The following code illustrates the general idea:

```
def sumFuture := calculator<-add(x,y);
def sqrtFuture := sumFuture<-sqrt();
when: sqrtFuture becomes: { |val|
  system.println("sqrt(x+y) = " + val);
} catch: Exception using: { |e|
  system.println("calculation error: "+e);
}
```

When `sumFuture` becomes resolved, the `sqrt` message is forwarded to its resolved value, in turn leading to the resolution of `sqrtFuture`. Should `sumFuture` be ruined with an exception, the futures of all messages sent to the future, like `sqrt` are likewise ruined with the same exception. Exceptions can thus propagate across a chain of futures.

Another form of chaining occurs when a future is itself resolved with another future. The following example demonstrates this:

```
when: a<-m() becomes: { |v|
  doSomethingWith(v);
}
```

```
def a := object: {
  def m() { b<-n() };
}
```

The future $f_{a \leftarrow m}$ attached to the message `a<-m()` is resolved with the return value of the method `m`, which is the future $f_{b \leftarrow n}$ attached to the message `b<-n()`. When $f_{a \leftarrow m}$ is resolved with $f_{b \leftarrow n}$, $f_{a \leftarrow m}$ is not considered resolved yet, so it will not yet forward accumulated messages, nor will it notify its registered **`when:becomes:`** observer. However, $f_{a \leftarrow m}$ will register *itself* as an observer on $f_{b \leftarrow n}$. When $f_{b \leftarrow n}$ finally becomes resolved with a proper value, it notifies $f_{a \leftarrow m}$ which in turn resolves itself with the same value.

It is not allowed to synchronously send a message to a future, even if the future is resolved. Conceptually, a future should be thought of as a special kind of far reference. As it is not known beforehand whether a future will resolve to a local or remote object, there is always the possibility that it will become a far reference. Therefore, a future is itself treated consistently as if it were a far reference. Adopting the terminology from E [Mil06], we will reserve the term *far reference* to denote a remote object reference to a concrete object and use the term *eventual reference* to denote the more general category of all kinds of object references which enforce asynchronous access to their designated object(s), which includes far references and futures.

4.3.4.3 Conditional synchronisation

Futures provide actors with a way of synchronising upon the return value of asynchronous message sends. However, there may be other synchronisation points in an application. For example, depending on the state of an object, it may disable or postpone replies of certain methods. This kind of synchronisation is known as conditional or behavioural synchronisation [BGL98]. The archetypical example is that of a bounded buffer whose `get` method must synchronise on an empty buffer and whose `put` method must synchronise on a full buffer.

AmbientTalk adopts the solution to conditional synchronisation proposed in E once more, by enabling the programmer to explicitly create futures to postpone replies. We illustrate this by means of a concrete example. Consider a new abstraction named `group:` that allows code to synchronise on multiple futures simultaneously. For example, an object may want to react only when the return values of *two* asynchronous sends have been computed:

```
when: (group: [ a<-m(), b<-n() ]) becomes: { |values|
  def [result1, result2] := values;
  ...
}
```

The call to `group:` takes as its argument a table of futures and returns a future itself. The returned future is resolved with a table of values, if and only if all of the argument futures are resolved. Its definition requires the use of explicit conditional synchronisation:

```
def group: futures {
  def [groupFuture, groupResolver] := makeFuture();
  def resolutions[futures.length] { nil };
  def numResolved := 0; // counts the number of resolved futures
  // apply a closure over all futures, passing index and value at index
  futures.eachWithIndex: { |i, future|
    when: future becomes: { |val|
      resolutions[i] := val;
      numResolved := numResolved + 1;
      if: (numResolved == futures.length) then: {
        // the last argument future has been resolved
        groupResolver.resolve(resolutions);
      };
    } catch: Exception using: { |e|
      groupResolver.ruin(e); // only the first exception is propagated!
    };
  };
  groupFuture;
};
```

The `makeFuture` function returns two values: a new, unresolved future and an object called a *resolver* that can be used to explicitly resolve or ruin the future. Note that the explicitly created `groupFuture` becomes the return value of the `group:` function. Objects can synchronise on this future in the usual way, via the registration of observers. However, unlike the futures from the previous sections, this future is not automatically resolved by the language, as it is not implicitly associated with an asynchronous message. Rather, it must be resolved (respectively ruined) explicitly, by invoking the `resolve` (respectively `ruin`) method of its corresponding resolver.

The `group:` function registers an observer on each of its argument futures. It counts the number of successfully resolved futures, and accumulates their values in the `resolutions` table. Note that each resolved value is stored at the same index of the original future, ensuring that the order of the resolutions matches that of the futures. The `groupFuture` is resolved explicitly only when all of the argument futures have been resolved, resulting in the desired synchronisation. If any of the asynchronous sends results in an exception, the first exception to trigger is propagated to the composite future.

4.3.4.4 Reflective Implementation

Futures are not part of AmbientTalk's kernel. By default, an asynchronous send returns `nil`. Futures are entirely implemented in AmbientTalk itself by means of *computational reflection* [Mae87]. We discuss their reflective implementation in section 5.2.3.1 as a concrete example of using AmbientTalk's metalevel architecture, which is the topic of the next chapter.

4.4 AmbientTalk: a Distributed Language

In this section, we proceed from concurrent to distributed programming. Actors can be distributed across a network, each hosted by different AmbientTalk interpreters. The major difference between single-machine and distributed programming is the possibility of partial failures, the phenomenon whereby remote objects may not respond to messages, due to either a network or a machine failure. The second important issue is service discovery, the goal of which is to acquire a first reference to a remote object.

Before continuing, a brief word on terminology. We have previously mentioned that an actor is said to *own* one or more objects. Likewise, an AmbientTalk interpreter is said to *host* one or more actors. Because AmbientTalk is currently implemented in Java (cf. section 4.8), a single Java Virtual Machine can be said to host one or more AmbientTalk interpreters. Two objects are said to be *local* when they are owned by the same actor. Objects are considered *remote* when they are owned by different actors, even if those actors are hosted by the same interpreter. Within one interpreter there is no notion of partial failure: connections between actors within a single interpreter never fail. Hence, interpreters are the unit of partial failure. Within one JVM, there is no notion of "crashes": either all interpreters within a single JVM are terminated, or none of them are. Hence, JVMs are the unit of termination. Table 4.1 summarises AmbientTalk's units of operation.

Object	Unit of designation
Actor	Unit of concurrency
Interpreter	Unit of partial failure
Java Virtual Machine	Unit of termination

Table 4.1: Overview of AmbientTalk's units of operation.

4.4.1 Far References and Partial Failures

Recall from section 4.3.2 that references between objects owned by different actors are always far references which only permit asynchronous access to their target object. Because objects residing on different devices (i.e. in distinct interpreters) are necessarily owned by different actors, far references are the only kind of remote object references in AmbientTalk. This ensures by design that all distributed communication is asynchronous, as required by the non-blocking communication characteristic of the AmOP paradigm (cf. section 2.3).

By allowing far references to cross machine boundaries, we must specify their semantics in the face of partial failures. AmbientTalk's far references are by default resilient to failures (including network partitions). When a (network or machine) failure occurs, a far reference becomes disconnected. A disconnected far reference buffers all messages sent to it. When the failure is restored at a later point in time (e.g. a network partition is healed), the far reference flushes all accumulated messages to the remote object in the same order as they were originally sent. Hence, messages sent to far references are never lost, regardless of the internal connection state of the reference. Making far references resilient to failures by default is one of the key design decisions that make AmbientTalk's distribution model suitable for mobile ad hoc networks, because temporary network failures have no immediate impact on the application's control flow. This behaviour is desirable in mobile networks because it can be expected that many partial failures are the result of temporary network partitions. However, perhaps a machine has crashed beyond recovery, or it has moved out of the wireless communication range and does not return. Such persistent failures also need to be dealt with. We postpone this discussion until section 4.4.4.

Even though it is often desirable not to disrupt an application's control flow upon a partial failure (because one wants to optimistically assume the failure will only be transient), it is nevertheless useful to be able to act upon the failure as an event. This allows applications to monitor their connectivity with remote objects, and potentially even fork a failure handling activity to handle a disconnection if it persists for a certain period of time. To enable such failure handling AmbientTalk introduces two *failure event handlers*, both showcased in the following example. In the example, the GUI of an instant messenger application uses these event handlers to indicate whether a buddy in the user's buddy list is online or not.

```
// buddy is a far reference to an instant messaging peer
whenever: buddy disconnected: {
  gui.markOffline(buddy);
};
whenever: buddy reconnected: {
  gui.markOnline(buddy);
};
```

The event handlers are registered by applying primitive functions that both take a far reference and a nullary closure as their arguments. The closure is applied *whenever* the interpreter detects the disconnection (respectively reconnection) of the object referred to by the far reference. Some precaution is needed here, as it may seem that race conditions could occur if these closures are allowed to execute concurrently with other activities in the same actor. However, AmbientTalk elegantly avoids this problem. Recall from section 4.2.2 that a closure is an object with an `apply` method. When the interpreter wants to invoke the closure, it simply evaluates `closure<-apply([])`,

thereby ensuring that the closure runs serially w.r.t. other activities in the actor owning the closure. We will refer to this asynchronous application of closures as *triggering* a closure (as opposed to *applying* a closure).

The return value of both failure event handlers is a *subscription* object with a single method, named `cancel`. When invoked, it cancels the registration of the event handler with the interpreter, such that the closure will no longer be triggered. However, previously triggered asynchronous applications may still be pending in the actor's message queue. These are not automatically retracted. Hence, the closure may still be applied even after `cancel()` was invoked.

4.4.2 Exporting Objects

Objects can acquire far references to objects by means of parameter-passing or return values from inter-actor message sends. However, it remains to be explained how objects can acquire an *initial* far reference to an object owned by a remote actor. In order to make it possible for an object to be discovered by remote actors, the object must be explicitly *exported*.

An object always has to be exported with a corresponding type tag. Here, the type tag is used to classify what kind of service the object provides. It plays a role similar to the *topic* in traditional publish/subscribe architectures [EFGK03]. The subtyping mechanism of type tags allows objects to be published in a hierarchical classification, a feature also supported by most publish/subscribe architectures. We do assume that all distributed peers attribute the same semantics to the names of type tags, and define the same type hierarchy. The following example shows how to export an object as an instant messenger chat peer:

```
deftype InstantMessenger;
def peer := object: {
  def chat(textMessage) { ... };
};
export: peer as: InstantMessenger;
```

From the moment an object is exported, it is discoverable by objects owned by other actors by means of its associated type tag. The **`export:as:`** function returns a publication object `pub` which can be used to "unexport" the object by invoking `pub.cancel()`. An unexported object can no longer be discovered by remote objects. However, far references referring to the unexported object remain valid, so an unexported object can still be remotely referred to.

In the following section, it is explained how remote objects can acquire a reference to an exported object.

4.4.3 Service Discovery

The AmbientTalk interpreter has a built-in service discovery algorithm that enables the discovery of remote objects in a peer-to-peer manner. The algorithm is fully decentralised, no servers or other infrastructure are required. Objects that want to be notified by the interpreter of available remote objects do so via a discovery event handler:

```
def subscription := whenever: InstantMessenger discovered: { |messenger|
  buddyList.add(messenger);
};
```

A discovery event handler is registered by calling the **whenever:discovered:** function which takes as arguments a type tag and a unary closure. Continuing our analogy with publish/subscribe systems, the closure plays the role of *subscriber* [EFGK03]. Whenever an actor is encountered in the network that exports a matching object, the closure is triggered on (i.e. asynchronously applied to) a far reference to the discovered object. Hence, the parameter of the block closure is the initial far reference to a remote object, from which other far references may be derived by message passing.

An object matches a discovery request if its exported type tag is a subtype of the type tag argument to the discovery event handler. This implies that service discovery is polymorphic: a discovery request for a `Printer` may be satisfied by a `ColorPrinter` object, provided that the `ColorPrinter` tag is a subtype of `Printer`. A discovery request only triggers on objects owned by other actors; an actor does not discover its own exported objects. It is possible for **whenever:discovered:** to trigger on the same remote object multiple times (e.g. when a temporary network partition is healed). The programmer should thus take this possibility into account.

Analogous to the return value of the failure event handlers (cf. section 4.4.1), the discovery event handler returns a subscription object whose `cancel()` method cancels the registration of the closure. There exists a variant event handler, accessible as a function named **when:discovered:** which only applies the closure to the first matching discovered object, and afterwards automatically cancels its registration.

Note that contrary to what the above examples may seem to indicate, the type tag argument to functions like **whenever:discovered:** and **export:as:** can be any expression. Because type tags are first-class objects, it is straightforward for code to abstract from the specific type tag to use. The type tag to use does not necessarily need to be a literal in the source text. This property enables the construction of more high-level abstractions on top of the primitive discovery event handlers provided by AmbientTalk. For example, we may define a function **whenAll:discovered:** that allows a programmer to express the following:

```
whenAll: [Camera, Laptop, GPS] discovered: { |services|
  def [camera, laptop, gps] := services;
  def pictureFut := camera<-takePicture();
  def taggedPictureFut := when: gps<-readCoordinates() becomes: {|coords|
    pictureFut<-geotag(coords);
  };
  laptop<-upload(taggedPictureFut);
}
```

The above example is based on the GeoTagger example of Svensson et al. [SHM07]. The code snippet is meant to be run on a handheld device in close proximity to a photo camera, a laptop and a GPS receiver. When all devices have been discovered, the camera is asked to take a picture which is subsequently geotagged with the GPS coordinates received from the GPS device. The geotagged picture is then uploaded to the laptop. Note how the above example maximally exploits all forms of chaining permitted by futures (cf. section 4.3.4). The new **whenAll:discovered:** abstraction is implemented as follows:

```
def whenAll: typetags discovered: handler {
  def compositeFuture := group: (typetags.map: { |type|
    def [discoveryFuture, resolver] := makeFuture();
    when: type discovered: { |service|
      resolver.resolve(service);
```

```
    };
    discoveryFuture
  });
  when: compositeFuture becomes: handler;
}
```

The implementation reuses the `group:` abstraction defined in section 4.3.4.3 to await the resolution of a table of futures. The table of futures is generated by spawning a discovery request for each type tag in the `typetags` table. Note that `type` is a variable, not a literal.

4.4.4 Partial Failures Revisited

In section 4.4.1 it was described how far references enable the programmer to optimistically abstract from transient failures by default, by buffering messages within the reference until the connection is restored. Despite the usefulness of this built-in failure handling strategy, it only works for transient failures. Unfortunately, it is a given that in distributed computing, one cannot distinguish network failures from device failures [WWWK96] and even if it is a network failure, one cannot distinguish transient from permanent failures ahead of time [Wal01]. The best the programmer can do is to choose a timeout period that will treat the failure as "permanent" when it persists for longer than that period.

AmbientTalk allows the expression of such timeouts either at the level of eventual references or at the level of individual asynchronous messages. Timeouts at the level of eventual references are part of a leasing strategy, which we will extensively discuss in section 4.6.4. Here, we focus on timeouts associated with individual asynchronous messages. If the future associated with a message annotated with a timeout is not resolved within the timeout period, the future is automatically ruined by the system with a `TimeoutException`. The following code shows how a timeout can be specified and dealt with:

```
when: buddy<-chat(aTextMessage)@Due(minutes(1)) becomes: { |ack|
  // message received successfully
} catch: TimeoutException using: { |e|
  // message timed out
}
```

In the above example, if the `chat` message is not replied to within 1 minute, the exception handler is triggered (i.e. asynchronously applied). The `@` syntax allows a message to be annotated with one or more type tags. The `Due` tag can be used to associate a timeout period with a message.

When performing failure handling, one should always be aware of the fact that the message *may* still have been received by the remote party. It may even be possible that the receiver sends a reply after the timeout period has already expired. In this case, the future will silently ignore the return value, as it has been previously ruined with a `TimeoutException`.

4.5 Discussion

In this section, we take a step back from the technicalities of AmbientTalk and argue why it can be called an event-driven object-oriented language for mobile ad hoc

networks.

4.5.1 Event-driven Object-oriented Programming

The event-driven concurrency model employed by AmbientTalk has the advantage that it maps well onto the inherently event-driven nature of distributed systems. Devices may join or leave the network and messages can be received from remote objects at any point in time. In contrast to multithreaded approaches, event loops are able to restrict non-determinism to the order in which events are received.

One disadvantage that is often attributed to event-driven concurrency models is that it causes an *inversion of control*, i.e. the programmer must explicitly partition the application into separate event handlers (e.g. callbacks) [HO06]. The root cause of the inversion of control problem is that event-driven programming is actually *programming without a call stack* [Hoh06]. Event handlers are usually triggered from a source external to the application, with no application-specific stack frames on the runtime stack[7]. In a short article, Hohpe describes the consequences of programming without a call stack. He notes that a call stack provides three implicit properties [Hoh06]:

Continuation The call stack implicitly encodes a "return address": the callee knows where to continue the execution after it completes.

Coordination A call stack supports an implicit form of synchronisation between caller and callee: the calling method waits for called method to complete.

Context Local variables are saved on the call stack[8]: when a callee returns, the execution context of its caller is restored. This "restoration" process only works for local variables. Instance variables of an object, for example, may have been changed by the callee.

With the implicit properties of a call stack now made explicit, the apparent problems related to event-driven programming also become explicit: all three of the above properties must be encoded manually on top of a pure event system! We claim that AmbientTalk mitigates the effects of the "inversion of control" because it reintroduces the three key features of a call stack in an object-oriented manner:

Continuation The implicit return address encoded by the call stack is made explicit in AmbientTalk as a future. A future is an explicit return address, it provides a first-class handle to which an asynchronously invoked callee can return the value of the asynchronous invocation.

Coordination The synchronisation implied by a synchronous method or function call is replaced by the ability to register *observers* in the form of *closures* on futures. These observers are in-line event handlers that enable the code to "wait for" the reply to arrive *without* blocking the underlying thread of control.

Context In AmbientTalk, the context of an outstanding asynchronous call can be captured by the lexical scope of a closure registered as an observer on the call's

[7] This phenomenon is easily observable in e.g. the Java Swing or AWT GUI libraries. A stack trace within an event notification method of a "listener" object will reveal that the only call frames on the stack pertain to the GUI's event loop framework. Thus, there is no implicit application context for the method to exploit.

[8] In languages that support closures, care must be taken with variables captured by a closure as they may out-live the lifetime of their stack frame.

associated future. The closure enables delayed code to refer to variables which were alive at the time the asynchronous call was made. If these variables are truly local to the scope in which the call was made, they are guaranteed not to have changed when the observer is eventually triggered[9]. If the variables are not local, care must be taken because they may have been assigned in the mean time (just like instance variables may have been assigned when a synchronous invocation returns).

The above analysis shows why anonymous (block) closures are so useful to express event-driven computations. They maintain coordination and context which are otherwise lost in a pure event system. They therefore prevent the control flow from becoming scattered across many event handlers. The use of true closures (rather than mere function pointers) is crucial here because they close over lexically free variables. Such variables are an important part of the execution context of an asynchronous call. To make the discussion more concrete, consider the following piece of code:

```
whenever: InstantMessenger discovered: { |messenger|
  when: messenger<-getName()@Due(seconds(10)) becomes: { |name|
    buddyList.put(name, messenger);
    def sub := whenever: messenger disconnected: {
      buddyList.remove(name);
      sub.cancel();
    }
  }
}
```

The above code features three nested event handlers. Whenever an instant messenger is discovered, the messenger is queried for its name and is added to a buddy list. If the query is not replied to within 10 seconds, the messenger is ignored. If the messenger replies but fails at a later point in time, it is removed from the buddy list and the innermost event handler de-registers itself. This behaviour can be expressed concisely because of the use of block closures as nested event handlers. Note how the variables `messenger` and `name` can be used without any explicit coding effort within the nested event handlers. In pure event systems, these variables would have to be explicitly parameter-passed in between event handlers.

In short, the inversion of control often attributed to event-driven systems can be mitigated in AmbientTalk by using block closures to represent event handlers.

4.5.2 Suitability for Mobile Ad Hoc Networks

In this section, we argue why AmbientTalk's language constructs are suitable for developing mobile ad hoc networking applications. Again, we distinguish between the two most apparent hardware characteristics of mobile ad hoc networks described at length in section 2.2.

4.5.2.1 Volatile Connections

The strictly asynchronous communication between objects owned by different actors is very suitable in mobile ad hoc networks. The built-in message queues of actors

[9] Note that while the variable is guaranteed to still refer to the same object, that object itself may of course have been changed.

and eventual references decouple communication in time and synchronisation, making the application resilient to transient network failures. The failure handling strategy of buffering messages optimistically while disconnected is a good default if failures are mostly engendered by temporary network partitions.

A traditional RPC or RMI communication model is not able to provide a similar decoupling. To abstract over temporary failures, objects would either remain blocked waiting for an outstanding RPC to a disconnected object (making the application unresponsive), or the RPC would fail, forcing the programmer to deal with every failure, even if it is only temporary.

4.5.2.2 Zero Infrastructure

In mobile ad hoc networks, services have to be discovered in the proximate environment as devices are roaming. A shared infrastructure is not always available, such that objects should not be required to rely on a third party to discover one another. To enable decentralised service discovery, each AmbientTalk interpreter is equipped with a topic-based publish/subscribe engine. The topics are the type tags used to classify objects in a meaningful way, independent of any particular device address, catering to anonymous interactions among objects. Each actor can independently export objects and subscribe to be notified of objects that become available. At the interpreter level, discovery is implemented by repetitively broadcasting "advertisement" messages. Fortunately, in wireless ad hoc networks, we can exploit the fact that the cost of broadcasting a message to one device is the same as that of broadcasting it to all devices [KB02].

4.6 Case Study: the Musical Match Maker

In this section, we combine the various language constructs introduced throughout this chapter and apply them to the musical match maker application (♫uMaMa) introduced in section 3.1. This exposition both serves to validate the practicality of AmbientTalk and to provide the reader with a coherent view on how to develop distributed applications with AmbientTalk. The ♫uMaMa case study has been used as a running example before in previous AmbientTalk publications [DVM+06b, VDD07, GVDD07].

Before delving into technical details, we briefly outline how the implementation of ♫uMaMa in AmbientTalk deals with the different aspects of coordination introduced in section 3.1:

Discovery The ♫uMaMa application is represented by means of a single *service object* which is explicitly exported to the network. Two ♫uMaMa applications discover one another by means of AmbientTalk's built-in service discovery event handlers. Once the objects have discovered one another, they set up a *session* to transmit their music libraries.

Communication Once the service objects have established a session, they need to transmit their library index. Recall from section 3.1 that ♫uMaMa should tolerate transient network failures. We will exploit the failure semantics of AmbientTalk's far references to achieve this goal.

Synchronisation In our implementation, the ♫uMaMa service objects transmit their library index incrementally (on a per-song basis). To synchronise on when they can send subsequent song information, the service objects will make use of futures.

Failure handling If a network partition does persist, far references still referring to a session should be eventually invalidated such that it becomes possible for the garbage collector to reclaim the allocated session object. Failure handling will be achieved by means of timeouts and leasing.

In the following section we first explain some simple abstractions used by ♫uMaMa. Subsequently, we describe how ♫uMaMa applications represent themselves as *service objects* which need to be exported to and discovered in the network. We are then ready to explain the distributed library transmission protocol. Finally, we add failure handling.

4.6.1 Data Abstractions

In order to implement ♫uMaMa, we need some simple data abstractions. The core abstraction of ♫uMaMa is the library of songs. We will represent it by a simple `Set` of song objects. To give a flavour of how to build data abstractions in AmbientTalk, we show the definition of a simple `Song` abstract data type:

```
def Song := isolate: {
  def artist;
  def title;
  def init(artist, title) {
    self.artist := artist;
    self.title := title;
  };
  def ==(other) {
    (other.artist == artist).and:
      { other.title == title }
  };
}
```

A song is represented as an object with two fields (initialised upon object creation). Note that the object is an *isolate* and thus a pass-by-copy object. Therefore the default implementation of the == method should be replaced to ensure equality of songs is based on their content and not on their object identity. Sending the message **new** to a song object creates a copy of that object, initialised using its `init` method.

4.6.2 Exporting and Discovering Service Objects

In order for two ♫uMaMa applications to discover one another, they have to *export* one or more service objects such that these objects can be discovered by objects owned by remote actors. We assume that a ♫uMaMa application is represented as a single service object which can be used to create new library transmission sessions, and that this object is exported by means of the following type tag:

```
deftype MuMaMaApp;
```

The name of this type tag represents global knowledge: all ♫uMaMa peers have to somehow know this name. With this type tag defined, the service object representing the interface to the application can now be exported as follows:

```
def interface := object: {
  def openSession() {
```

```
    // return a session object (explained later)
  };
};
export: interface as: MuMaMaApp;
```

Discovering proximate ♫uMaMa peers is a matter of defining a discovery event handler:

```
whenever: MuMaMaApp discovered: { |peer|
  // open a session to transmit the library
  when: peer<-openSession() becomes: { |session|
    // start transmitting songs (explained later)
  };
};
```

The `peer` argument of the above block closure is a far reference to the exported `interface` object of another ♫uMaMa application. Upon discovery, the `openSession` message is sent asynchronously to the remote interface object. The block closure registered as an observer on the future by means of **`when:becomes:`** represents the continuation of the message send.

Note that the code that exports the `interface` object, and the code above that discovers other such objects is executed by all ♫uMaMa peers in the network. Hence, these applications engage in peer-to-peer communication: when a peer `Alice` and a peer `Bob` enter one another's communication range, `Alice` will discover `Bob`'s interface object and `Bob` will discover `Alice`'s interface object. This simultaneous bidirectional matching is an emergent property of the application: it is not coded explicitly, but rather emerges naturally because network connectivity is often symmetric (if `Alice` can talk to `Bob`, chances are high that `Bob` can talk to `Alice`).

4.6.3 The Library Transmission Protocol

We now describe the implementation of the library transmission protocol between two peers. Once a peer has a reference to the `interface` object of another peer, it asks the remote peer to open a library transmission session by sending it the `openSession` message. The return value of this message send is a session object which allows song information to be uploaded on a per-song basis via its `uploadSong` method[10]. After all songs have been sent, the `endTransmission` method is invoked to signal the end of the transmission.

Figure 4.2 shows a sequence diagram of the library transmission protocol. For purposes of clarity the figure only shows the essential objects and describes the protocol from the point of view of `Alice`. In reality, this protocol is executed simultaneously by both `Alice` and `Bob`. Below is the definition of the `session` object at the site of the recipient peer.

```
def openSession() {
  def peerLibrary := Set.new(); // to store incoming songs
  // return a session object to collect the songs
  object: {
    def uploadSong(song) {
```

[10] A realistic implementation would probably improve upon this scheme by uploading multiple songs in one remote message send or by pipelining messages.

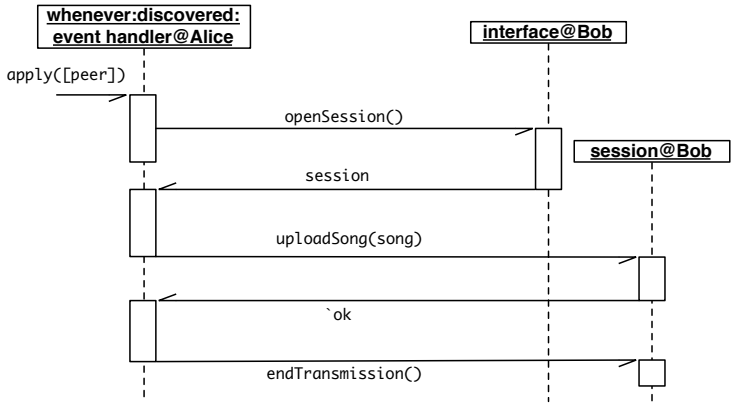

Figure 4.2: The library transmission protocol.

```
      peerLibrary.add(song);
      "ok" // acknowledge receipt
    };
    def endTransmission() {
      // calculate match percentage with my library
      if: (calculateMatch(peerLibrary, myLibrary) >= THRESHOLD) then: {
        // notify the user of the match
      }
    };
  };
};
```

Note that the `peerLibrary` is not directly accessible by remote peers, as it is hidden in the lexical scope of the anonymous session object. Still, it is a variable private to one particular session. Multiple sessions can be opened simultaneously and each session will encapsulate its own local state. The return value of `uploadSong` is a string representing a simple acknowledgement (see below). The sending peer uploads all of its own songs one by one to this session object upon discovery:

```
whenever: MuMaMaApp discovered: { |peer|
  when: peer<-openSession() becomes: { |session|
    def iterator := myLibrary.iterator();
    def uploadNextSong() {
      if: iterator.hasNext() then: {
        when: session<-uploadSong(iterator.next()) becomes: { |ack|
          uploadNextSong(); // recursive call to send next song info
        }
      } else: {
        session<-endTransmission();
      };
    };
    uploadNextSong(); // send the first song info
  };
};
```

Note how song information is sent sequentially from sender to receiver peer. This behaviour is guaranteed because `uploadNextSong` is only recursively invoked after the

current song has been successfully uploaded. It is easy enough to await this acknowledgement by registering an observer on the future returned by the `uploadSong` message send. Note that the use of an explicit (i.e. external) iterator over the collection is very useful here. If we were to write `myLibrary.each: {|song| ...}`, this would immediately generate an `uploadSong` request for all songs in the library, as `each:` iterates over the collection synchronously.

The above code once again demonstrates the usefulness of block closures to represent the continuation of an asynchronous computation. As argued in section 4.5.1, the use of block closures keeps the control flow of the application understandable (i.e. not fragmented). This is because block closures can be nested and because they implicitly capture the state (lexically free variables) needed to process the continuation.

4.6.4 Failure Handling

One of the major strengths of the ambient-oriented programming paradigm is that it assumes network partitions are omnipresent, and hence enables the programmer to abstract from them by default. Because far references mask failures by buffering messages while disconnected, the source code shown above is fully resilient to partial failures. Without *any* additional coding effort, the above protocol will continue to work if the connection between the peers is only intermittent.

Ironically, an AmOP programmer has to exert *more* coding effort in order *not* to make his code resilient to failures. Although the behaviour of the above source code is perfect for dealing with temporary network failures, it cannot cope with permanent failures. The problem is that, if a peer disconnects in the middle of the library transmission session, the session never terminates and unnecessarily consumes resources (e.g. the partially uploaded library of the sender peer).

In a first stage, we want the program to keep abstracting over the connection state of the underlying network for the purposes of exchanging the music library, yet we want to be able to detect and react upon changes in the environment. For example, we may want to inform the user that the transmission has temporarily stopped. We can express this by means of the failure event handlers introduced in section 4.4.1:

```
whenever: MuMaMaApp discovered: { |peer|
  // as before
  whenever: peer disconnected: {
    // inform the user that the transmission is paused
  };
  whenever: peer reconnected: {
    // inform the user that the transmission has resumed
  };
};
```

In a second stage, we need to gracefully terminate the transmission protocol if a disconnection lasts for too long. From the point of view of the sender peer, the protocol can easily be adapted by associating an upper bound with all of the asynchronous message sends (using a `Due(timeout)` annotation), as explained in section 4.4.4. From the point of view of the receiver, the problem cannot be solved by annotating outgoing messages with timeouts. Rather, what should be annotated with a timeout is the availability of the anonymous session object that is implicitly exported because it is the return value of a future-type message send. This can be achieved by not explicitly returning the session object, but rather a *lease* to the session object. Our initial support

for leasing in AmbientTalk using leased references [GVDD07] enables the code of the receiver peer to be rewritten as follows:

```
def openSession() {
  def peerLibrary := Set.new(); // to store incoming songs
  def lease := renewOnCallLease: LEASE_TIME for: (object: {
    def uploadSong(song) { /* as before */ };
    def endExchange() {
      // as before
      revoke: lease;
    };
  });
  when: lease expired: {
    // clean up resources associated with the session
    peerLibrary := nil;
  };
  // return a lease on the session object
  lease
};
```

The most important change to the code is that, rather than returning a far reference to the session object directly, a lease to that object is returned instead. A leased reference is an eventual reference to the session object until the lease time has elapsed. When the lease expires, access to the session object is denied and the leased reference no longer keeps the object alive for the purposes of garbage collection. Applications can schedule cleanup actions by registering closures as observers on leased references. Note also that the lease is revoked explicitly upon normal termination of the library transmission protocol.

The leased reference defined above is a "renew on call" lease, which implies that the lease is automatically renewed whenever a message is sent to the session object (i.e. it uses the heuristic that as long as an object is actively "in use" access to it should be prolonged). Hence, as long as the library transmission is in progress, the lease will not be prematurely revoked.

The precise details of the leasing language constructs can be found elsewhere [GVDD07]. Leasing is also discussed in more detail in section 4.9.2. The problems of AmbientTalk associated to garbage collection are discussed in the following section.

4.7 Limitations and Future Work

We now discuss the limitations of the AmbientTalk language and give some directions on how they can be addressed in future work.

Distributed Garbage Collection AmbientTalk currently does not feature a distributed garbage collector. There are conceptual reasons why this is the case. Recall that far references mask failures, which allows them to decouple the sender and receiver objects in time: messages can be sent to the remote target object even though it is disconnected. However, this very design decision prohibits one from reclaiming a target object even if all far references to it are disconnected – there is always a chance that one of them reconnects, requiring the object to be available still. In more general terms, the distributed garbage collector has to make a trade-off between *soundness* and *completeness* [AR98]: when a far reference to an object o becomes disconnected, either o

remains exported until the far reference reconnects (sacrificing completeness – the far reference may never reconnect, so o is never reclaimed) or it eventually takes o offline such that it can be reclaimed (sacrificing soundness – the far reference may reconnect and still refer to the object, hence becoming a broken pointer).

In the current implementation, we chose to sacrifice completeness: objects pointed to by far references are not automatically taken offline when all those far references become disconnected. As a result, these objects are not reclaimed. AmbientTalk does feature a low-level `takeOffline:` primitive that invalidates all far references referring to an object, enabling the object to be locally garbage collected. The goal is that using this primitive in conjunction with AmbientTalk's reflective capabilities, more high-level language constructs can be written in AmbientTalk itself.

Initial experiments to reconcile distributed garbage collection with AmbientTalk's far references have primarily focused on leasing [GVDD07]. Leasing is a time-based mechanism to perform fault-tolerant resource management [GC89]. A *lease* denotes the right to access a resource for a limited amount of time. At the discretion of the *lease granter* – the owner of the resource – a lease can be renewed, prolonging access to the resource. The advantage of leasing is that the lease granter retains control over the resource by maintaining the right to reclaim the resource once all of its leases have expired. Because of the lease time associated with a lease, the lease holder knows when its access rights have expired meaning it can no longer access the resource. Leasing solves the tension between soundness and completeness by weakening the notion of soundness: expired leases essentially are broken references.

The outcome of our experiments with leasing is the leased reference [GVDD07]. Here, an eventual reference plays the role of the lease and the object it designates plays the role of the resource. Hence, a leased reference only provides access to a remote object for a limited amount of time. However, as long as the leased reference is actively being used (i.e. messages are sent via the reference to the remote object), its lease is transparently renewed. Upon a network partition, the lease cannot be renewed and will expire if the disconnection outlasts the lease period. When a leased reference eventually expires, it ruins the future attached to any messages sent to it with an exception, signalling to the sender that its message could not be delivered. Once all leased references referring to an object have expired, the object is taken offline using aforementioned primitive, enabling the object to be garbage collected.

In future work, we would like to make leased references the default far references of AmbientTalk. However, what is lacking in the current proposal are expressive ways for programmers to declare and delimit leased references and how to deal with expired references. Further research is required in at least three areas. First, how can programmers expressively define a leasing policy for an entire group of references (e.g. all far references that make up an active session)? Second, what is the interplay between parameter-passing and leased references? Parameter-passing copies references, which in the case of leasing copies the *right* to access an object, which probably requires the lease granter to be involved. Third, how can programmers expressively deal with expired references? It may be useful to introduce language constructs to repair an expired reference by replacing it with a new one to a potentially different object. Or, the expiration of one reference may transitively invalidate the session of which it is a part.

Service Discovery The service discovery protocol of AmbientTalk implicitly assumes that distributed actors have a shared understanding of the type tags under which the objects are exported and discovered. The type tags used to classify exported objects

are assumed to be part of a universally shared ontology. There is a vast body of work on the limitations of these assumptions and on how to make peers coordinate when no such global knowledge is assumed. Work has been carried out to incorporate these results in the context of service discovery [CJF01, KK04], but the topic remains well beyond the scope of this dissertation.

The matching between exported AmbientTalk objects and subscribed event handlers is currently based entirely on connectivity as defined by the network layer underlying the AmbientTalk interpreter. For example, the matching range is entirely dependent on the fact whether e.g. Bluetooth or WiFi is used. Often, it is useful to be able to constrain service discovery to only discover services in a certain geographical region. For example, the Scalable Timed Events and Mobility (STEAM) middleware [MCNC05] introduces geographical locations as first-class entities named *proximities* which can be used for this purpose. In AmbientTalk, constraining service discovery based on geographical parameters has to be encoded on top of AmbientTalk's current discovery mechanism. Concrete experiments have already been conducted that show the feasibility of such an approach (cf. section 10.4).

Isolates Isolates are AmbientTalk objects that are passed by-copy in inter-actor message sends (cf. section 4.3.3). As previously discussed, in order for these objects to be copied without prohibitive costs, they are not allowed to access their enclosing lexical scope. In practice, this restriction is often a nuisance because if an isolate wants to call library functions, it has to import them in its own object scope, even though the functions may have already been imported in the outer lexical scope. On the other hand, because of this explicit import, the programmer is more aware of what code is copied along with the isolate.

The problem is that it is very difficult to separate pure code from objects in AmbientTalk, because of its prototype-based nature (code only exists within live objects). In a language like Java, the presence of static code entities like classes eases distributed deployment issues. Classes can be loaded, referenced and transmitted independently of live objects. When a Java object is serialised, classes referenced by the serialised object's class can be transmitted and loaded piecemeal, when they are first referenced by a byte-code instruction. On the other hand, keeping object and class as separate entities introduces versioning issues [Ded06]. For example, how should a serialised object of class `C` be treated when it is deserialised in a virtual machine that has already loaded an older or newer version of `C`? In AmbientTalk, because an isolate is self-sufficient, different versions of the same abstraction can coexist within the same actor.

The technical problems associated with isolates lead us to consider a more fundamental question: are objects really suitable as the unit of information interchange in a distributed system? A proper answer to this question is beyond the scope of this dissertation. However, the differences between regular objects and isolates lead us to believe that AmbientTalk objects, *in their current incarnation*, are indeed suboptimal to represent data to be exchanged. The difference in argument passing between objects and isolates is only a first indication. Another issue with copying objects is that they assume a totally different notion of *object identity*. Object identity is a crucial element of an object system, but when objects represent data to be distributed, a mismatch occurs: the data itself does *not* have an identity as such (e.g. an XML document does not itself have an identity). As a consequence, the built-in semantics for testing object equality is simply wrong for isolates, and we have wasted no small amount of time fixing the ensuing bugs.

Security Securing AmbientTalk is a topic that has not yet been thoroughly investigated. However, we have initial ideas to reconcile AmbientTalk with certain security properties. E, the language on which AmbientTalk's actor model is based, is a secure object-capability language [MMF00, MS03]. In a nutshell, the main idea is that a reference to an object also holds the authority to manipulate that object (strictly by means of its public interface). Hence, object references become the "resource" to be secured. In order to uphold security, objects can transmit and acquire object references only by a very specific set of operations (e.g. object creation and by-reference parameter passing). E abolishes global or static variables and functions, because they can be designated by *any* object, defeating the use of object references as the sole access mechanism. If one consistently models critical resources (such as files or network sockets) as objects that cannot be globally accessed, applications can be made capability-secure.

Object-capability security seems to us the most appropriate security model for AmbientTalk, because of the way it successfully unifies access control with object-orientation. As Miller explains in his dissertation, object-capabilities turn the access control graph referred to in the security community into the object-reference graph familiar to object programmers [Mil06]. However, AmbientTalk features language constructs not present in E, which are – to a lesser or larger extent – in conflict with object-capabilities. We sum up the most important issues below.

Delegation To a large extent, AmbientTalk's object model is similar to that of E: it is a lexically scoped language enabling objects to hide information by means of their lexical scope. Two features absent in E are delegation and trait composition. Because trait composition can be translated into a form of explicit delegation (cf. section 4.2.4), we only consider delegation. Delegation is potentially dangerous because any object may delegate a message to any other object, which causes the **`self`** pseudo-variable in the invoked method to refer to an arbitrary object. If the invoked method performs a self-send, as in **`self.`**`m(x)`, the programmer must be aware that the value of `x` (which may be part of the object's private state) is shared with potentially any object.

We do not consider delegation to be an extremely harmful language construct, for two reasons. First, in AmbientTalk, receiverless message sends (e.g. `m(x)`) are resolved purely lexically; they are not implicitly transformed into a self-send if `m` is not lexically visible. Hence, programmers can choose not to use **`self`** if late binding is not desired. Second, AmbientTalk does not enable distributed delegation, which implies that **`self`** can never be an eventual reference (i.e. it always refers to a local object). Because we assume that objects local to the same actor can be trusted, this does not introduce any major security issues.

Reflection AmbientTalk features an extensive metaobject protocol with a mirror-based architecture (cf. section 5.2). Mirrors are metaobjects which encapsulate the implementation of their base-level object. As such, even if an object `a` holds a reference to a mirror for an object `o`, `a` cannot convey more power via the metalevel interface than via the base-level interface. Even at the metalevel, the lexical scope of `o` remains opaque. However, at the metalevel, `a` can gain more knowledge about `o`, as it can introspect `o` to see to which messages it responds.

In a distributed context, reflecting upon an eventual reference only returns a mirror *on the reference* (rather than on the – possibly remote – object which the reference designates). As a result, simply having a far reference to an object does not convey the power to acquire a far reference to that object's mirror. A

mirror on a far reference only conveys the right to asynchronously send reified messages. While reflection certainly conveys more power over local objects, it does not by itself introduce security issues because objects cannot directly acquire mirrors on remote objects.

The security of delegation and reflection crucially depends on our assumption that all objects local to an actor can be trusted. However, this assumption is currently flawed because isolate objects enable code mobility: an isolate originating from a remote host is a local object yet it cannot be trusted. In order to secure AmbientTalk, we either need to remove our dependency on this assumption or otherwise ensure that isolates cannot violate the assumption.

Service Discovery Perhaps *the* most difficult feature of AmbientTalk to integrate with capability-security is service discovery. Service discovery enables objects to acquire object references simply by having knowledge of the type tag with which other objects have been exported. Hence, because of service discovery, type tags become an additional resource to be secured, next to object references. However, because type tags in AmbientTalk are identified simply by their name, type tags are globally accessible. As such, service discovery becomes a form of "ambient authority" [MYS03][11] and as such it is in direct conflict with the principle of object-capability systems that "only connectivity begets connectivity" [Mil06]. This dichotomy should come as no surprise, given that it is the explicit *goal* of service discovery to connect two a priori disjoint object graphs.

Because service discovery in AmbientTalk is currently a form of ambient authority, we have no adequate access control to restrict the visibility of exported objects or discovery queries. A different issue related to service discovery is that of trust. For example, how can the service discovery module running on a PDA trust the printing service it has just discovered to really *be* a printing service? The problem is exacerbated in pure ad hoc networks because one cannot rely on a globally accessible authoritative third party to authenticate discovered objects.

In short, service discovery introduces security issues which are currently not dealt with at the language level. Reconciling object capabilities with service discovery is an important topic for future research.

Now that the most apparent shortcomings of AmbientTalk have been discussed, we briefly describe the current implementation of the language.

4.8 Notes on Implementation Status

An interpreter for the AmbientTalk language has been implemented in Java. The implementation ships with a small library providing additional support functions and language constructs written in AmbientTalk itself (including the reflective implementation of futures and leasing). The interpreter can run on the Java 2 micro edition (J2ME) platform under the connected device configuration (CDC). This configuration targets

[11] The term "ambient" in this definition is totally unrelated to the one in "ambient-oriented" programming. Ambient authority indicates a form of global authority, outside of the program's scope. However, when considering the authority conveyed by service discovery, the term "ambient" *can* be taken literally, as the authority to access a resource is literally determined by the physical ambient of the device.

PDAs and high-end cellular phones. Our current experimental setup consists of a number of QTek 9090 smartphones which communicate by means of a wireless ad hoc WiFi network.

At the implementation level, AmbientTalk interpreters communicate with one another by means of sockets. AmbientTalk's topic-based publish/subscribe service discovery mechanism is peer-to-peer and does not require a centralised repository or directory service. AmbientTalk interpreters discover one another by means of the network's support for multicast messaging. After a successful discovery, the two interpreters exchange discovery information (e.g. registered subscriptions and exported objects) in order to find a match.

4.9 Previous and Related Work

In this section, we describe how the AmbientTalk language as described in this chapter differs from the prior AmbientTalk/1 language. We also provide pointers to prior work on the various concepts, languages and systems on which AmbientTalk is based.

4.9.1 AmbientTalk/1 versus AmbientTalk/2

Now that both AmbientTalk/1 and AmbientTalk/2 have been presented, we can highlight their differences and commonalities. Let us first discuss what important features of AmbientTalk/1 have been kept. Ultimately, the way in which both languages deal with the two hardware phenomena of MANETs remains the same. In section 4.5.2 we mentioned that AmbientTalk/2 deals with volatile connections by means of asynchronous message passing and by making far references resilient to failures by default. In AmbientTalk/1, remote references to active objects provide the same semantics. AmbientTalk/2 provides a topic-based publish/subscribe engine to cater to service discovery lacking any network infrastructure. AmbientTalk/1 provides similar functionality through its four discovery mailboxes discussed in section 2.4.3.

While the language features used by both languages are fundamentally equivalent, in practice they provide different levels of expressiveness. These will be brought to light in the following paragraphs. As most changes in going from AmbientTalk/1 to AmbientTalk/2 are motivated by limitations of the former, the rest of this section mirrors section 2.5 on the limitations of AmbientTalk/1.

Double-layered Object Model Recall that AmbientTalk/1's object model distinguishes between active and passive objects. In AmbientTalk/1, actors are modelled as ABCL/1-like active objects. However, AmbientTalk/2's concurrency model replaces the notion of actors as active objects with the notion of actors as vats, based on the vat model of the E language [MTS05]. In this model, actors become containers of passive objects. Despite these differences, in both versions each passive object is contained within exactly one actor. However, AmbientTalk/2 allows both sequential and asynchronous message sends between passive objects whereas AmbientTalk/1 only supports sequential message passing between passive objects.

Both versions also consider actors as the unit of distribution. However, in AmbientTalk/1, passive objects are not remotely accessible. Only the *behaviour* of an active object can be referenced from other actors. In AmbientTalk/2, passive objects owned by other actors can be individually designated by means of far references. This enables far more fine-grained interactions between passive objects.

Inter-actor Message Passing The parameter-passing semantics of inter-actor message sends in AmbientTalk/2 differs from those of inter-active object message sends in AmbientTalk/1. In AmbientTalk/1, passive objects are by default passed by copy. In AmbientTalk/2, objects are by default passed *by far reference*. Only isolates are passed by copy. The advantage of passing objects by far reference is that remote communication is more lightweight because the object-graph does not need to be completely copied. Moreover, by being able to partition the object graph of an actor into regular objects and isolates, the programmer has more control over which parts of the graph must be serialised and which not. In AmbientTalk/1, all passive objects are effectively isolates, explaining why its by copy semantics are heavyweight and less customisable.

Service Discovery In AmbientTalk/1, service discovery is expressed only in terms of meta-level entities such as the `requiredbox` and the `joinedbox`. AmbientTalk/2 features expressive language constructs like **`export:as:`** and **`when:discovered:`** directly at the base-level. Also, AmbientTalk/2 makes service discovery polymorphic because exported objects are matched with observers based on type tags, which support subtyping.

Mailboxes The mailboxes of AmbientTalk/1 have either been adapted or removed to fit with the new object model based on vats:

- The `inbox` has become the implicit incoming message queue of an actor. The `outbox` has become the implicit outgoing message queue contained within eventual references to objects owned by other actors.
- The `sentbox` and `rcvbox` have no direct representation. By default, message histories are not kept. However, they can be reconstructed by means of the metaobject protocol.
- The tags stored in the `requiredbox` and `providedbox` have been replaced by type tags used to subscribe closures as discovery event handlers and to export objects, respectively.
- The `joinedbox` and `disjoinedbox` have been replaced by failure event handlers on far references (cf. section 4.4.1).

Garbage Collection As discussed above, AmbientTalk/2 actors do not keep track of their communication history by default. This ensures that messages are normally automatically garbage collected. If they are needed for the purposes of a reflectively implemented language construct, they must be explicitly trapped via the metaobject protocol. This corresponds to the strategy discussed previously in section 2.5.2.

Recall that AmbientTalk/1 did not feature any distributed garbage collector. While AmbientTalk/2 does not feature automatic distributed garbage collection either, it enables the use of leases to partially resolve this issue (cf. section 4.7).

Metalevel Engineering AmbientTalk/2 has a mirror-based [BU04] meta-level architecture [MVTT07]. This removes many of the hurdles in engineering robust meta-level programs in AmbientTalk/1. Most notably, base- and meta-level code are completely stratified. Name clashes between the two cannot occur (cf. section 5.2.3). Also, mirrors

encapsulate implementation details: they provide a well-defined interface (the meta-object protocol) and can implement this interface however they want, without leaking implementation details. Thanks to traits, meta-level code can be composed in a principled way and conflicts between meta-level operations can be detected at composition time. Finally, AmbientTalk/2 consistently mirrors all base-level entities, not only actor-level entities, making the MOP more uniform.

Dynamic versus Lexical Scoping In AmbientTalk/1, all identifiers are resolved in the object scope. AmbientTalk/2 gives the choice between resolution in the lexical scope or in the object scope. This additional scope enables the useful pattern of nesting objects to share a private, lexical scope.

Linguistic Symbiosis AmbientTalk/1 does not provide any means to access the libraries of foreign languages. Although it has not yet been explained, AmbientTalk/2 can interoperate with the Java Virtual Machine through a mechanism known as linguistic symbiosis. This mechanism is discussed in section 5.3. A full account on this subject is given elsewhere [VMD07].

4.9.2 Notes on Related Work

In this section, we highlight the programming concepts and languages that have influenced the design of AmbientTalk in significant ways.

Actors AmbientTalk's integration of concurrent and distributed computing with object-oriented computing is founded on the actor model of computation [Agh86, Agh90], already touched upon in section 3.3.4.1. In the model actors refer to one another via *mail addresses*. When an actor sends a message to a recipient actor, the message is placed in a mail queue and is guaranteed to be eventually delivered by the actor system. Asynchronous communication by means of mail addresses decouples actors in time and synchronisation. This property makes the actor model in itself almost suitable for mobile networks.

The main feature lacking in the actor model to fit mobile networks is a means to perform service discovery, i.e. to acquire the mail address of a remote actor via anonymous communication. Mail addresses do not decouple sender and receiver in space. Extensions of the actor model have addressed this issue. We already previously mentioned the ActorSpace model [CA94] (cf. section 3.3.4.1) which enables messages to be sent to a *pattern* rather than to a mail address. While ActorSpace caters to anonymous communication between actors, it does not introduce an explicit form of service discovery.

E AmbientTalk's view of actors as communicating event loops is directly based on the communicating event loops model of the E programming language [MTS05, Mil06]. AmbientTalk also inherits from E the distinction between different types of references (near versus eventual references) and their associated message passing semantics. E also pioneered the `when` expression to access the value of futures (or promises, as they are called in E) in an entirely non-blocking, event-driven manner.

E is designed for writing secure peer-to-peer distributed programs for open networks, but not specifically for mobile ad hoc networks. That is why AmbientTalk

diverges from E's distribution model with respect to the failure semantics of far references. A network disconnection in E immediately *breaks* the far reference: any message sent after the disconnection is not stored, and the message's promise is resolved with an exception. Hence, E's far references do not decouple participants in time and are not designed to express communication over volatile connections.

E's references respond to a method named `__whenBroken` that plays a similar role to AmbientTalk's **`whenever:disconnected:`** failure event handler to enable the programmer to react upon the disconnection of a far reference. There is no corresponding method for reacting to a reconnection in E, because once broken, an E far reference remains broken.

To regain connectivity after a network failure, E features special references, known as *sturdy references*, which do survive network failures. Sturdy references, however, are created by means of an explicit address (in the form of a URL) and are meant to denote specific objects, so they do not decouple objects in space. When a host creates a sturdy reference for one of its local objects, it may associate a lease time with the reference, denoting how long the host should keep the object available. E features a persistence model with checkpointing, so its sturdy references may persist across crashes.

Futures The origin of futures can be traced back to their use in speculative parallel programming [BH77]. They are a frequently recurring abstraction in concurrent languages [BGL98]. They serve as an essential synchronisation tool when asynchronous message passing semantics are introduced. The use of futures as return values from asynchronous message sends can be traced back to actor-based languages such as ABCL [YBS86, Yon90, TMY94]. In Argus, futures are known as promises. Argus promises were the first to introduce a form of chaining [LS88]. Most future abstractions support synchronisation by suspending a thread that accesses an unresolved future. This is sometimes called *wait by necessity* [Car89, Car93]. This synchronisation may happen implicitly, as in Multilisp [Hal85] and ProActive [BBC+06] or explicitly, as in Argus [LS88] and Java 1.5 [GJSB05].

AmbientTalk's future chaining is similar to *promise pipelining* in E [MTS05] and *automatic continuations* in systems like Eiffel// and ProActive [EAC98]. Future chaining and promise pipelining differ in their treatment of messages sent to unresolved futures. In AmbientTalk, messages sent to an unresolved future are buffered in a proxy object local to the sender. In E, messages sent to an unresolved promise are immediately forwarded to the vat that will contain its resolved value. This is done to minimise round trips across the network. Incorporating pipelining in AmbientTalk is a topic of future work.

The technique used in AmbientTalk to register a closure as an observer on the future is directly derived from E's `when` expression. It is closely related to the notion of a customer actor in the actor model, which is an actor representing the continuation of an asynchronous request [Agh90]. Twisted is a well-known Python framework for network programming which makes heavy use of this technique to reduce the impact of network latency on programs. In this framework, *deferrables* play the role of AmbientTalk's futures [Fet05]. All of these techniques essentially express control flow in a continuation-passing style.

Leasing The initial experiments to manage the lifetime of AmbientTalk's far references by replacing them with leased references are primarily based on the use of leasing

in Jini [Wal99, Wal01]. Jini is a platform for service-oriented computing built on top of Java. In Jini, resources owned by a service should be accessed by clients by means of leases. This ensures that services can gracefully deal with unexpected disconnections. For example, services may advertise themselves by registering with a lookup service, but must explicitly renew their registration by means of a lease. Otherwise, the lookup service removes the advertisement, ensuring that it does not advertise stale information of services that have become unavailable [Wal01].

Leasing has already previously been integrated into programming languages for managing the lifetime of remote objects (e.g. in Java RMI [Sun98] and .NET Remoting [MWN02]). The idea of renewing the lease of a remote object upon each remote method invocation made to the object stems from the .NET Remoting framework. AmbientTalk's support for leasing differs from that of Java RMI and .NET Remoting in the sense that leases are used for managing valid disconnected remote object references. Java RMI and .NET Remoting do not provide time-decoupled remote object references that tolerate failures by default.

Publish/Subscribe Systems AmbientTalk's service discovery mechanism is based on the publish/subscribe paradigm [EFGK03]. More specifically, it is based on type-based publish/subscribe [Eug07], where the concept of topic is unified with the concept of type. The use of polymorphic type tags for service discovery was also inspired by Jini's use of Java interface types to achieve the same goals [Arn99].

Publish/subscribe systems perform well in mobile ad hoc networks because they decouple publishers and subscribers in time, space and synchronisation. The main difference between traditional, centralised publish/subscribe architectures and those for mobile networks is the incorporation of geographical constraints on the event publications and subscriptions. For example, in the location-based Publish/Subscribe (LPS) architecture [EGH05, EGH06], a publisher defines a *publication range* and a subscriber defines a *subscription range*. Only when the publication range of the publisher and the subscription range of the subscriber overlap is an event disseminated to the subscriber.

4.10 Conclusion

In his famous 1973 paper on *Hints on programming language design*, C.A.R. Hoare notes that the task of a language designer lies in the *consolidation* and *integration* of existing language features. He notes [Hoa73], p. 26:

> "The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best, and rejecting any that are mutually inconsistent. He must be capable of reconciling, by good engineering design, any remaining minor inconsistencies or overlaps between separately designed features. He must have a clear idea of the scope and purpose and range of application of his new language, and how far it should go in size and complexity."

As language designers of AmbientTalk/2, we have pursued the above advice to its maximal extent. When considered in isolation, none of AmbientTalk/2's language features are truly novel. However, combining the many interesting language features drawn from a variety of other languages and middleware into a consistent object-oriented language framework, and this in the specific domain of mobile ad hoc net-

works, is arguably an achievement in its own right. What is more, the very product of this integration has lead to novel scientific contributions in the following domains:

Object-orientation In the field of object-oriented programming, AmbientTalk/2 combines ex-nihilo objects, prototypes and classes in a unique manner. AmbientTalk/2:

- combines lexical scoping (engendered by nesting objects) with a restricted form of dynamic scoping (engendered by object-based delegation).
- unifies class instantiation with prototype-based cloning.
- transposes traits as defined by Schärli et al. [SDNB03] into an object-based setting. The main difference with Self's traits [UCCH91] is that AmbientTalk's traits are composed in terms of an explicit delegation message passing operator rather than via implicit multiple inheritance.
- provides a synergy between the traditional canonical syntax of functions and the keyworded messages characteristic of Self and Smalltalk. By representing "language constructs" as keyworded functions or methods, AmbientTalk/2 becomes an extensible "language laboratory".

Distribution In the field of distributed programming, AmbientTalk/2:

- integrates the service discovery abstraction traditionally provided by middleware or libraries directly in the programming language runtime.
- integrates Rover's queued RPC failure handling model (cf. section 3.3.6.1) and the time-decoupling properties of actor mail addresses (cf. section 3.3.4.1) with E's far references.

Concurrency AmbientTalk/2 is a proof-by-construction that E's concurrency model of communicating event loops can be applied to:

- another object model: AmbientTalk/2 sufficiently differs from E through its introduction of prototypes, delegation, traits, reflection,...
- another distribution model: AmbientTalk/2 is designed specifically for MANETs, which required us to modify the failure semantics of far references.
- represent service discovery as an asynchronous event, to which the programmer can react using event handlers similar to the `when`-expression in E to react on the resolution of promises.

Before turning our attention to ambient references in chapter 6, the following chapter continues our exposition of AmbientTalk. In particular, it describes AmbientTalk's support for metaprogramming and reflection. These abstractions are invaluable tools of the language, given its role as a language laboratory to experiment with novel language features. Later, in chapter 8, we apply these tools to implement ambient references reflectively in AmbientTalk.

Chapter 5

Metalevel Engineering in AmbientTalk

In the previous chapter, we have introduced AmbientTalk as an object-oriented, concurrent and distributed programming language. This chapter serves to establish AmbientTalk's role as a language laboratory: like its predecessor AmbientTalk/1, AmbientTalk/2 is an extensible research artifact with the goal of explicitly supporting the development and exploration of novel language constructs. To this end, the language is equipped with extensive metalevel programming abstractions, following a long-standing tradition of applying reflection to solve concurrent and distributed programming problems [OIT92, McA95, MMY96, BGL98, CBM+02]. This chapter contributes the following new concepts: first-class messages, the representation of custom object references as proxy objects with custom metaobjects and a mechanism to transparently yet safely compose JVM threads with AmbientTalk event loops. In chapter 8, we show how these concepts are applied to incorporate ambient references (introduced in the following chapter) as a language construct into AmbientTalk.

5.1 First-class Messages and Methods

We start our exposition of metalevel engineering in AmbientTalk by showing how object-oriented messages and methods can be manipulated by the programmer as first-class citizens of the language. While some object-oriented languages (most notably Smalltalk and Self) also reify messages, we will show that AmbientTalk goes further with respect to integrating them in the language.

5.1.1 First-class Messages

The messages sent and received by AmbientTalk objects are first-class objects in their own right. AmbientTalk is far from being the first language to introduce first-class messages. However, what is remarkable is that AmbientTalk features syntax for expressing *literal* messages. Consider the following methods defined on tables (AmbientTalk's arrays):

```
def eachSend: message {
  self.each: { |rcvr| rcvr <+ message }
```

```
};
def mapSend: message {
  self.map: { |rcvr| rcvr <+ message }
};
```

The expression `rcv <+ msg` sends a first-class message `msg` to an object `rcv` as if the message was literally invoked on the receiver in the source text. Hence, this expression provides the functionality of `perform:` in Smalltalk. The above methods, together with a literal syntax for messages, can be employed to express *higher order messages* [WD05]: messages taking other messages as their argument.

```
observers.eachSend: <-statusUpdated(newStatus);
[4,5,6].mapSend: .+(2); // returns [6,7,8]
```

A literal message is denoted by a message send expression lacking a receiver. AmbientTalk distinguishes between three kinds of literal messages: asynchronous ones (`<-m()`), synchronous ones (`.m()`) and delegated ones (`^m()`). The `<+` operator is polymorphic and expresses either an asynchronous, a synchronous or a delegated message send based on the type of its message argument.

A first-class message can be queried for its selector (its name) and the actual arguments it carries. The message object can also be classified according to all type tags with which it was annotated using the `@` syntax. Hence, for a message `msg` constructed as `.m()@Type`, evaluating `is: msg taggedAs: Type` yields `true`.

The benefit of having an expressive syntax for manipulating messages as first-class language citizens is that it becomes easy to express higher-order functions which can make direct abstraction from the *messages* sent back and forth between objects. Traditionally, this behaviour is incorporated in object-oriented languages by means of functional abstraction using block closures. Higher-order messages can be seen as an alternative abstraction mechanism which is purely object-oriented in nature [WD05]. Having an expressive syntax for messages is paramount for this abstraction to be usable in practice: requiring the programmer to write `Message.new('selector,[argument])` (as is done in Smalltalk) is as inexpressive as requiring the programmer to represent a lambda closure as the single method of an anonymous inner class (as is done in Java). Weiher and Ducasse achieve higher-order messages as a second-class abstraction in Smalltalk by means of argument currying and the ingenious use of Smalltalk's `doesNotUnderstand:` protocol [WD05]. Because AmbientTalk supports messages as first-class citizens of the language, they do not have to be represented implicitly in terms of argument currying.

In this section, we have shown how messages can be created and sent as first-class objects. We postpone a discussion on *intercepting* messages received by an object (cf. Smalltalk's `doesNotUnderstand:`) until section 5.2.2. In the following section, we first consider the first-class representation of methods rather than messages.

5.1.2 First-class Methods

In AmbientTalk, methods may be represented as first-class objects. Our representation is based on the first-class methods of the Pic% language [DD03], an object-oriented derivative of Pico [D'H96]. In AmbientTalk and Pic%, methods may be represented as closures. To see why this is useful, consider the following example which is typical of Smalltalk and Self:

```
dataset.map: { |data| converter.transform(data) }
```

In Pic% and AmbientTalk, the programmer's intent can be better expressed by replacing the block closure (whose sole task is to pass on the data from the collection on to the `converter` object) with the actual `transform` method of the `converter`. In AmbientTalk, we express this as follows:

```
dataset.map: converter.&transform
```

The `.&` operator selects a method from a receiver object and evaluates to a first-class closure representing that method. Importantly, when applying that closure, the caller only needs to pass regular parameters, it does not need to pass an explicit "receiver" argument to represent **`self`**. While a method does require such an additional receiver parameter, upon application the closure representing the method implicitly passes the receiver object from which the method was selected using `.&`. This ensures that a method always remains associated with a proper receiver object. This also forms the major difference between first-class methods and *reified* methods. In Java, for example, `java.lang.reflect.Method.invoke` requires the receiver to be passed explicitly as the first argument.

Note that a separate `.&` operator is required to indicate method selection. The expression `converter.transform` represents an invocation of `transform` with an empty parameter list. This is a direct consequence of AmbientTalk's adherence to the uniform access principle (cf. section 4.2.1).

Now that we have discussed both first-class methods and messages, we can turn our attention to AmbientTalk's support for reflection. As will be described later, first-class messages have an important part to play in the reification of asynchronous message sending in AmbientTalk.

5.2 Reflection

Computational reflection allows programs to reason about themselves [Smi84, Mae87]. AmbientTalk provides extensive support for reflection by means of a mirror-based architecture. Below, we explain the general ideas behind mirror-based reflection. AmbientTalk's metalevel architecture is novel in that it combines mirror-based reflection with *intercession* – the ability of programs to change the semantics of the programming language. Also, next to applying mirror-based reflection on objects, we also apply it on event loop actors. Finally, and most importantly, we show how reflection in AmbientTalk can be used to build custom object references as proxy objects with a custom metaobject. This implementation technique will be used in chapter 8 to implement ambient references.

5.2.1 Mirror-based Reflection

AmbientTalk is built upon a *mirror-based* reflective architecture. Bracha and Ungar define a mirror-based architecture as any reflective architecture whose metaobjects (called *mirrors*) adhere to three key design principles: *encapsulation*, *stratification* and *ontological correspondence* [BU04]. They are briefly summarised below:

Encapsulation requires mirrors to *encapsulate* their implementation details. This promotes the reuse of metalevel programs because metaprogrammers can code against an *interface* rather than against a specific *implementation*. One of the important aspects noted by Bracha and Ungar in this context is the role of the

type system: any metalevel API should not leak implementation details through its static type declarations.

Stratification requires mirrors to be cleanly separated from base-level functionality. We already briefly mentioned stratification in section 2.5.3 when discussing the limitations of AmbientTalk/1's metalevel architecture. The separation afforded by stratification ensures that when a base level method's name corresponds to a metalevel operation, this method is not accidentally regarded as part of the metaobject protocol. In a mirror-based architecture, any access to a mirror object should be a dedicated, explicit operation. Not only does this have benefits for deployment (e.g. only lazily enabling reflection support [BU04]), it also allows metalevel programs to intercept this operation to enforce encapsulation of the meta-level representation of base-level objects. For example, if p is a proxy for another object o and a client asks a mirror on p, an implementation could hide the existence of p for the client and directly return a mirror on o instead. This is only possible if the implementation can intervene in the operation that determines p's mirror.

Ontological Correspondence states that the metalevel should be structured according to the same concepts and rules that govern the base-level. Bracha and Ungar further distinguish between *structural* and *temporal* correspondence. In short, structural correspondence implies that each base-level construct in the language has a meta-level representation. Temporal correspondence requires a mirror-based architecture to make the distinction between code (a description of a computational process) and computation (the actual execution of that process) explicit.

In the following sections, we describe how these abstract design principles are concretised to reflect upon AmbientTalk objects and actors.

5.2.1.1 Mirrors on Objects

AmbientTalk's mirror-based architecture has been inspired by that of Self [ABC+00]. The following code excerpt shows how one may reflectively manipulate an instance of the `Point` object defined in section 4.2.1, page 60[1]:

```
def p := Point.new(2,3);
// retrieve a mirror by invoking reflect:
def mirrorOnP := (reflect: p);
// read the contents of a field via its mirror
mirrorOnP.grabField(`x).value; // 2
// retrieve a mirror on a method
mirrorOnP.grabMethod(`init); // <mirror on method:init>
// reflectively invoke a method
mirrorOnP.invoke(p, `distanceToOrigin, []);
// print all method names
mirrorOnP.listMethods.each: { |method|
  system.println(method.name)
};
// add a z coordinate
mirrorOnP.defineField(`z, 0);
```

[1] In AmbientTalk, a backquote character is used to quasi-quote an expression (cf. quasi-quoting in Scheme [ADH+98]). A quasi-quoted expression evaluates to an object representing the expression's abstract syntax tree. Quasi-quoting an identifier evaluates to a symbol.

A mirror is a metaobject which is *causally connected* [Mae87] to the object it mirrors: if the object is changed by base-level code, the changes can be observed via the mirror. Conversely, changes applied explicitly to the object via the mirror modify the actual base level object. The above examples illustrate many different forms of reflection. Using the terminology of Kiczales et al. [KRB91], AmbientTalk mirrors support:

Introspection : the retrieval of fields and methods (cf. `grabField`, `grabMethod` and `listMethods`).

Invocation : the explicit invocation of methods (cf. `invoke`). The arguments passed to `invoke` denote a receiver (any object), a selector (a symbol) and actual arguments (a table). The receiver parameter is the object to which **`self`** is bound during method invocation. If this receiver is the same as the object being mirrored (`p` in the example), the reflective call expresses a standard method invocation. If the receiver object is a different object, the reflective call expresses explicit delegation (cf. the ^ operator introduced in section 4.2.1).

Self-modification : the addition of new fields and methods (cf. `defineField`).

Mirrors on objects are accessed by calling the **`reflect:`** function. The **`reflect:`** function in turn creates a mirror by calling a factory method, which can be replaced by metaprograms. Because a mirror on an object `obj` is retrieved via a separate mirror factory (via **`reflect:`** `obj`), rather than by querying the object itself (e.g. via `obj.` **`reflect`**), the association between objects and their metaobjects can be separated from base level concerns. Separating mirrors from their associated base level objects in this way is what makes the mirror architecture stratified.

5.2.1.2 Mirrors on Actors

A novelty of AmbientTalk's metalevel architecture is that it enables programmers to reflect upon actors represented as event loops. Each actor hosts both base-level objects (representing an application) and metalevel objects (mirroring base objects). Furthermore, each actor hosts an *actor mirror*: a special object denoting the mirror on the actor as a whole. This mirror is special in that it does not reflect upon a concrete base-level object because an AmbientTalk actor is an event loop rather than a concrete object. The actor mirror allows manipulating the event loop without exposing its implementation, just like a `java.lang.Thread` object in Java allows for the manipulation of a thread without exposing its implementation.

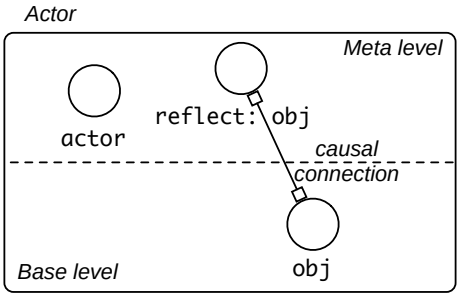

Figure 5.1: Reflective view on an AmbientTalk actor.

Figure 5.1 depicts the relationship between base and metalevel objects in an AmbientTalk actor. An actor's mirror is bound to the **actor** field in the actor's top-level scope. An actor mirror can be accessed without invoking **reflect:**. This does not violate stratification because **actor** is already a pure metalevel entity (it does not need to be reflected).

For the purposes of this dissertation, the most important aspect of an actor mirror is that it reifies the asynchronous message sending protocol used to communicate with remote objects. The other concepts reified by the actor mirror are discussed elsewhere [MVTT07]. The reification of asynchronous message sending in AmbientTalk is akin to that of other reflective architectures such as that of ABCL/R [WY88] and CodA [McA95]. In AmbientTalk, base-level code of the form `receiver<-message(argument)@Annotation` is reified by the asynchronous message sending metaobject protocol as:

```
actor.send(receiver,
           actor.createMessage('message,[argument],[Annotation]));
```

The `createMessage` method returns a first-class object representing the asynchronous message, while the `send` primitive enqueues the message in message queue of the receiver's owning actor. By overriding these methods, one may attribute new semantics to asynchronous messages and their transmission. Overriding these methods is done by dynamically installing a new actor metaobject. Section 5.2.3.1 presents a concrete example of the installation of a new actor protocol that shows how futures can be attached to asynchronous messages from within AmbientTalk. In the following section, we first continue our explanation of reflecting on regular objects.

5.2.2 Mirages: Mirror-based Intercession

One form of reflection which has not yet been discussed in section 5.2.1.1, again in terms of the classification of Kiczales et al. [KRB91], is *intercession*: the ability of metaprograms to modify the behaviour of objects. As a language laboratory, AmbientTalk relies heavily on intercession to develop new language constructs. For example, both futures and ambient references require intercession to be implemented reflectively. In this section, we describe how an AmbientTalk programmer can provide his own definition for methods of AmbientTalk's metaobject protocol, such as e.g. the `invoke` operation showcased in the `Point` example shown previously.

Early work on computational reflection already made the distinction between two processes called "reification" and "reflection" [FW84]. Reification allows programs to access data structures internal to a language interpreter. Mirrors provide such a reification for AmbientTalk objects. Reflection, as used by Friedman and Wand, refers to the reverse property of allowing a program to replace the data structures used by the language interpreter. Traditionally, support for such reflection in mirror-based architectures has been very limited [BU04]. AmbientTalk's contribution is that it reconciles mirrors with such a form of reflection by means of what we call a *mirage* [MVTT07].

A mirage is an object whose semantics is entirely described by another AmbientTalk object[2]. It is a base-level object causally connected to a mirror object that defines a custom MOP. In order to clarify this, consider the archetypical example of

[2]The name is both a pun on "reflection" (a mirage is an optical illusion arising from reflection) and signifies something that does not really exist. Because a mirage's semantics is specified in AmbientTalk itself, it does not exist as a concrete AmbientTalk object in the AmbientTalk interpreter.

intercepting and logging all methods invoked on an object. First, we define a prototype mirror object that encodes the logging behaviour by overriding the default implementation of the `invoke` metalevel operation:

```
def LogMirror := extend: actor.defaultMirror with: {
  def invoke(rcvr, selector, args) {
    system.println("invoked "+selector+" on "+self.base);
    super^invoke(rcvr, selector, args); // perform default behaviour
  }
}
```

To facilitate the development of mirror objects which require only small changes with respect to the default language semantics, the actor mirror contains a prototypical mirror object named the `defaultMirror` which encapsulates AmbientTalk's default metaobject protocol. The `defaultMirror` makes the native metaobject protocol implementation explicitly accessible while keeping it encapsulated behind the protocol's interface[3]. The `LogMirror` leaves all metalevel operations intact save `invoke`.

A mirror can refer to the object with which it is causally connected by invoking `self.base`. However, the above mirror is but a prototype implementation: it is not yet causally connected to any object. A mirror object can only be absorbed by the interpreter when a mirage object is defined to be explicitly mirrored by that mirror object. The code excerpt below redefines the `Point` prototype from section 4.2.1 as a mirage, whose behaviour is now defined by the `LogMirror`:

```
def Point := object: {
  /* the original implementation */
} mirroredBy: LogMirror;
```

The **`object:mirroredBy:`** language construct associates a mirage base level object with its corresponding mirror metaobject[4]. When the mirage is constructed, it becomes causally connected with its mirror. The latter then effectively becomes absorbed by the interpreter. For example, evaluating `Point.new(1,1).distanceToOrigin()` now triggers the custom `invoke` method defined by the `LogMirror`. The details on how the causal connection between a mirage and its associated mirror is constructed and maintained can be found elsewhere [MVTT07].

In short, previous mirror-based metalevel architectures provided only limited support for intercession. The contribution of mirages is that they combine a mirror-based architecture with intercession, thus making the architecture suitable to experiment with novel language constructs while maintaining the beneficial software engineering properties of a mirror-based design.

5.2.3 First-class References as Mirages

Now that AmbientTalk's metalevel architecture has been introduced, we demonstrate how it can be employed to create custom object references. In chapter 8, we will show how ambient references can be implemented as such custom object references. In

[3]The `defaultMirror` exists only for convenience. It is not necessary for the purposes of avoiding an infinite meta-regress [Smi84]. An infinite regress would occur if a mirage is itself mirrored by a mirage, and so on. To avoid infinite regress, a mirage must eventually be mirrored by a concrete (non-mirage) AmbientTalk object.

[4]It is an object-based equivalent of the `:metaclass` option in the CLOS metaobject protocol which associates a class with a custom metaclass [Pae93].

this section, we implement the (metalevel) behaviour common to all kinds of custom *eventual* references in AmbientTalk itself. In the following section, we show how futures can be implemented as such custom eventual references.

We define the behaviour common to all custom eventual references in a trait in order to promote its reusability. The trait can then be "mixed into" different language constructs.

```
def TEventualRef := object: {
  // disallow synchronous access to the object
  def invoke(rcvr, selector, args) {
    raise: XIllegalOperation.new(
      "Cannot synchronously invoke "+selector+" on " + self);
  };
  def receive(msg) {
    if: (is: msg taggedAs: MetaMessage) then: {
      self <+ msg; // process meta-message myself
    } else: {
      self.intercept(msg); // must be provided by the composite
    }
  };
  // disallow meta-level operations not applicable on eventual refs
  def newInstance(initargs) { raise: XIllegalOperation.new(...) };
  def addMethod(mth) { raise: XIllegalOperation.new(...) };
  ...
};
```

The AmbientTalk metaobject protocol reifies synchronous method invocation separately from asynchronous message reception. In particular, method invocation and delegation are reified via a mirror's `invoke` method, while asynchronous sends are reified via its `receive` method. The above trait makes use of this fact to disallow synchronous invocation on its associated mirage. Hence, an eventual reference enforces asynchronous access to its principal, which is its most distinctive feature in comparison to regular object references.

If an incoming asynchronous message is annotated with `@MetaMessage`, it is processed by the *mirror* itself, i.e. the message is resolved at the metalevel. If the incoming asynchronous message is not annotated with `@MetaMessage`, its processing is deferred to the composite using the `TEventualRef` trait. The method `intercept` is thus part of the required interface of the trait.

Finally, the `TEventualRef` trait provides an implementation for all metalevel operations that cannot be applied to eventual references. These implementations, like the implementation for `invoke` signal the erroneous operation to the caller by means of an exception. In effect, the only operations supported by eventual references are asynchronous message sending and equality testing.

A custom eventual reference can now be represented as a mirage whose mirror uses the above `TEventualRef` trait. The general pattern is as follows:

```
object: {
  /* empty proxy object */
} mirroredBy: (
  extend: actor.defaultMirror with: {
    import TEventualRef;
    def intercept(msg) {
      /* implement reference behaviour */
```

```
    }
  })
```

At the base level, the sole purpose of the above object is to act as an empty "proxy" (representing the object reference). Because it is a mirage, its semantics can be altered at the metalevel. Its asynchronous message reception behaviour is replaced by the definition provided by the `TEventualRef` trait. However, since this definition is incomplete (it depends on the trait's required interface), the mirror must additionally provide an implementation for the `intercept` method. The implementation of this method effectively encodes the behaviour of the object reference. The following section discusses a concrete instance of such an implementation, to wit the message passing behaviour of futures.

5.2.3.1 Case Study: Futures

Futures are not built into the AmbientTalk language. They have been implemented entirely by means of the metalevel architecture. We outline this implementation here to serve as a case study in applying the metalevel architecture to a concrete language construct. In chapter 8, we employ exactly the same techniques to implement ambient references. The integration of futures in AmbientTalk comprises two parts. First, we implement a future abstract data type as a special kind of object reference. Second, we adapt the actor metaobject protocol such that futures are automatically attached to outgoing asynchronous messages.

Recall from section 4.3.4 that futures are first-class placeholders to which (asynchronous) messages can be sent. Futures either forward the message to the value with which they are resolved or otherwise buffer the message in a message queue for as long as they remain unresolved. This semantics can be directly encoded by representing a future as an object reference, as follows:

```
def makeFuture() {
  def theFuture := object: { } mirroredBy: (
    extend: actor.defaultMirror with: {
      import TEventualRef; // use metalevel behaviour of eventual refs

      def state := UNRESOLVED;
      def resolvedValue := nil;
      def inbox := [];
      // implementation for the required method of the TEventualRef trait
      def intercept(msg) {
        if: (state == RESOLVED) then: {
          resolvedValue <+ msg; // forward the message
        } else: {
          inbox := inbox + [msg]; // buffer the message
        }
      };
      def resolveWithValue(value) { ... };
      def subscribe(observer) { ... };
      ...
  });
  def theResolver := object: {
    def resolve(val) {
      theFuture<-resolveWithValue(val)@[MetaMessage,OneWayMessage]
    };
```

```
    def ruin(exc) { ... };
  };
  [ theFuture, theResolver ]
}
```

The future's mirror is either in an unresolved or in a resolved state, as indicated by its `state` field. For didactic purposes, we omit the implementation of ruining futures with exceptions. Initially, the future is unresolved. The transition from an unresolved to a resolved state occurs when a `resolveWithValue` message is sent to the future's mirror. The `theResolver` object (acting as the second return value of `makeFuture`) implements `resolve` by sending a corresponding metamessage to the future object's *mirror*. This is accomplished by annotating the message with `@Metamessage`. Future-type message passing is disabled for messages annotated with `@OneWayMessage`. This annotation is useful if no return value is required for an asynchronous send. More fundamentally, the `resolveWithValue` metamessage sent to the future mirror requires this annotation to avoid an infinite loop. Without this annotation, the resolution of one future would require the creation (and resolution) of another future, whose resolution requires another future, and so on.

In addition to the `resolveWithValue` method, the future mirror also extends the default metaobject protocol with a `subscribe` method which can be used to register an observer with the future (normally by means of **`when:becomes:`**), to be notified when the future becomes resolved. The (simplified) definitions of these methods are shown below[5].

```
def observers := [];
def resolveWithValue(value) {
  if: (state == UNRESOLVED) then: {
    state := RESOLVED;
    resolvedValue := value;
    inbox.each: { |msg| value <+ msg };
    inbox := [];
    observers.each: { |obs| obs<-notifyResolved(value)@OneWayMessage };
    observers := [];
  }
};
def subscribe(observer) {
  // if future has already been resolved, notify immediately
  if: (state == RESOLVED) then: {
    observer<-notifyResolved(resolvedValue)@OneWayMessage;
  } else: {
    observers := observers + [ observer ];
  }
};
```

A future can be resolved only once. When it is resolved, all messages previously accumulated in the `inbox` are forwarded to the resolved value. Similarly, all registered observers are notified. An observer that subscribes on a resolved future is notified immediately. Otherwise, it is stored in a table until the future becomes resolved.

At this point, futures have been introduced as a new data type into the interpreter. However, we have yet to define how they are integrated into the actor's message sending

[5] We omit the case where a future is resolved with another future. In other words, we assume the `value` parameter of the `resolveWithValue` method not to be a future.

protocol. The following code excerpt shows how the language module defining futures refines these methods to attach a future to an outgoing asynchronous message.

```
actor.install: (extend: actor with: {
  def createMessage(sel,args,annotations) {
    // first, create a regular message
    def msg := super^createMessage(sel,args,annotations);

    if: (is: msg taggedAs: OneWayMessage) then: {
      msg;  // if msg is one-way, do not attach a future
    } else: {
      extend: msg with: { // attach a future to the message
        def [future, resolver] := makeFuture();
        // this method is invoked upon reception
        def process(receiver) {
          // delegate to actually invoke the method
          def result := super^process(receiver);
          resolver.resolve(result);
          result;
        }
      }
    };
    def send(receiver, msg) {
      def result := super^send(msg);
      if: !(is: msg taggedAs: OneWayMessage) then: {
        msg.future; // async send returns the future
      } else: {
        result; // nil, by default
      }
    }
  }
})
```

The code excerpt shows the installation of a custom actor mirror which overrides the default implementations of `createMessage` and `send` (cf. section 5.2.1.2). The `createMessage` method is specialised to return future-type messages. These are asynchronous messages extended with a `future` field and whose `process` method is overridden. The overridden `process` method is invoked when the message arrives at its receiver object. Its implementation ensures that the future attached to the message is automatically resolved with the return value of the invoked method. Finally, the actor's asynchronous message sending semantics is modified by overriding `send`. A future-type message send returns the future attached to the message rather than the default **`nil`** value.

5.2.4 Stratified Object References

From an implementation point of view, custom object references are nothing but placeholders or proxies for other objects. Implementing proxy objects by means of reflection as is done in AmbientTalk has several advantages over the more traditional method such as e.g. using dynamic proxies in Java [GJSB05] or the `doesNotUnderstand:` protocol in Smalltalk [GR89]. Object references are objects whose message reception semantics deviate from those of normal objects. By implementing them as mirages, we can change the *actual* message reception semantics which the language attributes to that

object, rather than using hooks provided by the language, as is done traditionally. As a result:

- base level code is not affected by the metalevel protocol. Because base and metalevel are cleanly stratified, the methods defined at the metalevel will not interfere with base level messages. For example, imagine an application involving newsfeeds to which users may subscribe (by invoking a newsfeed object's `subscribe` method). Assume that `feed` is a future to be resolved with a newsfeed object. When evaluating `feed<-subscribe(user)`, we can rest assured that the `subscribe` message will not accidentally be regarded as part of the metaobject protocol of the future. Contrast this with e.g. Smalltalk's approach to implementing proxies using `doesNotUnderstand:`. If `subscribe` is a method of the proxy object, then `doesNotUnderstand:` will simply not fire for this method, because the proxy object *does* understand the message. Even worse, if an application defines its own `doesNotUnderstand:` method (e.g. on an object representing a translator), this method may be erroneously invoked by the Smalltalk virtual machine.

- base level code does not affect the metalevel protocol. For example, in the implementation of **`when:becomes:`**, the observer object is registered with the future by invoking `future<-subscribe(observer)@MetaMessage`. The `@MetaMessage` annotation is crucial here, as the future's `subscribe` method is defined on the future's mirror, rather than on the base-level proxy object itself. As a result of this stratification, metalevel messages cannot be mistaken for base level messages: the implementation of **`when:becomes:`** can rest assured that the `subscribe` message triggers the metaobject protocol and *not* the `subscribe` method of a newsfeed object simply because the base application incidentally happens to implement a method with the same name.

In short, representing object references as mirages is a robust implementation technique. The key to this robustness lies in the principle of stratification which our architecture achieves through its mirror-based design. In the following section, we discuss a different metalevel engineering technique, one which allows AmbientTalk objects to interoperate with the underlying Java Virtual Machine.

5.3 Linguistic Symbiosis with the JVM

In this section, we describe how AmbientTalk objects can access objects in the underlying Java Virtual Machine (JVM) by means of a technique known as a *linguistic symbiosis* [GWDD06]. Such a mechanism enables us to access Java libraries from within AmbientTalk. In section 8.7.3, we will describe how the implementation of ambient references uses this mechanism to access the M2MI library (previously discussed in section 3.3.7.4). Not only does the symbiosis allow access to Java libraries, it allows AmbientTalk objects to interoperate with other languages as well, as long as these languages' data can be compiled into JVM objects.

While constructing a linguistic symbiosis with the JVM is not novel – as can be witnessed by the vast number of dynamic languages which provide access to the JVM (e.g. JRuby, Jython, JScheme, LuaJava, JPiccola, ...) – the AmbientTalk/JVM symbiosis goes further by also reconciling both systems' fundamentally different concurrency models with one another. While AmbientTalk is entirely built on an event-driven

actor-based architecture, the JVM employs a traditional multithreaded model. Composing such models is not straightforward: the event-driven model enforces certain concurrency constraints (cf. section 4.3.1) which could be violated by the JVM's multithreaded concurrency if JVM objects access AmbientTalk objects without proper provisions. Linguistic symbiosis acts as a mediator between the two systems to enforce a safe yet transparent composition.

5.3.1 Linguistic Symbiosis

Our model for explaining the AmbientTalk/JVM linguistic symbiosis is based on that of *inter-language reflection* [GWDD06], which is itself based on an open design of object-oriented languages [Ste94b]. In the inter-language reflection model, a linguistic symbiosis consists of:

- a **data mapping** which ensures that data in one language looks like data in the other language, such that the symbiosis becomes as syntactically transparent as possible. For example, it is desirable that JVM objects are equally represented as objects in AmbientTalk, such that messages can be sent to objects regardless of their native language.
- a **protocol mapping** between the metalevel representation of both languages' data. For example, both AmbientTalk and JVM objects communicate by sending messages, but AmbientTalk is dynamically typed while the JVM is statically typed and exploits type overloading during method lookup. A proper symbiosis needs to map these message sending protocols onto one another.

AmbientTalk has been implemented on top of the JVM. Because of this, the JVM plays two roles: it is both the symbiont system and the implementation host of AmbientTalk (and hence of the linguistic symbiosis itself). Figure 5.2 illustrates the different objects that play a part in the AmbientTalk/JVM symbiosis, according to the implementation model of inter-language reflection [GWDD06]. AmbientTalk objects are implemented as JVM objects. This is illustrated by means of the "represents" relationship. To enable symbiosis, additional objects are required which denote the *appearance* of objects from one language in the other language. At the implementation level, such appearances are implemented as *wrapper* objects, which wrap an object from a different language and which perform the protocol mapping which translates between the semantics of the symbiont languages. Below, we summarise the data and the protocol mappings of the symbiosis. A full account of the AmbientTalk/JVM symbiosis can be found elsewhere [VMD08].

Mapping objects from JVM to AmbientTalk AmbientTalk's data mapping is similar to that of other dynamic languages implemented on top of the JVM. In a nutshell, JVM values are either mapped to primitive AmbientTalk objects where possible (for example, a `boolean` is mapped onto the `true` or `false` prototype in AmbientTalk) or are otherwise represented as regular AmbientTalk objects (the "AT wrapper for JVM Object" in figure 5.2). The methods (resp. fields) of this wrapper object correspond to the non-static methods (resp. fields) of the wrapped JVM object. The parent object to which the AmbientTalk wrapper delegates is a wrapper for the *class* of the wrapped JVM object. The methods of this class wrapper correspond to the static methods and fields defined on the wrapped JVM class.

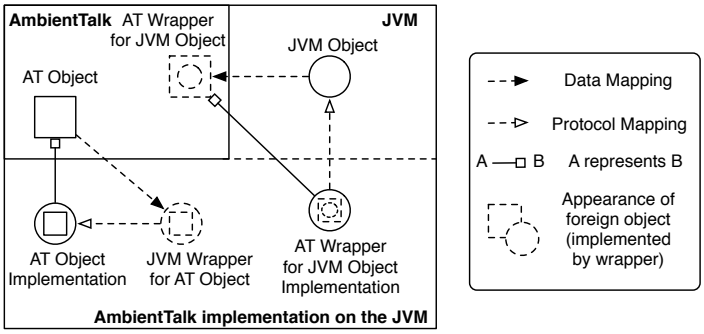

Figure 5.2: Entities in the linguistic symbiosis between AmbientTalk and the JVM.

Mapping messages from JVM to AmbientTalk When an AmbientTalk object invokes a method on the wrapper of a JVM object or class, the wrapper converts the invocation into a JVM method invocation by means of the `java.lang.reflect` API. If the method is overloaded, the symbiosis attempts to resolve overloading automatically by inspecting the type and number of actual arguments. If automatic overloading fails, the AmbientTalk programmer is forced to pass type information in the call (how this is done is explained elsewhere [VMD08]).

Mapping objects from AmbientTalk to JVM Primitive AmbientTalk objects are mapped to primitive JVM values when possible (e.g. an AmbientTalk integer is mapped to an `int`). If such a conversion is not possible, the AmbientTalk/JVM symbiosis can represent the AmbientTalk object as a regular JVM object, but only if the static type of the variable which is to hold that JVM object is an interface type[6]. Such wrapped AmbientTalk objects are represented by the "JVM wrapper for AT Object" in figure 5.2.

Mapping messages from AmbientTalk to JVM When a JVM object invokes a method on such a wrapper, the wrapper transforms the JVM method invocation into an AmbientTalk invocation by means of the `invoke` metalevel operation defined on the wrapped object's mirror (cf. section 5.2.1.1). Since AmbientTalk does not support overloaded methods, the method to be invoked can always be uniquely identified.

To illustrate how AmbientTalk objects can be passed to JVM objects, consider the following archetypical pattern of registering a listener object on a button GUI widget to act upon event notifications (written in AmbientTalk, but using the actual Java AWT framework)[7]:

```
def Button := jlobby.java.awt.Button;
def b := Button.new("Click me");
b.addActionListener(object: {
  def actionPerformed(actionEvent) {
    system.println("The button was pressed");
  }
})
```

[6]This limitation is due to the fact that the implementation uses the JVM's dynamic proxies to implement a wrapped AmbientTalk object. Such dynamic proxies can only be instantiated for interface types [GJSB05].

[7]The `jlobby` object is a special AmbientTalk object whose fields correspond to packages and classes available in the underlying JVM.

The above code demonstrates that an instance of the *Java* class `java.awt.Button` appears as an AmbientTalk object `b`. It also demonstrates that AmbientTalk text (`" Click me"`) is transparently converted by the symbiosis into a `java.lang.String`. The `addActionListener` method defined on instances of the Java `Button` class takes a parameter of type `ActionListener` as its argument, which is an interface type. As a result, it is allowed to pass any AmbientTalk object to this method; the object is not even required to implement all declared interface methods, although the anonymous object passed in the above code does properly implement the `ActionListener` interface. The symbiosis transparently wraps the AmbientTalk object into a wrapper implementing the `ActionListener` interface. The AWT framework will invoke the `actionPerformed` method on the wrapper whenever the button is pressed. A discussion on the concomitant threading issues is postponed until the next section.

5.3.2 Composing Threads with Actors

The linguistic symbiosis described above enables AmbientTalk objects to invoke methods on JVM objects and vice versa. Let us consider the resulting concurrency issues in both directions:

- When an AmbientTalk object ao invokes a method on a JVM object jo, it is the actor owning ao that will execute the method of jo. This does not fundamentally violate any concurrency properties of jo, as any JVM object that is to be manipulated by multiple threads of execution must be made thread-safe using the JVM's concurrency control primitives. However, the AmbientTalk programmer must be wary of the fact that, by making an actor execute JVM bytecode, the actor has to play by the rules of shared-state concurrency. Hence, the actor may need to acquire locks and may become blocked. As a result, the deadlock-freedom of the pure event loop model can no longer be guaranteed.
- When a JVM object jo invokes a method on an AmbientTalk object ao, we cannot simply allow the JVM thread to execute the method of ao. This would violate the Exclusive State Access property of the event loop model postulated in section 4.3.1. For example, if the thread executing Java AWT event notifications were allowed to execute the `actionPerformed` method of the anonymous AmbientTalk `ActionListener` object, the thread would operate concurrently on the same data as the actor owning the `ActionListener`. Since AmbientTalk does not have any thread synchronisation constructs (because it presumes the Exclusive State Access property) this could result in race conditions on the actor's state.

In order to safeguard the concurrency properties of AmbientTalk's event loop model, the AmbientTalk/JVM symbiosis automatically synchronises multithreaded access to AmbientTalk objects. We distinguish between two ways in which JVM objects may access AmbientTalk objects: via method invocation or via event notification. Both are discussed below.

5.3.2.1 Method Invocation

As previously remarked, when an AmbientTalk object (e.g. the `ActionListener` in the above example) is passed as an argument to a JVM object, the AmbientTalk/JVM symbiosis implicitly wraps this object in a JVM object implementing the required interface.

Figure 5.3 illustrates the sequence of events occurring when a JVM object invokes a method on the wrapper. Conceptually, this wrapper object represents an eventual reference to the AmbientTalk object because it enforces asynchronous access: whenever a JVM object invokes a method on the wrapper, the method invocation is transparently converted by the wrapper into an asynchronous AmbientTalk message send. This message send is properly enqueued in the message queue of the actor owning the wrapped AmbientTalk object. As such, it will be processed serially, and the above invocation of the `actionPerformed` method is treated simply as if it were triggered by a regular AmbientTalk object.

Of course, while the above solution satisfies AmbientTalk's event loop model, it does not fit with the JVM's multithreaded view on the object world: to the JVM thread, a method invocation should be executed immediately and may return a result or raise an exception. Therefore, the wrapper object transparently suspends the JVM thread on a synchronisation barrier until the AmbientTalk method has been processed. Any return value (or raised exception) is signalled to the barrier object which at that point resumes the JVM thread.

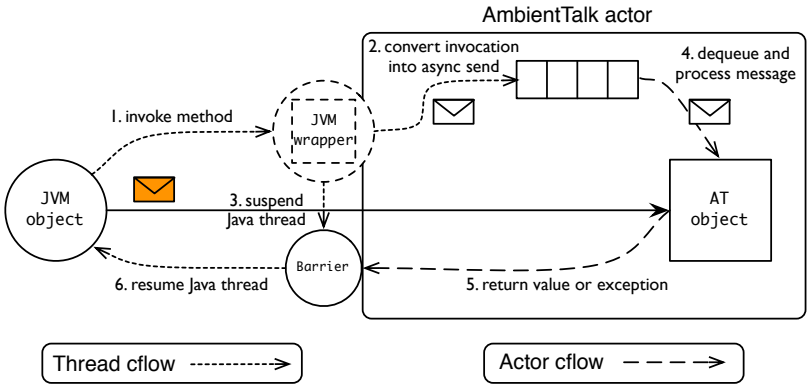

Figure 5.3: Mediating between JVM invocations and AmbientTalk messages.

5.3.2.2 Event Notification

The JVM's thread-based concurrency model is sometimes inappropriate even for Java applications. Many interactive applications (e.g. games, user interface frameworks) or discrete-event simulations require an event-driven approach. In Java, event-driven programming is not supported directly, but an event-driven style can be adopted by structuring code around an event loop framework. The Java AWT and Swing toolkits are the quintessence of such an approach.

In event-driven Java frameworks, asynchronous message sends are *second-class* language abstractions. They are represented indirectly in terms of synchronous method invocations on so-called *listener* objects. The documentation of most event-driven Java frameworks specifies that such methods must return as soon as possible, and should preferably only schedule tasks for later execution instead of calling other methods. Our symbiosis can detect such indirect (second-class) asynchronous message sends and map them onto *actual* asynchronous sends in AmbientTalk, without synchronising the JVM thread on the return value. This is highly desirable because it guarantees the

responsiveness of the event-driven Java framework.

Reconsider the anonymous AmbientTalk object registered as an `ActionListener` on an AWT `Button`. When the AWT event loop invokes the `actionPerformed` method on the AmbientTalk object, this implicitly indicates an event notification, which is conceptually asynchronous. However, applying the synchronisation described in the previous section would suspend the AWT event loop until the AmbientTalk actor has actually processed the `actionPerformed` method. In this case, the synchronisation is superfluous because the method represents an event notification, which does not need to return any result. The AmbientTalk/JVM symbiosis detects this and essentially converts the Java listener method invocation into a pure asynchronous AmbientTalk send by dropping steps 3, 5 and 6 in figure 5.3.

How does our symbiosis distinguish method invocations from event notifications? In order to qualify as an event notification, the invoked JVM method must **a)** belong to an interface extending the `java.util.EventListener` interface[8], **b)** have a `void` return type and **c)** have an empty `throws` clause. Because the `actionPerformed` method adheres to all three of these criteria, an invocation of this method on an AmbientTalk listener will not block the AWT framework, guaranteeing the overall responsiveness of the system. Also, AmbientTalk methods invoked in this way must not be restructured such that they should "return quickly".

To summarise, linguistic symbiosis is a mechanism that enables AmbientTalk and JVM objects to transparently communicate with one another. While this mechanism is common among dynamic languages implemented on top of the JVM, the novelty of the AmbientTalk/JVM linguistic symbiosis is that it enforces a safe composition of AmbientTalk event loop actors with JVM threads. In the following section, we discuss an embedding of AmbientTalk in Java which relies on the linguistic symbiosis outlined here.

5.3.3 Embedding AmbientTalk in Java

In the previous section, we have primarily regarded linguistic symbiosis as a mechanism for AmbientTalk objects to reuse existing Java libraries. However, it is equally viable for a Java programmer to embed AmbientTalk components into an existing Java application. This allows the Java programmer to benefit from e.g. AmbientTalk's support for distributed programming. In this section, we describe how an AmbientTalk interpreter can be *embedded* in a Java application.

As a concrete example, we illustrate how AmbientTalk unit tests can be combined with unit tests written in the JUnit unit testing framework, allowing a Java developer to integrate both unit tests written in AmbientTalk and written in Java in a consistent testing framework. In AmbientTalk, a unit test is an object whose methods are prefixed with `test`. All unit test objects delegate to the prototypical unit test object, which contains reflective code to invoke all test cases. This object also implicitly implements the `junit.framework.Test` Java interface type:

```
def UnitTestPrototype := object: {
  def testMethods; // a table of method metaobjects
  def init() {
    testMethods := retrieveTestMethods(self);
  };
```

[8] It is a convention of Java frameworks that classes whose instances represent event listeners implement (a subtype of) this empty interface.

```
    def countTestCases() { testMethods.length };
    def run(reporter) {
      reporter.startTest(self});
      testMethods.each: { |method| /* perform the test */ };
      reporter.endTest(self});
    };
  }
```

Now consider a `TestSuite` that is composed of both unit tests written in Java and unit tests written in AmbientTalk. All unit tests uniformly implement the `Test` interface. In order to incorporate an AmbientTalk unit test into the test suite, a Java object representing the AmbientTalk interpreter must be instantiated. As noted in section 4.3.2, every interpreter automatically creates a single actor which is responsible for evaluating AmbientTalk code.

```
public static void main(String[] args) {
  AmbientTalkInterpreter evaluator = new AmbientTalkInterpreter(...);
  Test exampleATTest =
    (Test) evaluator.evalAndWrap(new File("mytest.at"), Test.class);
  Test exampleJavaTest = new MyJavaUnitTest();
  TestSuite suite = new TestSuite();
  suite.addTest(exampleATTest);
  suite.addTest(exampleJavaTest);
  junit.textui.TestRunner.run(suite);
}
```

The ellipsis in the code snippet above abstracts from various configuration parameters which can be passed to the interpreter (e.g. which initialisation code to run, the directory path to AmbientTalk libraries, etc.). We further assume that the file `mytest.at` contains an AmbientTalk unit test. The AmbientTalk interpreter provides a method `evalAndWrap` which takes AmbientTalk code (in the form of a file or a string), evaluates this code and then wraps the resulting AmbientTalk object in a proxy implementing the given interface (cf. section 5.3.2.1).

A `TestRunner` executes the test suite by sequentially invoking each unit test's `run` method. This execution is performed by a Java application thread. Because an embedded AmbientTalk unit test should be run inside its owning actor, an invocation of the `run` method on the wrapped AmbientTalk unit test is transformed into an asynchronous message to be processed by the actor owning `exampleATTest` (as described in section 5.3.2.1). The JVM thread is transparently suspended by the wrapper until the AmbientTalk actor has executed the AmbientTalk unit test. The thread is suspended because the `run` method is not an event notification (the `Test` interface to which it belongs does not extend the `EventListener` interface). The JUnit test runner expects the unit test to run to completion before executing the next test or terminating.

The techniques put forward in the previous section greatly simplify the embedding of AmbientTalk code within Java applications. Since the `evalAndWrap` method of the interpreter object automatically creates a wrapper acting as a mediator between Java and AmbientTalk, neither AmbientTalk nor Java programmers must deal with the threading issues arising from embedding the event-driven AmbientTalk language into the multithreaded Java language.

5.4 Conclusion

In this chapter, we have presented metalevel engineering techniques in AmbientTalk. In particular, we have presented AmbientTalk's support for first-class messages, its mirror-based metalevel architecture and its support for linguistic symbiosis with the JVM. We discern the following scientific contributions:

- Through AmbientTalk's support for first-class messages, higher-order messages can be directly expressed in the language.
- AmbientTalk's reflective architecture reconciles mirrors with intercession. This marriage is accomplished via what we call a *mirage*: an immaterial object whose semantics is entirely described by a custom mirror object [MVTT07].
- We have shown how mirages can be applied to implement custom object references. The stratification of AmbientTalk's mirror-based architecture furthermore ensures that such object references avoid name clashes between base and metalevel message names.
- We have explained how AmbientTalk's futures can be implemented as custom object references and how they can be integrated into the asynchronous message sending protocol of actors.
- We have described a linguistic symbiosis between AmbientTalk and the Java Virtual Machine. This linguistic symbiosis is novel in that it allows for the transparent yet safe composition of threads with event loops [VMD08].

With both the AmbientTalk language and its reflective architecture properly introduced, the stage is set for describing ambient references. In the previous chapter, we have extensively described how AmbientTalk's far references are appropriate for modelling point-to-point communication which is resilient to partial failures. In the following chapter, we describe communication patterns that are inappropriate to express by means of far references. We will argue why and how ambient references complement far references to express coordination between objects which are distributed across a MANET. The concepts introduced in this chapter will only resurface later in chapter 8, when discussing how ambient references have been implemented reflectively in AmbientTalk.

Chapter 6

Ambient References

With AmbientTalk now introduced, we can use it as a technical platform on top of which ambient references are conceived. We start this chapter with a discussion on why AmbientTalk's built-in language constructs fail to provide the programmer with language constructs that deal with a number of the coordination criteria from section 3.2. Subsequently, we define two novel kinds of language abstractions: *ambient references*, which enable communication with a volatile group of proximate objects and *anonymous far references*, which make far references decoupled in space. While ambient references are but a single language abstraction, they enable messages to be delivered using a wide range of "delivery policies". These policies enable ambient references to express widely different interaction patterns among distributed processes within a single unifying object-oriented framework.

6.1 Motivation

Before describing the details of the ambient reference abstraction, we highlight the need for such abstractions in AmbientTalk. At first sight, it might seem that AmbientTalk is sufficiently high-level to deal with space-decoupled communication and arity decoupling directly. After all, the built-in service discovery primitives allow one to discover remote objects based on an abstract description (a type tag). In this section, we show by example that explicitly using the service discovery mechanism to communicate with remote objects can quickly lead to intricate code. This forms the main motivation for including ambient references in AmbientTalk: they allow the programmer to abstract from this intricate code such that they can better focus on application-semantic concerns.

6.1.1 Roaming

In a mobile ad hoc network, a single conceptual service may be offered by a multitude of different devices ("access points"). In a nomadic network, these access points may even cooperate using the network's infrastructure. In such a context, a mobile client of the service should be able to abstract from the different access points. We want to prevent code running on the mobile client from having to explicitly manage *rebinding* of service references based on their current connectivity and availability. This layer of abstraction is called *roaming* in the global system for mobile communications (GSM)

network, where it is used to transparently "reconnect" a cellular phone to a different base station when it approaches the boundary of its cell. In this context, such a "reconnection" is known as a *handover*. We give another example of roaming below.

6.1.1.1 Example: Location Service

Consider a location tracking service in a nomadic network. The idea is for mobile clients to periodically send their latest location information (e.g. GPS coordinates) to any nearby location tracking service. This service can then be queried either by other mobile clients or maybe even stationary desktop computers connected to the network. By querying the service, clients can retrieve the latest location of other mobile clients. We assume that the location tacking services themselves are connected via the nomadic network's infrastructure such that they can share or replicate the location data which they have gathered.

6.1.1.2 Implementation in AmbientTalk

We sketch a prototypical implementation of the mobile client in the above scenario. The mobile client's task is quite straightforward: it periodically sends its current location information to a nearby location tracking service. Of course, the code will have to tackle some border cases. When the client is first started, no location tracking service may be present. Similarly, while it is moving, the location tracking service with which it is communicating may no longer be in communication range. This requires the client to, at some point, *rebind* its connection to another location tracking service.

A second issue for the mobile client is how to treat location information that it cannot send to the remote service when no access point is available. The answer to this question is very application-specific. Possible semantics may be that location information is dropped while being disconnected. This is acceptable if the location service only keeps track of the latest location information anyway. If, on the other hand, the location service also keeps track of a client's path of movement (i.e. all locations a client has visited for a certain period of time), it is necessary to *buffer* location information until a new access point is available. For the sake of the arguments, we assume the latter semantics.

The following code snippet defines a function `createConnection` which, given a client identification, returns a conceptual connection to a nearby location tracking service. Location updates can then be submitted to the server by invoking the connection object's `updateLocation` method.

```
deftype LocationService;
def createConnection(clientId) {
  def availableServices := Set.new();
  def serviceProvider := nil;
  def locationBuffer := Vector.new();

  whenever: LocationService discovered: { |svc|
    availableServices.add(svc);
    if: (serviceProvider == nil) then: {
      serviceProvider := svc;
      locationBuffer.each: { |loc|
        serviceProvider<-submitLocation(clientId, loc);
      };
      locationBuffer.clear();
```

```
    };

    when: svc disconnected: {
      availableServices.remove(svc);
      if: (serviceProvider == svc) then: {
        serviceProvider := nil;
        if: !(availableServices.isEmpty) then: {
          serviceProvider := availableServices.selectRandom();
        }
      }
    }
  };

  object: {
    def updateLocation(location) {
      if: (serviceProvider == nil) {
        locationBuffer.append(location);
      } else: {
        serviceProvider<-submitLocation(clientId, location);
      }
    }
  }
}
```

We assume that location tracking services are exported and identified by means of the `LocationService` type tag. They implement a `submitLocation` method to store the coordinates of a client. All available location services are stored in the `available-Services` set. The connection object is conceptually in two states: if the `service-Provider` variable is set to **`nil`**, there is no service provider. Otherwise, the variable indicates the current service provider in use. Note how, in the `updateLocation` method, a distinction between these states is made to determine whether or not to buffer the location update. If a connection with a server is available, the location information is sent, along with the client's identification.

The more tedious part of the above code is the management of keeping track which services are available and which are not. The problem is that AmbientTalk provides no means to abstract over a set of available services. Rather, this set – represented here by the `availableServices` variable – has to be managed explicitly. Services are added upon discovery and removed upon disconnection. These events have to be trapped manually. Furthermore, these events generate "state transitions" that have to be dealt with: if the service being discovered is the only one available, previously buffered location information must be flushed to the server. Oppositely, if the service that disconnected was the client's current access point, the `serviceProvider` variable has to be manually rebound.

The above example shows that an AmbientTalk programmer must himself implement the layer of abstraction that supports roaming. This implies that the above code is a generic pattern that AmbientTalk programmers will find themselves writing every time they want their far references to rebind automatically to different remote objects. The pattern cannot be easily modularised by means of traditional abstraction mechanisms because different instances of the pattern deal with different kinds of services, messages and – as stated previously – multiple semantics regarding the treatment of messages sent to disconnected services. In the following section, we address another communication pattern which cannot easily be abstracted from in AmbientTalk.

6.1.2 One-to-many Communication

AmbientTalk in itself does not provide any means to express arity-decoupled, one-to-many communication. At first sight, this does not seem to pose much problems. After all, it is easy enough to iterate over a table of remote objects, and to send each of those objects a message, as in `group.each: {|o| o<-m()}`. However, this simple pattern does not deal with many of the issues that arise when communicating with a *volatile* group of objects. For example, how does one deal with objects joining or leaving the group *while* performing the "broadcast", and how should the multiple return values of the "broadcasted" message be aggregated? We discuss these issues in the context of a concrete application, introduced in the following section.

6.1.2.1 Example: Voting in an Ad Hoc Network

Consider a simple application for organising a poll among peers. One peer may start a poll, asking the opinion of nearby peers on a given subject. Nearby peers get notified of the poll, provide their vote and submit the result to the originator of the poll. After giving all peers a chance to vote, the originator can process the results and may even broadcast them to all nearby peers again. On-the-fly voting in an ad hoc network can be useful in e.g. a disaster relief situation. Paramedics, firemen and rescue workers could use it to decide on which actions to take next. In a more playful setting, voting can be used in a team-based mobile multiplayer game to decide on a group strategy among proximate players of the same team. We will use the latter as the context for a prototypical implementation in AmbientTalk.

6.1.2.2 Implementation in AmbientTalk

In order to implement voting between nearby team players, the very first step is to build a collection containing all nearby objects representing those players. AmbientTalk by itself does not provide any explicit means for such "groups" of proximate objects, requiring the programmer to create this abstraction himself. The code for implementing this is remarkably similar to the management of the service providers in the location service example explained earlier. In this example, it does not suffice to keep track of all services of a certain *type*, we also have to ensure that the services represent players *of the same team*. This is a dynamic attribute of the service that has to be retrieved and tested against explicitly in the code. We assume players in the multiplayer game are service objects exported under the `Player` type tag.

```
def nearbyTeamPlayers := Set.new();

whenever: Player discovered: { |player|
  when: player<-getTeam()@Due(TIMEOUT) becomes: { |team|
    if: (team == myTeam) then: {
      nearbyTeamPlayers.add(player);
      when: player disconnected: {
        nearbyTeamPlayers.remove(player);
      }
    }
  }
};
```

Note that a test for a dynamic attribute, like the player's team, is encoded by explicitly querying the player for his team. The asynchronous send is bounded in time by means of a timeout, which is application-specific: if the call is not replied to within the timeout period, the player is further ignored.

The above piece of code provides other components of the multiplayer game with a representation of all nearby team players. It is intended to be reused by different components of the multiplayer game. The module implementing votes is one of them. Its implementation is described below.

```
def broadcastVote(poll, maxVoteTime) {
  def [future,resolver] := makeFuture();
  def receivedVotes := Map.new();

  nearbyTeamPlayers.each: { |player|
    when: player<-askToVote(poll)@Due(maxVoteTime) becomes: { |vote|
      receivedVotes.put(player, vote);
    }
  };
  when: maxVoteTime elapsed: {
    resolver.resolve(receivedVotes);
  };

  future;
};
```

The `broadcastVote` function takes as parameters a textual description of the vote `poll`, and a time period `maxVoteTime` indicating how long incoming results should be accepted. It returns a future which will be resolved when the time period has elapsed[1]. The value of the future will be a mapping associating players with their corresponding vote. The caller of the `vote` function can then e.g. calculate the winning vote and/or broadcast this vote.

Note that the above code has to deal with a number of issues explicitly. First, a broadcast to all nearby players is represented implicitly by iterating over the previously constructed list of `nearbyTeamPlayers` and sending them an `askToVote` message. Not only is this broadcast implicit, it is also inefficient in terms of network load, as the broadcast is represented in terms of multiple low-level messages to be put on the network. Second, results are gathered explicitly in the `receivedVotes` map. This all seems easy enough to implement. However, the above code foregoes one important detail, which is the fact that `nearbyTeamPlayers` is a *volatile* set, which may change in between the time the vote is cast and the results are processed.

The major problem with representing one-to-many communication in AmbientTalk as a simple iteration is the following: the broadcast is only received by objects in range *at the time the broadcast was made*. If the poll lasts, say, 5 minutes and a new team player is discovered after the vote was initiated, he would simply not receive the vote, because only players in range at the time the poll was created received the message. This problem is a direct consequence of treating the set of proximate players as an explicit collection in the application code. An ad hoc solution to this problem is to keep track of changes in the set explicitly while the vote lasts. The updated code employing this solution is shown below.

[1] The function **when:elapsed:** is a library function which takes a time period and a closure and triggers the closure when the time period has elapsed. This timing behaviour is not real-time. It depends on the accuracy of the underlying JVM's timing capabilities.

```
def broadcastVote(poll, maxVoteTime) {
  def [future,resolver] := makeFuture();
  def receivedVotes := Map.new();

  def retrieveVote(player, timeLeft) {
    when: player<-askToVote(poll)@Due(timeLeft) becomes: { |vote|
      receivedVotes.put(player, vote);
    }
  };

  nearbyTeamPlayers.each: { |player|
    retrieveVote(player, maxVoteTime);
  };

  def alreadySent := nearbyTeamPlayers.copy();
  def voteStartTime := now(); // the current time

  // every subsequently discovered player should also receive the
  // vote while the vote time has not yet elapsed
  def discovery := whenever: Player discovered: { |player|
    if: !alreadySent.contains(player) then: {
      alreadySent.add(player);
      def timeLeft := maxVoteTime - (now() - voteStartTime);
      when: player<-getTeam()@Due(timeLeft) becomes: { |team|
        if: (team == myTeam) then: {
          retrieveVote(player, timeLeft);
        }
      }
    }
  };

  when: maxVoteTime elapsed: {
    discovery.cancel(); // stop temporary discovery
    resolver.resolve(receivedVotes);
  };
  future
}
```

The problem of broadcasting the vote to players entering communication range at a later point in time is tackled by creating a *temporary* discovery event handler which is notified of each new player in range from the time the vote is cast until the vote time is over. The handler's subscription is cancelled when the `maxVoteTime` has elapsed. Note that the above **`whenever:discovered:`** event handler has to duplicate the code shown in the first code excerpt of this section to check whether the player is a member of the same team, partially defeating the modularisation of the implementation of `nearbyTeamPlayers`.

Because notifying a newly discovered player is similar to notifying a previously discovered one, we use functional abstraction to factor out the common code in the function `retrieveVote`. One variability between these two cases is the upper bound on the time allowed to vote (which is used as the timeout period of the `askToVote` message). This upper bound has to be recalculated for each new discovered player, because such a player will have less time to vote than one already present at the time

the poll was created. Calculating the time left to vote (`timeLeft`) is done by subtracting from the original `maxVoteTime` interval the time elapsed between the start of the vote and the current time, as returned by the library function `now`. Hence, the longer it takes to discover a team member, the less time that player is allowed to cast his vote.

Note the explicit check in the **`whenever:discovered:`** event handler to filter out players to which the vote has already been sent. **`whenever:discovered:`** event handlers are independent of one another, in the sense that they are triggered for each discovered object, regardless of the fact that other handlers have already been triggered for that object. Hence, the new **`whenever:discovered:`** event handler will also discover all players previously collected in the `nearbyTeamPlayers` collection. Without the explicit check for duplicates, the event handler would send a vote to these players twice.

The above problem can be circumvented if the use of the `nearbyTeamPlayers` collection is simply dropped. In the above code, the `askToVote` message is first sent to all elements of the `nearbyTeamPlayers` set and then sent to every discovered player which is *not* in that set. A simpler solution would be to send the `askToVote` message to every discovered player. Because the above **`whenever:discovered:`** event handler also triggers for all members of the `nearbyTeamPlayers` collection, the explicit initial iteration using `each:` is not required, and one would not need to check for duplicates using the `alreadySent` collection. However, using this solution, the previously shown code snippet to construct and manage the `nearbyTeamPlayers` collection is no longer reused, while it was the goal of that piece of code to be reusable by different modules.

The above example illustrates the issues with which a programmer has to deal when representing a proximate set of objects (team players in the example) as an explicit collection. Depending on the application at hand, the collection must support different designation semantics. For example, in the voting example, the application wanted to designate all proximate team players while a poll was active. The `nearbyTeamPlayers` collection was unsuitable for the application because it only designated all team players at the time the poll was created. The explicit collection abstraction did not cater to any form of time-decoupling. Performing time-decoupled one-to-many communication could thus not be subsumed by a simple iteration over the collection, even if that was how the programmer wanted to express it conceptually.

In this section, we have described the difficulties in representing arity-decoupled, one-to-many communication abstractions as explicit collections of objects. In the following section, we describe a third pattern of communication which is not explicitly supported by any of AmbientTalk's language constructs.

6.1.3 Provisional Services

One of the virtues of futures is that they provide a convenient placeholder for referring to the result of a parallel computation. This placeholder is a first-class entity, which implies that it can e.g. be parameter-passed or bound in some data-structure for later use. This enables the initiator of the parallel computation to carry on computing while still being able to refer to the result of that parallel computation. This ability of futures to represent values which have not yet been computed forms the original motivation behind their use, for example in languages like Multilisp [Hal85] and Eiffel// [Car93].

In the case of future-type message passing, the spawned parallel computation is a remote method invocation. In AmbientTalk, service discovery is also a parallel computation, and for good reason: if no required service object is immediately available in the host device's proximity, it may take a very long time before a service discov-

ery request can be honoured. Unfortunately, AmbientTalk provides no analogue of the future abstraction for service discovery. A call to **when:discovered:** returns a subscription object, which can only be used to cancel the subscription. There is no direct support for "provisional" services which represent service objects that "have not yet been discovered". Again, we examine the problem by means of a concrete example.

6.1.3.1 Example: Rendering Product Prices

Consider a supermarket in which all products on the shelves have been tagged with Radio Frequency Identification (RFID) tags. The customer's shopping cart is equipped with an RFID reader that can scan the contents of the cart to construct a digital shopping cart. One of the functionalities of the cart is that it has a screen that can render all of the product details in the cart, including the total price of all the products in the cart[2]. We assume that the RFID tags themselves contain useful information about properties intrinsic to the product (e.g. its name, expiration date,...). Properties extrinsic to the product, such as its price, are stored in the supermarket's database and can be retrieved by means of a unique product code. The database is accessible to the cart via a wireless ethernet link.

The goal is to render the user interface on the cart's screen, displaying all of the information available directly in the RFID tags in a table overview. We would like the discovery of the database server and the retrieval of all the product prices to happen in parallel, as a background task. If a database is not yet available, the customer already has an overview of at least the intrinsic product details. When a database is available, missing product information is updated as and when it becomes available.

6.1.3.2 Implementation in AmbientTalk

We assume the availability of an object `gui` representing the table storing the items in the graphical user interface and an object `productsInCart`, representing the collection of objects currently in the cart. The following code renders the graphical user interface, assuming a reference `server` pointing to a database server.

```
def renderGUI(server) {
  productsInCart.each: { |product|
    gui.addRow(product.id, product.name, "??");
    when: server<-getPrice(product.id) becomes: { |price|
      gui.updateRow(product.id, product.name, price);
    }
  };
};
when: ProductDatabase discovered: { |db|
  renderGUI(db)
};
```

Product information is displayed and price information is gathered in parallel (because `getPrice` is sent asynchronously). However, note that products will only start to appear in the list when a service providing access to the database has been discovered. Without support for futures, rewriting the code to perform the discovery *in parallel* with rendering the GUI requires modifying the `renderGUI` function.

[2]Futuristic as this example may seem, actual experiments have been performed that have successfully prototyped such functionality, such as Metro's future store, cf. `http://www.future-store.org`.

```
def renderGUI() {
  productsInCart.each: { |product|
    gui.addRow(product.id, product.name, "??");
  };
};
def updateGUI(server) {
  productsInCart.each: { |product|
    when: server<-getPrice(product.id) becomes: { |price|
      gui.updateRow(product.id, product.name, price);
    }
  };
};
when: ProductDatabase discovered: { |db|
  updateGUI(db)
};
renderGUI();
```

The `renderGUI` code has been refactored into two functions: one performing all computation that does not require access to the `server` variable, and another one performing all computation dependent on this variable. This allows both functions to be invoked independently, allowing the discovery request to proceed in parallel with the initial construction of the GUI. Because AmbientTalk guarantees that any event handler is only triggered after the current computation of the actor runs to completion, the `renderGUI()` statement following the above **`when:discovered:`** event handler is always executed before the handler is triggered, even if a matching service would be available.

The problem with the above code is that a far reference to a `ProductDatabase` service must always be acquired from a type tag asynchronously by means of a discovery event handler, such that there can be no parallelism between the code requesting the reference and the code using the reference. The root cause of the problem is that far references must refer to a live object, they cannot represent "objects yet to be discovered".

The astute reader may have noticed that there exists a more elegant solution to the above problem, which is to make use of first-class futures (cf. section 4.3.4.3) to represent the server that has not yet been discovered:

```
def renderGUI(server) {
  /* as in the initial implementation */
};
def [future,resolver] := makeFuture();
when: ProductDatabase discovered: { |db|
  resolver.resolve(db);
};
renderGUI(future);
```

The result of an asynchronous service discovery request is captured by means of an explicitly created future. The future acts as an eventual reference to the database server while no such server has been discovered yet. The `getPrice` messages sent in `renderGUI` will thus be buffered until a database is available, without changes in the GUI code.

The above code is clearly a pattern that can be applied to the discovery of any kind of service. Anonymous far references, introduced in section 6.6.1, abstract this pattern

into an appropriate language construct that better captures the programmer's intent, which is to acquire a proxy for a service that is not yet discovered.

6.1.4 Summary

Even though AmbientTalk is a high-level ambient-oriented programming language, we have demonstrated that the language lacks direct support for expressing different useful communication patterns in mobile ad hoc networks:

Roaming AmbientTalk's far references do not support transparent rebinding to different objects. They designate a unique object throughout their lifetime.

One-to-many Communication Far references do not support communication with a group of proximate objects. They refer to only a single object throughout their lifetime.

Provisional Services Far references cannot be used as an ad interim communication channel when no matching services are available. They must always designate an object throughout their lifetime.

All of the above problems can eventually be traced back to the fact that far references do not cater to space-decoupling. In the following sections, we introduce the ambient reference abstraction, whose goal is to reconcile space decoupling with object referencing, thus filling the above gaps in AmbientTalk's communication abstractions.

6.2 Ambient References in a Nutshell

An ambient reference should be regarded as a reference to a *volatile set* of objects. This volatile set of objects often denotes "all objects of a certain type which are currently in communication range". The volatility of the set follows from the fact that the communication range is limited and because devices may physically move about in unpredictable ways.

6.2.1 Example: Broadcasting Stock Quote Updates

As a concrete example, suppose a stock quote server hooked up to the Internet through the infrastructure of a nomadic ad hoc network regularly receives stock quote updates from several stock markets. Mobile clients are interested in being notified whenever stock quotes change. In order to receive updates, a "listener" object should be explicitly exported by the client, as follows, assuming `StockQuoteListener` is a type tag known to both clients and server:

```
export: (object: {
  def quoteUpdated(code, price) {
    system.println("Price for "+code+" updated: "+price);
  };
}) as: StockQuoteListener;
```

The above code exports a stock quote listener object that simply echoes every stock quote update to the screen. Exporting objects is done through the **`export:as:`** construct explained in section 4.4.2. To address nearby listeners, the server may now construct an ambient reference as follows:

```
def clients := ambient: StockQuoteListener;
```

The variable `clients` designates all objects in communication range which are exported as a `StockQuoteListener`. The server can notify the clients by sending a message to the ambient reference, thereby implicitly referring to all clients currently within range. The management of this volatile set of objects is under the control of the ambient reference, shielding the server from having to deal explicitly with service discovery in terms of the low-level constructs described in section 4.4.3. The following code shows how the broadcast is achieved:

```
def makeQuoteServer() {
  def clients := ambient: StockQuoteListener;
  def quoteDB := ...;
  def refresh := minutes(5); // amount of time before refreshing info
  object: {
    def newPriceReceived(code, price) {
      if: (quoteDB.hasPriceChanged(code, price)) then: {
        def handle :=
         clients<-quoteUpdated(code,price)@[All,Transient(refresh),Oneway];
      }
    }
  }
}
```

Note that the broadcast is achieved by means of sending the message `quoteUpdated` asynchronously to the `clients` ambient reference. Because ambient references inherently represent *remote* objects, they are represented as eventual references. They require client code to use asynchronous message sends to communicate with the remote objects implicitly designated by the ambient reference. The message is annotated with additional information, telling the ambient reference how to handle the message. We will not yet give a full account of the semantics of the annotations here. For the purposes of this example, it suffices to understand that:

All indicates that the message should be sent to (but not necessarily received by) all objects in the ambient reference's set (i.e. it identifies the message send as a broadcast).

Transient indicates that the message should remain available to be sent to nearby clients until the given timeout period elapses. In the example, it is assumed that the stock quote information is refreshed after 5 minutes. In a more realistic setting, this timeout should be correlated with the rate at which the server itself receives new stock quotes.

Oneway indicates that the message does not need a result or acknowledgement, i.e. it can be regarded as a pure "event notification".

Every message send to an ambient reference immediately returns a *delivery handle*, which is a first-class representation of the delivery status of the message. A delivery handle provides functionality to cancel a message's delivery prematurely. For example, the server could use it to stop the broadcast of a stock quote before its timeout period of 5 minutes has elapsed. Furthermore, if the message requires a result or acknowledgement, any potential replies to the message can be accessed via this delivery handle. Concrete examples of the use of delivery handles are given in section 6.3.

We have intentionally left out the details of the message passing semantics of ambient references in the above discussion. The goal of this section is to give a general feel for the language abstraction's purpose and use. In subsequent sections, the semantics of ambient references is described in more detail.

6.2.2 Space-decoupled Object References

We now provide a definition for the term "ambient reference":

Definition 2 (Ambient Reference) *An ambient reference is a space-decoupled object reference designating a volatile set of proximate service objects.*

An ambient reference is an *object reference* (an eventual reference, to be precise): it has the ability to carry messages from a sender to a receiver object. However, it differs from AmbientTalk's far references in the following ways:

- An ambient reference decouples objects in space: remote objects are designated anonymously by means of an intensional description. *Any* object that adheres to the description can be designated by the ambient reference. This allows ambient references to cater to roaming.
- An ambient reference designates a *set* of objects. Hence, it may refer to more than one object, which allows it to cater to arity-decoupled, one-to-many communication.
- The set of objects designated by an ambient reference may be empty. Even then, an ambient reference remains a communication channel to the set. Hence, ambient references can be used to represent provisional services – services which are not (yet) available. This is in contrast to far references (acquired by means of service discovery) which only come into existence when a matching object actually becomes available.

Note that each of these differences between ambient and far references enables programmers to express a communication pattern described in section 6.1. We provide concrete examples to support this claim in section 6.4.

The most distinguishing feature of ambient references is that they are object references supporting a form of *connectionless* (as opposed to connection-oriented) object designation. In section 3.4.2 we have argued that there are advantages and drawbacks to both types of object designation. Ambient references provide space-decoupling but forego stateful communication, while far references provide stateful communication but forego space-decoupling. We will therefore explore a middle ground between both referencing abstractions, known as *anonymous* far references, in section 6.6.1.

From the discussion of the examples in section 6.1, it is clear that there is no single *right* abstraction for *all* kinds of collaborations. For example, whereas the mobile client in the location tracker example uses point-to-point communication to communicate with a server, the mobile client in the voting application clearly engages in one-to-many communication with nearby players. In the location tracker example, messages should be buffered if no server is nearby. In the voting application, there is no such requirement. The message sent to the server in the location tracker example requires no result, while the mobile client in the voting application needs to explicitly process replies to its message send.

In order to support these differing requirements, we will not introduce different kinds of ambient references, but rather different message passing operators that influence the delivery of messages sent via the ambient reference. In the following section, we discuss the salient features of the message passing semantics supported by ambient references.

6.3 Decomposing Ambient References

We now describe ambient references independent of their incarnation in AmbientTalk. More specifically, we introduce the necessary terminology to describe the key characteristics of ambient references. As described in the previous section, an ambient reference is a space-decoupled object reference. We therefore represent an ambient reference a as a tuple $\langle f, \mathcal{M} \rangle$ where f is the characteristic function defining the set of objects which the ambient reference may designate and $\mathcal{M}$ is a set of asynchronous messages sent to the ambient reference.

Ambient references designate *service objects*. A service object is any object exported to the network by an application.The set of all service objects is denoted $\mathcal{O}$ and a specific service object representing a service X is denoted s^X (in AmbientTalk, X could e.g. be a type tag). The characteristic function f forms the intensional description of the following set:

Definition 3 (Scope) *The scope $\mathcal{S}_a$ of an ambient reference a is the set of all service objects it may potentially refer to. That is, $\mathcal{S}_a = \{ s \in \mathcal{O} \mid f_a(s) \}$*

At any point in time, service objects may either be accessible or inaccessible to the ambient reference (that is, the device hosting the service object may be connected to or disconnected from the device hosting the ambient reference). This brings us to the definition of the following set:

Definition 4 (Communication Range) *The communication range $\mathcal{A}_a(t)$ of an ambient reference a at a given point in time t is the set of all service objects accessible to it by means of the underlying network.*

The communication range restricts the service objects that an ambient reference may designate at the level of the *physical* underlying network. Interpreting communication range in the context of wireless proximity ad hoc networks (where devices communicate by means of radio communication protocols like WiFi or ZigBee), objects in communication range are also *spatially proximate* to the ambient reference. In wired networks, objects may be logically connected yet not spatially proximate at all. Because ambient references are designed specifically for mobile ad hoc networks, we equate object accessibility with the fact that the object's host device is within communication range and hence proximate.

Combining an ambient reference's scope and communication range enables us to define the set of objects actually denoted by an ambient reference at any given point in time:

Definition 5 (Reach) *The reach $\mathcal{R}_a(t)$ of an ambient reference a at a given point in time t is the set of all objects in its scope that are accessible at that time, i.e. $\mathcal{R}_a(t) = \mathcal{S}_a \cap \mathcal{A}_a(t)$.*

We say that a service object s is *in reach of* an ambient reference a at time t if $s \in \mathcal{R}_a(t)$. Communication range and reach as defined above should be considered logical models rather than concrete representations. At any specific point in time, the exact contents of the sets defined by communication range and reach are unknown to the ambient reference. In a concrete implementation, the ambient reference will have to approximate the contents of these sets. We discuss two approaches to do so in chapter 8.

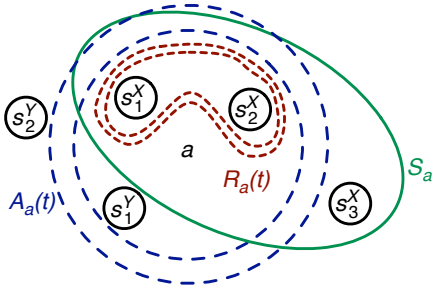

Figure 6.1: Scope $\mathcal{S}_a$, communication range $\mathcal{A}_a(t)$ and reach $\mathcal{R}_a(t)$ of an ambient reference a at a point in time t.

Figure 6.1 illustrates the relationship between scope, communication range and reach of an ambient reference graphically. The graphical notation abstracts from the difference between objects, actors and devices. The small circles represent service objects, each hosted on a different mobile device (and hence necessarily owned by a different actor). a represents (the location of) an ambient reference. The figure depicts the ambient reference's scope, communication range and reach, assuming that a's scope is delimited by all s_i^X services. As such, we get $\mathcal{S}_a = \{s_1^X, s_2^X, s_3^X\}$, $\mathcal{A}_a(t) = \{s_1^X, s_2^X, s_1^Y\}$ and $\mathcal{R}_a(t) = \{s_1^X, s_2^X\}$. Communication range and reach are represented as two concentric circles. Objects located fully inside both circles are considered in range or reach. Objects fully outside of both circles are not in range or reach. For objects located in between both circles, the semantics is undefined: these objects may or may not belong to the ambient reference's range or reach.

We refer to any message sent to an ambient reference as an *ambient message*. After an ambient message m is sent to an ambient reference $a = \langle f, \mathcal{M} \rangle$ it holds that $m \in \mathcal{M}$. a is referred to as the *carrier* of m. The set of service objects that are eligible to receive an ambient message m is derived from its carrier's reach and will be referred to as the set of *potential receivers* $\mathcal{PR}_m$ of the message. The subset of $\mathcal{PR}_m$ to which the message is actually sent is known as the set of *actual receivers* $\mathcal{AR}_m$.

In order to determine $\mathcal{AR}_m$ and $\mathcal{PR}_m$, the message's delivery must be bounded in a number of ways. This is done by means of a number of *message delivery policies* associated with each ambient message. The delivery policies of an ambient message are represented as a triplet $\langle \Delta t_d, n, \Delta t_c \rangle$. The different policies are:

Discovery Lifetime Δt_d is an upper bound on *how long* to keep the message available for delivery to objects entering the carrier's reach, known as the message's

discovery lifetime. The set of potential receivers of the message $\mathcal{PR}_m$ is constructed by means of its discovery lifetime.

Arity n is an upper bound on how *many* objects may receive the message, known as the message's *arity*. The arity indicates how many objects to select from $\mathcal{PR}_m$ to determine the set of actual receivers $\mathcal{AR}_m$.

Communication Lifetime Δt_c is an upper bound on *how long to wait* for a reply, if any, from any actual receiver of the message. It is known as the message's *communication lifetime* and is necessary to deal with failures.

We can distinguish two phases in the delivery of an ambient message. During the *speaking phase*, which starts when the ambient message is sent to the ambient reference, the ambient reference actively discovers receivers for the message. Whenever the message is sent to an actual receiver selected from the potential receivers, a *listening phase* starts during which a reply to the message is awaited. Both of these phases can be bounded in time. Discovery lifetime bounds the speaking phase while communication lifetime bounds the listening phase. We describe each of the above policies in further detail below.

Discovery Lifetime The discovery lifetime of an ambient message is the period of time in which it may be delivered to potential receivers. If t_s denotes the time at which an ambient message m is sent to an ambient reference a, we can define its set of potential receivers as:

$$\mathcal{PR}_m = \bigcup_{t_s \leqslant t \leqslant t_s + \Delta t_d} \mathcal{R}_a(t)$$

The discovery lifetime of an ambient message is a period of time Δt_d. Distinguish two border cases for this time interval, we consider the following choices regarding discovery lifetime:

- An ambient message has an *instant* discovery lifetime if $\Delta t_d = 0$. This signifies that the ambient message is sent *only* to objects in reach at the time the message is sent to its carrier.
- An ambient message has a *sustained* discovery lifetime if $\Delta t_d = \infty$. This signifies that the ambient message is sent to all objects entering its carrier's reach starting from the time it is sent.
- In all other cases, an ambient message is said to have a *transient* discovery lifetime, in which case it is sent to objects in reach in between the time it is sent and the timeout period elapses.

Arity The second delivery policy of an ambient message is its arity. An ambient message's arity $n \in \{1, \infty\}$ specifies an upper bound on the number of objects to elect as actual receivers from its set of potential receivers. We consider two choices:

- An ambient message is *point-to-point* if it has at most one actual receiver, i.e. $n = 1$. This receiver is a non-deterministically chosen element from the set of potential receivers $\mathcal{PR}_m$.

- An ambient message is *one-to-many* if it can have any number of receivers, i.e. $n = \infty$. For one-to-many messages, the set of actual receivers $\mathcal{AR}_m = \mathcal{PR}_m$.

If there are no actual receivers, i.e. $\mathcal{AR}_m = \emptyset$, an ambient message is said to be *lost*. A lost message is not sent to any service objects.

In principle, the arity n of an ambient message could be any natural number, thus specifying an exact number of receivers of a one-to-many message. We do not consider this semantics because ambient references are designed for anonymous interactions, where the amount of potential receivers is usually completely unknown. We further discuss this design decision in section 7.3.2.

Communication Lifetime The third delivery policy of an ambient message is its communication lifetime. Communication lifetime is a period of time Δt_c specifying how long to wait for a reply, if any, for each actual receiver $s \in \mathcal{AR}_m$. Again distinguishing two border cases for this time interval, we consider the following choices regarding communication lifetime:

- An ambient message is *one-way* if $\Delta t_c = 0$. This signifies that no reply to the message is required.
- An ambient message is *unbounded two-way* if $\Delta t_c = \infty$. This signifies that any reply to the message may take indefinitely long to arrive.
- In all other cases, an ambient message is *bounded two-way*, in which case Δt_c determines how long to await any reply to the message.

Ambient references never guarantee that an ambient message is ever delivered. The only way to be certain that an ambient message was delivered is to await a reply. When no reply is received, this may signify that either the message was not sent to any actual receiver, or it was lost during transmission, or it was delivered but its reply was lost during transmission or is still pending.

In this section, we have introduced ambient references in a very abstract way, without going into any technical detail. We have introduced the necessary terminology describing the important aspects of ambient references. Most importantly, an ambient reference is defined in terms of a characteristic function encoding an *intensional description* of its scope. At any point in time, the ambient reference denotes that subset of its scope which is in communication range, known as its reach. Messages sent via ambient references are known as ambient messages. The delivery of an ambient message may be influenced by means of three delivery policies: discovery lifetime, arity and communication lifetime. In the following section, we describe the particular incarnation of ambient references in AmbientTalk and in that particular context discuss each of the above aspects of ambient references in further detail.

6.4 Ambient references in AmbientTalk

As illustrated in the stock quote example of section 6.2.1, in AmbientTalk ambient references are represented as eventual references. An ambient message is simply a message sent via such an eventual reference. Its message delivery policies can be expressed at the level of an individual message send as *annotations* to the message. However, it is also possible to specify default values for these parameters at the level of

the ambient reference declaration. Annotations at the message-level can then override these defaults. Using defaults, if all messages are sent using the same delivery policies, the policies do not have to be repeated at the level of individual messages. If the policies need to be changed, it suffices to change the single ambient reference declaration rather than all individual message send expressions.

Being eventual references, ambient references may be parameter-passed as an argument or return value across other references. An ambient reference always designates a volatile set of objects which are proximate to the device hosting the ambient reference. Thus, because proximity is relative to the host device, when an ambient reference is parameter-passed across devices, the passed reference may designate a *different* set of objects than does the original ambient reference. We return to the parameter-passing semantics of ambient references in section 8.9.

In each of the following subsections, we examine the precise semantics of the scope and delivery policies of ambient references in AmbientTalk. For each delivery policy, we also describe when it is generally useful to use that policy. That is, we describe which delivery policies are appropriate for what use cases. After all delivery policies have been explained individually, we discuss their composition in section 6.4.8.

6.4.1 Scope

The scope of an ambient reference describes the set of objects to which an ambient reference can refer. In mathematics, sets are described either intensionally (by means of a characteristic function specifying which objects belong to the set and which do not) or extensionally (by explicitly enumerating all of the set's elements). Because of the volatile nature of the set of proximate objects, the scope of an ambient reference is specified intensionally. This is a fundamental difference with respect to traditional (object) referencing mechanisms in (object-oriented) programming languages, whose target is most often specified in terms of a specific identity. The power of an intensional description is that it allows remote objects to maintain their anonymity. It couples communicating parties only at the level of application-specific attributes, rather than at the level of object addresses. Furthermore, using an intensional description, one can abstract from the precise number of elements in the set. It is not possible to create an extensional description of the scope of an ambient reference. However, it *is* possible to create an extensional description of its reach, as explained later in section 6.6.2.

As explained above, a set is intensionally described by means of a characteristic function that, given an object, determines whether that object belongs to the set or not. In other words, the characteristic function is a predicate. From an operational point of view, the predicate is applied to each physically discovered object. If the object satisfies the predicate, it is part of the ambient reference's scope and can be added to the reference's reach. Ambient references support a number of built-in characteristic functions to delimit an ambient reference's scope. These characteristic functions classify objects according to:

Type tags These form the simplest and most direct way of characterising objects. We have already illustrated the use of a type tag as a characteristic function in the introductory example of section 6.2.1. Any given type tag `Type` induces a characteristic function `f` as follows:

```
def f(obj) { is: obj taggedAs: Type }
```

Recall that type tags support subtyping to introduce polymorphism: at runtime, `obj` may be an object tagged with a subtype of `Type`. At the implementation level, ambient references do not explicitly use the above characteristic function. The advantage of type tags is that they can be unified quite naturally with hierarchical *topics* in topic-based publish/subscribe engines [EFGK03], allowing the above type test to be pushed down to the level of the service discovery engine. We describe an implementation of this unification in section 8.7.3.

Even though type tags are efficient and mostly a straightforward solution for classifying and discovering objects, they introduce a number of issues. First, matching based on type tags is based on the implicit assumption that all devices in the network share the same semantics, i.e. the meaning of a type tag's name is globally unique. For example, the type tag `Scanner` does not enable a distinction between a service representing the hardware device and one representing the tokenizer of a compiler. The assumption of globally unique meaning may or may not be realistic, depending on the openness and the scale of the network on which the application is deployed.

Second, type tags cannot distinguish between different versions of the same service. For example, assume that there is a global consensus that `Scanner` represents a tokenizer service. Even then so, the type tag cannot distinguish between different versions of the same service and may match endpoints expecting different versions. This is also the reason why, in Java, distributed classes are not compared by name but by means of their `serialVersionUID` field (which, by default, is a "hashed" representation of the class structure) to be able to distinguish between different class versions [GJSB05].

Protocols A second kind of characteristic function allowed by ambient references classifies objects based on *protocols*. A protocol is a description of the set of selectors (message names) to which an object responds. The name is derived from its use in the Smalltalk community and the structural types of the StrongTalk language [BG93]. Whereas type tags introduce a form of nominal typing (the subtype relation between types is explicitly defined in a type hierarchy), protocols introduce a form of structural typing (the subtype relation is implicitly defined by means of the interface of the objects).

Protocols have been added reflectively to AmbientTalk. Like type tags, they are purely used for classifying objects, not for static type-checking. The following code illustrates their usage:

```
def Point := object: {
  def x := 0; def y := 0;
  def add(pt) { Point.new(x+pt.x, y+pt.y) };
};
// construct a protocol based on the
// interface of an existing object
def PointProtocol := protocolOf: Point;
def Point3D := object: {
  def x := 0; def y := 0; def z := 0;
  def add(pt) { Point3D.new(x+pt.x,y+pt.y,z+pt.z) };
};
does: Point3D implement: PointProtocol; // true
```

Even though `Point` and `Point3D` have no explicit relation, they both match the `PointProtocol`. In AmbientTalk, protocols are represented as sets of selectors (symbols). An object implements a protocol if the protocol is a *subset* of the set of selectors

defined by the object's fields and methods. If an object implements a protocol, this guarantees that the object responds to all messages defined in the protocol, but nothing more. In particular, no support is provided to recursively type-check arguments or return values of the methods of a protocol. Using protocols, ambient references can be defined without reference to a type tag. We can reformulate the stock quote example from section 6.2.1 as follows[3]:

```
def StockQuoteProtocol := protocol: {
  def quoteUpdated(code, price);
};
def clients := ambient: StockQuoteProtocol;
```

The `clients` ambient reference implicitly refers to all proximate objects implementing a `quoteUpdated` method. Any protocol `P` induces a characteristic function `f` as follows:

```
def f(obj) { does: obj implement: P }
```

Protocols integrate better with a dynamically typed language, because classifying objects is done based on an intrinsic property of the objects (their interface). The downside of protocols is that the evaluation of their corresponding characteristic function cannot be done directly within the service discovery engine. Unlike type tags, protocols cannot be directly mapped onto an analogue classification mechanism used by service discovery engines. Combining protocols with service discovery is further discussed in section 8.7.3.

The matching logic of protocols is based on the "duck test": "if it walks like a duck and quacks like a duck, it is a duck". This is a well-known form of inductive (yet unsound) reasoning. It leaves open the possibilities for incorrect matches, e.g. both the protocols for the scanner and tokenizer may consist of a single method `scan(document)`. Protocols have the advantage over type tags of being able to discriminate between different versions of the same service, but only if the interface changes as a result of a version change. For example, if the tokenizer in a later version adds the method `mark(position)`, to mark a starting position in the token stream, earlier versions of the tokenizer will not match the new protocol.

Filters The above two characteristic functions classify objects according to static attributes such as their type or their interface. Often, discriminating objects based on dynamic attributes is required, especially if those attributes have continuous rather than discrete values, e.g. matching a printer whose pending job queue size is smaller than a given integer value. To cater to such matching, ambient references enable any AmbientTalk predicate to be used as a characteristic function to filter matching objects. We refer to such predicates as *filters* and represent them as unary closures returning a boolean value.

Reconsider the voting example from section 6.1.2.1. Recall that a vote was to be launched to each proximate player *of the same team*. One may of course introduce discrete type tags to represent the different values of the team attribute (e.g. `BlueTeamPlayer` and `RedTeamPlayer`), but this is very ad hoc and works only for attributes with a discrete number of values. Furthermore, it complicates issues such as making a player switch teams, adding additional teams and filtering based on multiple

[3]The auxiliary function **protocol:** enables one to define ex-nihilo protocols, without reference to a live object implementing that protocol.

attributes. Assuming that players have been exported with their team as a public attribute (cf. the following section), an ambient reference that only refers to members of the sender's own team is created as follows:

```
def nearbyTeamPlayers := ambient: Player where: { |p| p.team == myTeam }
```

Filters work in conjunction with the characteristic functions of both type tags and protocols. Given one of the above characteristic functions `g` for type tags or protocols and any AmbientTalk predicate `p`, the compound characteristic function becomes[4]:

```
def f(obj) { g(obj).and: { p(obj) } }
```

Filters are very similar in nature to filters in content-based publish/subscribe systems, where events are matched based on the *content* of the event, not on their type [CRW01]. While such systems support a much more fine-grained event dispatch than topic-based publish/subscribe systems, they are harder to implement efficiently than their topic-based equivalent [EFGK03]. However, if the filter language is sufficiently restricted, these systems can be made very efficient as well [EG01], potentially outperforming topic-based systems because the more fine-grained dispatch leads to less redundant event notifications. Because we allow arbitrary AmbientTalk code to act as a filter, we currently forego any possibility of optimisation. We elaborate on this subject when discussing future work in section 7.4.

Note that filters subsume the characteristic functions of both type tags and protocols. These characteristic functions are treated separately because they describe purely static aspects of an object (i.e. its type or interface). An implementation may exploit this fact to optimise the runtime test that determines whether or not a service object belongs to an ambient reference's reach.

Summary In this section, we have focussed on how ambient references can *designate* service objects in AmbientTalk. As explained in section 6.3, ambient references are associated with a characteristic function which intensionally denotes its scope. In AmbientTalk, ambient references can be classified according to static attributes such as their type tags or their protocol. Filters augment these static classification schemes by allowing arbitrary AmbientTalk predicates to determine the scope of an ambient reference. In the following section, we discuss how service objects can be exported such that they satisfy the different types of characteristic functions of ambient references.

6.4.2 Service Objects

In AmbientTalk, service objects are represented as exported objects. If the characteristic function of an ambient reference uses type tags to classify objects, it suffices to export objects as (a supertype of) that type tag in order to make them receive messages sent via that ambient reference. We have already discussed how this is done in section 4.4.2. If a service object wants to make itself available for designation based on a protocol, it should be exported as that protocol. To do so, the **`export:as:`** function is overloaded and can both be invoked with a type tag or a protocol. Reconsider the client object from the stock quote example in section 6.2. It can be exported by means of a protocol rather than using a type tag as follows:

[4] AmbientTalk's boolean objects respond to the message `and: closure` representing (short-circuited) boolean conjunction.

```
def StockQuoteProtocol := protocol: {
  def quoteUpdated(code, price);
};

def service := object: {
  def quoteUpdated(code, price) {
    system.println("Price for "+code+" updated: "+price);
  };
};

export: service as: StockQuoteProtocol;
```

Equivalently, it is possible to omit the explicit protocol and export an object with an implicit protocol automatically derived from the methods implemented by the object. The **as:** part of the call is then omitted and the object is exported simply by evaluating **export:** `service`. It is possible for an object to be exported both under a type tag and under a protocol by exporting the same object multiple times with different arguments.

The return value of the **export:as:** function and its variants is a publication object, as explained in section 4.4.2. This publication object has a single `cancel` method that takes the publication offline. After invoking `cancel`, the object is no longer subject to receiving message sends originating from ambient references. That is, it is removed from the scope of any ambient reference that previously designated it. This removal is not atomic, in the sense that messages targeting the object may still arrive after it was unexported. Hence, the programmer should be aware of the possibility that the exported object's methods may still be invoked even after `cancel` was invoked.

Finally, both for type tags and protocols it is possible to attach *attributes* to the object. Recall that in the multiplayer game example in section 6.1.2.1, we assume players publish the team which they are currently in as an additional attribute. In order to export an object that matches the `nearbyTeamPlayers` ambient reference defined in the previous section, the `team` attribute must be exported as follows:

```
deftype Player;
def makePlayer(inTeam) {
  def player := object: {
    def askToVote(poll) { /* ... */ };
  };
  export: player as: Player with: {
    def team := inTeam;
  };
  player
};
```

The third argument to **export:as:with:** is a closure whose contents is used to construct an *attribute object* (much like the closure passed to the **object:** primitive is used to construct an ex-nihilo object). The closure is used to define this object's fields and methods. This attribute object corresponds to the `obj` parameter of the characteristic functions defined in the previous section. Depending on the particular implementation strategy of ambient references, this attribute object may or may not be an isolate object. The different options and their repercussions are discussed in detail in sections 8.3.7 and 8.4.6.

When a message is received by a service object, the message is added to the message queue of that object's owning actor, as is usual for asynchronous message recep-

tion in AmbientTalk. Hence, the method triggered by the message is executed by the receiver's owning actor serially, in mutual exclusion with other received messages. In this regard, message delivery via ambient references is entirely reminiscent of message delivery via far references. Thus, exported objects can abstract from the fact whether they are referred to by means of far or ambient references.

Now that we have described how service objects are both *designated* and *exported*, we turn our attention to the messages sent to these service objects via ambient references. In the following sections, we discuss the different message delivery policies supported by ambient references – arity, communication lifetime and discovery lifetime.

6.4.3 Arity

The arity of an ambient message denotes the maximum amount of receivers of the message. We distinguish two options: point-to-point or one-to-many messages. We discuss the details of each kind of message below.

Point-to-point In the case of point-to-point communication, a single object is non-deterministically chosen from its set of potential receivers. The resulting communication is point-to-point, like a regular message send via a far reference. The important difference between point-to-point communication via a far reference and point-to-point communication via an ambient reference is that in the former case, subsequent messages are guaranteed to be received by the *same* object. Ambient references provide no such guarantee.

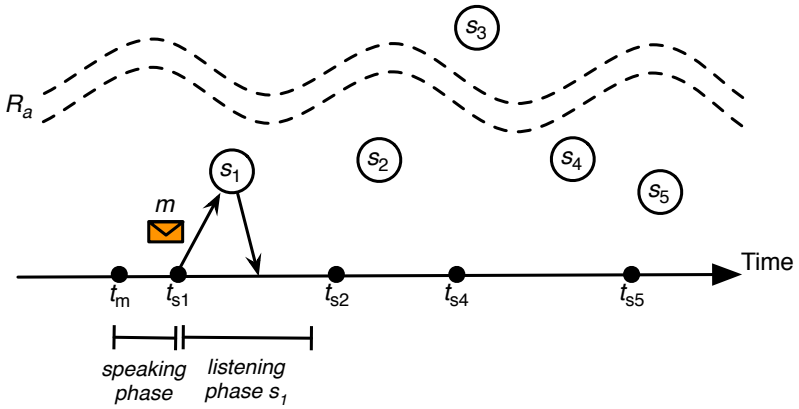

Figure 6.2: Point-to-point ambient message delivery.

Figure 6.2 depicts the delivery of a point-to-point ambient message. We assume that all depicted s_i services are within a's scope. The reach R_a is depicted by means of two waves (which correspond to the concentric circles in figure 6.1) and all service objects below both waves are considered to be in reach. The timeline denotes the time as measured on a's local clock[5]. t_m denotes the time at which a point-to-point message m is sent to a. t_{s_i} denotes the time at which service s_i is discovered by a. m's speaking

[5]We assume no knowledge about the time at which messages are received by service objects. We assume

and listening phases are depicted as intervals below the timeline. The speaking phase ends when at least one receiver has been discovered. The listening phase starts as soon as the message is sent to the discovered receiver. Note that, because the message is point-to-point, it is sent to the first discovered receiver and not to any other receiver that may be discovered during the message's discovery lifetime.

Point-to-point messages are expressed by annotating an ambient message with the `@One` annotation. An example of such messages is given in section 6.4.4.

In general, point-to-point ambient messages enable roaming. That is, they enable objects to seamlessly communicate with different service objects that offer the same service while they physically move about. This is useful when a mobile client requires access to a service which is offered by a (potentially large) number of different service providers. The intensional description of the ambient reference's scope enables the client to abstract from which service provider it is connected to. However, this setup only works when using either a stateless communication protocol between client and service provider or if the different service providers can synchronise their session state with one another (e.g. by means of a nomadic network infrastructure).

One-to-many In the case of one-to-many message sends, *all* objects in the ambient message's set of potential receivers are selected as actual receivers of the message. This enables ambient references to be used to perform one-to-many communication with proximate objects. In effect, a one-to-many message send is a controlled form of broadcasting, where the scope of the broadcast is delimited by the reach of the ambient reference at the time the broadcast is performed. Note that there is no guarantee that the message will effectively be delivered to all objects in reach.

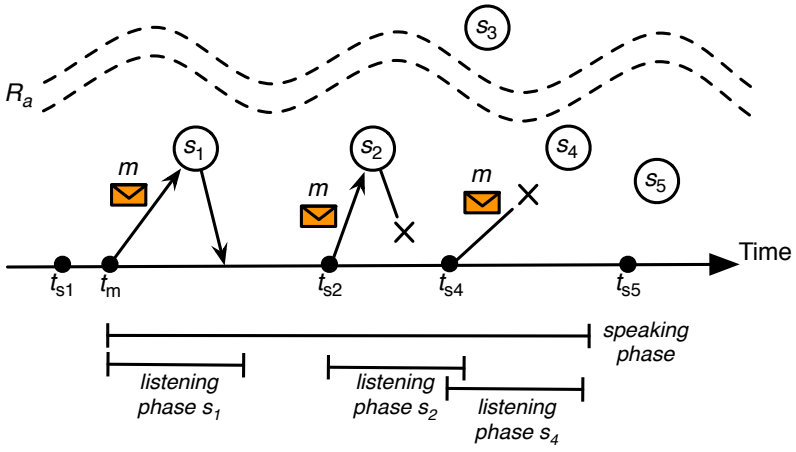

Figure 6.3: One-to-many ambient message delivery.

Figure 6.3 depicts the delivery of a one-to-many ambient message. Again, we assume that all depicted s_i services are within a's scope. Because the message is one-to-many, it is sent to all service objects in reach when the message is sent and to all

knowledge about the time at which receivers are discovered, messages are sent to them and replies are received from them.

service objects subsequently discovered during the message's speaking phase. The message is not sent to s_5 because it is discovered after the message's discovery lifetime has expired. Note that there is a listening period per message send to each discovered receiver. Arrows ending in a cross denote a failed message delivery. Thus, s_2 received m but failed to successfully send its reply back to a while s_4 did not receive m.

It is not possible to specify a precise number of receivers of a one-to-many message. If a message must be sent to a precise number of objects, it is more appropriate to construct a snapshot of an ambient reference (see section 6.6.2), which is an enumeration (i.e. an extensional representation) of all objects in its reach. A message can then be sent explicitly to all objects in the enumeration.

One-to-many messages are expressed by annotating an ambient message with the `@All` annotation. An example of one-to-many ambient messages was given in section 6.2.1 to broadcast the stock quote updates to all nearby clients.

In general, one-to-many messages allow a service to broadcast the same information to all proximate objects. This is useful for (stationary) services which can use it to notify all nearby (mobile) clients of certain events. For example, a shop service object might use it to broadcast advertisements to nearby customers. Alternatively, mobile clients can use one-to-many messages to communicate with all nearby services (e.g. a customer asking all nearby shops what products they have on discount).

Summary An ambient message's arity determines whether communication is point-to-point or one-to-many. In comparison to communication via far references, point-to-point messages enable roaming while one-to-many messages enable arity-decoupled communication with a group of objects. The following section discusses communication lifetime, a second type of delivery policy for ambient messages.

6.4.4 Communication Lifetime

The communication lifetime of an ambient message is an upper bound on each listening phase in the delivery of the ambient message. Communication lifetime specifies how long to wait for a possible reply, if any, from an actual receiver to which the message was sent. We distinguish between purely unidirectional, one-way message sends and bidirectional, two-way message sends. Two-way message sends make use of futures (cf. section 4.3.4) to deliver the reply to the sender. The communication lifetime of an ambient message specifies how long to wait for the future to become resolved with the reply. We distinguish between unbounded two-way communication, which implies an indefinite communication lifetime and bounded two-way communication, where the communication lifetime is specified by means of a timeout period.

6.4.4.1 One-way ambient messages

A one-way message send expects no reply from any potential receiver. As a result, no future is attached to the ambient message. If an acknowledgement to a one-way message is required, it must be acknowledged explicitly with a separate message. A one-way message does not necessarily imply that the message is broadcast once and then discarded, as is often the case in event notification services introducing one-way annotations. One-way messages combine with the discovery lifetime delivery policies described later (cf. section 6.4.5) and can be sent to potential receivers not in reach at the time the message was sent.

One-way message sends are annotated with the `@Oneway` annotation. The broadcast in the example of section 6.2.1 is an example of a one-way message: the server is not interested in which clients received the message, and it does not require a result.

In general, one-way messages are most useful when combined with one-to-many messages to represent a pure "event notification" from one service to multiple nearby clients. They may also be used in conjunction with point-to-point messages by clients to send status updates unidirectionally to nearby servers.

6.4.4.2 Unbounded two-way ambient messages

Two-way message sends enable the transmission of a reply from receiver to sender by means of futures. A future is automatically attached to an ambient message that is annotated with the `@Reply` annotation; an annotation specifying that a reply to the message is expected. The kind of future attached to the message directly depends on the ambient message's arity. A point-to-point message send receives a regular future while a one-to-many message send receives a multifuture. Both are described in further detail below.

Futures A point-to-point ambient message is received by at most one receiver object. As a result, at most one reply to the message can be received. In this case, a regular AmbientTalk future (cf. section 4.3.4) is attached to the ambient message. As is the case in standard AmbientTalk, the actual return value can only be accessed by registering an observer closure with the future using the **`when:becomes:`** function.

Reconsider the example of section 6.1.3.1 in which the GUI of a shopping assistant application was to be rendered in parallel with the discovery of a database server hosting additional product information. Using futures and point-to-point message passing, we can exploit an ambient reference's support for anonymous communication as follows:

```
def db := ambient: ProductDatabase;
def renderGUI(server) {
  productsInCart.each: { |product|
    gui.addRow(product.id, product.name, "??");
    def handle := server<-getPrice(product.id)@[One,Sustain,Reply];
    when: handle.future becomes: { |price|
      gui.updateRow(product.id, product.name, price);
    }
  };
};
renderGUI(db);
```

Two changes have been made to the original code from section 6.1.3.1. First, instead of using **`when:discovered:`**, we use an ambient reference to refer to all proximate `ProductDatabase` servers. Since ambient references are valid references even if no matching server has been found, we may readily invoke the `renderGUI` method, passing the `db` ambient reference as an argument. Moreover, the application additionally supports roaming, as it can now transparently query different product servers (of potentially different super markets).

A second change to the code is the addition of annotations to the `getPrice` ambient message in the method body of `renderGUI`. The `@Reply` annotation ensures that a future is associated with the message, which is retrieved by means of the delivery

handle returned by the ambient message send. As explained in section 6.2.1, message sends to ambient references always return delivery handles rather than futures directly. This is because a delivery handle provides more functionality than a future (e.g. cancelling message delivery prematurely) and because one-way ambient messages do not even have an associated future. The `@Sustain` annotation is discussed in more detail in section 6.4.5.3.

Futures are only attached to two-way point-to-point messages. Two-way one-to-many messages are associated with a more general type of future, discussed below.

Multifutures A one-to-many message send annotated with `@Reply` is associated with a *multifuture* [Ded06]. A multifuture is a future that can be resolved or ruined multiple times. The multifuture itself represents the collection of values and/or exceptions as a whole. A multifuture supports the same operations as a regular future. That is:

- it acts as a proxy to which messages may be sent. This message is subsequently forwarded to *every* value with which the multifuture is resolved.
- it allows for the registration of observer closures. These closures trigger on *every* value or exception with which the future is resolved or ruined.

A multifuture can be unresolved, *partially* resolved or totally resolved. It can only be totally resolved if the amount of time to wait for replies is bounded by means of a timeout period. This is described in further detail in section 6.4.6.2 when the appropriate mechanisms to express ambient message lifetimes have been introduced. There exist three different ways in which observer closures can be registered with a multifuture:

- **`when`** observers are triggered only on the *first* value or exception with which the multifuture is resolved or ruined. This ensures that a multifuture can be used anywhere a regular future is expected.
- **`whenEach`** observers are triggered on *each* value or exception of the multifuture. Hence, these closures can trigger an unspecified number of times.
- **`whenAll`** observers are triggered at most once, when the multifuture can guarantee that no further results will be gathered. These observers receive as an argument a table of values and/or exceptions such that all results can be accessed simultaneously.

Regular futures can be regarded as a special case of multifutures. A regular future behaves like a multifuture that is resolved at most once. As a result, it is legal to register **`whenEach`** and **`whenAll`** observers on regular futures. In this case, both behave like normal **`when`** observers which are triggered at most once. In the case of a **`whenAll`** observer, the value passed to the observer is wrapped in a unary table.

Because a one-to-many message is targeted at *all* of its potential receivers, and because this set of receivers is derived from its carrier's reach, the number of potential receivers is unknown. Consequently, the sender has to deal with an unknown number of replies to the message. Multifutures are a convenient abstraction to gather all of the replies to a broadcast at a single point in the code. To illustrate the use of multifutures, consider a virtual shopping assistant example originally proposed by Dedecker [Ded06]. The shopping assistant runs on e.g. the users' mobile phone equipped with WLAN. Users can specify their interest in a particular item. While

strolling in the shopping mall, the shopping assistant queries nearby shops for the item's price. At any point in time, the shopping assistant displays the store offering the best price for the item. The code below shows how the shopping assistant can keep track of the best offer thus far for a given `item`, which represents a unique product ID.

```
def bestPriceSoFar;
def bestShopSoFar;
def shops := ambient: Shop;
def handle := shops<-query(item)@[All,Sustain,Reply];
whenEach: handle.future becomes: { |reply|
  if: ((bestPriceSoFar == nil).or:{ reply.price < bestPriceSoFar }) then:{
    [bestPriceSoFar, bestShopSoFar] := [reply.price, reply.shopId];
    updateGUI(bestPriceSoFar, bestShopSoFar);
  }
}
```

The variable `shops` contains an ambient reference referring to nearby shops (assuming objects representing shops are exported by means of the `Shop` type tag). The `query` message is sent to all shops and expects replies to be sent to a multifuture. A **`whenEach`** listener on the multifuture is triggered each time a shop responds to the query. We assume the reply to the query is an object containing both the item's price and shop information. An example showcasing **`whenAll`** observers is given in section 6.4.6.2. Finally, the `@Sustain` annotation specifies that the message may be delivered indefinitely, until explicitly retracted (cf. section 6.4.5).

In general, multifutures enable many-to-one communication between peers, allowing for the expressive aggregation of replies. For example, a mobile client can use a multifuture to gather replies to a request that was broadcast to all stationary services in the nearby environment. The shopping assistant exemplifies such a communication pattern. Conversely, a stationary service could use a multifuture to gather replies to a request that was broadcast to all nearby mobile clients.

This section discussed *unbounded* two-way ambient messages. Such messages are associated with a future to gather replies, but there is no upper bound on the amount of time to wait until a reply arrives. Restricting this time period brings us to *bounded* two-way ambient messages, discussed below.

6.4.4.3 Bounded Two-way ambient messages

Once a potential receiver for an ambient message has been discovered by an ambient reference, the message is sent to that receiver. If the message is a one-way message, this ends the message delivery. However, if the message is a two-way message, the ambient reference needs to devote resources to await the reply. A bounded future-type message send enables control over how long the ambient reference ought to wait for that reply.

An ambient message can be annotated with `@Due(t)` where `t` denotes a timeout period (in milliseconds), known as the message's communication lifetime. It indicates the amount of time to wait for a reply after the ambient message was sent to a receiver. Hence, the timeout period `t` is relative to the time at which a receiver for the message was found. It is not relative to the time at which the message was sent to the ambient reference; such timeouts are discussed later in section 6.4.6.

An ambient message annotated with `@Due(t)` always has a future associated with it. After all, the possible expiration of the message requires a future to report this status to.

The expiration period indicated by `@Due` interacts with futures as follows. In the case of a point-to-point message, when the expiration period has elapsed and no reply has been received, the future attached to the ambient message is automatically ruined with a `TimeoutException`. This mimics the familiar way of dealing with overdue messages in AmbientTalk (cf. section 4.4.4). The interaction behaviour with multifutures is more complex, because it also depends on the discovery lifetime of an ambient message, a property which is explained in the next section. We will return to the interaction between communication lifetime and multifutures in section 6.4.6.2.

6.4.4.4 Summary

Communication lifetime allows peers to specify an upper bound on the listening phase, i.e. how long to wait for replies to an ambient message. Messages without a listening phase are one-way messages. For such messages, no reply is expected and there is no way of checking whether the message was successfully delivered. Messages with a listening phase are two-way messages and are always associated with a future. The type of future (regular or multifuture) depends on the message's arity. If, furthermore, the communication lifetime of the message is bounded (i.e. not infinite), the future can also be used to perform failure handling when replies to the message fail to arrive within the listening phase.

In the next section, we turn our attention to the third delivery policy of ambient messages, discovery lifetime, which enables one to bound an ambient message's speaking phase rather than its listening phases.

6.4.5 Discovery Lifetime

Communication lifetime bounds the listening phase by specifying an amount of time to wait for a reply to an ambient message, once that message was sent to an *actual* receiver. The discovery lifetime of an ambient message bounds the speaking phase during which service objects entering an ambient reference's reach are regarded as *potential* receivers for the message. Discovery lifetime is a crucial delivery policy because it forms a *temporal scope* in which to deliver the message. Without such a temporal scope, the delivery time of an ambient reference cannot be bounded. This is because, due to the intensional description of an ambient reference's scope, it is impossible to know how many objects the scope contains. As a consequence, it is impossible to put an upper bound on the number of potential receivers of an ambient message.

Discovery lifetime bounds the potential receivers of an ambient message in time rather than in number. The programmer can specify three policies, each of which is discussed in detail below.

6.4.5.1 Instant Delivery

An ambient message annotated with the `@Instant` type tag is only sent to objects within the ambient reference's reach when the message is sent to the ambient reference. Any objects that enter the ambient reference's reach after the message was sent are not considered potential receivers of the ambient message. In figure 6.3, if the message m were to have an instant discovery lifetime, it would only be sent to s_1 because this is the only service object in reach at the time m is sent (s_1 was discovered at time t_{s_1} prior to t_m).

To illustrate instant discovery lifetime, reconsider the location tracking service example introduced in section 6.1.1.1. In the example, mobile clients roam within a nomadic ad hoc network. The infrastructure of that network connects a number of access points to a location tracking service, which may be queried by clients to retrieve location information of another client. Roaming clients periodically submit their GPS coordinates to a proximate access point. Recall that we mentioned several possible (application-specific) semantics in the case that no access point is available. The coordinates could be buffered, or they could simply be dropped. Dropping coordinates is fine if the location server only keeps track of a client's latest position and not its entire path. The following code snippet implements this semantics:

```
def accessPoint := ambient: LocationService;
def updateLocation(loc) {
  accessPoint<-submitLocation(clientId, loc)@[One,Instant,Oneway];
};
```

Note how the entire implementation of the `createConnection` function defined in section 6.1.1.2 is subsumed by ambient references and the ability to send point-to-point ambient messages. Because the message is further annotated with the `Instant` type tag, the location information is dropped if there exist no proximate `LocationService` objects.

An ambient message with an instant discovery lifetime does not decouple sender and potential receivers in time: if such a message is sent when its carrier's reach is empty (i.e. there are no objects in communication range), the message is lost. However, messages with an instant discovery lifetime are useful for representing pure event notifications which are only relevant at the time the message is sent. Often, these notifications are broadcast repeatedly, but with constantly changing argument values. The quicker a message or event's content grows stale after it is sent, the more appropriate it becomes to deliver it to services available at that instant only.

In the following section, we discuss how the delivery of ambient messages can be decoupled in time, by widening the temporal scope during which the message can be delivered.

6.4.5.2 Transient Delivery

The potential receivers for an ambient message annotated with `@Transient(t)` are all of the objects in its carrier's reach from the moment it is sent until the timeout period Δt_d (represented as an amount of milliseconds `t`) has elapsed. Hence, an ambient message with a transient discovery lifetime may be received by objects which are not in communication range at the time the message was sent, but which do enter communication range before Δt_d elapses. Instant message delivery can now be regarded as transient delivery with a timeout period equal to zero. Figure 6.3 adequately depicts a message with a transient discovery lifetime.

Transient message delivery decouples sender and receiver in time, but only up to a given period of time. Hence, messages with a transient discovery lifetime may still be lost if no objects are in reach when the timeout period has elapsed. Transient message delivery generalises instant delivery in the sense that the sender can more accurately describe when the information represented by the message becomes stale and should be dropped. An example of transient message delivery was given in section 6.2.1, where it was used to broadcast a stock quote update to clients accessible within the next 5 minutes.

In the following section, we describe the final kind of discovery lifetime for an ambient message.

6.4.5.3 Sustained Delivery

A message annotated with the `@Sustain` type tag is considered to have an infinite discovery lifetime[6]. The delivery of the message is sustained unconditionally, until the sender explicitly retracts the message. The potential receivers of a sustained message are all objects in its carrier's reach from the the moment it is sent to the ambient reference. In figure 6.3, if the message m were to have a sustained discovery lifetime, it would also be sent to s_5 because its discovery lifetime has no upper bound.

Sustained message delivery totally decouples sender and potential receivers in time. Only if the sender explicitly retracts the ambient message is the delivery cancelled. In order to illustrate the retraction of sustained messages, reconsider the shopping assistant example from section 6.4.4.2. The code below shows the necessary additions to stop the delivery of the `query` message.

```
def ShoppingAssistant := object: {
  def bestPriceSoFar;
  def bestShopSoFar;
  def deliveryHandle;
  def startSearch() {
    def shops := ambient: Shop;
    deliveryHandle := shops<-query(item)@[All,Sustain,Reply];
    whenEach: deliveryHandle.future becomes: { |reply|
      if:((bestPriceSoFar == nil).or:{reply.price < bestPriceSoFar}) then:{
        [bestPriceSoFar, bestShopSoFar] := [reply.price, reply.shopId];
        updateGUI(bestPriceSoFar, bestShopSoFar);
      }
    }
  };
  def stopSearch() {
    deliveryHandle.cancel(); // stop broadcasting the query message
  };
}
```

In the above example, message delivery is explicitly cancelled when `stopSearch` is invoked. It could for instance be invoked implicitly by the user via the shopping assistant's GUI. Cancellation is achieved by invoking the `cancel` method of the delivery handle returned by the ambient message send. If an ambient message has an instant lifetime, invoking `cancel` has no effect, as the message is already considered cancelled. In the case of a transient lifetime, an invocation of `cancel` is treated as if the timeout period had elapsed, causing the speaking phase to end.

We can use sustained discovery lifetime to encode the delivery semantics of the original location service example from section 6.1.1.1. In the original example, if a mobile client was disconnected from any location service, its location updates were buffered. The code using ambient references is the same as that shown previously when discussing instant discovery lifetime, only now a sustained discovery lifetime is used to ensure that the message is not lost upon disconnection.

[6] We first considered the term "persistent delivery" but refrained from using it as the term is heavily overloaded and hints at the fact that the message would be stored on persistent storage, which is not the case.

```
def accessPoint := ambient: LocationService;
def updateLocation(loc) {
  accessPoint<-submitLocation(clientId, loc)@[One,Sustain,Oneway];
};
```

In general, sustained message delivery is useful for publishing information and for keeping it online, not for representing "events" that are only relevant for a short period of time. For example, mobile clients can use it to keep information available for one or more remote services (like the client's location coordinates in the above example), without any assumptions on when that information will be consumed.

Summary An ambient message's discovery lifetime acts as a temporal scope during which objects entering its carrier's reach are regarded as potential receivers for the message. While point-to-point messages may be retracted from an ambient reference after they have been delivered once, one-to-many messages must remain available throughout their discovery lifetime. We distinguish three possible choices: message delivery may be instant (the message is sent only to objects in reach at the time it is sent), transiently (it remains available for a given timeout period) or sustained (it remains available until explicitly cancelled).

We have now discussed the three delivery policies of ambient messages – arity, communication lifetime and discovery lifetime. In the following sections, we study their interactions. In the following section in particular, we study the relationship between the speaking and listening phases during the delivery of an ambient message.

6.4.6 Relating Discovery Lifetime and Communication Lifetime

As previously stated in section 6.3, we distinguish two phases in the delivery of an ambient message. During the speaking phase, the carrier of the ambient message actively discovers potential receivers for the message. Whenever a receiver has been discovered, the ambient message is sent to that receiver. If a reply to the message is required, a listening phase is then started to await that reply. These phases are depicted in figures 6.2 and 6.3. For a point-to-point message, the speaking phase lasts until at least one potential receiver has been discovered. For a one-to-many message, the speaking phase may proceed in parallel with multiple listening phases, as the message may be sent to multiple actual receivers. In the previous two sections, we have described the different mechanisms to control the lifetime of each of these phases.

With both communication and discovery lifetime explained, we now discuss how they combine. The combined lifetimes determine the total delivery time of the message. The total delivery time provides an upper bound on the resolution of any future associated with the message. Table 6.1 gives an overview of the different combinations between the choices for communication lifetime (the rows) and discovery lifetime (the columns). The values depicted are the sum of the corresponding lifetimes. Δt_c and Δt_d represent timeout periods specified by the programmer.

Evidently, the total delivery lifetime of a message is only bounded if both its communication and discovery lifetimes are bounded. However, because communication and discovery lifetimes are specified using separate annotations by the programmer, the total delivery time of a message can be very implicit in the code. This is especially the case when combining a bounded discovery or communication lifetime with an unbounded one (because the result will be unbounded as well). For example, a program-

	`@Instant`	`@Transient(`Δt_d`)`	`@Sustain`
`@Oneway`	0	Δt_d	∞
`@Due(`Δt_c`)`	Δt_c	$\Delta t_c + \Delta t_d$	∞
`@Reply`	∞	∞	∞

Table 6.1: Combining communication (rows) and discovery (columns) lifetimes.

mer may annotate a message with `@Due(minutes(5))` and `@Sustain`. The resulting future is not ruined with a `TimeoutException` 5 minutes after sending the message to the ambient reference, but 5 minutes after sending it to an *actual receiver*. Since it may take forever to discover an actual receiver (because of the sustained discovery lifetime), the future may never be ruined at all. In order to explicitly correlate communication and discovery lifetimes, we introduce the *expirable* delivery policy, discussed below.

6.4.6.1 Expirable Ambient Messages

An expirable ambient message is annotated with `@Expires(t)`. Contrary to the timeout periods of `@Due(t)` or `@Transient(t)` which only specify the lifetime of the communication or speaking phases respectively, the timeout period of `@Expires(t)` specifies the entire lifetime of the message delivery process. Whereas the timeout period of `@Due(t)` is relative to the time at which the message is sent *to a discovered receiver*, the timeout period of `@Expires(t)` is relative to the time at which the message is sent *to the ambient reference* itself. The longer it takes to discover a potential receiver for a message, the less time is allotted for that receiver to reply to the message. An ambient message annotated with `@Expires` is implicitly also annotated as `@Due` *and* `@Transient`. Hence, annotating a message with `@Expires(t)` fixes both its communication lifetime and its message lifetime.

We now specify how discovery and communication lifetime are related by `@Expires(`Δt`)`. Δt is the total delivery lifetime of the ambient message. The communication lifetime of such a message can only be determined once a potential receiver is discovered. Because Δt covers the entire delivery lifetime, we get $\Delta t = \Delta t_c + \Delta t_d$ and it directly follows that $\Delta t_c = \Delta t - \Delta t_d$. The time interval Δt_d denotes the amount of time it took the ambient reference to discover the receiver. It can be rewritten as the difference of two absolute time values: t_d, the time at which the receiver was discovered, and t_s, the time at which the message was sent. Hence, $\Delta t_d = t_d - t_s$ and by substitution $\Delta t_c = \Delta t - (t_d - t_s)$.

Figure 6.4 illustrates the delivery of an expirable one-to-many ambient message. Note that the communication lifetime (the length of each listening phase) is now correlated with the time at which the receiver is discovered. The longer it takes to discover a receiver, the shorter the listening period and the less time is available to wait for the reply (for example, s_4's reply arrives too late and is discarded). Also note that the total delivery lifetime of the message does not exceed the timeout period specified by the programmer (which corresponds to the length of the speaking phase).

In short, an ambient message's total delivery time is only bounded if it is not annotated with either `@Reply` or `@Sustain`. While it is possible to define the total delivery time implicitly in terms of Δt_c and Δt_d, it is often more relevant to express the total delivery time Δt explicitly. Expirable ambient messages cater to this by explicitly correlating discovery lifetime with communication lifetime. In the following section, we

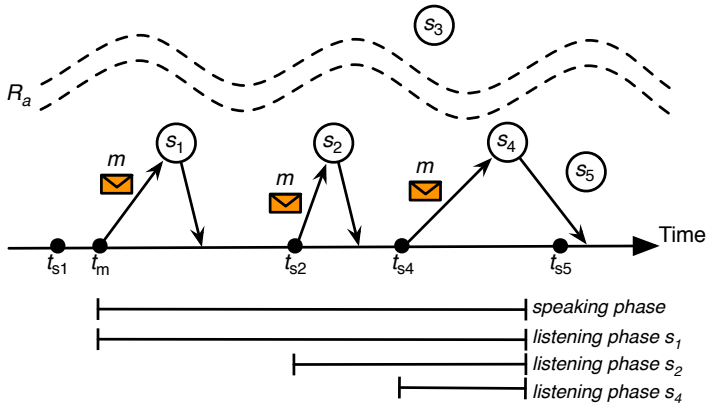

Figure 6.4: Delivery of an expirable one-to-many ambient message.

describe the effect of these lifetimes on the futures associated with two-way ambient messages.

6.4.6.2 Lifetime and Futures

If an ambient message's total delivery lifetime has expired, its future becomes totally resolved. In the case of a point-to-point message, the future becomes ruined with a `TimeoutException` at that point. In the case of a one-to-many message, any **`when`** observers registered on the multifuture are equally notified with a `TimeoutException` to ensure that multifutures remain interchangeable with regular futures.

Multifutures provide more possibilities for synchronisation than regular futures. We previously described the possibility to register a **`whenAll`** observer on a multifuture. Such observers are not triggered by incoming results or exceptions, because it is never known how many replies should be received (because the total number of receivers of a one-to-many message is not known a priori). Rather, these observers are triggered when the multifuture becomes totally resolved. In other words, **`whenAll`** observers are triggered when the ambient message's total delivery lifetime has passed.

To illustrate the combined use of multifutures and the expiration delivery policy, we revisit the voting application introduced in section 6.1.2.1. Recall that the goal of the application is to enable a team player in a mobile multiplayer game to issue a poll to nearby team players, allow them to reply within a limited period of time and then collect the results.

```
def nearbyTeamPlayers := ambient: Player where: { |p| p.team == myTeam };
def broadcastVote(poll, maxVoteTime) {
  def [future,resolver] := makeFuture();

  def handle :=
    nearbyTeamPlayers<-askToVote(poll)@[All,Expires(maxVoteTime)];
  whenAll: handle.future resolved: { |receivedVotes|
    resolver.resolve(receivedVotes);
  } ruined: { |exceptions|
    // ignore the votes of faulty players
  };
```

```
    future
};
```

It is instructive to contrast the above implementation with the original ones defined on pages 121–122. Note that `nearbyTeamPlayers` is no longer an explicitly managed collection, but rather an ambient reference. The `askToVote` message is sent to all team players in `nearbyTeamPlayers`' reach at the time the message is sent *and* all team players entering the reach before the vote time has elapsed. The explicit management of timeouts is avoided by means of the `@Expires` delivery policy.

The **`whenAll`** observer on the multifuture attached to the handle returned by the ambient message provides a convenient hook to close the vote and gather the results. In the original version, `receivedVotes` was a map from players to their corresponding answer. In the above version, it is simply a table of the answers. If information about the players that voted is required, the return value of `askToVote` can be changed into a tuple `[receiver,vote]` indicating who replied to the `askToVote` message. Finally, note that the `nearbyTeamPlayers` ambient reference is a reusable abstraction: it may be used by other parts of the application which can use different message delivery policies for their own ambient messages.

6.4.7 Summary

Ambient references always reference a set of proximate objects. However, they do not support a single kind of message passing semantics. Rather, the programmer can select a set of message delivery policies organised in a hierarchy. This hierarchy is shown in figure 6.5. Note that making a message expirable implies both a bounded communication lifetime and a transient discovery lifetime. The classification is exclusive, e.g. a message cannot be annotated both as a point-to-point and as a one-to-many message.

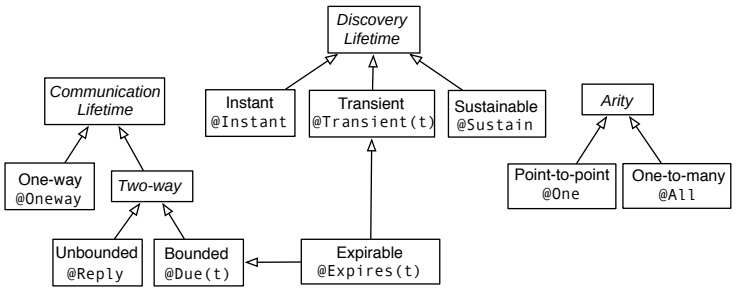

Figure 6.5: Taxonomy of Ambient Message Delivery Policies.

6.4.8 Interactions between Delivery Policies

One advantage of specifying the three kinds of message delivery policies independent of one another is that they can be combined to express a great variety of delivery semantics. From figure 6.5, we can derive the total number of possible message delivery policies: 3 (communication lifetime) $\times$ 3 (discovery lifetime) $\times$ 2 (arity) already makes for 18 possible basic combinations. Adding the possibility to annotate messages

with `@Expires` adds 1 (communication and discovery lifetime) $\times$ 2 (arity) is 2 more combinations, bringing the total to 20.

We will not go into details on each of the 20 specific combinations. Rather, we make some abstraction and discuss the pairwise interactions at the abstract policy level, resulting in $C_3^2 = 3$ possible combinations. As we will describe, some combinations cater to particularly useful kinds of interactions, while other combinations may prove dangerous, introducing pitfalls for the unwary programmer.

Arity and Discovery Lifetime The combination of these two delivery policies is perhaps the most important, largely determining the kind of interaction. For example, a point-to-point message with a sustained discovery lifetime expresses a time, space and synchronisation-decoupled communication with a remote object. The time decoupling can be relaxed by shortening the discovery lifetime.

A one-to-many message with an instant discovery lifetime adequately captures the event dissemination from a publisher to nearby interested subscribers. Depending on the kind of event, its temporal scope can be widened by using a transient discovery lifetime, increasing its chances to be delivered to nearby subscribers. An event is not usually broadcast with a sustained discovery lifetime. Sending a one-to-many message with a sustained discovery period can more be regarded as posting a message in a news feed or virtual bulletin board: the message remains available for any encountered object to read. For example, an ad hoc agenda application can use this pattern to broadcast appointments in its public agenda to all agenda peers it encounters. If the appointment is cancelled, the broadcast can be retracted.

Arity and Communication Lifetime The relation between these two policies has already been uncovered when discussing communication lifetime in section 6.4.4. Arity has no influence on one-way message sends, but it determines the result of a future-type message send. Because a point-to-point communication naturally results in at most one reply, a future is an adequate abstraction for the return value. A one-to-many communication may cause any number of replies to be generated, requiring multifutures instead. When using multifutures, the programmer should be aware that **`whenAll`** observers are only triggered implicitly if the ambient message's total delivery time is bounded. Otherwise, they are only triggered when message delivery is cancelled explicitly via the delivery handle.

Discovery and Communication Lifetime When combining these two policies, the programmer must be wary. When combining a temporary discovery lifetime (e.g. instant or transient discovery) with an unbounded communication lifetime, this may lead to a situation where a message is not delivered to any receiver (because the reach of the ambient reference remains empty), yet the sender relies on the future's resolution to trigger the remainder of the computation. In such a situation, we know for sure that the future will never be resolved (there is no receiver to resolve it). Hence, when using a temporary discovery lifetime in combination with futures, it is advisable to *always* bound the communication lifetime as well (using `@Due`).

One way of avoiding this pitfall is to change the semantics such that an ambient reference ruins a future with a special "message lost" exception if it knows that a message it is about to discard was not sent to any receiver. Nevertheless, using futures without bounding communication lifetime remains dangerous: the message may be delivered to a receiver who moves out of communication range before being able to reply. As

a result, the "message lost" exception is not triggered (a receiver was found), yet the reply might never be received so the future might never be resolved as well.

The relationship between discovery lifetime and the communication lifetime introduced by bounded future-type message sends has been extensively discussed in section 6.4.6. The two represent the lifetime of different stages in the delivery of the message: the time until a receiver is discovered and the time to wait for a reply. Communication lifetime relative to message sending time (`@Expires`) necessarily constrains discovery lifetime to be transient. It puts an upper bound on message delivery (ruling out `@Sustain`) yet allows the message to be sent to any receiver in reach until the timeout has elapsed (ruling out `@Instant`).

Communication lifetime by itself (`@Due`) does not influence discovery lifetime. However, the programmer has to be wary that using `@Due` in combination with `@Sustain` does *not* put an explicit upper bound on the resolution of the future associated to the message (cf. table 6.1). However, the situation is less hazardous than the one described previously because there is always a chance that a potential receiver can still be found, because the message remains available for delivery.

6.5 Delivery Guarantees

In the previous section, we have discussed at length how ambient references discover potential receivers for their ambient messages. However, we have yet to explain the precise delivery semantics of messages transmitted when a receiver is found.

In the literature, the most widely known message delivery semantics are *best effort* (the system tries hard to deliver the message, but does not give any other guarantee), *at least once* (the message may potentially be delivered multiple times), *at most once* (the message is either delivered once, or not at all) and *exactly once* (the message is guaranteed to be delivered without duplicates, provided that any system or network failure is finite) [CDK05]. In ad hoc networks, it is extremely hard to provide strong delivery guarantees, because network partitions can often be permanent (i.e. devices move out of range and never meet again) [MMH05]. Moreover, because it is generally undecidable to discriminate between network and machine failure, even in stationary networks an *exactly once* semantics is rarely guaranteed. Java RMI for example, employs *at most once* semantics [Sun98]. Stronger semantics are often found in message queuing systems like JMS, where messages can be made persistent to survive the failure of both message receiver and the delivery system itself.

Ambient references provide an *at most once* delivery guarantee for point-to-point messages, but only guarantee *best effort* delivery for one-to-many messages. Replies to ambient messages are delivered to the ambient message's future at most once. We discuss these choices in more detail below.

6.5.1 Point-to-point Messages

As discussed in section 6.3, a point-to-point message m is delivered to an actual receiver s selected from the set of potential receivers, i.e. $s \in \mathcal{PR}_m$. If $\mathcal{PR}_m = \emptyset$ then the message is not sent to any receiver. Otherwise, it is delivered at most once to the single receiver. The message may get dropped due to transmission errors, in which case it is also lost. In general, one can assume that the methods of exported objects invoked by means of point-to-point messages are triggered at most once per message send.

The benefit of at most once semantics is that it does not require heavyweight techniques to implement (e.g. persistence, transactions) [GDL+04], while it provides a manageable abstraction for the programmer which is relieved from the burden of filtering out duplicate (retransmitted) messages. For two-way point-to-point ambient messages either the future attached to the message is resolved with a reply (this could be a simple acknowledgement) or it is ruined with a `TimeoutException` if no reply arrives during the listening phase.

Note that a future ruined with a `TimeoutException` *does not imply* that the message was not received. It may be the case that a message has been successfully received by a remote party, but that the reply (acknowledgement) is lost or arrives late. Hence, the sender regards the message send as failed without the receiver being aware of this. The message may thus trigger a computation which remains unacknowledged to the sender. Such computations are known in the literature as *orphans* [CDK05] (figuratively speaking, these are computations "without a parent" to report their results to). One of the major difficulties in distributed programming remains the design of applications that can gracefully recover from such situations in which different parties have a diverged view on the state of the distributed system.

6.5.2 One-to-many Messages

For one-to-many messages, the delivery semantics is more complex. Recall from section 6.3 that such messages are delivered to all potential receivers. In order to prevent a message from being delivered more than once to the same receiver, either the ambient reference must remember *all* previous receivers to which it already sent the message, or the receivers must remember *all* ambient messages which they have previously received. In either case, bookkeeping information is necessary to filter out duplicate messages. If the lifetime of an ambient message is finite, then the set of potential receivers is similarly finite and as a result the bookkeeping information can be removed at some point in time.

The problem lies with ambient messages with an unbounded lifetime (e.g. a message annotated with `@Sustain`). For such messages, the set of potential receivers is unbounded and as a result there is no predefined point at which the bookkeeping information can be forgotten. Because it is not realistic to store such bookkeeping information indefinitely, we weaken the delivery guarantees for one-to-many messages. More specifically, a one-to-many message is delivered at most once only if its lifetime is smaller than some predefined constant, which we term the *recall period*. The recall period indicates the maximum amount of time until which an ambient reference can "recall" the identity of a previous receiver or alternatively, until which a receiver "recalls" a previously received message. If a one-to-many message's lifetime exceeds the recall period, it may be delivered multiple times to the same receiver. Such duplicate message delivery only occurs after an object becomes disconnected and then reconnects at a later point in time. Nevertheless, in general, one-to-many messages have only best effort semantics.

Figure 6.6 depicts the impact of the recall period on message delivery. We assume that *m* is a one-way, one-to-many message with a sustained discovery lifetime. The dotted arrows denote the movement of service objects. *m* is sent to both s_1 and s_2. Both services subsequently move out of reach. Because s_1 reconnects within the duration of its recall period, it is "recalled" by *a* and does not receive *m* again. s_2 on the other hand, receives *m* twice because it is regarded by *a* as a "new" service (s_2 is not recalled

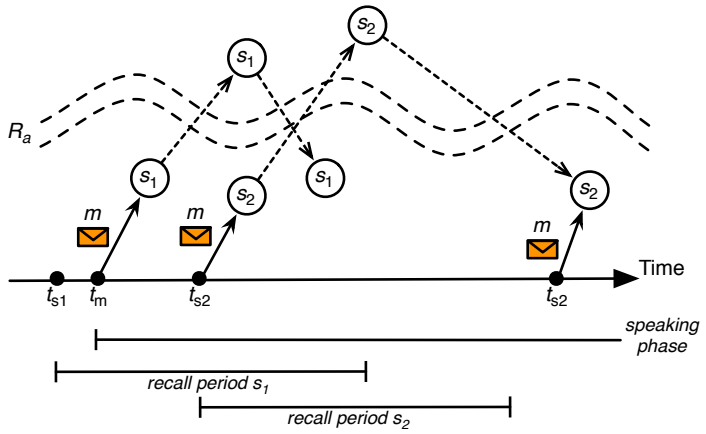

Figure 6.6: Impact of an ambient reference's recall period on message delivery.

by *a* because its disconnection outlasted its recall period).

In short, when performing one-to-many communication using transient or sustained ambient messages, the AmbientTalk programmer must be aware that such messages may trigger the method of actual receiver objects multiple times. This requires the programmer to either take duplicate filtering measures at the application-level or to ensure that the invoked method is *idempotent* [CDK05], i.e. that duplicate invocations have no inadvertent effects (e.g. because the method performs no side-effects).

6.5.3 Delivery Order

A final aspect of message delivery is the *order* in which ambient messages are received by their receiver(s) with respect to the order in which the messages were sent. Here, again, the discovery lifetime plays an important role. Multiple messages with an instant discovery lifetime sent to the same ambient reference will arrive in the same order at actual receivers.

For messages with a prolonged (i.e. transient or sustained) discovery lifetime, ambient references do not specify any order on the way ambient messages are delivered to potential receivers. When a new potential receiver enters an ambient reference's reach, it may receive any pending ambient messages sent to that ambient reference in any order.

Note that when using transient messages, any delivery order is always relative to the particular discovery lifetimes of the messages at hand. Consider the following example:

```
a<-first()@Expires(seconds(5));
...
a<-second()@Expires(seconds(1));
...
a<-third()@Expires(seconds(3));
```

Given code like this, we can never guarantee that a receiver always receives the sequence `first`,`second`,`third`. A potential receiver entering `a`'s reach after 2 seconds may receive `first` and `third`, but no longer `second`. The programmer must thus be

aware that for messages with a prolonged discovery lifetime, the time at which the message is sent to the ambient reference is irrelevant in terms of message ordering.

The lack of any delivery order on prolonged messages can be compared to the lack of any order on tuples in a tuple space [Gel85]. In tuple spaces, subsequent `in(tuple)` operations performed by a consumer process are not necessarily correlated with subsequent `out(tuple)` operations performed by a producer process: every `in(tuple)` operation may retract *any* matching tuple in the tuple space. Stronger guarantees must be incorporated at the application-level.

Finally, note that an explicit order can be defined on messages by using a pattern of nested **`when:becomes:`** observers. For example, the following code guarantees that `second` is sent to a remote service only after `first` was successfully received and processed by an actual receiver:

```
when: a<-first()@[One,Sustain,Reply] becomes: { |ack|
  when: a<-second()@[One,Sustain,Reply] becomes: { |ack2|
    // first and second have been processed in order
  }
}
```

In short, it is guaranteed that `@Instant` messages are delivered to actual receivers in the order in which they were sent to the ambient reference. For messages with a prolonged discovery lifetime no such ordering is guaranteed. If a delivery order is required, it must be implemented either by means of acknowledgements (**`when:becomes:`**) or by passing application-level sequencing information in the ambient messages.

6.6 Reintroducing Connection-oriented Designation

In section 6.2.2, we highlighted that the most distinguishing feature of ambient references is that they are object references supporting a form of connectionless (as opposed to connection-oriented) object designation. This feature is at the heart of their ability to support roaming. However, we also mentioned that roaming is not always a desirable abstraction as it only works for either stateless communication or for services which can synchronise by means of infrastructure (e.g. in a nomadic network). In this section, we introduce abstractions that reintroduce connection-oriented designation but *without* sacrificing the space decoupling afforded by ambient references.

6.6.1 Anonymous Far References

In section 6.2.2 we pointed out that ambient references trade stateful communication for space-decoupling while far references trade space-decoupling for stateful communication. In this section, we introduce an abstraction that strikes a balance between both, which we shall refer to as *anonymous far references*.

Anonymous far references are required when communication must be space-decoupled yet stateful. Recall the RFID shopping cart example from section 6.1.3. In the example, we required an ad interim remote object reference to a `ProductDatabase` service to act as a stand-in while no `ProductDatabase` was available. We named such references *provisional* because they act as a stand-in that allows clients to send messages to and pass around the stand-in object when a real service is not yet discovered. Far references fail to represent such provisional references because they are not decoupled in space, they must always refer to a known object. In section 6.4.4.2 we have

shown that ambient references – in combination with sustained message delivery – can be used to represent a provisional reference to a `ProductDatabase`.

However, the problem is that ambient references form a provisional reference to *any* service object in their scope. That is, ambient references cannot guarantee statefulness: subsequent point-to-point messages are not guaranteed to be received by the *same* service object (recall that point-to-point messages are delivered to a non-deterministically chosen potential receiver). While this behaviour enables roaming, recall that roaming is only a suitable abstraction when the communication protocol between client and service object is stateless or if the different designated service objects can exchange session information. In the shopping cart example, if a reference to only *one* `ProductDatabase` service is required, ambient references are not the right abstraction either.

In section 6.1.3.2 we already provided a pattern for implementing provisional references in AmbientTalk without reference to ambient references, by combining first-class futures with a service discovery event handler. We pointed out that what is lacking is a proper referencing abstraction that allows programmers to abstract from the pattern. Anonymous far references form such an abstraction.

6.6.1.1 Anonymous Far References in AmbientTalk

An anonymous far reference is a remote object reference that transparently discovers and binds to a service object by means of an intensional classification mechanism, like a type tag or protocol. Hence, an anonymous far reference, like an ambient reference, has a scope. However, unlike an ambient reference, the anonymous far reference does not designate *all* objects in its scope but rather *one* non-deterministically chosen object from its scope (the first object to enter communication range, to be precise). Once an anonymous reference is bound to a remote object, it behaves just like a far reference to this object. Reconsidering the RFID shopping cart example from section 6.1.3.1, we may construct a provisional reference to a proximate product database server as follows:

```
def renderGUI(server) {
  /* as in the initial implementation */
};
def database := discover: ProductDatabase;
renderGUI(database);
```

The expression **`discover:`** `ProductDatabase` initiates a service discovery request for a remote object exported as a `ProductDatabase` and immediately returns an anonymous far reference to such an object. Initially, this reference is unbound, i.e. it does not designate any object yet. Only when a matching object is found does the reference become bound. However, objects can safely send messages to an anonymous far reference even if it is unbound. As can be expected, messages sent to an unbound reference are buffered and forwarded later, when the reference becomes bound.

Figure 6.7 depicts the situation where an anonymous far reference is created, but where no matching service object has yet been found. The type tag with which the anonymous far reference is initialised is depicted as a diamond shape. The object `A` refers to an unbound anonymous far reference that will bind to objects whose type tag "matches" the diamond shape. Conceptually, an unbound anonymous far reference is

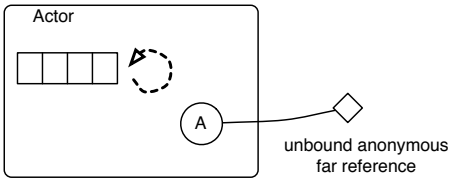

Figure 6.7: An unbound anonymous far reference.

a dangling remote reference. Any messages sent to this reference will be accumulated until it is bound.

Figure 6.8 depicts the situation where an actor hosting a matching exported service object enters communication range. The object is shown to be exported under a type tag "matching" the diamond shape of the reference. Because a matching service object has been found, the anonymous far reference at `A` becomes bound to this remote object. Any messages that were previously accumulated are forwarded to the remote service object.

When the two actors move back out of one another's communication range, the anonymous far reference does *not* revert to an unbound status. Rather, it maintains the bond with the remote service. Hence, once an anonymous far reference becomes bound, its behaviour is indistinguishable from that of a regular far reference. This behaviour is what makes anonymous far references a stateful, connection-oriented communication channel, like a far reference. Hence, the `database` variable in the code excerpt above refers to *at most one* `ProductDatabase` object, and all messages sent to it are sent to that same object. We thus traded the roaming abstraction of ambient references for the statefulness of far references.

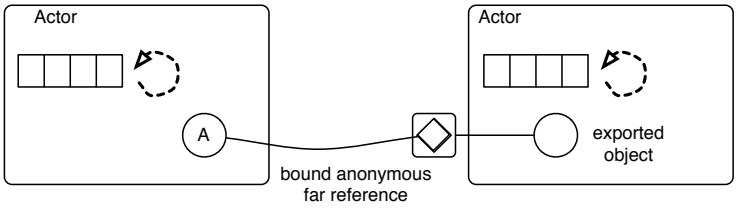

Figure 6.8: A bound anonymous far reference.

6.6.1.2 Futures for Service Discovery

There is an interesting parallel to be drawn between traditional futures on the one hand and anonymous far references on the other hand:

- In the same way that futures allow one to abstract from the return value of an asynchronous message send (i.e. the result may or may not yet have been computed), anonymous far references allow one to abstract from the status of an asynchronous discovery request (i.e. a suitable service has or has not yet been found).

- Futures start off in an unresolved state and can later become resolved. Anonymous far references start off in an unbound state and can later become bound.
- Once resolved, a future essentially becomes a proxy for the computed value. Once bound, an anonymous far reference essentially becomes a proxy for the discovered service.
- A future is a first-class reference to a value that is yet to be computed. Anonymous far references are a first-class reference to a value that is yet to be discovered.
- Messages sent to an unresolved future (in AmbientTalk and E) are buffered and forwarded later, when the future becomes resolved. Messages sent to an unbound anonymous far reference are buffered and forwarded later, when the anonymous reference becomes bound.

Strengthened by the above parallel, we claim that anonymous far references are to service discovery what futures are to asynchronous method invocations. As such, they bring about the same advantages:

- Futures allow a caller to refer to the result of an asynchronous computation in the same scope as where that computation was started, and not in a separate callback. Likewise, an anonymous far reference can be used in the same scope as where a remote service is required.
- A future allows a caller to proceed in parallel with some asynchronous computation. Likewise, declaring an anonymous far reference initiates an asynchronous service discovery process, but the declaring object can immediately proceed.

To summarise, anonymous far references form a middle ground between far references and ambient references. They combine the space-decoupled designation of ambient references with the stateful, connection-oriented communication provided by far references. Anonymous far references are not a special kind of ambient references because they do not designate a volatile *set* of objects, but rather *one* non-deterministically chosen element from a volatile set of objects.

6.6.2 Snapshots

Ambient references designate a volatile set of objects and therefore cater to anonymous collaboration with an unknown number of proximate objects. Sometimes, even though a collaboration starts off with an unknown number of peers, it may at some point in time be necessary to fix the objects communicated with. For example, in a multiplayer game for mobile phones, before a game can be started, a number of opponents need to be discovered. As the number and identity of other peers is not yet known, this requires some form of anonymous communication. However, once a game session starts, the set of players is usually fixed and the game must ensure that further session-specific information is only routed to the players that joined the game.

To this end, it is possible to acquire an extensional representation of the objects designated by an ambient reference by making a *snapshot* of its reach. This representation is an enumeration of all the potential receivers at the time the snapshot is made. It is represented as a standard AmbientTalk table of far references to the remote objects. Using a snapshot, one converts communication with a volatile set of objects that

is decoupled in space into communication with a precise set of objects that is no longer decoupled in space.

As a concrete example, if `players` is an ambient reference to all `Player` objects, one may create a snapshot as follows:

```
def enum := snapshot: players;
```

The variable `enum` is a reference to a (possibly empty) table which can then be iterated over using e.g. the standard `each:` method. It is never guaranteed that the enumeration accurately reflects the current availability of services. The moment the snapshot is made, objects contained within it may move out of range. However, because the snapshot contains far references, communication with the remote objects is still decoupled in time and synchronisation.

In general, snapshots enable one to convert arity-decoupled communication (i.e. communication with an unknown number of services) into arity-coupled communication (i.e. communication with a known number of services). Furthermore, a snapshot reveals the identity of the objects communicated with.

6.6.3 Multireferences

As explained in the previous subsection, a snapshot represents the extensional enumeration of the reach of an ambient reference as a simple table of far references. While this representation gives full control over the set to the AmbientTalk programmer, the downside of such a representation is that the programmer can no longer use the expressive one-to-many communication abstractions provided by ambient references to communicate with objects in that set. In particular, one can no longer represent a broadcast to the set by means of a single message send (cf. the `@All` annotation for ambient messages) and one can no longer use multifutures and **`whenEach`** or **`whenAll`** observers to expressively gather results.

We reintroduce these abstractions by means of *multireferences* [Ded06]. A multireference – like an ambient reference – denotes a group of objects. However, unlike an ambient reference, it denotes a *fixed* rather than a volatile group of objects. Any message sent to a multireference is automatically broadcast to all objects in its set and returns a multifuture which can be used to gather all replies. A multireference can be created from any table of objects.

If multireferences are created from a table of far references constructed by the application, they represent a communication channel which is connection-oriented (they refer only to the given objects) but no longer anonymous (the objects in their set are explicitly enumerated). However, multireferences can also be used in tandem with **`snapshot`** to create a communication channel which is both connectionless and anonymous: while the table returned by **`snapshot`** is fixed, its contents is generated by an ambient reference and never explicitly enumerated in the application code. As a concrete example, one may create a multireference to all nearby players as follows (still assuming `players` represents an ambient reference to nearby players):

```
def opponents := multiref: (snapshot: players);
```

The `opponents` variable contains a multireference encapsulating the table of players. Rather than using explicit iteration to communicate with the opponents, one may now use familiar asynchronous message sends to communicate with all objects encapsulated by the multireference.

In general, multireferences support one-to-many communication with a *fixed* group of objects. When combined with snapshots, they form a communication abstraction that maintains both the statefulness and the anonymity of the communication.

6.6.4 Summary

While ambient references cater to anonymous interactions, they do not cater to stateful, connection-oriented designation. Statefulness is reintroduced by means of anonymous far references, snapshots and multireferences. Figure 6.9 depicts a flowchart that relates different kinds of designation with the object referencing abstractions that have been introduced. In the iconic representation of the referencing abstractions, a cloud denotes a volatile set while a circle denotes a constant set. A dashed line indicates an intensionally specified set while a solid line indicates an extensionally specified set.

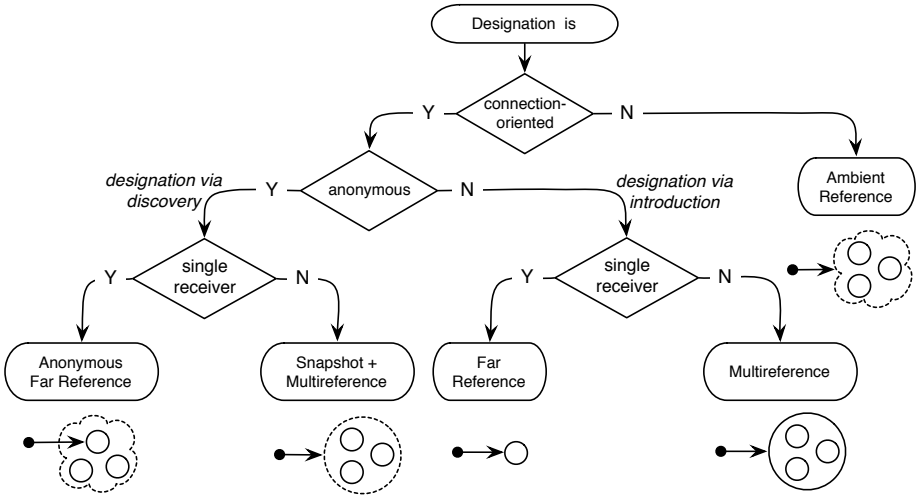

Figure 6.9: Flow chart classifying referencing abstractions according to designation.

6.7 On the Scale of Time and Space

Throughout this chapter, we have often referred to measures of time and space when discussing ambient references. For example, a one-to-many ambient message can be sent to "many" different encountered devices and the lifetime of an ambient message is "some period" of time. But how much is "many", and how long is "some period"? As language designers, we cannot decide on the precise scale of these values in the application programmer's stead, as they are application-dependent. What we can do, however, is to put the notions of time and space in perspective with respect to the kind of applications for which ambient references have been designed.

As noted by Grimm et al., what characterises ubiquitous and pervasive computing applications is the *human* scale at which they operate [GDL+04]. Because such applications interact with humans, the measures they use for time and space are similar to those employed by humans. So, when considering interactions among people, we can

expect interactions among tens to hundreds of devices. AmbientTalk and ambient references are designed for ad hoc networks connecting people (through a mobile device they carry with them), so it is designed for the human scale of such networks. It is not designed to connect thousands of nodes in a wireless sensor network, or to scan tens of thousands of RFID tags on containers in a harbour.

Similarly, with respect to time we expect timeouts for ambient messages to be in the order of human time scales, such as seconds, minutes, hours or even days or weeks. The timing abstractions for ambient references were not designed for dealing with sub-second time frames. They were definitely not designed for real-time systems.

6.8 Conclusion

At the start of this chapter, we set out to explore the basic problems that an AmbientTalk programmer has to deal with when creating applications that require the coordination of proximate objects in an ad hoc network. We demonstrated the various issues by means of concrete examples. Subsequently, we introduced the abstraction of an ambient reference: a space-decoupled object reference to a volatile set of proximate objects. Ambient references carry ambient messages: object-oriented messages without a receiver, but annotated with a set of policies that steer their delivery process. They deal with the discussed issues as follows:

Roaming is achieved by sending point-to-point messages to an ambient reference designating the set of service providers. Because the scope of an ambient reference is specified intensionally, one may abstract from the particular service instances to which a message is sent.

One-to-many Communication is achieved by means of one-to-many messages. Multifutures enable the programmer to perform arity-decoupled communication without losing the beneficial request-response interaction pattern normally associated with an object-oriented message send.

Provisional Services can be represented by ambient references to which messages are sent with a sustained discovery lifetime. Messages are buffered within the ambient reference until one or more matching services become available.

By providing the programmer with a wide range of options to control the delivery process, the at first disparate set of problems can all be succinctly expressed by means of ambient references. Even though the set of delivery policies to choose depends on the specific problem at hand, the core abstraction is always the same: programs send messages to references representing a volatile set of proximate objects.

When stateful communication is required, the abstraction of communicating with a volatile set of objects is not appropriate. While far references can guarantee statefulness, they do not cater to any form of anonymous communication. As a result, we have introduced anonymous far references, snapshots and multireferences whose goal is to reconcile stateful communication with space decoupling.

In this chapter, we have primarily focussed on the different kinds of communication patterns which ambient references can express. We have not yet described how ambient references help in resolving the object-event impedance mismatch. This is one of the focal points of the next chapter, which describes ambient references in a more general context and relates them to other referencing abstractions.

Chapter 7

Ambient References in Context

In this chapter, we abstract from the technicalities of AmbientTalk and ambient references and put the novel language abstraction in context. First, we describe how a programmer, armed with the AmbientTalk language and the ambient reference abstraction can conveniently express coordination in a MANET. We do this by means of an evaluation with respect to the criteria postulated in section 3.2. Subsequently, we describe how ambient references combine the power of event-based publish/subscribe systems with an object-oriented programming style. Finally, we relate ambient references to our own prior work and to related research.

7.1 Evaluation

In section 3.2, we postulated six criteria to which a good coordination abstraction for mobile ad hoc networks should adhere. In this section, we evaluate how AmbientTalk and ambient references conform to these criteria.

7.1.1 Decentralised Discovery

AmbientTalk is equipped with a type-based publish/subscribe engine that directly caters to decentralised service discovery. By registering discovery event handlers which trigger upon the presence of a nearby service, an application can react to changes in its physical environment. While the built-in discovery functionality is limited to describing remote objects by means of a type tag, library functions exist which augment this functionality with discovery based on protocols and additional filtering using AmbientTalk predicates.

Ambient references themselves do not directly provide service discovery. They always designate a collection of proximate objects implicitly. This implicit collection can be made explicit by means of snapshots if required. Of course, ambient references can be used to implement service discovery. For example:

```
def whenever: TypeTag discovered: handler {
  def nearbyServices := ambient: TypeTag;
  def hdl := nearbyServices<-ping()@[All,Sustain,Reply];
  whenEach: hdl.future becomes: handler;
}
```

In the above code snippet, the primitive function **whenever:discovered:** (cf. section 4.4.3) is redefined in terms of ambient references. The `ping` ambient message is continuously broadcast to all nearby services of the given type. By registering a **whenEach** observer on the multifuture associated with the one-to-many ambient message, we can conveniently keep track of replies to this message and hence of the presence of new services. For completeness, the above implementation relies on the fact that services are exported as follows[1]:

```
def export: service as: TypeTag {
  primitiveExport: (object: {
    def ping() { service }
  }) as: TypeTag;
}
```

If only a single service object needs to be discovered, anonymous far references are a more suitable referencing abstraction. An anonymous far reference combines service discovery with object designation such that it can represent a service object "yet to be discovered". Moreover, an anonymous far reference is always time-decoupled, such that messages sent to the reference are never lost.

In section 3.2.1, we noted the importance of a decentralised discovery mechanism for acquiring references to proximate services in MANETs. Such decentralised discovery is provided as a primitive abstraction in AmbientTalk but can also be expressed in the language itself by means of ambient references.

7.1.2 Loosely-coupled Communication

As noted in section 3.2.2, coordination abstractions between objects across a MANET should be decoupled as much as possible in time, space and synchronisation to reduce the impact on the application of the intermittent connectivity and the scarce infrastructure of the underlying network. Below, we discuss how AmbientTalk and ambient references achieve such decoupling.

7.1.2.1 Decoupling in Time

In AmbientTalk, eventual references by default fully decouple sender and receiver in time. The transmission time of a message may be bounded by annotating a message send with a `@Due` annotation, which eventually ruins the future associated with the message with a `TimeoutException`. An anonymous far reference caters to even more time-decoupling than a far reference because it also buffers messages sent to it while no matching object has been discovered yet. As stated previously, it additionally allows client objects to abstract from the availability of the "to be discovered" service.

The degree to which communication over an ambient reference is decoupled in time is controlled by the lifetime of an ambient message. Since ambient references distinguish between two phases in the delivery of an ambient message – the speaking phase and the listening phase(s) – there are two time periods that can be bounded. An ambient message with a sustained discovery lifetime and no bounded communication lifetime (`@[Sustain,Reply]`) essentially fully decouples sender and potential receivers in time, while a one-way ambient message with an instant discovery lifetime (`@[Instant,Oneway]`) does not decouple them in time at all. Sometimes neither of

[1] The function `primitiveExport:as:` is assumed to be an alias to the original **export:as:** function.

these choices is appropriate: we want to abstract over temporary disconnections, but at a certain point we need to consider the option that the disconnection may well be permanent and react upon it.

Thanks to the whole event-driven object-oriented framework underlying AmbientTalk, the various timeouts specified by means of the `@Due`, `@Transient` and `@Expires` annotations do not need to be implemented by suspending the control flow of the sender object. A timeout can be represented as an event that needs to be processed by the actor. Thanks to futures and their observers, a timeout can be dealt with in the appropriate (lexical) scope, where all relevant context-information regarding the original send is still available. Finally, note that the `@Expires` policy manages the correlation of discovery and communication timeouts such that the programmer does not need to deal with this explicitly.

7.1.2.2 Decoupling in Space

AmbientTalk enables a limited form of space decoupling through its explicit support for service discovery via type tags. It allows objects to coordinate without knowing one another's exact address. However, service discovery in itself is not a communication channel (i.e. an object reference). Rather, it is a mechanism by which communication channels (far references) to objects can be *acquired*. All effective communication is still expressed in terms of far references, which designate a well-defined receiver throughout time. For some applications, this stateful behaviour is useful. For applications that require support for roaming (i.e. being able to use equivalent services transparently), it is not.

Ambient references are decoupled in space by definition. They designate an abstract set of proximate objects by means of an intensional description of their scope (a type tag or protocol together with an optional filter predicate that may query the attributes of a service object). Whereas far references are useful for setting up a resilient point-to-point communication link with a specific service, as the musical match maker example of section 4.6 illustrates, ambient references provide a volatile but more opportune communication link with any matching service that happens to be available, regardless of which device hosts the service.

We argue that such opportunistic communication is often more appropriate when the dynamic execution context changes frequently, as in pervasive or ubiquitous computing applications. Grimm et al. [GDL+04] support this claim when discussing the difference between what they call "early" versus "late" binding in their one.world system for pervasive computing. In their system, early binding causes discovery to trigger once when routing an event, after which all subsequent events are routed to that same discovered service, while late bound events are always routed to any remote resource (service) matching a certain description (see section 7.5.5 for a more thorough explanation). The resulting behaviour is very similar to that of sending a message to an (anonymous) far reference versus sending a message to an ambient reference. Quoting Grimm. et al [GDL+04], p. 449:

> "In our experience, late binding is generally preferable over early binding for pervasive applications, as it is more responsive in an ever changing computing environment. However, if an application sends many, possibly large messages to the same receiver in short succession, the overhead of repeatedly resolving discovery queries becomes noticeable, and early binding represents the more appropriate choice. At the same time, with

> early binding, the application needs to be prepared to rediscover the receiver if its computing context changes."

The opportunistic addressing of ambient references provides direct support for roaming. Objects communicate with whatever matching service is available and as they roam, they continue their collaboration with potentially different instances of the same service. Hence, ambient references enable an application to adapt itself to a new environment without explicitly coding for reconfigurability. This advantage is not to be underestimated in a mobile network that is in a state of constant flux. For example, consider a not so distant future where cars will be communicating by means of a VANET (vehicular ad hoc network) with nearby cars on the highway or even with the infrastructure itself (roads, bridges, traffic signs, ...). In such a network, services that report accidents, road works, traffic jams, ... will be available on a multitude of different hosts; they will be updated without global administration and peers will join and leave the network unannounced at high frequency. In such networks, roaming becomes a key abstraction to manage the sheer complexity arising from such changes.

Anonymous far references are introduced in AmbientTalk to fill the gap between far references (which introduce no decoupling in space, but enable stateful communication) and ambient references (which are totally decoupled in space, but do not enable stateful communication). An anonymous far reference is decoupled in space, but once bound it remains bound to the same service object, guaranteeing statefulness. Hence, anonymous far references correspond to "early binding" in one.world, which as Grimm et al. mention can improve efficiency but at the cost of having to manually reconfigure the application should the service to which the anonymous far reference is bound disappear.

7.1.2.3 Decoupling in Synchronisation

AmbientTalk's event-driven execution model is totally aimed at decoupling sender and receiver objects in synchronisation. This is one of the key features of AmbientTalk. Because AmbientTalk is a programming language, this property can be *enforced*. While programmers can of course still simulate a "blocked" process, the language's design causes such behaviour to be more difficult to express, hence making programmers refrain from using that solution unless it is really critical to their problem at hand.

Any (proper) abstraction built on top of AmbientTalk maintains the synchronisation decoupling provided by the language. This includes ambient references, anonymous far references and multireferences. All three are deliberately represented as eventual references: they only support asynchronous message sending and they retain the use of futures and the associated observer closures to express non-blocking synchronisation. Ambient references and multireferences augment the synchronisation capabilities of futures via their support for multifutures. The **`whenEach`** and **`whenAll`** observers that can be registered on multifutures provide convenient synchronisation points for one-to-many communication.

From the receiver objects' point of view, messages originating from ambient references are not processed differently from messages originating from far references. Both kinds of messages are enqueued in the message queue of the actor owning the receiver ensuring that messages can be received even while the receiver object is busy processing another message. The receiver can implicitly reply to an ambient message by means of the return value of its triggered method.

7.1.2.4 Arity Decoupling

Arity decoupling enables the expression of a collaboration between an a priori unknown number of services. AmbientTalk by itself does not directly support any form of arity decoupling. Built-in far references only support point-to-point communication with a well-defined receiver. Anonymous far references also do not cater to arity decoupling, although they can bind with any *one* object chosen from an unknown number of proximate objects.

Ambient references support arity decoupling by introducing one-to-many ambient messages. Communication can only be expressed with a volatile group of proximate objects. However, by attributing a longer discovery lifetime to the ambient message, the programmer can increase the chances that the message is received by remote objects. Communication with a fixed group of objects is supported by multireferences (cf. section 6.6.3). The introduction of multifutures to represent the replies to a one-to-many ambient message allows the sender of a broadcast message to easily gather any replies.

In section 6.1.2, we already argued why a sequence of point-to-point messages falls short of representing the broadcast of information to a group of volatile objects. The advantage of representing broadcasting as a first-class operation in its own right is that the language construct can both hide the total number of receivers communicated with and the actual delivery of the message to the appropriate receivers [CA94]. The management of a (volatile) set of receivers in the application's stead is the key design goal behind ambient references. This management is not trivial: it requires a significant amount of coding effort in order to discover proximate objects and to explicitly manage the various delivery policies, as can be witnessed from the implementation of ambient references which is described in chapter 8.

7.1.3 Connection-Independent Failure Handling

One of the design goals of AmbientTalk (and AmOP in general) is to not regard network disconnections as "failures". Most obviously, far references are resilient to temporary disconnections. The failure event handlers introduced in section 4.4.1 enable a program to *react* to disconnections orthogonal to the communication that occurs along the far reference. This reaction does not by default cause any part of the application to abort (which would be the case if disconnections were modelled as programming language exceptions). The use of `@Due` to limit the lifetime of a message sent via an eventual reference also does not cause that eventual reference to become unusable once the message expires. It only causes the particular annotated message to time out, and it does not necessarily imply that message delivery failed.

The use of timeouts in combination with eventual references enables one to deal with failures orthogonal to the underlying state of connectivity of the ad hoc network. Of course, most often a timeout triggers because there is a network partition, so a message (or its reply) cannot be delivered in due time. However, it is perfectly possible for a timeout to occur when sender and receiver are connected. The receiver could have discarded a message, or it may not have been able to process the message in time because it first had to process other pending messages. Note that even in the case of anonymous far references or ambient references, failure handling is expressed at the messaging level and not at the reference level: while individual messages may time out, this does not render the underlying referencing abstraction invalid.

7.1.4 Summary

The language abstractions built into AmbientTalk already deal with four of the criteria proposed in section 3.2. Eventual references introduce time and synchronisation decoupling, service discovery introduces a form of space decoupling and timeouts and leased references introduce a form of connection-independent failure handling. However, AmbientTalk lacks communication abstractions which are explicitly space decoupled. Such abstractions are necessary to express roaming, one-to-many communication and provisional services. As ambient references enable the expression of exactly these communication patterns (cf. section 6.8), together AmbientTalk and ambient references are able to directly address the six criteria for adequately expressing coordination in mobile ad hoc networks. What is more, they express this coordination in an event-driven yet object-oriented way, as will be discussed in the following section.

7.2 The Object-Event Impedance Mismatch Revisited

We now turn our attention once more to the issues in combining objects with events, as explained in section 3.4. Before explaining how ambient references help to solve the issues, we describe the relationship between ambient references and event systems. How is it that ambient references enable AmbientTalk programmers to program in an event-driven yet object-oriented way?

Objects communicate via object references. Event-driven systems communicate via event brokers (the mediator between publishers and subscribers). The crux of ambient references is that they *represent the event broker as an object reference*. Therefore, an ambient reference can be regarded as a little publish/subscribe engine of its own. Table 7.1 contrasts the concepts of an event-based, publish/subscribe abstraction with those of ambient references. In a publish/subscribe system, publishers send events to an event broker, which is responsible for delivering those events to interested subscribers. With ambient references, the act of sending a message to an ambient reference represents the publication of an event for consumption by nearby interested objects.

	Publish/Subscribe	**Ambient References**
Data are	Events	Messages
To send	Publish an event	Send an ambient message
To receive	Register a callback	Export an object
To handle	Invoke a callback	Trigger (asynchronously invoke) a method

Table 7.1: Comparing Ambient References to Event notification systems.

In each of the following sections, we revisit the problems from section 3.4 in light of the ambient reference abstraction.

7.2.1 Specific versus Generic Communication

In section 3.4.1 we contrasted the specific communication interface of objects with the generic communication interface that is often provided by publish/subscribe architectures. Ambient references maintain the specific interface of objects by representing events as messages. This has both drawbacks and advantages. A drawback is that

this makes the representation of events explicit in the code. In the introductory stock quote example in section 6.2, the event `quoteUpdated(code, price)` is represented as a message send of which the message selector identifies the kind of event and the message arguments constitute the event data. Both sender and receiver need to be fully aware of the structure (selector and arguments) of the message.

There are two major advantages when representing events as messages. First, event publication can be unified with object-oriented message sending. Rather than having to explicitly construct an event as an object of a certain type and then invoking a generic `publish(Event)` method, a message is sent to an object representing the subscribers and the event's type becomes the selector of that message. Second, event handling can be unified with object-oriented method invocation. Rather than having to represent event notification by subscribing a generic `reactTo(Event)` callback method to an event type, it is represented by asynchronously triggering a method whose selector corresponds to the kind of event.

In short, ambient references resolve the impedance mismatch by representing events as messages, thus maintaining the specific communication interface of object-oriented message passing.

7.2.2 Connection-oriented versus Connectionless Designation

Recall from our discussion in section 3.4.2 that while object-oriented referencing abstractions provide connectionless communication but no space decoupling, pure event systems provide space decoupling but do not cater to any connection-oriented communication.

It should be clear from our exposition of anonymous far references that they have been explicitly designed to combine the advantageous properties of both object and event systems. They provide both space decoupling and statefulness, because once an anonymous reference is bound, it remains bound to the same object. Hence, while communication along an anonymous far reference is evidently anonymous, it is guaranteed that subsequent message sends are received by the same service. Snapshots together with multireferences enable stateful and anonymous communication with more than one object.

While anonymous far references and snapshots cater to connection-oriented designation, ambient references enable connectionless designation by definition. Ambient references designate a volatile set of objects. Because of this, they provide direct support for one-to-many communication. As a result, one-to-many messages can represent event notifications to nearby interested subscribers. While a pure event notification is best implemented using an instant discovery lifetime, time decoupling can be introduced in the event system by prolonging the discovery lifetime of the ambient message. Because communication across an ambient reference is stateless, multiple point-to-point messages may be received by different service objects, enabling roaming.

Even though ambient references provide connectionless designation, the programmer can still control what *kind* of objects the ambient reference may designate. Recall that an ambient reference is initialised with a characteristic function defining its scope; the set of objects which it may designate. Objects can be designated based on static attributes (type tags and protocols) or dynamic attributes. Ambient references share this property to restrict which objects they may designate with the ActorSpace model. As noted explicitly by Callsen and Agha [CA94], this is an improvement over the privacy provided by tuple spaces: tuple space-based communication provides no way of

specifying what kind of processes may or may not consume a tuple. In this regard, publish/subscribe systems are much like tuple space systems.

In tuple spaces and publish/subscribe systems it is generally the receiver of data that specifies what kind of *data* (tuples or events) it wants to accept, rather than the sender specifying what kind of *receiver* (subscriber) can accept the data. Ambient references invert this relationship. As a result, they have the advantage that their scope can delimit what objects are eligible to receive their messages. The downside is that service objects have to encode their interest in particular data sent via ambient messages by means of explicit conditional tests in their triggered methods.

In short, ambient references resolve the impedance mismatch by providing a form of stateless and connectionless designation, like event brokers. If stateful communication is required, anonymous far references and snapshots provide a middle ground between the space-decoupled communication provided by ambient references and the stateful communication provided by far references.

7.2.3 Bidirectional versus Unidirectional Communication

Publish/subscribe systems are good at broadcasting information from publishers to subscribers. However, if subscribers need to pass information to event publishers, this can only be accomplished by turning the subscribers themselves into publishers and by turning event publishers into subscribers explicitly to gather the replies. Ambient references avoid this pattern by means of futures. Using futures, the return value of an asynchronously invoked method can naturally serve as an implicit reply from receiver (subscriber) to sender (publisher).

Classic futures are not directly applicable to one-to-many communication, because a future can only be resolved with a single value (or ruined with a single exception)[2]. Multifutures solve this problem because they can become resolved multiple times. They remain object-oriented in the sense that a multifuture is a first-class representation of the reply, and any messages sent to it are forwarded to all values with which it becomes resolved. Via their support for **`whenEach`** and **`whenAll`** observers, they also enable the sender to expressively link event notification with reply processing and this without giving up on the anonymous style of communication promoted by publish/subscribe systems.

In short, ambient references resolve the impedance mismatch by using futures to express bidirectional communication without giving up on the full synchronisation decoupling afforded by event brokers.

7.2.4 Threads versus Event Loops

In section 3.4.4 we noted that event-driven frameworks are mostly incorporated into (multithreaded) object-oriented languages by means of callback methods. However, because the callback method is invoked synchronously by a thread which is not managed by the application, the application developer must be aware of the resulting concurrency issues. By unifying event notification with the asynchronous invocation of a receiver's method, such issues are avoided. In particular:

[2]The futures of ABCL/1 [YBS86] form a notable exception. However, while these futures can be resolved multiple times, they follow the traditional semantics that accessing an unresolved future suspends the control flow of the accessor.

- because the AmbientTalk language ensures that incoming messages are processed serially by an actor, the receiver object does not need to guard against race conditions on its data. While it is true that the serial execution of incoming messages conservatively limits the overall concurrency of the system (i.e. some methods are safe to execute in parallel), the resulting system is arguably safer and easier to understand for the programmer.

- because methods are processed asynchronously by the actor owning the receiver object, the thread of control of the event broker remains responsive. Hence, a method that takes a long time to complete does not monopolise the resources of the entire event delivery subsystem.

The major drawback of event-based systems is that they suffer from an inversion of control. In section 4.5.1 we extensively described how AmbientTalk avoids the adverse effects of inversion of control (e.g. code fragmentation) by means of futures and (block) closures. Ambient references retain the use of futures and (block) closures as observers on futures (even for multifutures).

In short, because of the event loop architecture of AmbientTalk, service objects must not take any additional synchronisation precautions when being designated by one or more ambient references. This is in contrast to multithreaded object-oriented programs where explicitly subscribing to an event broker introduces concurrency issues.

7.2.5 Reconciling Objects with Events

In section 3.4.5, we explicitly stated which properties of objects and events we wanted our unification to exhibit. Here, we discuss how ambient references achieve these properties by representing the event broker as an object reference:

- Ambient references maintain the *specific* communication interface of object-oriented message passing by representing events as messages.

- Ambient references, like event brokers, provide *connectionless* designation, catering to anonymous interactions with a volatile group of proximate objects. Anonymous far references provide *connection-oriented* designation while remaining space-decoupled.

- Ambient references make use of futures to retain the *bidirectional* communication of message passing, without sacrificing the time and synchronisation decoupling afforded by event brokers.

- Thanks to AmbientTalk/2's *event loop* concurrency, ambient references can compose gracefully with other concurrent processes: ambient messages are delivered to remote objects like any other message via the incoming message queue of actors and impose no additional synchronisation burden on the "subscribers" (the actual receivers of the ambient message).

As can be observed from the above discussion, ambient references are not solely responsible for resolving the impedance mismatch. Rather, it is the combination of AmbientTalk's event loops, (anonymous) far references *and* ambient references that together bridge the gap between event-driven and object-oriented abstractions.

7.3 Relation to Prior Work

We have already explored ambient references in previous work [VDMD05, DVM+06a, VDM+06, Ded06, VDD07]. However, the exposition of ambient references in prior publications differs from the account given in this dissertation in a number of significant ways. An early exposition of ambient references [VDM+06] considered a hierarchy of ambient references with a single, built-in message delivery policy rather than one single ambient reference abstraction supporting a wide range of message delivery policies. We briefly give an overview of what we will call the "early taxonomy" below such that we can more accurately contrast it with the description of ambient references given in this chapter – the "revised taxonomy".

In previous work, the focus was on different kinds of ambient *references* rather than on different kinds of *delivery policies*. The early taxonomy builds upon three principal design dimensions [VDM+06]:

Scope of Binding is similar to the *scope* as defined in section 6.3. It determines the set of objects to which the ambient reference may bind. The early taxonomy only considers type tags and filters to delimit the scope of an ambient reference, not protocols.

Elasticity defines how resilient an ambient reference is to *disconnections* of the objects it has previously discovered. There is a strong relationship between elasticity and the discovery lifetime of an ambient message, but the two differ in significant points, as discussed later.

Cardinality defines the *maximum number* of objects to which an ambient reference can refer. It corresponds closely with an ambient message's *arity* as defined above, but again there are some important differences.

In each of the following subsections, we discuss these design dimensions in further detail and contrast them with the design of the revised taxonomy.

7.3.1 Elasticity

Three different values for elasticity lead to the following kinds of ambient references (in the early taxonomy):

Fragile ambient references stop designating an object the moment moves out of communication range. In light of the early taxonomy, an ambient reference in the revised taxonomy is "fragile". If one sends a message to a fragile ambient reference, it is only delivered to objects that are proximate at that time. The same behaviour is achieved in the revised taxonomy using an instant discovery lifetime for the message.

Elastic ambient references keep on designating a disconnected object until a specified timeout period has elapsed. As a result, a message sent to an elasic ambient reference might still be delivered to a remote object if it reconnects within the timeout period, otherwise the message is lost. The revised taxonomy introduces this behaviour by means of the `@Transient(t)` delivery policy.

Sturdy ambient references keep on designating any object they have previously discovered, even if those objects have long moved out of communication range. If

one sends a message to such an ambient reference, the message is stored until delivered to any matching object. A similar behaviour for ambient messages can be specified using the `@Sustain` delivery policy.

Fragile ambient references are the only kind of ambient reference in the early taxonomy that truly denote a "volatile set of proximate objects". Elastic and sturdy references may designate objects that have already moved out of reach. As such, they no longer accurately reflect the abstraction of ambient references as a reference to proximate (accessible) objects only. In the revised taxonomy, an ambient reference is always fragile. Rather than being able to send a message to previously reachable objects, revised ambient references enable the programmer to send a message to objects that may become reachable in the future.

7.3.2 Cardinality

The early taxonomy distinguishes between the following ambient references for different values of cardinality:

Uni ambient references refer to *at most one* remote object at any point in time.

Multi ambient references refer to a maximum predefined number of objects at any point in time.

Omni ambient references have no upper bound on the number of objects they may refer to, and usually represent *all* remote objects within scope.

In the revised taxonomy, cardinality of references is replaced by the arity of ambient messages, where a point-to-point message most closely corresponds to the behaviour of a unireference and a one-to-many message most closely corresponds to the behaviour of an omnireference. Multireferences have no direct analogue in the revised taxonomy because experience showed that they made little sense in ad hoc networks. The point is that ambient references add space decoupling and anonymity to remote object references, which is useful for precisely those situations where it is *a priori* not known with how many participants to interact. Also, if more than the maximum allowed services would be available, which ones should the reference bind to, and which not? Only distinguishing between targeting *one* or *all* objects in range considerably simplified the taxonomy[3]. If communication with an exact number of receivers is required, snapshots can be used in the revised taxonomy to turn an intensional ambient reference into an extensional collection.

7.3.3 Reference-centric versus Message-centric View

Another fundamental difference between the early and the revised taxonomy lies in the way they enable the programmer to define variabilities in semantics. The early taxonomy is reference-centric: a programmer defines a certain *kind* of ambient reference which applies a certain semantics on *all* messages it receives. The revised taxonomy is message-centric: there is only one kind of ambient reference and each message carries its own instructions on how to be delivered to the objects abstracted from by the

[3]This design can be considered an application of MacLennan's *zero-one-infinity* principle of good programming language design, which goes as follows: "The only reasonable numbers in a programming language design are zero, one, and infinity" [Mac86].

receiver ambient reference. To exemplify the difference, reconsider the introductory example from section 6.2:

```
def clients := ambient: StockQuoteListener;
...
clients<-quoteUpdated(code, price)@[All,Transient(timeout),Oneway];
```

This example code reflects the following abstractions:

1. `clients` refers to stock quote listeners currently in reach.
2. `quoteUpdated` should be sent to all stock quote listeners in reach now, or within the next `timeout` milliseconds.

In the early taxonomy, this example would have been written as follows:

```
def clients := ambient: StockQuoteListener
               withCardinality: Omni
               withElasticity: Elastic(timeout);
...
clients<-quoteUpdated(code, price);
```

This code reflects the following abstractions:

1. `clients` refers to all stock quote listeners that are in reach now or that were in reach during the last `timeout` milliseconds.
2. `quoteUpdated` should be sent to these listeners.

The difference does not appear to be large, but there are some important consequences. In the revised taxonomy, a single ambient reference can be used with different messaging policies in the same program. In the early taxonomy, this requires the declaration of multiple ambient references because a single ambient reference is inherently associated with a delivery policy that is applied to *all* messages sent via that reference. Having to declare a separate ambient reference per delivery policy is less efficient because each ambient reference has to separately determine its reach. Using the revised ambient references, the costs associated with service discovery can be shared by different messages with their own delivery policies. Also, in the early taxonomy, ambient references *cannot* be accurately described as references representing "a volatile set of proximate objects". In that taxonomy, an ambient reference can only be understood in terms of all of its properties. In the revised taxonomy, an ambient reference can be understood without reference to any delivery policy.

There are more subtle differences between the early and revised taxonomies. For example, fragile ambient omnireferences had ad hoc support for "sustained message delivery". In the revised taxonomy, sustained message delivery is an integral part of the taxonomy and no longer an ad hoc delivery mode for one specific combination of ambient reference types. Furthermore, the revised taxonomy introduces delivery modes that were lacking in the early taxonomy, such as one-way messages and communication lifetime. The new system introduces a clear distinction between discovery and communication lifetime and provides a message sending mode (`@Expires`) which correlates the two. Bounding the resolution of multifutures in time based on the associated message's lifetime is also a novel feature of the revised taxonomy.

Finally, in the early taxonomy, an anonymous far reference is represented as a *sturdy* ambient *uni*reference [VDD07]. Such a reference binds to at most one remote object (uni) and once bound never rebinds to another object (sturdy). In this

dissertation, we no longer consider this abstraction as an ambient reference, because an ambient reference now *consistently* designates a volatile set of proximate objects, while a sturdy ambient unireference designated a single, fixed object. Instead, we provide anonymous far references as a separate abstraction. This abstraction is named an anonymous *far* reference rather than an *ambient* reference, because we use the latter term solely for referring to object references designating a *volatile set* of objects.

7.4 Limitations and Future Work

In this section, we highlight aspects of ambient references which can definitely be improved upon. We postpone a discussion on the conceptual drawbacks of using ambient references until section 10.3.

Proximity Ambient references allow a programmer to designate proximate objects intensionally, via a type tag or a protocol. Any additional constraints that cannot be conveniently encoded as a type tag or a protocol must be specified by means of a predicate, which is opaque to the ambient reference. There are many constraints that one must encode as a predicate for which it would be better to make them explicit. For example, ambient references do not provide direct support for expressing geographical proximity constraints on the designated objects. Event systems like STEAM [MCNC05] and LPS [EGH05] have already proven that making geographical location explicit enables the event notification engine to exploit this information (e.g. to limit the propagation of events by means of geographical bounds).

Security When considering security, the two major issues are the following. First, some exported objects may want to control which parties may send it ambient messages. Second, some ambient references may want to control which parties can receive their messages. Ambient references provide direct support for neither of the above: security measures must be built on top of ambient references (e.g. by parameter-passing public cryptographic keys in ambient messages to verify the authenticity of a sender or receiver's identity). In this way, ambient references are no improvement over most tuple space-based or publish/subscribe communication abstractions, which also do not offer any built-in security guarantees. An interesting topic of future work would be to study whether and how security measures could be built into the scope of an ambient reference, since this scope already acts as a kind of contract on which both senders and receivers of ambient messages must agree in order for a message to be transferrable between them.

7.5 Notes on Related Work

Ambient references can be regarded as a referencing mechanism that generalises a number of related approaches. In this regard, referencing abstractions in related work can be considered "special cases" of ambient references. In subsequent sections, we describe the work most closely related to ambient references (i.e. other referencing abstractions). Where possible, we identify how the abstraction under scrutiny can be expressed in terms of ambient references.

7.5.1 Tuple Spaces

In this section, we encode tuple space-based communication [Gel85] in terms of ambient references in order to gain insight in how communication via tuples relates to communication via ambient messages.

Assume that the objects sending and receiving ambient messages are the "processes" or "agents" from the tuple space paradigm. Assume further that ambient messages are the tuples (i.e. a message `<-m(arg)` is represented as a tuple `(m, arg)`). The tuple spaces themselves can be implicitly represented by the set of ambient messages whose discovery lifetime has not yet expired (the set denoted $\mathcal{M}$ in section 6.3). Note that this tuple space is only writeable by processes (objects) that have access to the ambient reference, and that it is only readable by processes (objects) that are within the ambient reference's reach. Hence, the reach of an ambient reference introduces a form of scoping in the tuple space model.

Sending a message to an ambient reference corresponds to adding a tuple to a tuple space. Exporting a service object corresponds to perpetually querying the tuple space for tuples matching the methods of the exported object. A receiver process (exported service object) implicitly accesses the tuple spaces defined by all ambient references that designate it. Because of the mobility of physical devices, the set of tuples that a process can read or take thus varies in time. This bears much resemblance to the engagement and disengagement of the interface tuple spaces in LIME [MPR01] (cf. section 3.3.5), even though in our encoding tuple spaces are never explicitly merged. Our encoding now being complete, we can examine the different message delivery policies in the context of tuple space communication:

- The arity of an ambient message corresponds to specific tuple space communication patterns. A point-to-point message send corresponds to an interaction where a process writes a single tuple into the tuple space and *one* of a number of processes (the potential receivers of an ambient message) *takes* the tuple from the tuple space. This corresponds to "one-for-all" communication [EFGK03]. A one-to-many ambient message send corresponds to an interaction where the sending process writes a single tuple into the tuple space and multiple processes *read* the tuple, consuming only a *copy* of the tuple and leaving the original for other processes to consume, naturally leading to "one-for-each" communication [EFGK03]. Note that using ambient references, it is the sender that decides on the arity of the communication. In tuple spaces, the situation is reversed.

- The discovery lifetime of an ambient message corresponds to a "time to live" value attached to a tuple inserted into a tuple space. When the time to live elapses, the tuple is removed from the tuple space and no longer readable by processes.

- With respect to the reply destination of an ambient message, a one-way ambient message corresponds most closely to traditional tuple space-based communication. Indeed, writing a tuple into a tuple space is a one-way asynchronous operation. A future-type ambient message must be encoded in a tuple space interaction by having the sending process perform a (non-blocking) read on a special tuple representing the reply, which is put into the tuple space by the receiving process.

- The communication lifetime of an ambient message provides an upper bound for the non-blocking read operation on a reply tuple.

The above encoding enables us to compare the expressiveness of ambient references to that of tuple spaces. On the one hand, tuple spaces are more flexible in the sense that they enable the expression of more fine-grained interaction patterns. For example, there is no delivery policy for ambient messages that directly corresponds to a tuple space communication where multiple processes post tuples and a single process consumes them (this would require multiple ambient references). Also, consuming but a single tuple from a tuple space is easy to express in tuple spaces, while using ambient references this requires the export of a service object which is immediately unexported when it first receives an ambient message.

Reversing the line of thought that tuple spaces are more flexible, ambient references can be regarded as more high-level: a single ambient message send mostly requires multiple primitive operations in the tuple space model. Moreover, an exported service object with two or more methods can directly express the reception of different kinds of messages. In the tuple space model, there is no primitive to directly express communication of the form `in(tuple1 OR tuple2)`. Futhermore, as already described in section 3.3.5, the traditional tuple space communication primitives need fundamental adaptations (i.e. they must be made synchronisation-decoupled) in order to scale in mobile ad hoc networks.

Matsuoka and Kawai have previously studied the incorporation of tuple space communication in an object-oriented language [MK88]. However, their system does not make any attempt at unifying concepts of tuple space-based communication with object-oriented language features. Rather, they describe how to develop an appropriate object-oriented interface to a tuple space. More concretely, they represent tuples and tuple spaces as first-class objects. The methods defined on the classes of these objects correspond to the primitive operations of Linda.

7.5.2 Actors and Far References

In the original actor paradigm, actors refer to one another using mail addresses [Agh86], as previously explained in section 3.3.4.1. A mail address uniquely identifies a remote actor. Moreover, the asynchronous `send` operator of the actor model allows programs to abstract over volatile connections because messages are implicitly buffered in the actor's mailbox. Actor mail addresses, like far references, do not cater to space decoupling but enable stateful designation. In section 6.6, we have discussed that ambient references do not guarantee a stateful connection, which has lead us to introduce anonymous far references. In the following, we discuss an encoding of actor mail addresses in terms of ambient references to show how one can make the designation of an ambient reference *logically* stateful.

From the point of view of our ambient reference abstraction, mail addresses can be reconstructed as ambient references whose scope is a singleton (i.e. a set consisting of a single, unique object). An actor with mail address `addr` could for instance be represented by means of the ambient reference `ambient: Actor where: { |a| a.address == addr }`. Assuming that `addr` identifies a unique actor, the reference can only ever refer to that actor. In order to achieve delivery properties similar to that of the actor model's `send` primitive, every message sent over such an ambient reference should be a point-to-point message with a sustained discovery lifetime and an unbounded communication lifetime. Being a point-to-point message, it will be delivered to the sole possible receiver and then discarded. The sustained discovery lifetime ensures the message remains buffered until the matching actor is in reach. Because the message has an unbounded communication lifetime, its associated future never times

out, allowing the acknowledgement of the message to take forever (as in the original actor model). While the above delivery policy most closely matches message delivery in the actor model, it is not entirely similar. As discussion in section 6.5, point-to-point messages are delivered at most once, while in the original actor model a message is guaranteed to be delivered eventually [Agh86], which is a stronger delivery guarantee.

Because AmbientTalk's far references are inherently based on mail addresses and the actor `send` primitive, the same strategy applies to turn an ambient reference into a far reference to an object. In order for an ambient reference to designate a unique object, the identity of that object must be made first-class, e.g. by associating each exported object with a globally unique identifier (GUID). Note that such a "connection-oriented" ambient reference still supports roaming: if *another* object is exported by means of the *same* GUID, it can be assumed to logically represent the same object and the ambient reference may designate it as well.

7.5.3 M2MI Handles

M2MI handles [KB02] have previously been discussed in section 3.3.7.4. We briefly recall their characteristic properties here. An M2MI handle is an anonymous reference to remote objects exported by means of a Java interface type. M2MI distinguishes between uni, multi and omnihandles. Unihandles refer to one specific proximate object, multihandles to a specific group of proximate objects and omnihandles to all proximate objects of the handle's interface type. Communication over a handle is unreliable: messages sent to receivers which are out of communication range are lost.

Many of the ideas behind ambient references are based on the different kinds of handles provided by M2MI. However, ambient references and M2MI handles differ in the level of abstraction they provide to the application developer. Kaminsky and Bischof explicitly note that M2MI handles form an expressive yet low-level referencing abstractions, more high-level abstractions like stronger delivery guarantees, service discovery and failure detection should be built on top [KB02]. Ambient references literally build upon M2MI handles. As we will discuss in chapter 8, ambient references have been implemented on top of M2MI handles, exploiting the fact that these references are designed specifically for ad hoc wireless proximal networks to the fullest extent.

The major strength of an M2MI handle is that it accurately reflects the connectivity of the underlying ad hoc wireless proximity network. As the authors put it themselves, a message sent to an omnihandle literally means "every object out there that implements this interface, call this method". The key point here is the term "out there", which characterises the fact that communication through handles is purely achieved through the underlying wireless communication range of the host device. Ambient references share this fundamental aspect with M2MI handles. An ambient reference is an abstraction for the set of objects that is reachable through wireless broadcast communication. Because the wireless communication range of the transceivers of mobile devices is inherently limited, it follows that those objects must be physically proximate. Being an abstraction for (a set of) proximate objects is the essence of both M2MI handles and ambient references.

To show that object designation via ambient references subsumes object designation via M2MI handles, we describe how the latter can be expressed by appropriately delimiting the scope of ambient references:

- A unihandle refers to a unique proximate remote object. To make ambient refer-

ences exhibit such a behaviour, the reference's scope must again be restricted to a singleton, requiring a technique similar to that for representing mail addresses as shown in section 7.5.2.

- A multihandle refers to a unique *group* of proximate objects. Objects can join the group by *attaching* themselves to the multihandle. In order to represent multihandles as ambient references, this group of objects must be captured as its scope. Such an ambient reference can be thought of as having a scope whose elements are described *extensionally* rather than intensionally. In other words, the characteristic function of its scope is a test whether the argument object is a member of the group.

- An omnihandle refers to *all* proximate objects of a certain type. It is most easily represented by an ambient reference whose scope is intensionally specified by means of a type tag. AmbientTalk's type tags assume the role of Java interfaces in M2MI to determine the type of the exported objects.

The message passing semantics of handles can be accurately captured by sending one-to-many, one-way messages with an instant discovery lifetime. The one-to-many semantics ensures that the message is broadcast to a group of nearby objects, which is the semantics of both multi and omnihandles. Even though broadcasting might not seem appropriate for unihandles, it does not matter whether the message is point-to-point or one-to-many because there can be only a single receiver, by construction of the reference's scope as a singleton set. The one-way semantics reflects the fact that M2MI handles only support the asynchronous invocation of methods that do not return a value or raise no declared exceptions. Finally, and most importantly, the instant discovery lifetime implies that messages are delivered only to receivers currently in range, and otherwise immediately discarded. Hence, M2MI handles do not support time decoupling.

Multihandles seem to closely correspond to multireferences as discussed in section 6.6.3. Indeed, with respect to object *designation*, they both represent a fixed group of objects. However, both abstractions differ in terms of object *communication* because multireferences support decoupling in time while multihandles do not.

7.5.4 ActorSpace

The ActorSpace extension to the actor paradigm has been previously discussed in section 3.3.4.1. To recapitulate, ActorSpaces extend the actor model by allowing messages to be sent to *patterns* as well as mail addresses. Patterns are an abstract representation for a *group* of similar actors. The original `send` primitive of the actor model can be used to send a message to a pattern. The message is then delivered to an actor matching the pattern. An additional `broadcast` primitive allows sending a message to *all* actors matching a given pattern [CA94].

Clearly, the ActorSpace model shares similar goals with the ambient reference abstraction, but in a different context. ActorSpace does not give up on the strong delivery guarantees of the actor model because no ad hoc networking model is assumed. Ambient references, on the other hand, provide a wider range of message passing options to the programmer, even though the core idea of both abstractions is the same: enable actors to send messages to other actors whose mail addresses are not known (i.e. they are both an anonymous message passing abstraction).

In order to adequately capture the semantics of ActorSpace by means of ambient references, the notion of an ActorSpace pattern must be unified with the notion of an ambient reference's scope. What exactly constitutes a pattern in ActorSpace is a little unclear in the exposition given by Callsen and Agha. In an earlier publication on ActorSpace, actors can be named with an arbitrary string and a pattern is literally a regular expression over strings [AC93]. In a later publication, actors can be described by means of *attributes* and a pattern becomes a conjunction or disjunction of such attributes [CA94]. Both semantics can be expressed by means of filters on the scope of an ambient reference and by exporting objects together with a set of properties.

Defining a representation for patterns does not suffice to capture the semantics of ActorSpace. The model further defines the resolution of a pattern to a group of mail addresses to be *local* to a specific actorspace. Moreover, actorspaces can be naturally nested. It therefore seems appropriate to unify the notion of an actorspace with the notion of a type tag. Type tags also form a scope in which to resolve discovery requests, and the scopes they define can be structured hierarchically by means of subtyping.

We illustrate how ambient references can express communication in ActorSpace by means of a small example. Assume that actors are named after the user owning them, and that we are only interested in actors that are published in the actorspace with the name `MySpace`. Referring to either the user Alice or Bob is achieved by means of the following ambient references (depending on which representation for patterns is chosen):

```
// if patterns are regular expressions over strings
ambient: MySpace where: {|a| a.pattern ~= ".* (Alice | Bob) .*" };
// if patterns are (con|dis)junctions of attributes
ambient: MySpace where: {|a| (a.name == "Alice").or: { a.name == "Bob" }};
```

As the example shows, type tags represent actorspaces while filters select only those actors from the space whose attributes match a specified pattern[4].

In order to encode the message passing semantics of the ActorSpace `send` and `broadcast` primitives, we have to specify the message delivery policies of messages sent over the above ambient references. Naturally, the difference between `send` and `broadcast` is expressed by means of the arity of an ambient message: `send` sends point-to-point messages while `broadcast` sends one-to-many messages. As for the temporal delivery aspects of the ambient messages, Callsen and Agha note that the precise delivery semantics of both `send` and `broadcast` are not fixed by the ActorSpace model. In their future research directions, they explicitly note that ActorSpace communication could be made more flexible if messages with different delivery behaviours were allowed [CA94]. Ambient references cater to exactly such different delivery behaviours by allowing messages to be annotated with different delivery policies. This enables the programmer to choose which delivery semantics is most appropriate for the communication at hand.

7.5.5 One.world

One.world [GDL+04] has already been briefly discussed as a system architecture for pervasive computing in section 3.3.7.5. It provides a framework in which applications use a common data format (tuples) and in which they communicate by sending asynchronous events to one another. Events are themselves represented as tuples. Events

[4] In AmbientTalk, the operator $\sim=$ is defined on strings and returns `true` if and only if the receiver string matches the regular expression given as an argument.

in one.world include a *source* field, which references the event handler to which information about the delivery of the event, or a potential *response* to an event (in the case of request/reply interactions) should be sent. Hence, this source field corresponds to a future associated with an ambient message. Additionally, events have a *closure* field which represents a dictionary that can be used to store the state necessary to adequately process a potential reply. This is entirely reminiscent of the use of closures as observers on futures in AmbientTalk and E. Events are delivered with at most once delivery semantics. Grimm et al. note that such a semantics is appropriate because it is lightweight (it does not require heavyweight distributed transactions) and because it allows events to be explicitly dropped to reduce the load on the system if necessary [GDL+04].

Discovery takes on a central role in the one.world infrastructure, quoting Grimm et al. [GDL+04], p. 448:

> "The primary challenge in designing the communications facilities for one.world is to provide services that are more flexible than established point-to-point communications technologies and support a rich set of communication patterns. In particular, as people and devices move through the physical world, service discovery assumes a critical role for pervasive applications. After all, if an application cannot locate necessary resources in an ever changing computing environment, it cannot function."

Hence, one.world shares with ambient references the goal of providing a framework in which different communication patterns with mobile services (resources) can be described. One.world, like ambient references, provides the programmer with different options along a number of principal design axes:

Binding Time determines *when* to perform a discovery request with respect to event notification. Two options are available (cf. the quote in section 7.1.2.2, p. 165):

- *Early binding* implies that a discovery request is resolved once and subsequently direct point-to-point communication is used to interact with the resolved resource. That is to say: once a matching service is discovered, all subsequent communication is with the same service. As already discussed in section 7.1.2.2, early binding corresponds to communication via an anonymous far reference.
- *Late binding* implies that every event notification is combined with discovery: a late bound event is always routed toward any matching service and successive events may be routed to different services. Late binding corresponds to communication via an ambient reference.

Early bound events and anonymous far references ensure that the target to which messages are sent does not change, while late bound events and ambient references provide maximum chances for a message (an event) to be delivered to any matching service (i.e. they support roaming).

Specificity determines the *number* of recipients of an event. This corresponds to the arity of an ambient message. In one.world a distinction is made between:

- *Anycast*, where the event is sent to a single matching recipient (cf. point-to-point ambient messages).

- *Multicast*, where the event is sent to all matching recipients (cf. one-to-many ambient messages).

Query Target determines the entity on which to perform a discovery query. One may register a discovery query on:

- *Resource descriptors*. A resource descriptor is the external representation of a service. This resource descriptor can be a globally unique ID or a tuple. The GUID is used for point-to-point communication, while services exported by means of a tuple can be "discovered" by sending events to a matching tuple. In AmbientTalk, GUID-based communication would be performed in terms of far references. Using a tuple as a resource descriptor corresponds to using type tags, protocols and filters to intensionally represent the scope of an ambient reference.
- *Events* themselves. Using this option, an event handler becomes a listener for a particular kind of (late bound) event. This option enables late bound events to be intercepted by other applications. It is most useful for logging or protocol translation purposes. The functionality provided by using events as query targets cannot be directly expressed using ambient references. One could export a "proxy" service whose sole purpose is to trap ambient messages, but intercepting messages in this way influences the delivery semantics. For example, if a point-to-point ambient message is sent to a logging proxy service, the message is considered "delivered" and will not be sent to the real service – unless the logging proxy broadcasts the message anew.

One.world being a system architecture and ambient references being a language construct, they evidently provide different levels of abstraction. For example, one.world requires applications to listen for events by means of a generic `EventHandler` interface supporting a single `handle(Event)` method. Ambient references unify events with object-oriented messages. The closure of an event must be managed explicitly as a dictionary in one.world, whereas in AmbientTalk, closures are an integral part of the language and close over all lexically visible state automatically. Taken together, all of these differences lead to a substantial difference in expressive power. Consider the following example code [GDL+04], p. 450:

```
SymbolicHandler destination = new DiscoveredResource(new
  Query(new Query("",
                  Query.COMPARE_HAS_SUBTYPE,
                  UserDescriptor.class),
        Query.BINARY_AND,
        new Query("user", Query.COMPARE_EQUAL, fetchUser)));
operation.handle(new RemoteEvent(this, closure, destination, msg));
```

The purpose of the above code is to send the message `msg` to a remote service representing a certain user (as indicated by the `fetchUser` string). In AmbientTalk we can express this example as follows:

```
def user := ambient: UserDescriptor where: {|u| u.user == fetchUser};
user<-selector(args)@[One,Instant,Oneway];
```

Rather than having to send a generic `msg` object, we can directly encode the message represented by `msg` as a base-level AmbientTalk message. Note that we have

chosen to model the event with an instant discovery lifetime. The precise lifetime of the one.world event depends on the settings of the `operation` object. This object will deliver the event and can perform failure detection and recovery (e.g. retry sending the event until it is successfully received). The differences between both of the above code snippets would become even more dramatic if we would include the code necessary to initialise the `closure` and `msg` variables and the code to process a reply.

The above example is a concrete illustration of how proper language design can realistically augment the programmer's expressive power. In absolute terms, one.world can express the same communication as ambient references. However, in relative terms, the level of abstraction provided by ambient references is significantly higher. Arguably, the better the code matches the intended communication, the more understandable it becomes. This is not a critique on one.world in particular, but rather a reinforcement of our assumption that domain specific languages are an appropriate software engineering tool, as stated in the introductory chapter.

7.5.6 Distributed Asynchronous Collections

Distributed asynchronous collections (DACs) are object-oriented abstractions for expressing different publish/subscribe interaction styles and qualities of service [EGS00]. They relate to ambient references because DACs also provide a framework for expressing different message delivery policies.

A DAC extends the notion of a traditional object-oriented collection to a distributed context. In a nutshell, objects can add elements to the collection (which is equivalent to publishing an event) and can register a callback to be invoked asynchronously whenever elements are added to the collection (which is equivalent to subscribing to events). A DAC is shared between multiple distributed objects. Importantly, a DAC is not a centralised collection accessible by distributed objects but rather an inherently distributed collection. While traditional object-oriented collections differ in terms of whether or not they accept duplicates (e.g. bags versus sets) or the order in which they store their elements (e.g. stacks versus queues), DACs differ in terms of delivery guarantees (e.g. at-least-once versus at-most-once delivery) and delivery order (e.g. FIFO versus Total order). A DAC is constructed by means of a *topic* (as used in topic-based publish/subscribe systems [EFGK03]), which is functionally equivalent to the use of type tags by ambient references.

Ambient references share with DACs the goal to make publish/subscribe-based interactions easier to accomplish in an object-oriented language. Both approaches do have their differences:

- DACs – unlike ambient references – have not been designed specifically for MANETs. For example, DACs do not support time decoupling: messages are considered volatile and are dropped once delivered to all connected subscribers.
- Ambient references go further in integrating publish/subscribe with object-orientation than DACs. For example, ambient references unify events with messages and event delivery with method invocation. No such unification is provided by DACs. Also, DACs provide no support for dealing with replies to events. Adding an element to the collection is a unidirectional operation.
- The delivery policies provided by DACs are different than those provided by ambient references. For example, DACs support different delivery guarantees and delivery orders but do not support the analogue of arity or discovery lifetime.

The differences between ambient references and DACs notwithstanding, they share the common goal of providing a distributed programming abstraction which is customisable to the problem at hand, by enabling the programmer to specify different delivery policies. Whereas DACs unify publish/subscribe communication with object-orientation by representing the event broker as a *collection*, ambient references unify publish/subscribe communication with object-orientation by representing the event broker as an *object reference*. Because object references lie at the heart of an object-oriented language, ambient references can additionally unify events with messages and method invocations.

7.5.7 Joule Channels

Multireferences and multifutures are akin to the *channels* abstraction in the Joule language [TMHK95]. Like multireferences and multifutures, Joule channels may convey messages to more than one object and enable the sender to expressively gather all replies to the message. Unlike multireferences, but like multifutures, the set of objects to which the channel conveys messages may grow over time.

A Joule channel has two separate interfaces: an *acceptor* and a *distributor*. The acceptor forwards any message sent to it to all objects designated by the channel. The channel can be made to designate additional objects via the distributor. This is reminiscent of the distinction between futures and resolver objects in AmbientTalk (cf. section 4.3.4.3) and the promise-resolver pairs in E [Mil06] to distinguish manipulating the object designated by the future from manipulating the future itself.

7.6 Conclusion

In this chapter, we have put ambient references in perspective. We discussed how AmbientTalk and ambient references adhere to the criteria postulated in section 3.2, thus arguing that in combination they form a powerful platform to express coordination between objects in a MANET. We also discussed how ambient references import the beneficial properties of event-driven publish/subscribe systems into an object-oriented language, without sacrificing the object-oriented message passing metaphor. They resolve the object-event impedance mismatch by representing the event broker as an object reference. Finally, we discussed how ambient references as described here resemble and differ from related and prior work. In the following chapter, we delve back into more technical details when discussing the implementation of ambient references in AmbientTalk.

Chapter 8

Implementing Ambient References

With both the AmbientTalk language and ambient references fully explained we can now turn our attention to the implementation aspects of the language abstraction. Two implementations for ambient references have been devised, each on the basis of a different design. We discuss both designs in detail and compare their relative advantages and drawbacks.

It should be clear from their exposition in previous chapters that ambient references, anonymous far references and multireferences are special kinds of object *referencing* abstractions. Hence, implementing them requires support for first-class object *references* which have the ability to intercept and reify any messages sent to them. We will therefore apply the technique used to represent futures as object references (cf. section 5.2.3.1) once more in this chapter.

At a lower level of abstraction, ambient references should somehow communicate with remote objects by means of an appropriate network protocol. The choice of the underlying network protocol is important given the peculiarities of mobile ad hoc networks. Concretely, we will make use of the M2MI library already discussed in sections 3.3.7.4 and 7.5.3. Given that M2MI is a library developed for Java and not for AmbientTalk, we will make use of AmbientTalk's support for linguistic symbiosis (cf. section 5.3) such that AmbientTalk objects can safely use the M2MI API.

We conclude this chapter with a discussion on the implementation of anonymous far references and multireferences which, as we shall see, do not require the more elaborate implementation techniques required for ambient references.

Note that the implementations discussed in this chapter are "proof of concept" implementations: they illustrate that it is feasible to implement the concept of an ambient reference, as described in chapter 6, using contemporary technology. In particular, we are well aware that the implementation as described here is by far not optimal or scalable up to thousands of network nodes. The implementation rather serves as a proof by construction that ambient references are a feasible language construct. For the sake of reproducibility, the complete source code of the implementation is listed in appendix A.

8.1 Implementation Strategies

We describe two different strategies to implement ambient references, based on two different representations for the *reach* of an ambient reference. Recall from section 6.3 that the reach of an ambient reference represents that subset of the ambient reference's scope which is in communication range at a particular point in time. An ambient reference's reach can be specified:

Extensionally In this design, the reach of an ambient reference is represented as an explicit collection of far references. In other words, the ambient reference's reach is represented as an extensionally specified set of elements. In order to represent the elements of this set, the ambient reference must use a service discovery engine to keep track of new services joining the network (which are then added to the set) and of discovered services leaving the network (which are then deleted from the set). Section 8.3 elaborates on this implementation.

Intensionally In this design, the reach of an ambient reference is represented implicitly by means of a broadcast channel. Everyone listening on the channel is implicitly part of the ambient reference's reach. The implementation is broadcast-based because communicating with objects in reach entails sending a message on the broadcast channel. Section 8.4 provides an in-depth discussion on this implementation.

The choice of representation for the ambient reference's reach has a large impact on how the different delivery policies for ambient messages can be achieved. The details can be found in the respective implementation sections. Section 8.5 provides a comparative overview after the details of both implementations have been discussed.

8.2 Implementation Outline

Even though there exist two different implementations of ambient references, depending on how their reach is represented, both implementations are built around the same design. We sketch this generic design here, before delving into technical details in the subsequent sections. Figure 8.1 provides an illustrative overview of the entire process of sending a message via an ambient reference to potential receivers in the ambient reference's reach.

We discuss each of the steps indicated in the figure below:

1. A message is sent to an ambient reference. The ambient reference is a mirage representing a proxy object that can intercept and reify the message (cf. the representation of custom object references as discussed in section 5.2.3).

2. Based on the annotations of the message (cf. the overview in section 6.4.7), the message is associated with a dedicated *message handler* object that encodes the delivery policy for that message. This handler object is implemented by means of trait composition: for each of the three delivery policies (arity, discovery lifetime and communication lifetime), the handler uses a separate trait.

3. The ambient reference dispatches the message to its associated handler, passing itself as an argument. The precise delivery semantics of the message is dependent

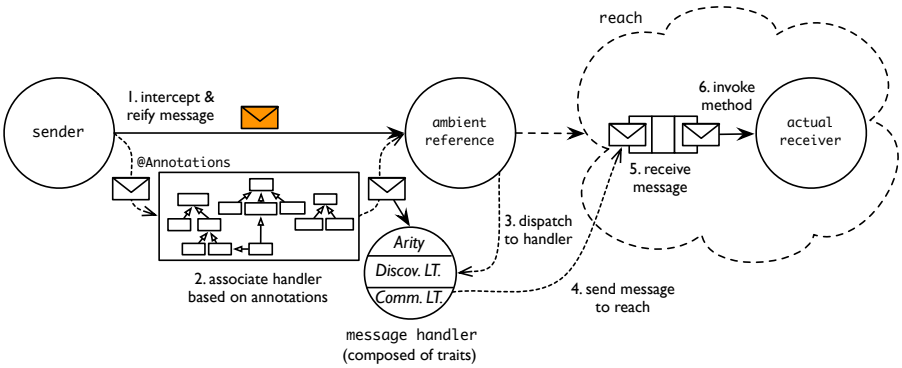

Figure 8.1: Overview of the implementation of ambient references.

on the interplay between the different delivery policies composed together in the handler.

4. The handler can eventually send the message to potential receivers by means of the ambient reference's reach. The details of how the reach is used depend on whether it is specified either intensionally or extensionally.

5. The message is subsequently buffered in the message queue of actors owning actual receivers to which the message could be delivered successfully.

6. When the actors processes the message, the corresponding method is triggered in the actual receiver.

Note that the above sketch still abstracts from a lot of technical details. In particular, we did not specify how replies are dealt with and how delivery handles fit into this picture. Both will be explained later on in this chapter. The three most important aspects of the implementation are the following:

- In the spirit of the work on modular interpreters [Ste94a, LHJ95, Esp95], where monads are used to construct a language interpreter using modular building blocks, we have used trait-based composition to factor out the behaviour specific to each of the three delivery policies of ambient references (arity, communication and discovery lifetime) into separate objects. These objects act as basic building blocks with which a handler for an ambient message can be constructed. Ambient messages receive a handler whose behaviour is composed of a trait for each kind of delivery policy. While the delivery policies are not entirely orthogonal (there is a definite interplay between them, cf. section 6.4.8), the advantage of using trait-based composition is that each kind of delivery policy is specified in a *modular* way, making the interplay between the different policies *explicit* by means of the composition interface defined by the different traits.

- An ambient reference is essentially implemented as a *proxy* to (a set of) remote objects. It intercepts each message sent to it and subsequently delivers the message to its actual receivers according to a specified delivery policy. As described previously in section 5.2.3, a proxy representing a custom object reference is

implemented by means of a mirage. Just like we discussed the reflective implementation of futures in section 5.2.3.1, this chapter discusses the reflective implementation of ambient references as custom object references.

- An ambient reference designates objects in its reach. As stated in the previous section, the implementation of this designation has a large effect on the resulting properties of the ambient reference. The following two sections discuss the repercussions of the choice of designation on the trait-based implementation of the handlers.

8.3 Extensional Reach

We now describe an implementation of ambient references where the reach of the reference is represented by means of an explicit collection of references to remote objects. Because the objects which the ambient reference designates are made explicit using this representation, the implementation is very receiver-centric: the different message handlers describe their delivery policies in terms of the arrival and departure of potential receivers because these events can be explicitly captured by means of a service discovery engine.

8.3.1 Representing Reach

The reach of an ambient reference can be represented by means of a collection, e.g. a vector. Service objects are added to the collection each time a service discovery engine detects the presence of a new matching service. Service objects are removed from the collection each time the service discovery engine detects that a previously discovered service is no longer responsive. The following code encodes this behaviour[1]:

```
def ambient: typetagOrProtocol where: filter {
  ...
  def reach := Vector.new();
  whenever: typetagOrProtocol discovered: { |descriptor|
    if: filter(descriptor.attributes) then: {
      // add the discovered service to the AR's reach
      reach.add(descriptor.service);
      // notify discovery observers (explained later)
      self.notifyObservers(descriptor.service);
    };
  };
  whenever: typetagOrProtocol lost: { |descriptor|
    if: reach.contains(descriptor.service) then: {
      // remove the lost service from the AR's reach
      reach.remove(descriptor.service);
      ...
    };
  };
  ...
}
```

[1]Most of the code depicted in this chapter is part of a module implementing ambient references. To use ambient references, application programmers can simply import this module and as such access its public definitions.

In the above code, `reach` is defined to be an instance variable of the ambient reference. Upon creation, the ambient reference makes use of two functions provided by a service discovery module to keep track of appearing and disappearing services. The **`whenever:discovered:`** and **`whenever:lost:`** functions shown here are not built-in AmbientTalk functions. Rather, they are the public interface to a discovery engine written in AmbientTalk itself (cf. section 8.7.2). One difference with the built-in **`whenever:discovered:`** function is that the above function accepts type tags as well as protocols as its first argument, while the built-in function accepts only type tags (cf. section 4.4.3).

The ability of ambient references to restrict their scope based on arbitrary AmbientTalk predicates is not directly supported by the service discovery module. That is why, in the above code, an additional if-test is performed to check whether the discovered service object actually belongs to the scope of the ambient reference. This is done by applying the `filter` predicate, which is passed as an initial argument to the ambient reference, to the attributes with which the service object was published. We discuss service descriptors and their attributes in more detail in section 8.3.7.

Note that the discovery of a new service in turn triggers observers registered on the ambient reference itself. These observers are registered by message handlers to be able to react upon the appearance of a new potential receiver. Their use will become clear throughout the subsequent sections.

8.3.2 Delivery Policies as Traits

Every ambient message receives a message handler which encodes its overall delivery policy. As discussed in section 6.3, an ambient reference supports three kinds of policies: arity, communication lifetime and discovery lifetime. In the implementation, we represent each policy as a trait. A handler is thus represented as:

```
def handler := object: {
  import TArity;
  import TCommunicationLifetime;
  import TDiscoveryLifetime;
};
```

The `TDiscoveryLifetime` trait defines a `dispatch` method which, given an ambient reference and an ambient message as arguments performs the actual message delivery. This `dispatch` method can be regarded as a multimethod: its behaviour actually depends on the "type" of message it receives. In a language with direct support for multiple dispatch one would be able to modularly implement an ambient message's dispatch without having to use traits. AmbientTalk lacks direct support for multimethods. However, rather than implementing the multiple dispatch by means of inflexible if-tests, we represent the multiple dispatch by a series of single dispatches, implemented as late bound self-sends to auxiliary methods. Each trait implements its own version of the appropriate auxiliary methods. Figure 8.2 provides an overview of the provided and required interface of the traits used by the handler. The traits' methods are explained throughout the following sections.

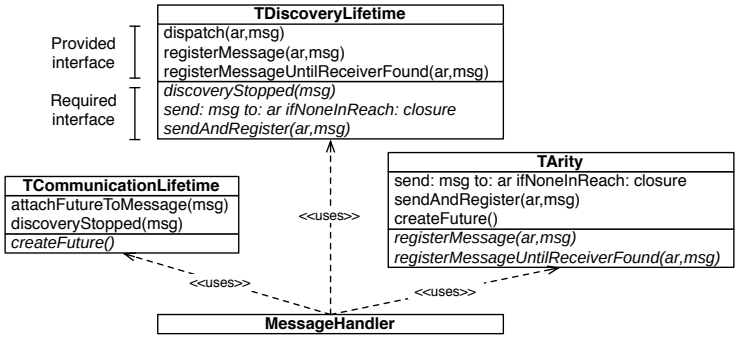

Figure 8.2: Delivery policy traits used by a message handler (extensional impl.).

8.3.3 Representing Discovery Lifetime

The discovery lifetime trait manages the prolonged registration of an ambient message with its carrier (i.e. the ambient reference to which it is sent) in order to get notified of new potential receivers becoming available. Moreover, it provides the `dispatch` method which is the entry point of the message delivery process. The `TDiscoveryTrait` introduced in the previous section is an abstract entity. There exist three concrete implementations of the trait, corresponding to the three values for discovery lifetime. We discuss each concrete implementation in turn.

Instant First, consider the code for ambient messages with an instant discovery lifetime. Note that whenever we discuss a trait implementation like the one below, we explicitly indicate further dispatches to other traits.

```
def TInstant := object: {
  def dispatch(ar, msg) {
    self.send: msg to: ar ifNoneInReach: { }; // dispatch TArity
    self.discoveryStopped(msg); // dispatch TCommLT
    object: { def cancel() { false } };
  }
}
```

When an ambient message with an instant discovery lifetime is received by an ambient reference, it is sent to one or all potential receivers currently in reach (depending on the arity of the message, cf. the following section). Because the message has an instant discovery lifetime, it is not registered with its carrier for prolonged delivery. By invoking `discoveryStopped`, the discovery lifetime trait signals to the communication lifetime trait that no more receivers will be discovered. Note that, if the reach is empty at the time an `@Instant` message is sent, it is not received by any object.

The return value of the `dispatch` method is a registration object that is used by the delivery handle to cancel the delivery of the message. As will be shown later, for prolonged (transient and sustained) messages, this object is used to unregister an ambient message with its carrier. In the case of an instant discovery lifetime, the message is never registered with the ambient reference, so it does not need to be unregistered upon cancellation. A dummy object is returned simply to achieve consistency across the different delivery policies.

Sustain Next, we discuss the implementation of messages with a sustained discovery lifetime.

```
def TSustain := object: {
  def dispatch(ar, msg) { self.sendAndRegister(ar, msg) };//dispatch TArity
  def registerMessageUntilReceiverFound(ar, msg) {
    def registration := ar.addDiscoveryObserver: { |rcvr|
      rcvr <+ msg;
      registration.cancel();
    };
    def stopDiscovery() { self.discoveryStopped(msg) };// dispatch TCommLT
    object: {
      def cancel() {
        registration.cancel();
        stopDiscovery();
      }
    };
  };
  def registerMessage(ar, msg) {
    def registration := ar.addDiscoveryObserver: { |rcvr|
      rcvr <+ msg;
    };
    def stopDiscovery() { self.discoveryStopped(msg) };// dispatch TCommLT
    object: {
      def cancel() {
        registration.cancel();
        stopDiscovery();
      }
    };
  };
}
```

When a message with a sustained discovery lifetime is received by an ambient reference, the message is first sent to one or all service objects in reach (depending on the message's arity). Since the message has a sustained lifetime, it should be registered with its carrier, such that it is notified each time a new potential receiver comes in reach. This is done by means of the `addDiscoveryObserver` method defined on the ambient reference. Note that there are two ways in which a message can be registered: it can either be registered until a receiver is found, or until explicitly cancelled. Which of the two methods is invoked depends on the arity of the ambient message.

Every time a new potential receiver is discovered, the message associated with the handler is forwarded to the receiver. As explained in section 5.1.1, the expression `rcvr <+ msg` sends the first-class message object `msg` to the receiver `rcvr` as if the message send were performed directly in the source text. One important property implicit in the above code is whether the `addDiscovery-Observer` can trigger twice for the same receiver. The service discovery engine itself detects duplicate advertisements and hence only fires the observer once per unique receiver. However, if the receiver becomes disconnected for longer than a certain amount of time (which we call the discovery engine's *recall period*), the engine forgets about the remote object and notifies the ambient reference of the object's disappearance (cf. the **`whenever:lost:`** observer in section 8.3.1). Should that object later reconnect, it will not be "recalled" by the discovery engine. This causes the ambient reference to be notified of a new potential receiver, and as a result the `addDiscoveryObserver` may be invoked multiple times on

the same receiver. This implementation corresponds to our explanation in section 6.5 that one-to-many message delivery may violate *at most once* delivery semantics.

Transient The final option is for the ambient message to have a transient lifetime. We can conveniently express a transient lifetime as a sustained lifetime which is automatically cancelled once the transient message's discovery lifetime period has expired:

```
def makeTTransient(initTransientPeriod) {
  def makeAutoExpirable(originalRegistration) {
    def timerRegistration := when: initTransientPeriod elapsed: {
      originalRegistration.cancel();
    };
    object: {
      def cancel() { // signals a premature cancellation
        timerRegistration.cancel(); // stop the timer
        originalRegistration.cancel(); // cancel the registration now
      }
    }
  };
  extend: TSustain with: {
    def transientPeriod(msg) { initTransientPeriod };
    // override the methods defined by the sustainable lifetime trait
    def registerMessageUntilReceiverFound(ar, msg) {
      def reg := super^registerMessageUntilReceiverFound(ar, msg);
      makeAutoExpirable(reg);
    };
    def registerMessage(ar, msg) {
      def reg := super^registerMessage(ar, msg);
      makeAutoExpirable(reg);
    };
  }
};
```

Note that the above trait is parameterised with the timeout period specified in the corresponding `@Transient(t)` annotation, which is why the trait is represented as a function rather than as a singleton. The trait uses delegation to acquire the methods defined for a sustained discovery lifetime. However, it overrides the two registration methods in order to make the returned publication "auto-expirable". It does this by starting a timeout (by means of the **`when:elapsed:`** library function) which simply cancels the original registration when the timeout period has elapsed.

8.3.4 Representing Arity

The trait representing the arity delivery policy of an ambient reference provides three methods:

- The `createFuture` method returns a particular kind of future to be attached to the message.
- The `send:to:ifNoneInReach:` method sends the message to one or all objects in reach of the given ambient reference. If the reach is empty at the time this method is invoked, its third argument is applied.

- The `sendAndRegister` method sends a message once to one or all objects in reach of the given ambient reference. Then, depending on the arity, the message is registered with the ambient reference until a receiver is found or until its lifetime expires (by invoking one of the two registration methods discussed in the previous section).

One Below is the implementation of the arity trait for point-to-point messages:

```
def TOne := object: {
  def createFuture() { makeFuture() };
  def send: msg to: ar ifNoneInReach: closure {
    if: (ar.reach.isEmpty) then: closure else: {
      def receiver := (ar.reach)[1 ?? (ar.reach.length + 1)];
      receiver <+ msg;
    };
  };
  def sendAndRegister(ar, msg) {
    def registration := object: { def cancel() { false } };
    self.send: msg to: ar ifNoneInReach: {
      // dispatch TDiscoveryLT
      registration := self.registerMessageUntilReceiverFound(ar, msg);
    };
    registration
  };
}
```

The `createFuture` method simply returns a regular future, as can be expected for a point-to-point message. The `send:to:ifNoneInReach:` method either invokes its argument closure if the collection representing the reach is empty, or it delegates to a communication lifetime trait to send the message to a non-deterministically chosen receiver by means of the `??` operator[2]. The `sendAndRegister` method first sends the message to one potential receiver currently in reach. If such a receiver exists, this stops the delivery process (a point-to-point message can have at most one receiver). If no receiver is in reach, however, it asks the discovery lifetime trait to register the message with the ambient reference until a potential receiver is found (or until the discovery lifetime of the message expires, whichever comes first).

All Below is the implementation of the arity trait for one-to-many messages.

```
def TAll := object: {
  def createFuture() { makeMultiFuture() };
  def send: msg to: ar ifNoneInReach: closure {
    if: (ar.reach.isEmpty) then: closure else: {
      ar.reach.each: { |receiver|
        receiver <+ msg;
      }
    };
  };
  def sendAndRegister(ar, msg) {
    self.send: msg to: ar ifNoneInReach: { };
```

[2]The expression `a ?? b` is equivalent to the expression `a.??(b)`. The method `??` is defined on AmbientTalk integers and returns a random integer drawn from a uniform distribution over $[a, b[$.

```
    self.registerMessage(ar, msg); // dispatch TDiscoveryLT
  };
}
```

The `createFuture` method now returns a multifuture rather than a regular future because a one-to-many message can generate multiple replies. In the `send:to :ifNoneInReach:` method, rather than selecting one receiver from the current reach, the message is "broadcast" to all receivers in the current reach by sending the message to all elements in the collection. The `sendAndRegister` method first sends the message to all current members of the ambient reference's reach. Then, regardless whether the reach contained potential receivers or not (a one-to-many message can have any number of receivers), it asks the discovery lifetime trait to register the message with the ambient reference until its discovery lifetime expires.

8.3.5 Representing Communication Lifetime

The communication lifetime trait defines two methods. `attachFutureToMessage` takes a message as its argument and returns a tuple `[future,message]`. Here, `future` denotes the future that is attached to the message (for two-way messages) or **`nil`** (for one-way messages) and `message` denotes a novel message to which the `future` has been attached (for two-way messages) or the original message (for one-way messages). This method is invoked when the message is originally intercepted by an ambient reference receiver (cf. section 8.6).

The second method, `discoveryStopped`, is used by the discovery lifetime traits and signals to the communication lifetime trait that no more potential receivers for the message will be found (i.e. that the last message has been sent). This enables any future attached to the ambient message to become totally resolved.

Oneway We start with describing one-way ambient messages:

```
def TOneway := object: {
  def attachFutureToMessage(msg) { [nil,msg] };
  def discoveryStopped(msg) { };
}
```

A one-way ambient message does not attach a future to a message. Neither does it need to take any special actions when the last message has been sent.

Reply A two-way message with an unbounded communication lifetime uses the following trait:

```
def TReply := object: {
  def attachFutureToMessage(msg) {
    def [fut,res] := self.createFuture(); // dispatch TArity
    [fut, futurize(msg, fut)];
  };
  def discoveryStopped(msg) { };
}
```

A two-way message requires a future. What kind of future (regular or multifuture) depends on the arity. The self-send to `createFuture` is a dispatch to one of the arity traits defined previously. When a future for the message is constructed, it is attached

to the message by a call to the auxiliary function `futurize`. This function adapts the behaviour of the message such that after being processed, the message resolves the associated future with the result of the invoked method. This is made possible because AmbientTalk's metaobject protocol provides the necessary hooks to trap asynchronous message processing.

Because the communication lifetime is unbounded, no actions need to be taken when the last message is sent. The future attached to the message will not time out automatically.

Due Finally, we consider two-way messages with a bounded communication lifetime:

```
def makeTDue(initDuePeriod) {
  extend: TReply with: {
    def timeLeftForReply(msg) { initDuePeriod };
    def discoveryStopped(msg) {
      when: self.timeLeftForReply(msg) elapsed: {
        (msg.handle.future)<-becomeResolved()@MetaMessage;
      }
    };
  }
};
```

The above trait is parameterised with a timeout period corresponding to the value in the associated `@Due(t)` annotation. The trait is further defined as an extension of an unbounded communication lifetime trait. It overrides `discoveryStopped` to start a timeout which, upon triggering, notifies the future associated with the message to become totally resolved. In the case of a regular future, this ruins the future with a `TimeoutException`. In the case of a multifuture, this message disallows any further results to be processed and hence enables the multifuture's **`whenAll`** observers to be triggered. Note the `@MetaMessage` annotation to the `becomeResolved` message. As discussed in section 5.2.3 this annotation signifies that the message is destined *for the future itself* and not for the object *represented by* the future.

8.3.6 Representing Expirable Messages

The final type of messages that can be processed are expirable messages, tagged with the `@Expires` annotation. Recall from the diagram in section 6.4.7 that an expirable message fixes both the message's communication and discovery lifetimes. This is accurately reflected in its implementation below, which uses trait composition to implement this "multiple inheritance" relationship:

```
def makeTExpires(expPeriod) {
  object: {
    import makeTTransient(expPeriod);
    import makeTDue(expPeriod) exclude timeLeftForReply;
    def timeLeftForReply(msg) {
      self.transientPeriod(msg) - (now() - msg.handle.sendTime)
    };
  }
};
```

An expirable message is constructed with an `expPeriod` argument, signifying the expiration period specified in the annotation. An expirable message inherits all of the behaviour of a message with a transient discovery lifetime where the maximum discovery lifetime equals its expiration period. It also inherits the behaviour of a message with a bounded communication lifetime, save the `timeLeftForReply` method. The expirable message provides its own definition for the time that is left for a reply to be received, as it is no longer a constant. Note that the variable `sendTime`, which is stored in the message's delivery handle, denotes the time at which the message was sent to the ambient reference (cf. section 8.6).

As we have discussed at length in section 6.4.6, the communication lifetime of an expirable message grows shorter the longer it takes to discover a potential receiver. If, for example, a potential receiver exists at approximately the time the message was sent, then `now() - msg.handle.sendTime` will be close to 0 and the message's communication lifetime roughly equals the maximum value, which is its discovery lifetime. The longer it takes to find a receiver, the later the `timeLeftForReply` method is invoked and hence the larger the value of `now()`, causing the value of `timeLeftForReply` to shorten.

Note that, because an expirable message fixes both the communication and discovery lifetimes, some care is required when composing the actual traits into a composite handler (cf. the composition code in section 8.3.2). For the purposes of composing the message handler, an expirable message has an empty discovery lifetime trait and uses the above trait to represent its communication lifetime. If the above trait were used both as the composite's communication *and* its discovery lifetime, the implementation would be composed twice, which would immediately lead to name clashes.

8.3.7 Representing Exported Objects

We now turn our attention to the receivers of ambient messages. Recall from section 6.4.2 that objects are eligible for receiving ambient messages only after they have been explicitly exported, potentially specifying a number of attributes. Below is the implementation of the **`export:as:with:`** function:

```
def export: service as: typetagOrProtocol with: closure {
  def attributes := isolate: closure;
  def descriptor := ServiceDescriptor.new(service, attributes);
  export: descriptor as: typetagOrProtocol;
};
```

Again, the invoked **`export:as:`** method is part of the interface of a custom discovery engine written in AmbientTalk itself, which enables objects to be exported as either a type tag or a protocol (cf. section 8.7.2). The closure describing the attributes of the service is used to construct an isolate object, which is exported together with a reference to the service object. Because the attributes are modelled as an isolate, they are passed by copy to remote discovery engines. This enables the `filter` predicate of remote ambient references to locally query those attributes, without requiring further communication.

8.3.8 Snapshots

If the reach of an ambient reference is represented by means of an explicit collection, creating snapshots of the ambient reference's reach becomes extremely easy. The im-

plementation is as follows:

```
def makeSnapshot() { reach.asTable() };
```

The method turns the collection object representing the ambient reference's reach into an AmbientTalk table of far references. By not simply returning the `reach` collection directly, we ensure that the caller of the `makeSnapshot` method cannot modify the original collection, which should be maintained solely by the ambient reference.

8.3.9 Summary

Figure 8.3 gives a complete overview of the delivery process of an ambient message by means of the extensional implementation described above. Note that the messages exchanged between the `TArity` and `TDiscoveryLifetime` traits depend on the kind of message. For example, if the message `msg` is tagged with the `Instant` annotation, the message `send:to:ifNoneInReach:` is sent to the `TArity` trait. The margin on the left indicates the time intervals corresponding to communication and discovery lifetime.

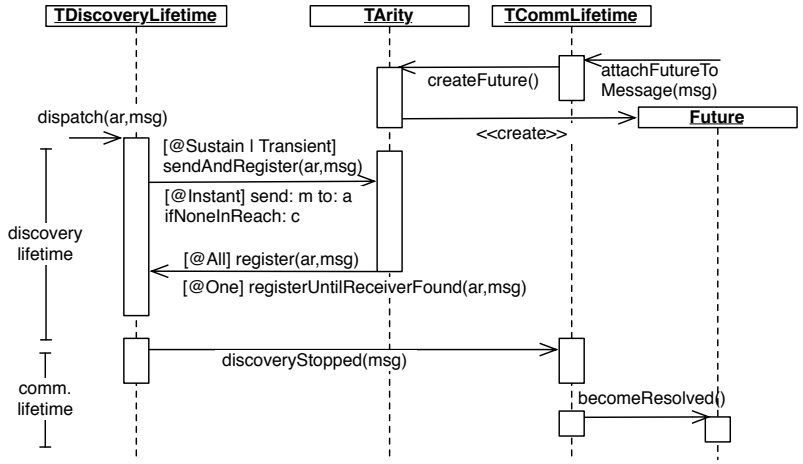

Figure 8.3: Multiple dispatch implementing ambient message delivery.

To summarise, using the extensional implementation of an ambient reference's reach, the ambient reference can refer to all service objects in reach at any point in time. Instant message delivery is expressed simply by sending a message to one or all elements of the `reach` collection (depending on the arity). Transient and sustained delivery are achieved by registering an observer with the ambient reference, which is triggered each time a new potential receiver is discovered. This allows the ambient message to be delivered to receivers that come in reach after the message was sent to the ambient reference.

We have now discussed the complete details of delivering ambient messages sent via ambient references whose reach is represented by means of an explicit collection object. The implementation of ambient references as a whole still remains incomplete. For example, it is not yet explained when and by whom the `dispatch` and `attachFutureToMessage` methods are invoked. This will be explained later, in section 8.6. In the following section, we first discuss an alternative implementation of ambient references whose reach is only represented implicitly.

8.4 Intensional Reach

In this section, we discuss an implementation of ambient references where the reach of the reference is represented intensionally. As a result, it is closer to the intuitive definition of ambient references whose scope is also specified intensionally. Because the implementation no longer directly refers to objects in reach, the implementation is no longer "receiver-centric" but rather "message-centric": ambient references simply broadcast their messages to anyone that is subscribed to a broadcast channel, and it is up to the receivers to e.g. filter out duplicates based on previous received messages. The events of subscribing to or unsubscribing from this broadcast channel are not made explicit anywhere in the implementation. Hence, no service discovery engine is used to keep track of "nearby" service objects.

8.4.1 Representing Reach

In the intensional implementation of ambient references, the reach is represented as a "broadcast channel". We implement such a channel by means of M2MI omnihandles. Recall from section 3.3.7.4 that sending a message to an M2MI omnihandle broadcasts that message to all nearby exported Java objects of a certain interface type. M2MI handles are Java objects. We defer an explanation of how such objects can be accessed by AmbientTalk objects until section 8.7.3. For now, it suffices to know that we wrap an M2MI omnihandle in a small abstraction which we name an *omnireference* which allows AmbientTalk code to treat the Java M2MI handle as if it were an AmbientTalk eventual reference. Below is the definition of the reach of an ambient reference:

```
def ambient: typetagOrProtocol where: filter {
  ...
  def reach := omnireference: typetagOrProtocol;
  def performAnycast(replyHandler) {
    reach<-anycast(filter, replyhandler);
  };
  def performBroadcast(msg, id, ttl) {
    reach<-broadcast(filter, msg, id, ttl);
  };
  ...
}
```

The omnireference wrapper takes care of converting the type tag or the protocol into a Java interface type with which the Java M2MI handle can be initialised. The ambient reference further defines two auxiliary methods which are used by the handlers defined below to communicate with objects in reach. `performAnycast` broadcasts the message `anycast` to any object that happens to be proximate enough to receive the broadcast. Any object that receives the broadcast should reply to the `replyhandler` object. This method is part of a simple anycast protocol that enables an ambient reference to get a view on which objects are in reach. The `performBroadcast` method broadcasts the `broadcast` message to nearby receivers (cf. section 8.4.3). Any object receiving this message can then deliver `msg` to exported application objects. The additional `id` and `ttl` arguments specify a unique message identifier and a time to live value respectively. Their role will be explained in due course.

Note that in both of the auxiliary methods, the `filter` predicate is passed as an argument to the broadcasted message. Because M2MI only enables one to delimit the

scope of a broadcast at the level of types (i.e. Java interfaces) and not at the level of dynamic attributes, some objects may receive the broadcasted messages even though they do not satisfy the `filter`. Receivers of `anycast` or `broadcast` messages should therefore first check whether they actually belong to the scope of an ambient reference by applying the passed `filter` predicate to the attributes of locally exported objects. Only if the objects satisfy the `filter` will the objects reply (in the case of `anycast`) or deliver `msg` (in the case of `broadcast`). For this strategy to work, however, `filter` must be passed by copy, requiring the predicate to be implemented as an isolate[3].

8.4.2 Delivery Policies as Traits

The intensional implementation of ambient message delivery is also based on a multiple dispatch over the different delivery policies. However, in this implementation, the initial dispatch is on arity, rather than on discovery lifetime as in the extensional implementation. Figure 8.4 shows the modified trait hierarchy. Again, we discuss the individual methods throughout the following sections.

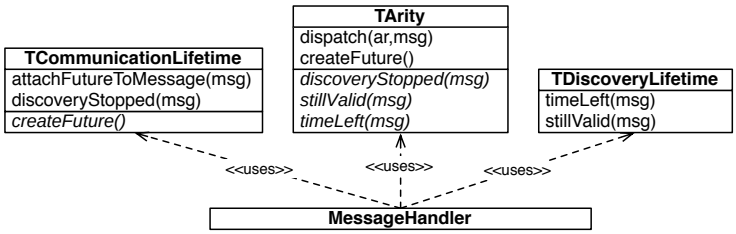

Figure 8.4: Delivery policy traits used by a message handler (intensional impl.).

8.4.3 Representing Arity

As in the extensional implementation, each of the three kinds of traits discussed above has a number of implementations depending on the different kinds of delivery policies. We first discuss arity because the initial dispatch is now on this trait.

All Representing the ambient reference's reach as a broadcast channel naturally has the advantage that one-to-many messages become very easy to implement. Below is the implementation of the arity trait for one-to-many messages:

```
def TAll := object: {
  def createFuture() { makeMultiFuture() };
  def dispatch(ar, msg) {
    def continuation;
    def id := generateMessageId(msg);
    def sendOnce() {
      def ttl := self.timeLeft(msg); // dispatch TDiscoveryLT
```

[3]In AmbientTalk, closures are by default pass-by-reference, but there exist library functions to express pass-by-copy closures as well (cf. section 9.1.1). Since closures are objects which define an `apply` method, pass-by-copy closures are nothing but isolate objects which define an `apply` method. However, because isolates do not have access to their enclosing lexical scope, isolate closures must explicitly import any lexical variables on which they depend as instance variables of the closure object.

```
      ar.performBroadcast(msg, id, ttl);
      continuation := when: BROADCAST_RATE elapsed: {
        if: self.stillValid(msg) then: { // dispatch TDiscoveryLT
          sendOnce(); // recursive call to start new broadcast
        } else: { // stop broadcasting
          self.discoveryStopped(msg); // dispatch TCommLT
        }
      }
    };
    sendOnce(); // start broadcasting
    def stopDiscovery() {
      continuation.cancel();
      self.discoveryStopped(msg); // dispatch TCommLT
    };
    object: { def cancel() { stopDiscovery() } }
  };
};
```

The trait provides a single `dispatch` method. The inner `sendOnce` function is used to start an asynchronous iterative process. Upon each invocation of `sendOnce`, the message is broadcast to the ambient reference's reach. Subsequently, after a fixed timeout period, the process is repeated (by recursively invoking the method), but only if the message's discovery lifetime has not yet expired. This is tested by dispatching the `stillValid` message to a discovery lifetime trait (discussed later). If a message is no longer valid, the `sendOnce` function is no longer recursively invoked, and the communication lifetime trait is notified that no more messages will be sent (using the `discoveryStopped` message, which serves the same purpose here as it did in the extensional implementation).

Because the above implementation periodically broadcasts the message to any receiver in reach, regardless of whether they already previously received it or not, the broadcasted message includes a unique message identifier `id`. This identifier allows receivers to check whether they previously received the given ambient message. The `ttl` variable is the ambient message's "time to live": it denotes how much longer the message will be broadcast by the handler (this depends on the message's discovery lifetime). This value is also passed to any remote receiver, as a hint for how long they should store bookkeeping information regarding this message.

One For point-to-point messages, the delivery protocol is slightly more complex, because such messages should be delivered to at most one receiver. Figure 8.5 depicts the anycast protocol by which a single receiver is elected from the group of nearby potential receivers. The object in the centre represents the ambient reference. The dotted circle represents its communication range. All other objects are assumed to have a matching type and are hence eligible to receive broadcast messages if they are in range. The object marked `C`, however, is assumed not to match the `filter` predicate of the ambient reference (i.e. it is not in the reference's scope).

The protocol goes as follows. First, the `anycast` message described previously is broadcast via an M2MI omnihandle to all nearby objects. Each receiver locally filters its exported objects and if they satisfy the filter, the objects send a `reply` message to the ambient reference, passing along a reference to themselves. The ambient reference awaits replies for a predefined amount of time before processing them. If no object

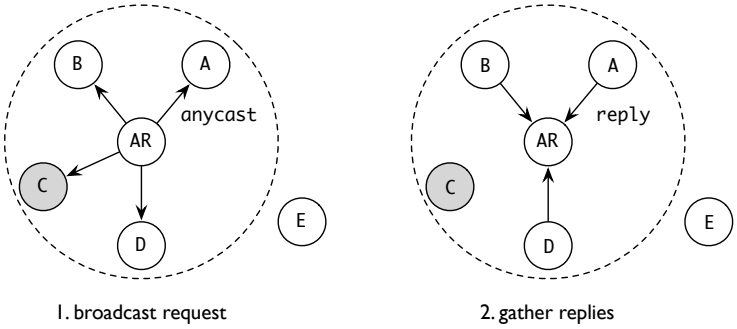

Figure 8.5: Anycast protocol to select a single receiver.

replied, this process of receiver selection is repeated. If there is at least one receiver, one is non-deterministically chosen to receive the point-to-point message[4]. The equivalent AmbientTalk code is shown below:

```
def TOne := object: {
  def createFuture() { makeFuture() };
  def dispatch(ar, msg) {
    def continuation;
    def sendOnce() {
      def receivers := [];
      ar.performAnycast(object: {
        def reply(rcvr) { receivers := receivers + [rcvr] };
      });
      continuation := when: ANYCAST_TIMEOUT elapsed: {
        if: !receivers.isEmpty then: {
          receivers[1 ?? (receivers.length + 1)] <+ msg;
          self.discoveryStopped(msg); // dispatch TCommunicationLT
        } else: {
          continuation := when: BROADCAST_RATE elapsed: {
            if: self.stillValid(msg) then: { // dispatch TDiscoveryLT
              sendOnce(); // recursive call to start new anycast
            } else: { // stop the anycast protocol
              self.discoveryStopped(msg); // dispatch TCommunicationLT
            }
          }
        }
      }
    };
    sendOnce();
    def stopDiscovery() {
      continuation.cancel();
      self.discoveryStopped(msg); // dispatch TCommLT
    };
    object: { def cancel() { stopDiscovery() } }
  };
};
```

[4]Because ambient references do not specify which potential receiver is elected as the actual receiver of the message, we could as well simply have awaited just one reply. The message would then always be sent to the first potential receiver to respond to the anycast.

The above code is structured according to the same pattern of periodically invoking an auxiliary `sendOnce` function. However, rather than performing a broadcast during each iteration, the anycast protocol described above is performed: the `anycast` message is sent to all receivers in reach, after which replies are gathered by the anonymous object passed as an argument to the `performAnycast` method. After a fixed timeout, if at least one receiver is available, one of them is non-deterministically chosen to receive the message. If no receiver replied *and* if the message's discovery lifetime has not expired, the process is repeated.

8.4.4 Representing Discovery Lifetime

The discovery lifetime trait implicitly controls how long to continue the repetitive broadcasting of the ambient message by means of its `stillValid` method. This method returns a boolean indicating whether the message's discovery lifetime has expired. It also provides a `timeLeft` method that returns the amount of discovery lifetime left, which is used in the arity trait for one-to-many messages defined in the previous section.

Instant A message with an instant discovery lifetime is easily represented as having a lifetime of 0 seconds, as follows:

```
def TInstant := object: {
  def timeLeft(msg) { 0 };
  def stillValid(msg) { false };
};
```

Note that an instant message is always broadcast or anycast at least once, because the arity traits perform at least one broadcast or anycast unconditionally before checking whether the message has expired.

Transient An ambient message with a transient lifetime has a predefined timeout period which can be used to derive whether the message is still valid if we know the time at which the message was sent. The time at which the message is sent is stored in the message's delivery handle in a field named `sendTime`. Given this information, a transient lifetime can be encoded as follows:

```
def makeTTransient(timeout) {
  object: {
    def transientPeriod(msg) { timeout };
    def timeLeft(msg) { (msg.handle.sendTime + timeout) - now() };
    def stillValid(msg) { self.timeLeft(msg) > 0 };
  }
};
```

Sustain A message with a sustained discovery lifetime does not expire automatically. This is reflected in the implementation as follows:

```
def TSustain := object: {
  def timeLeft(msg) { RECALL_PERIOD };
  def stillValid(msg) { true };
};
```

Note the implementation of `timeLeft`. Strictly speaking, invoking `timeLeft` on a message with a sustained discovery lifetime should return ∞. However, since the return value of this method is used as a lower bound on the "time to live" value by means of which potential receivers can know how long to remember the message for filtering out duplicates, we choose to return a predefined timeout period. The choice of the name is not accidental: this timeout period closely corresponds to the "recall period" of the service discovery engine in the extensional implementation: it is the maximum amount of time a receiver is willing to store information about previously received message sends. This implies that, if a receiver receives a sustained ambient message `m`, goes out of reach for longer than `RECALL_PERIOD` milliseconds and then moves back in reach, it will have forgotten about `m` and will receive the message twice.

8.4.5 Communication Lifetime and Expirable Messages

With respect to communication lifetime and expirable messages, the implementation for intensional ambient references is entirely reminiscent of the implementation for extensional references given in sections 8.3.5 and 8.3.6. Hence, we will not repeat this part of the implementation here.

8.4.6 Representing Exported Objects

In the extensional implementation, it was the ambient reference itself that for the most part avoided duplicate messages by only sending messages to new potential receivers as signalled by the discovery engine. This discovery engine hides a lot of the bookkeeping code necessary to filter out duplicate advertisement messages, etc. In the intensional implementation, ambient references no longer use a discovery engine and hence put the burden of keeping track of duplicate messages on the potential receivers. Therefore, we can expect the code to export objects to be more complex than the quite straightforward implementation discussed for an extensional reach in section 8.3.7. The following code exports object for use with the intensional implementation of ambient references:

```
def export: serviceObject as: typetagOrProtocol with: closure {
  def attributes := object: closure;
  def alreadyReceivedMessages := makeLeasedEntryTable();
  def broadcastMsgHandler := object: {
    def anycast(scopeFilter, replyHandler) { ... };
    def broadcast(scopeFilter, msg, id, ttl) { ... };
  };
  def pub := M2MI.export: broadcastMsgHandler as: typetagOrProtocol;
  object: {
    def unexport() {
      pub.unexport();
      alreadyReceivedMessages.deactivateLeaseRenewal();
    }
  }
};
```

The `closure` argument is used to initialise an object representing the exported service object's attributes. The variable `alreadyReceivedMessages` stores the list of message identifiers corresponding to those ambient messages that have previously been delivered to the exported `serviceObject`. Entries in this list are leased: each entry has a corresponding time to live (which can be prolonged). Once this time to live

elapses, the entry is removed from the list. Note that the object being exported is not the `serviceObject` directly, but rather a proxy object named `broadcastMsgHandler`. This proxy object understands the anycast and broadcast messages discussed in section 8.4.1. The proxy is not exported by means of AmbientTalk's built-in **`export:as:`** function but rather by means of the M2MI library[5]. This makes the object eligible for the reception of any messages sent to an M2MI omnihandle whose Java interface matches the Java representation of `typetagOrProtocol` (cf. section 8.7.3).

We now discuss the two methods of the `broadcastMsgHandler` object. Below is the implementation of the `anycast` method which implements part of the anycast protocol:

```
def anycast(scopeFilter, replyHandler) {
  if: scopeFilter(attributes) then: {
    replyHandler<-reply(serviceObject)
  }
};
```

Recall from section 8.4.3 that the `anycast` method is used to determine all potential receivers for a point-to-point message. As explained in section 8.4.1, the receiver first has to check whether it actually belongs to the scope of the ambient reference by applying the given `scopeFilter` predicate to the attributes of the exported service object. Only if the predicate is satisfied is a reference to the actual service object sent to the `replyHandler`. The service object can then receive a point-to-point message directly if it is elected as the receiver. It is not necessary to check for duplicate messages as a point-to-point message is guaranteed to be delivered at most once.

The `broadcast` method is invoked whenever a one-to-many message is broadcast by an ambient reference. Its implementation is shown below.

```
def broadcast(scopeFilter, msg, id, ttl) {
  if: scopeFilter(attributes) then: {
    if: alreadyReceivedMessages.containsEntry(id) then: {
      // msg already previously received, update lease time
      alreadyReceivedMessages.renewEntry(id, ttl);
    } else: {
      serviceObject <+ msg;
      alreadyReceivedMessages.addEntry(id, ttl);
    }
  }
};
```

Again, the implementation ensures that the message is only processed if the receiver is in scope of the ambient reference. The second if-test ensures that duplicate messages are weeded out: if the identifier associated with the message was previously received, the time to live of that identifier's entry is renewed and the message is further ignored. By renewing the leased entry for the message in the table of received messages, we ensure that no duplicates can ever be received as long as the receiver does not move out of reach of the ambient reference. If the message was not previously received, it is delivered to the actual service object and an entry is added to the table of received messages to filter out subsequent duplicate messages.

[5] The `M2MI` object is a facade object that mediates between AmbientTalk and the M2MI Java API.

8.4.7 Snapshots

Recall from section 8.3.8 that in the extensional implementation the `reach` collection itself represented the snapshot. In the intensional implementation, such a representation is not readily available. However, it is easy enough to think about the anycast protocol to select a receiver for a point-to-point message as building a temporary snapshot of the reference's reach. Hence, we can readily reuse this protocol to actually return a snapshot to an interested client, as follows:

```
def makeSnapshot() {
  def [fut,res] := makeFuture();
  def theSnapshot := [];
  self.performAnycast(object: {
    def reply(rcvr) { theSnapshot := theSnapshot + [rcvr] }
  });
  when: ANYCAST_TIMEOUT elapsed: {
    res.resolve(theSnapshot);
  };
  fut;
}
```

The above code returns a future which is eventually resolved with a table of references to all objects that replied to the anycast message before the anycast timeout elapsed.

8.4.8 Summary

We have sketched an alternative implementation of ambient references whose reach is represented only implicitly by means of M2MI omnihandles. Because the ambient reference no longer explicitly refers to potential receivers in reach, the delivery of a point-to-point message requires an anycast protocol to elect a suitable receiver. Broadcasting, on the other hand, is much easier and more efficiently implemented by broadcasting the message via the omnihandle. Because no underlying service discovery engine is used to detect new potential receivers, messages with a prolonged discovery lifetime are broadcast at regular intervals, until their discovery lifetime expires or their delivery is cancelled. This requires potential receivers to explicitly filter out duplicate messages. While this filtering of duplicate messages also happens in the extensional implementation, it is dealt with at the level of the service discovery engine and therefore was only implicit in our discussion. Now that both the extensional and intensional implementations have been explained, we can compare them with one another.

8.5 Evaluation

Having explained two alternative implementation strategies for ambient references, we can now describe their relative advantages and disadvantages:

- Because the extensional implementation has explicit references to all potential receivers at each point in time, expressing a point-to-point message is easy. However, the downside is that a one-to-many broadcast must be implemented by sending a unicast message to all receivers in the collection representing the reach. This is a very inefficient way of implementing broadcasting in terms of

network load. Especially in a wireless proximal ad hoc network, where all communication is inherently broadcast via radio, a broadcast to n receivers results in n consecutive radio signals, each only processed by one device and discarded by all others.

In the intensional implementation, expressing a point-to-point message send requires a two-phase anycast protocol to elect a receiver on the fly. The downside here is that, even though one device will be able to process the message, all devices need to reply to the single sending device at about the same time, potentially congesting the network. On the other hand, expressing one-to-many communication is both extremely expressive and extremely efficient: a single broadcast message addresses all potential receivers in one shot.

- In the extensional implementation, snapshots are readily available as the collection representing the reach *is* a continuously updated snapshot of the environment. In the intensional implementation, snapshots must be constructed on the fly, although this does not take much effort as the anycast protocol used to deliver point-to-point messages can be readily reused.

- In the extensional implementation, potential receivers pass their attributes by copy to ambient references that discover them. It is the ambient reference itself that applies its filter locally to the copied attributes. In the intensional implementation, the ambient reference periodically passes its filter by copy to potential receivers. It is the potential receivers themselves that apply the copied filter to their local attributes.

  The advantage of the extensional implementation is that it is often easier to pass state (attributes) by-copy than to pass code (the filter predicate) by-copy. Also, the filter is applied only once per receiver entering the reach of the ambient reference. The downside of this scheme is that if an ambient reference has a large number of potential receivers, it can quickly become swamped with discovery events and essentially becomes a bottleneck for processing all of the filters. In addition, by checking the predicate only once, changes in a remote service's provided attributes are *not* taken into account, such that over time the ambient reference may refer to receivers which are strictly speaking no longer in its scope.

  For the intensional implementation, all of these advantages and disadvantages are reversed: because the filter must be applied remotely, it should be self-contained and cannot depend on location-specific values which cannot be copied along. Also, the filter is passed along and checked in *each* message send, which can be a serious performance hit. On the upside, the scheme of simply broadcasting the filter predicate along with messages makes for a decentralised solution which scales to a large number of receivers: all receivers will process the filter predicate in parallel and the ambient reference is no longer a bottleneck. Moreover, by checking the predicate continuously, changes in the attributes of nearby services are always taken into account. Hence, an ambient reference's reach is always consistently a subset of its intended scope.

  If it can be guaranteed that the filter predicate of an ambient reference is referentially transparent (i.e. it is free of side effects and thus always returns the same output for the same input), receivers could cache the outcome of the filter, remembering for each ambient reference in communication range whether or not they belong to its reach. This would avoid any redundant filter checking. Changes in an exported object's attributes (which can be detected purely

locally, without any remote communication) could then invalidate this cache if necessary.

We have now described the ambient message delivery process in full. However, there are still some loose ends to tie up. For example, when is the `dispatch` method invoked upon the traits to start the delivery process, and by whom? And where is the return value of this method (the "registration" object) stored? How are futures and reply handles attached to ambient messages? The following section will fill in these details. It discussed how all previously explained code is integrated into AmbientTalk's metaobject protocol by representing ambient references as custom eventual references.

8.6 Ambient References as Custom Eventual References

As discussed in chapter 6, an ambient reference is represented as an eventual reference to the AmbientTalk programmer. To achieve this, we reuse the metalevel engineering techniques discussed in section 5.2.3 to represent object references by means of mirages. This enables ambient references to intercept and reify any asynchronous messages sent to them. With the ambient message made explicit, ambient references can add metadata to the message (such as a delivery handle) and can then deliver the message to the remote objects they designate based on the delivery policies indicated by the message's annotations.

In section 5.2.3.1 futures were defined as custom eventual references by means of mirages. Here, we use the same pattern to define ambient references:

```
def ambient: typetagOrProtocol where: filter {
  object: { /* empty proxy object */
  } taggedAs: [ AmbientReference ] mirroredBy: (
    extend: actor.defaultMirror with: {
      import TEventualRef; // use metalevel behaviour of eventual refs

      def intercept(message) {
        def handler := createHandler(message);
        def [fut, newMsg] := handler.attachFutureToMessage(message);
        def registration;
        def handle := object: {
          def future := fut;
          def sendTime := now();
          def cancel() { registration.cancel() };
        };
        registration := handler.dispatch(self,
                                         extendWithHandle(newMsg, handle));
        handle // the value of an ambient message send is always a handle
      };
      def makeSnapshot() { ... };
      ...
    })
}
```

An ambient reference is a mirage object whose mirror imports the `TEventualRef` trait to acquire the metalevel behaviour of eventual references. Recall that `intercept` is part of that trait's required interface, so an implementation must be provided by the composite object. Upon intercepting an asynchronous message sent to it, the ambient

reference first constructs a handler for the message. Based on the annotations of the message, a message handler is created for the given combination of annotations by means of trait composition, as noted in section 8.3.2[6].

When a handler for the message has been constructed, the ambient reference attaches the appropriate future to the message by invoking the `attachFutureToMessage` method on one of the communication lifetime traits discussed in section 8.3.5. This method returns a tuple consisting of the future to be attached to the message's handle, and a new message object to which the returned future is attached.

As discussed in section 6.2.1, a message send to an ambient reference always returns a delivery handle. This object allows the sender to access any future associated with the ambient message and to prematurely cancel the message delivery, which is primarily useful in the case of ambient messages with an unbounded lifetime. The handle object is explicitly constructed by the ambient reference upon message interception. Note that the handle object also stores the time at which the message was sent to the ambient reference, which is used by the `@Expires` delivery policy to calculate the appropriate expiration time (cf. section 8.3.6).

Finally, the ambient reference dispatches to the message's handler to start the delivery process. The auxiliary `extendWithHandle` method extends the message object with a `handle` field referring to its handle. This allows the traits to access the message's future and the `sendTime` field. Recall that the return value of the `dispatch` method is a registration object that enables the delivery policy to be cancelled. This object is used by the `cancel` method of the message's delivery handle to stop the message delivery process.

The above discussion on reifying ambient messages and attaching delivery handles fills in the missing details in the implementation outline sketched in section 8.2. Before concluding this section, we highlight once more the advantages of the stratified design of AmbientTalk's custom object references. Because base and metalevel are cleanly stratified:

- methods defined at the metalevel do not interfere with base level messages. For example, imagine an ad hoc application in which an ambient reference `cameras` designates nearby photo cameras in the ad hoc network. Also imagine these camera objects to implement a `makeSnapshot()` method to take a picture. When evaluating `cameras<-makeSnapshot()`, we can rest assured that `makeSnapshot` will not accidentally trigger the metaobject protocol of the ambient reference.

- methods defined at the base level do not interfere with metalevel messages. Consider the following function which is part of the ambient reference language construct module:

```
def snapshot: ambientRef {
  ambientRef<-makeSnapshot()@MetaMessage;
}
```

  Because the `makeSnapshot` method is defined at the metalevel, the corresponding message must be explicitly annotated with `@MetaMessage` such that it is sent to the mirror rather than the base level proxy. Continuing our previous example, the ambient reference language module can rest assured that the `makeSnapshot`

[6]For didactic purposes, our explanation abstracts from the fact that message delivery policies can be incomplete at the message-level and may depend on the default policies specified at the ambient reference level. The complete implementation can be found in appendix A.

message above triggers the metaobject protocol and not the methods of nearby cameras simply because they incidentally happen to have the same name.

We have now provided a complete overview of the implementation of ambient references in AmbientTalk. However, the role played by M2MI in this implementation still remains vague. The following section discusses where and how our implementation relies on the features provided by this Java library.

8.7 Many to Many Invocations

We now briefly describe the role of M2MI in the implementation of ambient references. First, we motivate why ambient references are built on top of M2MI. Second, we describe their particular use in the extensional and intensional implementations of ambient references. Finally, we describe how they can be technically used in AmbientTalk because M2MI has been designed as a Java library, not as an AmbientTalk library.

8.7.1 Motivation

Many to many Invocations are built upon a custom network protocol named the "many to many protocol" (M2MP). M2MP has been designed specifically for wireless proximal ad hoc networks. Its design is based upon the following assumptions, as stated by Kaminsky and Bischof [KB02]:

- The M2MP protocol does not assume device addresses. This allows devices to enter and leave ad hoc networks without having to maintain any routing information and without having to acquire or release e.g. IP addresses. This is an important advantage because infrastructure to maintain this information is unavailable in pure ad hoc networks.
- The M2MP protocol always broadcasts all messages to all nearby devices. This does not introduce additional performance penalties compared to other protocols because wireless radio transmissions are inherently broadcast in the sender's proximity.
- Whether or not a device should actually process a received message depends solely on the message's contents. Unlike traditional network protocols where message relevancy is determined by means of a recipient address, an application using M2MP must use the message's contents (preferably its initial bytes) to determine whether it should process or discard the message.
- M2MP assumes that message delivery is *mostly* reliable. As a result, it only includes minimal reliability measures. M2MP uses checksums to ensure that corrupted packets are discarded. However, unlike high-level protocols such as TCP/IP, M2MP does not perform automatic retransmission in case of dropped packets or because of packets arriving out of sequence.

Due to its lightweight nature and because of the above assumptions, M2MP forms a good substrate on top of which to build communication abstractions for mobile ad hoc networks. M2MI makes use of M2MP to transmit Java invocations, while ambient references in turn make use of M2MI to transmit AmbientTalk messages. We could

have also used M2MP directly, bypassing the M2MI layer. While this would probably have resulted in a more efficient system, the implementation on top of M2MI is simpler to understand and does not give up on any of the essential benefits of using M2MP.

8.7.2 Applying M2MI

M2MI is used in different ways in the different implementations of ambient references:

Intensional Reach In this implementation, M2MI is used explicitly. M2MI Omnihandles are used to represent the reach directly. The major advantage of this approach is that a one-to-many message can be broadcast directly and efficiently via the omnihandle.

Extensional Reach In this implementation, M2MI is only used implicitly through the service discovery engine. We have employed M2MI to construct an implementation of the discovery algorithm which is built into the AmbientTalk interpreter (written in Java) in AmbientTalk itself. Services discover one another by repetitively broadcasting AmbientTalk query messages to nearby devices and awaiting replies. M2MI readily supports this implementation strategy through its omnihandles. Services are considered unavailable when they become unresponsive to such query messages for longer than a given timeout period. This timeout period is what we have called the *recall period* in sections 8.3.3 and 8.4.4. Our main motivation behind implementing a service discovery engine in AmbientTalk itself is that this engine becomes much more amenable to customisation from within AmbientTalk.

As an example of such a customisation, based on the discovery engine written in AmbientTalk using M2MI, we have implemented the more energy-efficient DEAPSpace service discovery algorithm. This algorithm is developed specifically for ad hoc networks of resource-scarce devices [Nid01]. In this implementation, services still discover one another through broadcasts. However, rather than having every device advertise its own services to all other devices periodically, in DEAPSpace every device broadcasts advertisements of services offered by *all* nearby devices periodically. Hence, n broadcasts each containing 1 service advertisement are replaced by 1 broadcast containing n service advertisements. Because of this reduced number of broadcasts, individual devices eventually save more energy. However, even in the DEAPSpace implementation, the role of M2MI remains unchanged: M2MI omnihandles are still used to broadcast the aggregated service advertisements. Note that DEAPSpace allows objects to discover service objects which may be beyond their physical communication range (because announcements may be effectively forwarded by intermediate parties). The extensional implementation of ambient references does not use this implementation because discovered objects would not necessarily be in communication range, defeating the abstraction that ambient references only designate physically proximate objects.

8.7.3 Using M2MI

Recall the use of omnihandles in section 8.4.1 to represent the reach of an ambient reference:

```
def reach := omnireference: typetagOrProtocol;
...
reach<-anycast(...);
```

Using the M2MI library in Java directly, we would write:

```
Type reach = M2MI.getOmnihandle(Type.class);
...
reach.anycast(...);
```

Note that while in AmbientTalk `typetagOrProtocol` is really a variable, the type declaration `Type` in the Java code represents an actual static type (i.e. `Type` must be some concrete application-specific interface). Also note that while `anycast` is sent asynchronously even in the Java implementation, this cannot be directly expressed because Java offers only synchronous method invocation syntax and semantics. M2MI handles must be initialised with an object representing a Java interface type. The `omnireference:` wrapper in the above AmbientTalk code is responsible for mapping the above AmbientTalk code onto the corresponding Java code. To this end, AmbientTalk type tags and protocols must be converted into Java interfaces. Below, we discuss how the different options (discussed in section 6.4.1) to express an ambient reference's scope are implemented in terms of M2MI:

Type Tags For type tags, the conversion process is quite straightforward because Java interfaces, like type tags, are nominal types supporting multiple supertypes. A type tag `T1` defined as **`deftype`** `T1 <: T2, T3` is converted into the following Java interface type:

```
public interface T1 extends T2, T3 { }
```

Every AmbientTalk type tag which does not explicitly define a list of supertypes is implicitly a subtype of a root type. This root type is represented as the following Java interface:

```
public interface RootType extends EventListener {
  public void invoke(ATObject message);
}
```

Every generated interface eventually inherits from the above interface. Hence, every Java interface representing a converted AmbientTalk type tag automatically declares a method `invoke` which takes an argument of type `ATObject`, which is the AmbientTalk interpreter's root type for any value representing an AmbientTalk object. This `invoke` method is called from within AmbientTalk code to send an AmbientTalk message via an M2MI handle, e.g. `reach<-anycast(...)` is translated by the `omnireference:` wrapper into `handle.invoke(<-anycast(...))` where `handle` is the wrapped M2MI handle.

The `invoke` message is subsequently transmitted across the network using the M2MP protocol underlying M2MI. At its destination, this message is received by an M2MI message delivery thread and the `invoke` method is invoked by that thread on the exported AmbientTalk object. When an AmbientTalk object is exported by means of the M2MI layer (cf. the exported `broadcastMsgHandler` object in section 8.4.6), the object is not exposed directly to M2MI. Rather, a small wrapper (AmbientTalk) object is exported which implements a single `invoke` method. The conversion described

in the previous paragraph ensures that all invocations on M2MI handles from within AmbientTalk are converted into calls to this `invoke` method. This method essentially acts as a callback for the M2MI framework. While this callback is invoked from within a JVM thread (the M2MI delivery thread), the AmbientTalk/JVM symbiosis converts this method invocation into an asynchronous AmbientTalk message send such that it is properly scheduled in the actor's message queue.

Recall from section 5.3.2.2 that the linguistic symbiosis between the JVM and AmbientTalk considers a Java method invocation as an asynchronous event notification if the invoked method returns no value, throws no exceptions and is part of (a subtype of) the `java.util.EventListener` interface. `RootType` extends this interface such that an invocation of the above `invoke` method is treated by the symbiosis as an asynchronous event notification. Thus, when the M2MI framework delivers messages sent via M2MI handles to exported AmbientTalk objects, these invocations will not block the M2MI delivery thread, guaranteeing the responsiveness of the M2MI framework because the call returns immediately.

The conversion from AmbientTalk type tags to Java interfaces is done at runtime by synthesising the interfaces using a byte code generation library. The JVM's support for dynamic class loading ensures that these generated interfaces can be loaded when required to construct an appropriate M2MI handle.

Protocols For protocols, there exists no direct encoding into Java interface types. At first sight, one may expect to be able to convert a protocol into an equivalent Java interface whose method names match the selectors of the AmbientTalk protocol. However, Java features nominal subtyping, where types are only considered subtypes if this is explicitly stated in their declaration. One Java interface is not a subtype of another Java interface simply because the methods it defines are a subset of the methods defined by the other interface.

Protocols are thus represented in terms of a generic `Protocol` type tag plus an additional runtime type test which compares two protocols based on structural subtyping. The type tag is converted into a Java interface type using the conversion method described in the previous section. Hence, the implementation of protocols on top of M2MI handles is suboptimal, because any object exported by means of a protocol will receive all messages sent via M2MI handles for the `Protocol` interface. The objects then have to perform an additional type test to check whether they really are destined to receive the message.

Filter Predicates As previously explained in section 8.4.1, the optional filter predicates supported by ambient references cannot be used to efficiently scope message delivery itself. Filters are arbitrary AmbientTalk closures which are used in a conditional expression to explicitly ignore those messages which have been received by an object which is not actually in an ambient reference's scope. This may lead to inefficient use of the network since messages may sometimes be delivered needlessly.

Predicates could be used to efficiently scope message delivery itself if they are no longer regarded as opaque arbitrary AmbientTalk predicates but rather as a restricted query language. Such a representation allows the application of event routing techniques in content-based publish/subscribe systems [CRW01, EG01] to improve upon the accuracy of the message delivery. Integrating such techniques in the implementation of ambient references is left as a topic for future research.

To summarise, ambient references make use of M2MI handles to transmit AmbientTalk messages via the low-level M2MP protocol to remote objects. In order to achieve this, the characteristic function defining the scope of an ambient reference must be represented in Java. Type tags are directly compiled into Java interface types by means of bytecode generation. Protocols and filter predicates are not converted in this way, requiring additional runtime checks when messages are delivered using M2MI handles.

Now that both ambient references and the role of M2MI in their implementation have been explained, we turn our attention to the implementation of anonymous far references and multireferences.

8.8 Implementing Connection-oriented References

This section discusses the implementation of the referencing abstractions from section 6.6 whose goal was to reintroduce stateful communication. We discuss anonymous far references below and multireferences in the following subsection.

8.8.1 Anonymous Far References

Recall from section 6.6.1 that an anonymous far reference is constructed by invoking `discover:` `Type`, where `Type` can be a type tag or a protocol. This function call immediately returns a reference to any discovered object matching the given type. If no such object is available, the reference is unbound and transparently buffers asynchronous messages sent to it.

We already hinted at the implementation for anonymous far references in section 6.1.3.2. The code snippet below generalises the code snippet from section 6.1.3.2 such that it becomes applicable for any type tag or protocol:

```
def discover: typetagOrProtocol {
  def [future,resolver] := makeFuture();
  when: typetagOrProtocol discovered: { |service|
    resolver.resolve(service);
  };
  future
}
```

We use a first-class *future* to represent the anonymous far reference. We explicitly stated in section 6.6.1.2 that anonymous far references are to service discovery what futures are to asynchronous method invocations. The implementation reveals that the correspondence is such that one may readily use futures to represent anonymous far references: an unresolved future *is* an unbound anonymous far reference. It simply buffers incoming messages and forwards them to an actual far reference once the far reference has been acquired via service discovery.

Next to the above `discover:` function, the ambient reference language module also provides a `discover:where:` function that allows one to refine to which remote services the anonymous far reference may bind. The implementation of this function is more complex because the first service to be discovered may not immediately be a match. However, unbound anonymous references are still represented as first-class futures, so the essential aspects of the implementation remain unchanged.

8.8.2 Multireferences

Briefly recapitulating from section 6.6.3, a multireference is an object reference to a *fixed group* of objects. Any message sent to a multireference is automatically broadcast to all objects in the group and returns a multifuture which can be used to gather replies. A multireference can be created from any table of objects. The following code snippet shows how multireferences are implemented as yet another kind of custom object reference, entirely similar to the pattern previously shown in section 5.2.3.1 for futures and section 8.6 for ambient references:

```
deftype MultiReference;
def multiref: farrefs {
  object: { } taggedAs: [MultiReference] mirroredBy: (
    extend: actor.defaultMirror with: {
      import TEventualRef;
      def intercept(msg) {
        def [multifut,res] := makeMultiFuture(msg,farrefs.length);
        // attach the above multifuture to the message
        def newMsg := futurize(msg, multifut);
        // forward the message to the entire group
        farrefs.each: { |farref| farref <+ newMsg };
        multifut; // the return value is the multifuture
      };
      ...
    })
}
```

As can be seen, a multireference encapsulates a table of far references and explicitly forwards each message sent to it to all objects in the table. The forwarded message is equipped with a multifuture which may be used by the sender to conventiently gather any replies to the message. Note that the total number of receivers (`farrefs.length`) is passed to the `makeMultiFuture` method. This number serves as a hint to the multifuture as to how many replies it may expect. The `futurize` method has been explained before (cf. section 8.3.5) and is used to attach the multifuture to the message, resulting in an augmented `newMsg` which notifies its associated future when it has been processed.

To summarise, both the implementations of anonymous far references and multireferences use AmbientTalk's built-in far references to perform the actual (stateful) communication. Both abstractions are also implemented as custom eventual references. This fact is implicit in the implementation of anonymous far references because we have reused futures to represent unbound anonymous far references. An alternative strategy would have been to represent anonymous far references as explicit custom eventual references, like we did for futures, ambient references and multireferences. We have now described the implementation of all the different referencing abstractions from chapter 6 in terms of custom eventual references. Before concluding this chapter, we briefly discuss two more technical aspects of ambient references, to wit marshalling and garbage collection.

8.9 Marshalling Ambient References

We have yet to explain what semantics we attribute to the marshalling (also known as the serialisation or pickling) of ambient references. That is, when an ambient reference

itself is passed as an argument in a message to a remote object, should the ambient reference be passed by copy or by (far) reference?

It is customary in distributed object systems that objects representing references (i.e. proxies) are passed by copy [CDK05]. That is to say: the proxy object is passed by copy, not the object which it designates. Ambient references adhere to this semantics. Thus, when an ambient reference is parameter-passed as an argument (resp. as a return value via a future) in an inter-actor message send, the receiver (resp. sender) receives its own copy of the reference. It does not receive a far reference to the proxy object representing the ambient reference.

In order to copy an ambient reference, it suffices to copy the type tag or protocol and the filter predicate with which it has been initialised. By means of this data, the copy can re-initialise itself. As previously stated in section 6.4, the copy may designate a different set of objects than the original does, because the notion of proximity is relative to the device hosting the ambient reference. When an ambient reference is passed from a sender to a receiver, the original reference designates objects proximate to the sender, while the copied reference designates objects proximate to the receiver.

Type tags and protocol objects are isolates and can be copied without problems. However, copying the filter predicate is a more delicate issue. To copy the predicate, it must be implemented as a pass-by-copy closure (cf. section 8.4.1), since regular closures are passed by far reference. Therefore, the marshalling semantics of ambient references specifies that it is an error to parameter-pass an ambient reference whose scope is described by means of a regular AmbientTalk closure.

8.10 Garbage Collection

We briefly discuss how ambient references influence the garbage collection of the different objects that play a part in an ambient message send. Most importantly, any object that is exported in order to receive incoming ambient messages is no longer subject to automatic garbage collection. From a garbage collection point of view, the ambient of an actor acts as an additional "root" from which the GC algorithm derives which objects can still be referred to. To make an exported object subject to garbage collection, it must be explicitly unexported.

It is possible to build more sophisticated abstractions on top of the simple export/unexport operations provided by ambient references. For example, we could introduce leasing such that an exported object is automatically unexported after a certain period of time, unless it regularly receives ambient messages. We can readily reuse the "renew on call lease" abstraction discussed in section 4.6.4 to express such behaviour:

```
def export: serviceObject as: typetagOrProtocol for: timeout {
  def lease := renewOnCallLease: timeout for: serviceObject;
  // export the lease rather than the actual service object
  def registration := export: lease as: typetagOrProtocol;
  when: lease expired: {
    registration.cancel(); // unexport the object
  };
  registration
}
```

Note that by exporting a leased reference to the service object rather than the object itself, the object can be automatically unexported once the lease expires.

Ambient references themselves only become subject to garbage collection if they are not referred to by local objects *and* they do not contain any pending messages. As long as an ambient reference contains messages which it still has to deliver, the ambient reference may influence the distributed application and should not be reclaimed prematurely. A similar situation occurs when garbage collecting actors: actors should also not be garbage collected as long as there are messages pending in their mailbox, even if they are not referred to by other actors [KWN90].

Ambient messages themselves are also objects. These become subject to garbage collection once their lifetime has expired. As long as their lifetime has not expired, they are referred to by an ambient reference which prevents them from being collected prematurely.

8.11 Conclusion

We discussed two implementation strategies for ambient references. Both strategies differ in the way they represent the ambient reference's reach: either as an *extensionally* specified collection or as an *intensionally* specified broadcast channel. In either case, the different message delivery policies supported by ambient references are implemented in a modular way by means of trait composition. The advantage of such an implementation is that it makes the interplay between the different messaging policies explicit in the composition interfaces of the traits.

Ambient references, anonymous far references and multireferences are a new kind of *object reference*. While such abstractions are traditionally implemented by means of ad hoc implementation hooks, the aforementioned abstractions can be integrated much more tightly in the language by means of computational reflection. In particular, using AmbientTalk's support for intercession via mirages, these references can robustly *reify* and *intercept* any messages sent to them without interference from base level objects.

Both the extensional and the intensional implementations of ambient references also depend on the Many to Many Invocations library. The extensional implementation uses M2MI implicitly by means of a service discovery engine, while the intensional implementation directly uses M2MI omnihandles to broadcast messages to nearby devices. Accessing the Java objects that represent M2MI handles is possible because of the linguistic symbiosis between AmbientTalk and the JVM.

In the following chapter, we cross the abstraction barrier once again and examine ambient references from the programmer's rather than the implementor's point of view by using ambient references to implement a number of concrete ad hoc networking applications.

Chapter 9

Ambient References in Action

We validate ambient references by employing them to build two concrete ad hoc networking applications: a chat application featuring multiple chat rooms and a collaborative slideshow application, both designed for use by multiple collocated mobile devices. These applications are based on demo applications shipped with the M2MI library (which was previously introduced in section 3.3.7.4).

After discussing how these applications can be implemented by means of ambient references, we discuss how ambient references subsume M2MI handles, which can be seen as state of the art referencing abstractions for MANETs. We do so by means of the communication patterns (introduced in section 6.1) for which ambient references have been designed. We implement each communication pattern in Java using M2MI. As such, we can specify precisely how ambient references improve upon M2MI handles. M2MI is an apt candidate for a direct comparison with ambient references because:

- M2MI handles, like ambient references, have been designed from the ground up to be used in a mobile ad hoc networking setting. Comparing ambient references with an approach that is also developed for MANETs ensures that the comparison is biased toward neither system from the start.
- The survey summarised in table 3.2 shows that M2MI is the only approach to coordination designed specifically for MANETs which is based on the message passing metaphor. As a result, coordination is expressed similarly to how it is expressed by means of ambient references. On the other hand, this also implies our comparison is not suitable to bring to light all of the difficulties of integrating object-orientation with events discussed in section 3.4, because M2MI partially addresses some of those issues as well (e.g. representing events as messages).
- Ambient references and M2MI handles are of a comparable scale. Both are designed as relatively specific language constructs which can be applied to solve a concrete interaction between objects "in the small". Other systems, like Jini or one.world, are less amenable to a *direct* comparison with ambient references because they cover a wide range of aspects not covered by ambient references, such as deployment, persistency and security issues.

In each of the following sections, we first describe the example ad hoc networking application and then its implementation in AmbientTalk via ambient references. We conclude each section by highlighting the beneficial properties of ambient references on the structure of the distributed application.

9.1 Collaborative Chat

Consider a collaborative chat application running on mobile devices such as laptops, PDAs or cellular phones[1]. The goal of such applications is to allow people in one another's vicinity to chat with each another. As noted by Kaminsky and Bischof [KB02], this could be useful when discreet communication is required (e.g. in a library, in a conference during a presentation,...) or to communicate in very noisy places (e.g. in an engine room). The chat application under consideration has two use cases:

- Users should be able to create and join different chat rooms. Conversations should be private to a chat room, i.e. they should not be visible to users in other chat rooms.
- Users must be able to discover all proximate chat rooms without any preliminary configuration. Note that we do not consider any security issues; we consider all chat rooms to be "public".

9.1.1 Implementation via ambient references

We assume that each chat room is uniquely identified by means of a name (a string). We refer to the chat room in which a user is logged in at a given point in time as the user's *active* chat room. The implementation of the chat application is structured around two main communication patterns, each expressed by means of ambient references:

- An ambient reference designating all proximate chat applications is used to advertise the name of the active chat room. Peers who receive this advertisement message add the chat room name to their list of available chat rooms. By using the `@Sustain` delivery policy, the advertisement message is automatically broadcast by the ambient reference, without the programmer having to explicitly schedule broadcasts repeatedly.
- To communicate with all members of the active chat room, an ambient reference is used which designates all proximate chat applications whose active chat room name matches that of the sender. Broadcasting a message to nearby members of the active chat room is implemented by sending a one-to-many ambient message to this ambient reference.

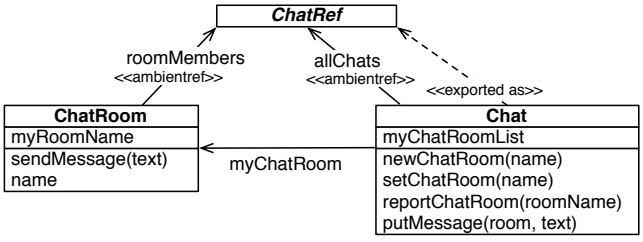

Figure 9.1: Structure of the collaborative chat application in AmbientTalk.

Figure 9.1 depicts the structure of the application. Note the following conventions:

[1] This application is based on the demo application in the `edu.rit.chat2` package distributed with build 20060329 of the M2MI library available at `http://www.cs.rit.edu/~anhinga/m2mi.shtml`.

- We assume classes in the UML class diagram to represent prototype objects in AmbientTalk.
- We represent type tags as abstract classes with no methods. An "exported as" stereotype indicates that an object is exported as a certain type tag.
- Associations between objects implemented by means of ambient references are attributed with the "ambientref" stereotype.

A `ChatRoom` object represents a chat room identified by means of a name. Below is the implementation for this abstraction in AmbientTalk:

```
deftype ChatRef;
def makeChatRoom(myRoomName) {
  def roomMembers :=
    ambient: ChatRef
    where: (script: { |chat| chat.chatRoom == myRoomName }
            carrying: `[myRoomName]);
  object: {
    def name := myRoomName;
    def sendMessage(line) {
      roomMembers<-putMessage(myRoomName,line)@[All,Instant,Oneway];
    };
  }
};
```

The ambient reference `roomMembers` designates all proximate `ChatRef` objects whose `chatRoom` attribute equals its own room name. We assume the intensional implementation of ambient references where the filter predicate is passed by copy to exported remote objects to check whether their chatroom matches the user's current chatroom. Therefore, the argument to **`where:`** must be a pass-by-copy closure[2]. The `sendMessage` method is invoked by the `Chat` object (which is explained below). The method is responsible for sending a text message to proximate members of the chat room. It does so by sending `putMessage` as a one-way, one-to-many ambient message with an instant discovery lifetime to the encapsulated ambient reference.

The `Chat` object in the diagram above is responsible for advertising the user's active chat room to nearby peers. Its implementation is shown below:

```
def Chat := object: {
  def username;
  def myChatRoom;
  def myChatRoomList;
  def reportHandle := object: { def cancel() { } };
  def allChats := ambient: ChatRef;
  ...
  def setChatRoom(roomName) {
    reportHandle.cancel(); // cancel previous advertisement message
    myChatRoom := makeChatRoom(roomName);
    reportHandle :=
```

[2]The library function **`script:`** body **`carrying:`** variables returns an isolate object representing a closure. It implements an `apply` method whose method body corresponds to `body`. The `variables` parameter denotes the names of variables which are lexically free in `body` that must be copied along with the closure. These variables become fields of the isolate object representing the closure. A pass-by-copy closure can be thought of as a script that may be executed remotely.

```
      allChats<-reportChatRoom(roomName)@[All,Sustain,Oneway];
    };
    def sendText(line) {
      myChatRoom.sendMessage(username + "> " + line);
    };
    ...
  }
```

The `Chat` object encapsulates an ambient reference `allChats` designating all proximate objects exported as a `ChatRef`. Its `myChatRoom` variable refers to the `ChatRoom` object in which the user is currently logged in (i.e. the active chat room). The `setChatRoom` method is triggered by the GUI each time the user selects a different chat room; the `sendText` method is triggered by the GUI each time the user has entered a text message[3]. Advertising the presence of the chat room is done by sending a one-way, one-to-many ambient message with a sustained discovery lifetime to all proximate `ChatRef` objects. Because message delivery is sustained, there is no need for the client object to repetitively broadcast it to nearby objects; the ambient reference does this on behalf of the client. However, because a sustained message send has no upper bound on its lifetime, its delivery should be explicitly cancelled at some point in time. Therefore, its associated delivery handle is stored in the variable `reportHandle`. Whenever the user switches chat rooms, the delivery of the old chat room's `reportChatRoom` advertisement message is explicitly cancelled.

The `Chat` object also stores a list of the names of all nearby `ChatRoom`s, including the ones created by itself. It also exports a nested object as a `ChatRef` such that its methods may be triggered by ambient messages:

```
def Chat := object: {
  ...
  export: (object: {
    def reportChatRoom(roomName) {
      myChatRoomList.containsKey(roomName).ifFalse: {
        myChatRoomList.add(roomName);
        // update the GUI
      }
    };
    def putMessage(senderRoom, line) {
      if: (senderRoom == myChatRoom.name) then: {
        /* display line in GUI */
      }
    };
  }) as: ChatRef with: {
    def chatRoom() { myChatRoom.name }
  }
}
```

The `reportChatRoom` method allows the chat application to react to the advertisement of nearby chat rooms. The implementation simply adds the broadcasted chat room name to its list of chat rooms if it did not do so before. Hence, the list of chat rooms can be conveniently used to filter out duplicate advertisements for the same chat room. The `putMessage` method, triggered whenever an ambient message is received

[3]The AmbientTalk implementation of the chat application reuses the GUI code of the original M2MI implementation in Java via linguistic symbiosis (cf. section 5.3).

from a proximate chat room's `roomMembers` ambient reference, simply displays the text in a chat window. Note that the text is only displayed if the sender's chat room matches the user's currently active chat room. This check is necessary to avoid a race condition not addressed in the original M2MI implementation, which could cause a user to receive a message sent by a member of its *previous* chat room, just after the user has switched chat rooms.

Note that the above object is exported together with a `chatRoom` attribute. This attribute is queried by the `roomMembers` ambient reference encapsulated by each chat room. Note that the attribute is actually implemented as a method rather than as a field. This is important, because it ensures that each time the filter of `roomMembers` is evaluated, `chat.chatRoom` transparently retrieves the user's current chat room. If it were implemented as a field, `chat.chatRoom` would once and for all refer to the chat room joined by the user at the time the object was exported.

9.1.2 Evaluation

The above implementation of the chat application in AmbientTalk boasts the following strengths:

- As noted several times in chapter 6, ambient references naturally cater to roaming. That is, the set of objects designated by e.g. the `roomMembers` ambient reference changes over time, without the programmer having to manage these changes explicitly. Furthermore, because the `roomMembers` ambient reference only designates objects whose chat room name matches the user's active chat room, service objects whose chat room attribute changes are dynamically excluded from or included into the scope of the ambient reference. Nevertheless, while the `roomMembers` ambient reference can guarantee to the sender that the receiver of the `putMessage` belonged to its reach at the time that message was sent, it does not guarantee that the receiver still belongs to its reach when the message is eventually delivered, since the receiver's attributes may have changed meanwhile. That is why, in the `putMessage` implementation, an additional test is necessary to ensure that the message does not arrive in the wrong chat room.

  If one were to implement a referencing mechanism akin to `roomMembers` in M2MI, the functionality of the filter predicate of ambient references would have to be implemented on top of more low-level designation mechanisms. In M2MI, the "scope" of a handle is either specified by means of an interface type (via omnihandles) or otherwise by means of specific object identifiers (by "attaching" or "detaching" objects to/from a unihandle or multihandle). To represent filter predicates, one must explicitly discover nearby objects via an omnihandle, test whether the discovered object matches the predicate and then attach the object to a multihandle. The multihandle then encodes an ambient reference whose scope is demarcated by a filter predicate.

- By using a sustained ambient message send to advertise the availability of a chat room, the `Chat` object does not have to manually reschedule the advertised message. Whenever the content to be advertised changes (e.g. whenever the user selects a new active chat room), a new ambient message is sent and the old ambient message's delivery is cancelled. If one were to implement such repeated advertising using M2MI directly, the time-decoupling afforded by sustained message

delivery must be implemented by hand. A timer thread must be used to repetitively broadcast an advertisement message via an omnihandle. Furthermore, this introduces concurrency issues because of the introduction of a separate timer thread to wake up the application.

- The above application does not decouple chat participants in time: a message is only propagated to peers which are proximate at the time the message is sent. However, because time decoupling can be explicitly controlled using the lifetime of an ambient message, time decoupling can be easily introduced. It suffices to simply prolong the lifetime of the `putMessage` message:

```
def sendMessage(line) {
  roomMembers<-putMessage(myRoomName,line)@[All,Transient(t),Oneway];
};
```

  By making the message delivery transient instead of instant, the chat message can be delivered to peers which enter communication range at a later point in time[4]. If communication between peers would be via M2MI handles, which do not cater to time decoupling, the code would have to be significantly refactored: one would need to add a timer thread and arrange for the `putMessage` message to be repeatedly broadcast. The introduction of a new thread also has repercussions on the application's concurrency control. This brings us to the final strength of the above AmbientTalk implementation.

- The chat application is concurrently manipulated by multiple entities: messages may arrive from the network and events may arrive from the GUI at any point in time. Thanks to the event loop architecture, objects never have to explicitly take any precautions with respect to concurrency control. In a corresponding Java implementation using M2MI, careful use of `synchronized` blocks is required to avoid race conditions between the application thread, the GUI event notification thread and any M2MI thread delivering messages arriving from the network.

9.2 Collaborative Slideshow

The second M2MI demo application under scrutiny is a collaborative slideshow application[5]. The goal of this application is to allow different devices representing physical screens and projectors to collaborate to display a slideshow. For example, a professor could use this application to broadcast a slideshow of his course both to a physical projector and to the laptops of nearby students. Moreover, multiple slideshows may be broadcast simultaneously and their slides can be displayed next to one another (or even overlaid) on the different screens. The three key entities in this system are:

Projectors which *provide* slideshows for display. Projectors correspond to devices (e.g. laptops, PDAs) running some slideshow presentation software.

Screens which display available slideshows on a physical screen.

[4]The programmer must be wary that prolonged ambient messages are not guaranteed to be delivered in sending order (cf. section 6.5), so the `putMessage` method's argument list is best extended with a sequence number identifying the ordering of the text messages.

[5]Our exposition is based on the code in the `edu.rit.slides` package distributed with build 20060329 of the M2MI library available at `http://www.cs.rit.edu/~anhinga/m2mi.shtml`.

Theatres which group different projectors and screens for the purposes of a collaborative slideshow. Theatres are identified by a user-defined name and are simply a means to scope the interaction between different projectors and screens such that they do not interfere with other ongoing collaborative slideshows.

In spite of what the above terminology might suggest, note that a physical projector should be considered to be a screen, not a projector: a physical projector's job is to display slides on a physical screen (which has no digital representation), not to broadcast them to a virtual screen.

A slideshow consists of a number of slides. When a projector advertises a slideshow, screens in the theatre incrementally prefetch and cache its slides. This way, when a projector later instructs all screens to display a certain slide, chances are high the slides are already downloaded at the screen's device, avoiding network latency when displaying a new slide.

9.2.1 Implementation via Ambient References

Figure 9.2 shows the structure of the key objects in the AmbientTalk implementation.

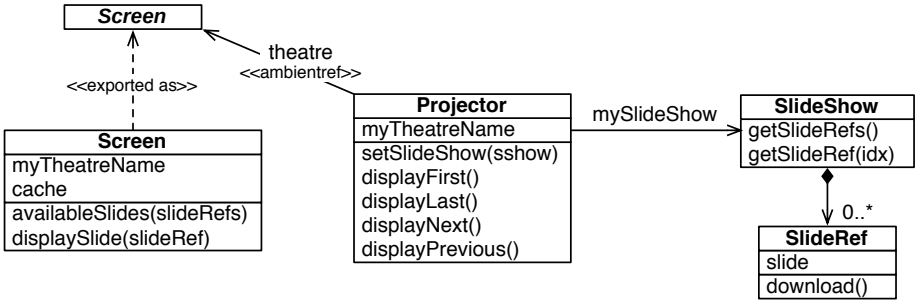

Figure 9.2: Structure of the collaborative slideshow application.

The code involves two distributed protocols. The first protocol is a service discovery protocol for theatre names. Both `Screen` and `Projector` objects advertise the existence of their active theatre (that is, the theatre to which they currently belong). Using AmbientTalk's built-in service discovery event handlers, the application can keep track of available theatre names. To join a theatre, the application can instantiate either a projector or a screen parameterised with the name of the joined theatre (see below).

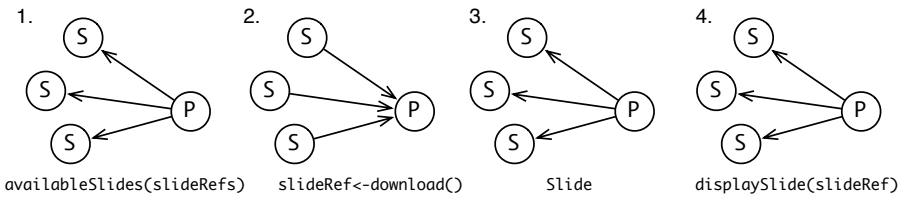

Figure 9.3: Protocol to transmit a slideshow from a projector to a group of screens.

The second protocol allows `Projector`s to send their slides to the active theatre incrementally. This protocol is depicted in figure 9.3 and will be explained throughout

the following paragraphs. In the figure, one projector transmits a slide show to three nearby screens in its theatre. A theatre is essentially a collection of `Screen` objects and is thus implemented by means of an ambient reference designating objects of type `Screen` whose theatre name matches the active theatre name (cf. the implementation of chat rooms in the previous section):

```
deftype Screen;
def makeProjector(myTheatreName) {
  // theatre designates all nearby Screen objects belonging to my theatre
  def theatre :=
    ambient: Screen
    where: (script: {|screen| screen.theatreName == myTheatreName}
            carrying: `[myTheatreName]);
  def mySlideShow;
  // delivery handle for theatre advertisement message
  def advHandle := object: { def cancel() { } };
  object: {
    // invoked by the GUI
    def setSlideShow(sshow) {
      mySlideShow := sshow;
      advHandle.cancel(); // stop advertising previous show, if any
      advHandle :=
       theatre<-availableSlides(sshow.getSlideRefs())@[All,Sustain,Oneway];
      ...
    };
  }
};
```

A `Projector` advertises that it has slides available for download by sending a sustained `availableSlides` message to its theatre. This advertisement message does *not* contain the slides themselves, as these are potentially large objects and broadcasting them repeatedly would congest the network. Instead, the advertisement message contains a table of far references to the slide objects.

When a `Screen` object receives the advertisement message, it prefetches any available slides which are missing from its cache. Slides are prefetched such that they are cached locally in the `Screen` before they actually need to be displayed (avoiding large network latencies when the presenter changes slides). To actually download a specific slide object, the screen sends a `download` message to the far reference to the slide object. The actual slide object is then passed by copy as the return value of this message (and is retrieved by the `Screen` via a future):

```
def makeScreen(myTheatreName) {
  def cache := HashMap.new();
  ...
  export: (object: {
    def availableSlides(slideRefs) {
      def fetchNextSlide(idx) {
        if: (idx <= slideRefs.length) then: {
          if: !cache.containsKey(slideRefs[idx]) then: {
            when: slideRefs[idx]<-download() becomes: { |slide|
              cache.put(slideRefs[idx], slide);
              when: seconds(1) elapsed: {
                fetchNextSlide(idx+1);
              }
```

```
            } catch: Exception using: { |e|
              // abort the download, clear all slides
              cache.removeAll(slideRefs);
            }
          } else: { // already got slide idx, fetch the next one
            fetchNextSlide(idx+1);
          }
        }
      };
      fetchNextSlide(1); // start downloading the slideshow
    };
    ...
  }) as: Screen with: {
    def theatreName() { myTheatreName };
  }
};
```

The `fetchNextSlide` function incrementally downloads slides from the projector to the screen (the recursive implementation is reminiscent of the library transmission protocol discussed in section 4.6.3). By means of far references and futures, the request/response prefetching protocol can be succinctly expressed, without the need for explicit callbacks or explicit session state to identify the slides.

The above prefetching process does not cause the screens to actually *display* the slides. In order to display a slide, the projector sends a `displaySlide` message to its theatre, again passing the identity of the slide to display. This message is sent using a sustained message delivery such that the application does not have to repeatedly broadcast the message by hand:

```
def makeProjector(myTheatreName) {
  ...
  def slideIndex := 0;
  def displayHandle := object: { def cancel() { } };
  def displayNext() {
    slideIndex := slideIndex + 1;
    displayHandle.cancel(); // cancel previous advertisement
    def slideRefToDisplay := mySlideShow.getSlideRef(slideIndex);
    displayHandle :=
      theatre<-displaySlide(slideRefToDisplay)@[All,Sustain,Oneway];
  };
  ...
};
```

If the screens have previously prefetched the slide, they can display it immediately; otherwise, they need to download it first:

```
def makeScreen(myTheatreName) {
  ...
  def displaySlide(slideRef) {
    if: (cache.containsKey(slideRef)) then: {
      gui<-display(cache.get(slideRef));
    } else: {
      when: slideRef<-download() becomes: { |slide|
        cache.put(slideRef, slide);
        gui<-display(slide);
```

```
        }
      }
    }
  }
```

9.2.2 Evaluation

The following aspects of the above implementation are noteworthy:

- The `theatre` ambient reference encapsulated by a projector object designates all proximate screens in the active theatre. Screens joining or leaving the theatre have no impact on the projector: the ambient reference allows the programmer to make abstraction from these events. Again, in M2MI these events must be dealt with explicitly. Lacking filter predicates to demarcate the scope of an M2MI omnihandle, all proximate screen objects must be explicitly discovered, queried for their theatre attribute and, if their theatre name matches that of the projector, attached to a multihandle. The multihandle is then used to communicate only with screens belonging to the same theatre. The programmer must take care that, when a screen switches theatres, the screen must be explicitly detached from the multihandle of its previous theatre (otherwise it may still receive slideshows from the previous theatre).

- We make use of sustained message sends to notify screens in a projector's theatre of slides available for download or which slide should be displayed. Because message delivery is sustained, this introduces time decoupling between the projector and the screens. If the information to be broadcast needs to change, we use the same pattern as used previously in the chat application and explicitly cancel the delivery of the previous ambient message before broadcasting an updated one. Using M2MI handles, sustained message sends must be implemented by explicitly scheduling repeated broadcasts in the code.

- Ambient references can serve as a communication channel by which other eventual references are passed to proximate objects. This allows those proximate objects to engage in a connection-oriented request/response interaction with a unique object. In the prefetching protocol, this is used to download a specific slide from the specific projector advertising that slide. While M2MI unihandles cater to connection-oriented interaction, they do not cater to request/response interactions: replies must be represented explicitly in the code by means of callbacks. The callback must also be parameterised with an identifier that correlates a call with its callback.

- As in the chat application, ambient references benefit from a tight integration with their event-driven host language, AmbientTalk. In particular, no explicit locking of objects is required, while a corresponding implementation in Java using M2MI would have to be carefully made multiple thread-safe.

In each of the above two sections, we have discussed how ambient references can be used in the construction of two concrete ad hoc networking applications. However, while we have briefly contrasted ambient references with M2MI, it remains vague how ambient references exactly improve upon this state of the art MANET abstraction at the level of concrete source code. In the following section, we will discuss how the

communication patterns introduced in section 6.1 have to be encoded in terms of M2MI handles, thus showing precisely the difference in level of abstraction between the two referencing abstractions.

9.3 Comparing Ambient References with M2MI

In section 6.1 we have introduced three communication patterns which constitute the motivation for including ambient references in AmbientTalk. In each of the following subsections, we show how to implement these patterns in Java using M2MI. In doing so, we illustrate precisely how the abstractions provided by ambient references have to be reconstructed in state of the art technology in a mainstream language.

9.3.1 Roaming

In section 6.1.1.1 we introduced the location service example. In the example, a mobile client continuously submits its latest location information to any `LocationService`. As the client roams, it may submit this information to different location services. In section 6.4.5.3 we have shown how ambient references directly cater to expressing such behaviour:

```
def accessPoint := ambient: LocationService;
def updateLocation(loc) {
  accessPoint<-submitLocation(clientId, loc)@[One,Sustain,Oneway];
};
```

Here, we discuss how such a communication pattern can be implemented in Java using M2MI. Point-to-point communication is expressed by means of M2MI *uni*handles. However, unihandles do not cater to roaming directly: they are associated with a unique object and in this respect resemble far references rather than ambient references. Therefore, the following implementation achieves roaming as follows: first, an omnihandle is used to ask all nearby location services to reply to the sender with a unihandle to themselves. When the sender receives a reply, it can deliver the message via the unihandle. When the sender does not receive any reply, it retries the whole process (to mimic the `@Sustain` message delivery policy of ambient references). The following class encapsulates the implementation:

```
class UpdateLocation implements ReplyHandler, TimerTask {
  private static final long ANYCAST_TIMEOUT = 2000L; // 2 seconds
  private boolean isMessageSent = false;
  private final ID clientId;
  private final Location loc;
  private final ReplyHandler replyHandler =
    (ReplyHandler) M2MI.getUnihandle(this, ReplyHandler.class);
  private final Timer myTimer =
    TimerThread.getDefault().createTimer(this);

  public UpdateLocation(ID clientId, Location loc) {
    this.clientId = clientId;
    this.loc = loc;
  }
  public void sendTo(LocationService omnihandle) {
    performAnycast(omnihandle, myTimer);
```

```
    }
    ... // continued below
  }
```

By invoking **new** `UpdateLocation(id, loc).sendTo(locationServices)`, the client can submit its location to any nearby service (provided `locationServices` denotes an omnihandle to all objects of type `LocationService`). This starts the anycast protocol which is implemented by the following methods:

```
private void performAnycast(LocationService omnihandle, Timer repeater) {
  omnihandle.anycast(replyHandler);
  repeater.start(ANYCAST_TIMEOUT);
}
public synchronized void reply(LocationService s) {
  if (!isMessageSent) {
    isMessageSent = true;
    s.submitLocation(clientId, loc);
  }
}
public void action(Timer theTimer) {
  if (theTimer.isTriggered()) {
    synchronized(this) {
      if (!isMessageSent) {
        performAnycast(theTimer); // repeat
      }
    }
  }
}
```

First, the `performAnycast` method broadcasts the `anycast` message to all proximate `LocationService`s. Since M2MI handles cannot directly express return values, the sending object must implement a callback method to process the reply. In the above example, the callback method is named `reply`. It is meant to be invoked with a unihandle to a location service. The boolean `isMessageSent` ensures that the `submitLocation` message is sent to at most one location service (this corresponds to the `@One` annotation of the above ambient message).

To deal with the case that no replies to the `anycast` message are received, the `performAnycast` method schedules the `action` method to be invoked by a separate `repeater` timer thread when a certain timeout period has elapsed. The `action` method checks whether a reply was received and if not, repeats the anycast protocol. The conditional test on `isTriggered()` is necessary to avoid a race condition that can occur when another thread concurrently cancels the timed action[6]. The `synchronized` block is necessary because access to the boolean `isMessageSent` must be made multiple-thread safe: the `action` method is executed by the timer thread while the `reply` method may be concurrently invoked by an M2MI thread.

The following code snippet sketches the implementation of the location service:

```
class LocationServiceImpl implements LocationService {
  public void anycast(ReplyHandler r) {
    r.reply((LocationService)M2MI.getUnihandle(this,LocationService.class));
```

[6]More information regarding this race condition can be found in the developer documentation of M2MI at `http://www.cs.rit.edu/~anhinga/m2miapi20040302/doc/edu/rit/util/TimerTask.html`

```
  }
  public void submitLocation(ID clientId, Location loc) { ... }
}
```

The service responds to the `anycast` message by invoking the callback with a unihandle to itself.

The astute reader will have noticed that the above implementation of roaming using M2MI handles almost directly corresponds to the anycast protocol used to implement point-to-point ambient message delivery (cf. section 8.4.3). This is not surprising and shows that the implementation of a language construct closely corresponds to the pattern of code which the construct is trying to hide.

To summarise, the above implementation in Java hints at the fact that ambient references can effectively abstract from the following patterns. First, point-to-point messages have to be encoded in terms of an anycast protocol via omnihandles, because unihandles do not directly cater to roaming. Second, request/response interaction must be implemented in terms of calls and callbacks. Third, prolonged message sends must be implemented explicitly by means of a timer thread.

9.3.2 One-to-many Communication

In section 6.1.2.1, we introduced the example of launching a vote in an ad hoc network to nearby team players in a mobile multiplayer game. We first implemented the example in AmbientTalk and later, in section 6.4.6.2, we have shown how ambient references can succinctly express the broadcasting and gathering of the votes as follows:

```
def nearbyTeamPlayers := ambient: Player where: { |p| p.team == myTeam };
def broadcastVote(poll, maxVoteTime) {
  def [future,resolver] := makeFuture();
  def handle :=
    nearbyTeamPlayers<-askToVote(poll)@[All,Expires(maxVoteTime)];
  whenAll: handle.future resolved: { |receivedVotes|
    resolver.resolve(receivedVotes);
  } ruined: { |exceptions|
    // ignore the votes of faulty players
  };
  future
};
```

The implementation in Java using M2MI is structured as follows: when the vote is initiated, a `report` message is broadcast at a fixed rate to all nearby players until the vote time elapses. Team members reply to this message and are subsequently asked to vote. Replies are gathered explicitly in a `HashMap` until the vote time elapses.

```
class BroadcastVote implements VoteReplyHandler, TimerTask {
  private static final long BROADCAST_RATE = 2000L; // 2 seconds
  private final String myTeam;
  private final String poll;
  private final long deadline;
  private final VoteReplyHandler replyHandler =
    (VoteReplyHandler) M2MI.getUnihandle(this, VoteReplyHandler.class);
  private final Player nearbyPlayers =
    (Player) M2MI.getOmnihandle(Player.class);
  private final HashMap receivedVotes = new HashMap();
```

```
  public BroadcastVote(String myTeam, String poll, long maxVoteTime) {
    this.myTeam = myTeam;
    this.poll = poll;
    this.deadline = System.currentTimeMillis() + maxVoteTime;
    performDiscovery(TimerThread.getDefault().createTimer(this));
  }
  public boolean stillValid() {
    return System.currentTimeMillis() <= deadline;
  }
  ... // continued below
}
```

When a `BroadcastVote` instance is created, the discovery process is started by invoking `performDiscovery`. This method broadcasts a discovery message (`report`) via the `nearbyPlayers` omnihandle. Since this omnihandle refers to *all* players (not only players of team `myTeam`), the discovery message contains `myTeam` as an argument and expects only team members to reply to the discovery request. Below are the methods implementing the calls and callbacks to communicate with remote players:

```
private void performDiscovery(Timer repeater) {
  nearbyPlayers.report(myTeam, replyHandler);
  repeater.start(BROADCAST_RATE);
}
public synchronized void playerDiscovered(Player p) {
  if (!receivedVotes.containsKey(p) && stillValid()) {
    receivedVotes.put(p, null); // ensures p can vote only once
    p.askToVote(poll, replyHandler);
  }
}
public synchronized void replyToVote(Player p, String answer) {
  if (stillValid()) { receivedVotes.put(p, answer); }
}
public void action(Timer theTimer) {
  if (theTimer.isTriggered()) {
    if (stillValid()) {
      performDiscovery(theTimer);
    } else {
      // vote expired, process receivedVotes
    }
  }
}
```

Upon receiving a `report` message, remote team players return a unihandle to themselves (via the `replyHandler` unihandle). This in turn triggers the `playerDiscovered` callback which checks whether the player has been discovered before. If this is not the case and the vote has not expired yet, the player is asked to vote (by sending a `askToVote` message to its unihandle). By adding the player to the `receivedVotes` map, the sender ensures that `askToVote` cannot be sent more than once to the same player. Again, the `replyHandler` is passed in the `askToVote` message such that the remote team player can reply his or her answer. Replying is done via the `replyToVote` callback. The vote is only taken into consideration if the vote deadline has not yet passed. Finally, recall that the `action` method serves as the callback for a separate

timer thread which repeatedly invokes the `performDiscovery` method until the vote deadline has passed.

The code for the `Player` service is given below. The `Player` only replies to `report` messages if its team matches the team of the sender. Replies are always explicitly performed by means of a unihandle to a `VoteReplyHandler`.

```
class PlayerImpl implements Player {
  private final String myTeam;
  private Player myHandle = (Player)M2MI.getUnihandle(this,Player.class);
  public PlayerImpl(String team) { myTeam = team; }
  public void report(String team, VoteReplyHandler r) {
    if (myTeam.equals(team)) {
      r.playerDiscovered(myHandle);
    }
  }
  public void askToVote(String poll, VoteReplyHandler r) {
    String answer = processPoll(poll);
    r.replyToVote(myHandle, answer);
  }
}
```

Comparing the above implementation to that using ambient references, we can note the following differences. First, service discovery must be performed explicitly by repeatedly broadcasting discovery requests. Issues such as filtering out non-team players and previously discovered team players must be dealt with explicitly. Second, replies must again be encoded explicitly via callbacks whereas multifutures serve this purpose when employing ambient references. Third, the **`whenAll:resolved:ruined:`** synchronisation provided by multifutures in combination with the `@Expires` policy must be explicitly encoded in terms of a timer thread. Fourth, the above code must be carefully made thread-safe since `playerDiscovered` and `replyToVote` may be invoked concurrently (by multiple M2MI invocation threads) and both manipulate the `receivedVotes` map.

9.3.3 Provisional Services

In section 6.1.3 we introduced the concept of a provisional service: an ad interim representation of a service that is not yet available. In the RFID shopping cart example, such a provisional service is used to both render a GUI and retrieve product prices from a `ProductDatabase` server in parallel, even if no such server is available yet. In section 6.4.4.2, we have shown how ambient references can play the role of a provisional service, as follows:

```
def db := ambient: ProductDatabase;
def renderGUI(server) {
  productsInCart.each: { |product|
    gui.addRow(product.id, product.name, "??");
    def handle := server<-getPrice(product.id)@[One,Sustain,Reply];
    when: handle.future becomes: { |price|
      gui.updateRow(product.id, product.name, price);
    }
  };
};
renderGUI(db);
```

If we were to implement this example in Java using M2MI, the implementation would be very similar to that of the roaming example displayed at length in section 9.3.1: the sustained message send (cf. `@Sustain`) would be represented by repeatedly broadcasting a discovery request via an omnihandle. Exported `ProductDatabase`s would reply to this message with a unihandle to themselves. This allows the sender to send the `getPrice` message to one such service (since the message is point-to-point, cf. `@One`). The only addition to the code in the roaming example is that the `getPrice` message send must additionally return the product `price` as a result (it is a two-way message, cf. `@Reply`). As we have seen in the previous section, such a return value can be implemented by means of an additional callback.

In short, M2MI handles cannot directly represent provisional services because they do not cater to time decoupling: if a message is sent to an M2MI handle when no matching exported objects are proximate, the message is lost. The time decoupling property must be implemented in terms of repeated message sends. The code examples given in the previous two sections clearly show how this can be accomplished.

9.4 Conclusion

We have shown ambient references in action by employing them to construct two small but representative, concrete applications for mobile ad hoc networks. Subsequently, we have compared M2MI handles with ambient references in terms of the generic communication patterns for which ambient references have been designed. This comparison confirms our statement that ambient references and M2MI handles are designed for providing different levels of abstraction. Whereas M2MI handles are extremely lightweight and flexible, they require the programmer to encode more high-level patterns by hand. We summarise the most noteworthy differences below:

- A recurring pattern in M2MI is the implementation of service discovery. Often, services are discovered by broadcasting discovery requests via an omnihandle. Subsequent communication occurs only via uni or multihandles which denote only a subset of the proximate objects. Ambient references can denote this subset expressively by means of its support for filter predicates. Filter predicates directly delimit the scope of the ambient reference.

- M2MI handles do not cater to request/reply interactions. Return values or exceptions must be explicitly encoded in terms of callbacks. While this approach works, it obscures the overall control flow of the application. It also proves our point that straightforwardly combining an object-oriented application with events requires the application to abandon bidirectional communication (cf. section 3.4.3). Ambient references use the machinery of non-blocking futures to maintain bidirectional communication and to counteract the partitioning of code in fragmented callbacks.

- M2MI handles do not support time decoupling directly. Time decoupling is always represented by means of separate timer threads which periodically call back on the application to trigger repeated message sends. The discovery lifetime of ambient messages has been designed to avoid such patterns in the application code itself.

- All M2MI applications are multithreaded. There is always at least one application thread and at least one other M2MI invocation thread (which delivers mes-

sages arriving from the network). Often, additional concurrency is spawned by the user interface (e.g. the AWT or Swing event loop thread) and one or more timer threads (cf. the `action` callback in the above examples). As a result, M2MI applications must always be carefully kept thread-safe. While it is tempting to "merely" add the `synchronized` modifier to all methods, this solution comes at the cost of an increased risk of deadlock. The communicating event loops model employed by AmbientTalk enforces this synchronisation pattern (actors process messages sequentially) *and* abolishes deadlocks by enforcing all communication among concurrent entities to be asynchronous.

The above comparison between ambient references and M2MI – a state of the art referencing abstraction for MANETs in a mainstream language – forms the final contribution of this dissertation. The following chapter summarises all previously made contributions and concludes the text with a discussion on limitations, influenced work and future work.

Chapter 10

Conclusion

In this concluding chapter, we revisit our research goals as stated in the introduction with hindsight and highlight the contributions of this dissertation once more. We discuss the rough edges to our proposal and outline those aspects of ambient references which may lead to interesting avenues of future research. We also point out other research which has been – to some extent – influenced by the work described in this dissertation.

10.1 Research Goals

In section 1.3, we explicitly stated our intended research goals. We briefly recapitulate these goals and address to what extent they have been achieved.

- It was our goal to be able to discriminate which coordination abstractions are appropriate for use in MANETs and which are not. In chapter 3, we put forth six criteria for which we extensively motivated why they are critical for coordination in MANETs. Our survey of related work further highlighted that classic object-oriented abstractions fail to adhere to the postulated criteria. Event-driven abstractions, on the other hand, appeared to be more suitable. Therefore,

- It was our goal to uncover why the object-oriented message passing metaphor for distributed communication was unsuitable for coordination in a MANET. Contrasting them with the more suitable event-driven coordination abstractions brought to light a number of fundamental differences. These differences complicate the composition of object-oriented with event-driven systems and have lead us to define the object-event impedance mismatch. Based on this,

- It was our goal to resolve the mismatch by developing a coordination abstraction that is both object-oriented and event-driven. As such, we wanted to prove by construction that it is possible to make distributed object technology scalable in mobile ad hoc networks.

We have achieved our proof by construction by developing a novel language construct in a novel programming language. Together, they form the chief contributions of this dissertation:

AmbientTalk/2 is a novel ambient-oriented programming language. Its important property with respect to our research goals is that the language combines objects with a pure event-driven (as opposed to multithreaded) concurrent execution model for the specific purposes of expressing coordination in MANETs. Furthermore, the language provides built-in support for publish/subscribe-based service discovery and is equipped with an extensive metalevel architecture. AmbientTalk/2 is in itself already suitable for expressing complex communication patterns in a MANET, as the musical match maker case study from section 4.6 indicates. However, primarily its lack of any direct space-decoupled communication abstractions have lead us to design and implement ambient references.

Ambient References are space-decoupled object references designating a volatile set of proximate service objects. Their expressive power lies in their unification of object-orientation with event-driven concepts. Their most distinguishing feature is that, unlike traditional object-oriented abstractions, they designate objects based on an *intensional* description. It is this property which decouples their clients and the service objects they designate in space. This space-decoupling, in turn, enables the direct expression of communication patterns – such as roaming, one-to-many communication and provisional services – which are otherwise absent from the AmbientTalk/2 programmer's toolbox.

While ambient references introduce space-decoupling, they do not cater to stateful communication. To this end, we have introduced anonymous far references, snapshots and multireferences. The merit of these abstractions is that they combine space decoupling with statefulness. Together with ambient references and far references, they form an extensive set of object designation abstractions from which a programmer can choose the most appropriate for the task at hand.

In chapter 7, we have shown how AmbientTalk/2, when extended with the different object designation abstractions introduced in chapter 6, forms a suitable platform for expressing collaboration between objects which are distributed across a MANET. We did so by discussing how the abstractions offered by this platform adhere to each of the six criteria put forward in chapter 3.

10.2 Restating the Contributions

For each chapter, we summarise what that chapter contributes to this dissertation's research goals:

- In chapter 2, we recapitulated the motivation behind ambient-oriented programming. In particular, we discussed how mobile ad hoc networks are characterised by the fact that connections between devices are volatile and that infrastructure to support communication is scarce. We also highlighted the limitations of AmbientTalk/1, the first ambient-oriented programming language.
- In chapter 3, we described six criteria that characterise coordination abstractions which are suitable for use in mobile ad hoc networks. Decentralised discovery and space-decoupled communication are key for coordination without infrastructure. Time- and synchronisation decoupling and connection-independent failure handling allow processes to communicate in the face of volatile connections. Finally, arity decoupling enables processes to engage in many-to-many interactions with an unknown number of proximate peers.

- In the same chapter, we used the criteria to extensively survey related work. Our summary in table 3.2 indicates that contemporary object-oriented message passing abstractions do not scale in MANETs, while event-based abstractions do.

- The above observation forces object-oriented programs to adapt to event-based communication abstractions. This adaptation is hindered by fundamental differences between the two paradigms, a phenomenon which we have titled the *object-event impedance mismatch*.

- In chapter 4, we introduced AmbientTalk/2 as the successor to AmbientTalk/1. While AmbientTalk/2 uses the same building blocks as its predecessor, the limitations discussed in chapter 2 have lead AmbientTalk/2 to employ a different object model, based on the communicating event loops of the E language [MTS05]. As a result, the language allows objects and events to be gracefully combined for the purposes of communication in MANETs.

- In chapter 5 we introduced metalevel engineering techniques in AmbientTalk. We primarily focussed on AmbientTalk's support for first-class messages, its reconciliation of mirror-based reflection with intercession through mirages and applying mirages to represent custom object references. We discussed how AmbientTalk provides interoperability with the Java Virtual Machine through linguistic symbiosis based on inter-language reflection. Importantly, the linguistic symbiosis enforces a safe composition of AmbientTalk's event loop actors with the JVM's threads.

- Chapter 6 introduced ambient references, which constitute our attempt at unifying the event-driven properties of publish/subscribe systems with the message passing abstraction of object-orientation. We described ambient references from a designer's point of view, discussing in detail the different message delivery policies and the communication patterns which they support.

- In chapter 7, we described how AmbientTalk, when extended with support for ambient references, satisfies the six criteria for coordination in MANETs. Furthermore, this coordination is done in an event-driven yet object-oriented way. The effects of the object-event impedance mismatch are avoided because ambient references unify objects with events by representing the event broker of publish/subscribe systems as an object reference. Chapter 7 concluded with a discussion on how related work can be recast in terms of ambient references.

- Chapter 8 extensively elaborated on the implementation details of ambient references. We discussed how the representation of an ambient reference's reach leads to two different implementation strategies. What both strategies have in common, however, is their use of traits to express the different message delivery policies in a modular way. Finally, we discussed the finer points of representing ambient references as first-class object references (by means of reflection) and how the implementation can safely reuse the many to many invocations (M2MI) library [KB02] by means of the linguistic symbiosis explained in chapter 5.

- In chapter 9, we implemented two concrete ad hoc networking applications by means of ambient references. Subsequently, we contrasted ambient references with M2MI, a state of the art referencing abstraction for MANETs in Java. By

re-implementing the motivating examples for ambient references introduced in chapter 6 via M2MI, we have demonstrated precisely where ambient references gain in expressiveness.

10.3 Limitations of our Approach

We already highlighted specific technical shortcomings of ambient references in section 7.4. Important as they may be, we will not repeat them here, but rather focus on the limitations of the concept of ambient references as a whole.

10.3.1 Language Integration versus Language Separation

Our language-oriented approach seeks to unify objects and events by means of dedicated programming language constructs. As a result, our solution requires a homogeneous software platform: all distributed services are assumed to be implemented as AmbientTalk objects. This is in contrast with coordination abstractions like that of the original Linda language which strive for a clean separation between application and coordination aspects [GC92].

Separating programming from coordination language has benefits which are lacking in an integrative approach. A separate coordination abstraction can be used in conjunction with any programming language; it is not a language construct tailored to fit the design of a single language. This makes such a coordination abstraction suitable for use in a heterogeneous network where it cannot be assumed that all software is written in the same or a similar platform. This may well prove to be an important asset in a ubiquitous computing context. On the other hand, as we have extensively discussed in section 3.4, separating the programming language from the coordination abstraction leads to an impedance mismatch: their integration will not be without problems as both systems were designed in separation, with different goals and different driving forces shaping the design.

10.3.2 Custom Message Delivery Policies

As they are presented in this work, ambient references provide a fixed set of message delivery policies for the programmer. While the programmer has some degree of freedom in selecting between a number of predefined design dimensions (i.e. an ambient message's arity, communication and discovery lifetime), ambient references do not allow the programmer to compose his or her own delivery policies.

Because our framework of message delivery policies is fixed, it may be that ambient references will not be directly usable for some application scenarios because they abstract from too much interaction details. Comparing them with a more low-level abstraction like the M2MI handles (cf. section 9.3), it is clear that a gain expressiveness comes at the cost of less fine-grained control over the communication.

One way to provide the programmer with the best of both worlds (i.e. a referencing abstraction which is both high-level *and* customisable) is to restructure the implementation of ambient references according to the rules of an open implementation [KP96]. An open implementation provides the programmer with a well-defined API through which the implementation may be adjusted in a very controlled way. While ambient references have been *implemented in* an open implementation (the metaobject protocol of AmbientTalk), they are themselves not *designed as* an open implementation. One

important aspect of the current implementation is that the different delivery policies are at least already specified modularly as traits. In an open implementation, programmers would be able to add their own traits to the implementation, together with an annotation that would select the corresponding trait.

While the current trait hierarchy most probably requires a redesign to accomplish an open implementation of ambient references (because it is presently too biased towards expressing only the envisioned message delivery policies), we see no fundamental issues that would prohibit us from doing so. AmbientTalk's metaobject protocol is flexible enough to allow new delivery policies to be composed at runtime.

An alternative to introducing custom delivery policies as special annotations is to consider what minimal subset of the current taxonomy can be provided to the programmer such that he can build more useful delivery policies on top of ambient references, without modifying the implementation of the language construct itself. We are aware that the current taxonomy of delivery policies may overwhelm the programmer. In hindsight, the taxonomy of delivery policies should be considered a roadmap that has helped us in exploring the design space of sending messages to a volatile group of proximate objects. From our experience, in most cases expirable ambient messages are the most appropriate delivery policy. By promoting this policy as the default, the programmer is left solely with the task of determining an appropriate timeout period and to choose between point-to-point or one-to-many communication.

10.4 Work influenced by Ambient References

We highlight recent work that was influenced by the work put forward in this dissertation.

Ambient Bindings Plšek et. al [PMS07] have applied the notion of an ambient reference in a component-oriented software engineering context. As a point in case, they have extended Fractal [BCL$^+$06], a component model for the Java programming language with what they call *ambient bindings*. In a component-based program, components are linked via bindings (also known as connectors). Bindings are often associated with contracts (e.g. provided and required interfaces) to which both parties have to adhere before they can be connected. Ambient bindings transpose loose coupling between objects into loose coupling between distributed components. We point out that the experiment of Plšek et. al is based on the exposition of ambient references preceding this dissertation, as discussed in section 7.3. Also, as can be expected, the concept of an ambient binding is less tightly integrated with the host language than in the case of ambient references and AmbientTalk, as can be witnessed from the absence of non-blocking futures to support asynchronous request/response interactions (they employ explicit callbacks instead).

Not surprisingly, the implementation of ambient bindings is quite reminiscent of the implementation of ambient references. In Fractal, components can be controlled via what is known as a *control membrane*. This membrane acts as a reflective interface to the component, enabling the expression of non-functional component requirements separate from the base-level component code [PMS07]. The implementation of ambient bindings entails the deployment of a base-level proxy component, called an ambient *interceptor* in their system. The re-binding behaviour of this interceptor is controlled by an ambient *controller*, which is situated at the membrane-level. This is entirely

reminiscent of our mirage/mirror distinction in the reflective implementation of ambient references (cf. section 8.6).

Proximity References In his masters dissertation, Ramiro has made initial attempts at integrating proximity with ambient references and service discovery [Ram08]. In his extended system, service discovery in AmbientTalk can be scoped according to physical proximity measures. Both exported objects and discovery listeners can indicate the physical boundaries of their interactions. In his experimental implementation, these boundaries are implemented as arbitrary relations over the Euclidean distance between interacting mobile devices. In Ramiro's experimental setup, this distance is approximated by means of the devices' GPS coordinates.

Next to scoping AmbientTalk's service discovery protocol, Ramiro also introduces the notion of a *proximity reference*. A proximity reference is an anonymous far reference (as discussed in section 6.6.1) with the difference that the anonymous far reference's scope can additionally be restricted by means of physical proximity relations. For example, one may create a reference to a `Printer` service which may be at most 10 meters away by means of the following code:

```
def printer := proximity: Printer
               attach: myGPSCoordinates
               in: euclidean_distance(meters(10));
```

One limitation of Ramiro's experimental setup is that it is only possible to express proximity as a *subset* of the device's physical communication range. For ad hoc scenarios, this is not a problem because in most cases, devices cannot communicate with devices outside of their communication range anyway. However, in a nomadic networking setup, where devices can potentially communicate with a great number of other devices by means of shared infrastructure (e.g. a base station), other implementation techniques are required such that proximity can be expressed independent of the physical range of a device's wireless communication link. This requires looking into novel routing techniques such as *geocasting*, where the destination of network packets can be constrained by means of the geographical location of the receivers [Mai04].

Supporting Mobile Actors Ambient references have been used to incorporate transparent network reconfiguration in a mobile actor system [VVG+07]. In AmbientTalk/1, Vallejos implemented migration of mobile actors, much in the spirit of migratable actors in Salsa [VA01] and mobile agents in ProActive [BBC+06]. One problem in such systems is the updating of actor references after moving actors from one host to another. While the sending and receiving hosts involved in the migration process can locally update their references, third party hosts may be unaware of the migration and keep on referring to the moved actor at its original host. By replacing actor mail address-based communication by sustained point-to-point communication over ambient references, programmers can effectively abstract from actor mobility because the ambient reference can be made to denote a unique service object without reference to any device address.

In this dissertation, we have stressed the use of space-decoupled communication to make an object unaware of different *objects* that represent conceptually the same service (cf. section 6.1.1). The abstraction afforded by ambient references is depicted in figure 10.1 on the left: the ambient reference may deliver messages to either service

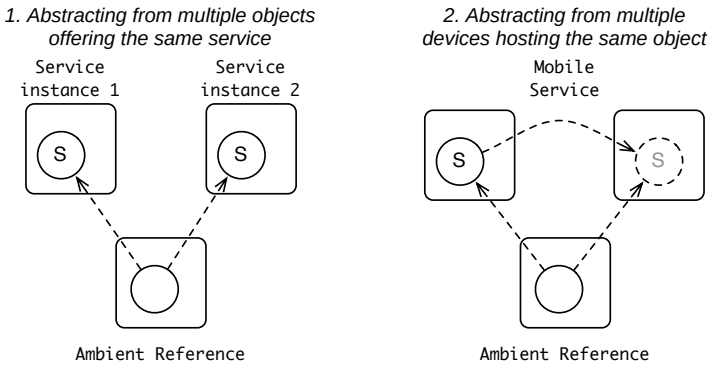

Figure 10.1: Abstraction afforded by space-decoupled communication.

(the boxes represent different actors or devices). The experiment with mobile actors shows that space-decoupled communication is equally important to be able to abstract from different *devices* that host what is conceptually the same object. This is depicted on the right-hand side of figure 10.1: the ambient reference will deliver messages to the object regardless of the device that hosts it.

The White Language The AmbientTalk language in itself has spawned third party interest in language design for ubiquitous computing. White [Qui07] is a nascent programming language for mobile networks, whose language design is influenced by AmbientTalk. While the language still appears to be in its very early stages of development, with no concrete implementation for download or publications for documentation, it is nevertheless promising to see that our approach instigates other researchers to tackle the same problems and improve upon our work by means of a similar, language-driven approach.

10.5 Avenues for Future Research

In this section, we discuss how our research could be extended or studied in a different context without the emphasis on addressing limitations, as was the case in section 10.3.

10.5.1 Aspect-oriented Programming

One research domain which has been left unexplored in this dissertation is that of aspect-oriented programming (AOP) [KLM+97]. The goal of AOP is to be able to modularise crosscutting concerns. A crosscutting concern is an aspect of an application which is difficult to capture using traditional modularisation techniques such as traits, classes, methods or functions. Traditionally, distributed programming has served as a resourceful application domain for AOP, because it engenders a lot of non-functional aspects in addition to the application's core logic. Examples are taking care of object serialisation during parameter passing, performing concurrency control, enforcing certain security properties, etc. In fact, one of the first aspect-oriented languages, D, was a domain-specific aspect language for tackling exactly these issues [LK97].

In general, AOP techniques have not yet been applied to do ambient-oriented programming. In Dedecker's dissertation as well as in this dissertation, metaobject pro-

tocols are used as a more general mechanism to capture crosscutting aspects in distributed programming such as message sending and message reception, following a long-standing tradition of applying reflection to solve concurrent and distributed programming problems [OIT92, McA95, MMY96, BGL98, CBM+02]. It remains to be investigated whether the full power of a metaobject protocol is really required to express certain ambient-oriented language constructs. In this dissertation, they key exploited feature of the metaobject protocol is the ability to intercept message sends. There is a strong correspondence between the ability to intercept message sends and the use of "around advice" to instrument a method invocation with additional code. Also, ambient references make explicit use of annotations on message sends to direct the message sending process. Such annotations could similarly be exploited by an aspect weaver to weave in the appropriate aspect.

While AOP can definitely be used to provide a modular *implementation* of a language construct like ambient references, a more fundamental consideration is whether AOP can replace the language construct *itself*, or indeed to abolish the need for an ambient-oriented programming language by advising programs written in a general-purpose language. It is our conjecture that AOP ultimately depends on a suitable base language whose aspects it is to describe. The intent of AOP is to modularise crosscutting concerns, not to turn a general-purpose language into a domain-specific language. If aspects are used to encode "language features" (e.g. asynchronous message passing), these features still need to be composed with the base language. In this light, AOP is prone to the impedance mismatch resulting from the composition of two systems discussed in section 1.4.2.

10.5.2 Service Selection

One issue currently overlooked by ambient references is how to *rank* available nearby services. That is to say, if multiple matching services are available in an ambient reference's reach, which one should the ambient reference prefer as the actual receiver of e.g. point-to-point messages? Currently, this selection is non-deterministic, i.e. at the implementation's discretion.

Closely related to ranking services in reach is the idea of making the definition of the reach *itself* dependent on ranking metrics. The idea here is to replace the boolean filter predicate of ambient references by a metric that returns how well a nearby object matches certain requirements (e.g. in the form of a number between 0 and 1). This would result in the scope (and consequently also the reach) of an ambient reference becoming a *fuzzy set* [Zad65]. Fuzzy sets are sets whose "characteristic function" is exactly such a metric. Investigating the impact of this change in representation on message passing remains a topic for future research.

10.5.3 Session Types

Ambient references currently support designation based on nominal typing (using type tags) and structural typing (using protocols). One could envision designation based on more advanced typing schemes. The work on *session types* shows how static types can be used to describe the *behaviour* of processes in terms of sending and receiving messages [Hon93] and has recently been applied to distributed object-oriented languages [DCYAD05]. For example, the session type `?int?int!bool` describes an object that first receives two messages containing an `int` and subsequently outputs a message containing a `bool`. It matches the dual type `!int!int?bool`. Session types may

also be conditional or iterative, such that complex interactions between objects (e.g. the library transmission protocol discussed in section 4.6.3) can be expressed as finite state machines whose state transitions correspond to message sending and message reception.

Traditionally, session types have been studied in a synchronous, two-party interaction context. An extension of session types to support asynchronous, multi-party interactions as supported by ambient references has only been proposed recently [HYC08]. It remains to be seen how this state of the art research can be used for the purposes of making interactions among objects using ambient references more type-safe.

10.6 Concluding Remarks

Today, we find ourselves at the dawn of the ubiquitous computing era. While Weiser's vision will probably take many years still to become reality, the research on mobile ad hoc networking technology today already provides us with a glimpse of what future applications may offer. At the same time, it provides us with a glimpse of the *difficulties* with which the developers of those applications will be confronted. Next to all the difficulties already engendered by distributed computing, the very properties of the ad hoc network make it difficult – or indeed impossible – to use tried-and-tested patterns such as naming and directory servers or client-server interactions. MANETs are particularly unforgiving with respect to the traditional distributed computing abstractions put forth by the otherwise so successful object-oriented paradigm.

The goal of ambient-oriented programming is to make the metaphor of distributed computing as a natural extension of object-oriented message passing as accurate as is practically possible in MANETs. The seminal work of Dedecker has laid the foundation for this dissertation by showing how active object models, asynchronous message passing abstractions and some form of distributed naming form the suitable ingredients for objects to scale in a MANET [Ded06]. While his investigation of asynchronous message passing and its integration into an OO model was extremely thorough, the combination of distributed naming (space-decoupled communication) with object-oriented programming left much to be desired still.

This work can be seen as the continuation of Dedecker's work in unifying OOP with distributed computing for MANETs. We have primarily focused on the question of how to address objects whose identity is a priori unknown. Our investigation has lead us to evaluate different coordination abstractions, and to conclude that event-driven, publish/subscribe approaches scale best. While Dedecker already recognised this, what was still lacking was a mechanism to transpose this event-based, publish/subscribe communication into an object-oriented language abstraction. Ambient references have been designed exactly to fill this gap in the ambient-oriented paradigm. Employing ambient references, programmers can denote a volatile set of proximate objects by means of an intensional description. Communication with objects in the set can be customised along various design dimensions by annotating the messages sent to these references.

In the introductory chapter, we discussed the gap between ubiquitous computing and contemporary software development tools. We noted how, ultimately, our research is about closing that gap. While the tools that are described in this dissertation are state of the art research vehicles, we argue that, as this technology matures, it can be the basis of a solid software development platform for constructing applications in the coming age of calm technology.

Appendix A

Ambient References Source Code

This appendix contains the complete implementation of ambient references in AmbientTalk as they are discussed in chapter 8. The implementation is divided among four main modules: the language module implementing all code common to both the extensional and intensional implementation of ambient references, a module implementing the generic behaviour of custom eventual references in AmbientTalk and finally the modules containing the implementation details of the extensional resp. intensional implementation. Each of the following sections contains the complete source code listing for each module.

A.1 Ambient References Language Module

File `at/lang/ambientrefs.at`

```
/* This module implements the behaviour common to both the extensional
 * and intensional implementation of ambient references and provides the
 * public interface of the ambient references language construct.
 */
def Util := /.at.support.util;
def FuturesModule := /.at.lang.futures;
def MFuturesModule := /.at.lang.multifutures;
def OneWayMessage := FuturesModule.OneWayMessage;
def ProtocolM := /.at.lang.structuraltypes;
import /.at.support.timer;
import /.at.lang.firstclassrefs;

deftype IllegalAnnotation <: /.at.types.Exception;
def XIllegalAnnotation :=
  /.at.exceptions.createException(IllegalAnnotation);
deftype ARAnnotation;

deftype MsgArity <: ARAnnotation;
deftype One <: MsgArity;
deftype All <: MsgArity;
```

```
deftype MsgDiscoveryLifetime <: ARAnnotation;
deftype Instant <: MsgDiscoveryLifetime;
deftype TransientT <: MsgDiscoveryLifetime;
deftype Sustain <: MsgDiscoveryLifetime;

def Transient(period) {
  extend: TransientT with: { |period|
    def ==(other) { super == other };
  } taggedAs: [/.at.types.Isolate, /.at.types.TypeTag ];
};

deftype MsgCommunicationLifetime <: ARAnnotation;
deftype Oneway <: MsgCommunicationLifetime;
deftype Twoway <: MsgCommunicationLifetime;
deftype Reply <: Twoway;
deftype DueT <: Twoway;

def Due(timeout) {
  extend: DueT with: { |timeout|
    def ==(other) { super == other };
  } taggedAs: [/.at.types.Isolate, /.at.types.TypeTag ];
};

deftype Expirable <: TransientT, DueT;
def Expires(timeout) {
  extend: Expirable with: { |timeout|
    def ==(other) { super == other };
  } taggedAs: [/.at.types.Isolate, /.at.types.TypeTag ];
};
def extendWithHandle(msg, handle) { extend: msg with: {|handle|} };

def AmbientRefsModule(ARImplModule := /.at.ambient.ar_extensional_impl) {
  // a pass-by-copy object that resolves to the correct
  // implementation module when deserialised
  def ARImplModuleCapsule := ARImplModule.capsule;
  def arityTraitFor(annotation) {
    // ARITY = ONE | ALL
    if: (annotation.isSubtypeOf(One)) then: {
      ARImplModule.TOne;
    } else: {
      if: (annotation.isSubtypeOf(All)) then: {
        ARImplModule.TAll;
      } else: {
        raise: XIllegalAnnotation.new(
          "Illegal arity annotation: " + annotation);
      }
    };
  };
  def commLTTraitFor(annotation) {
    // REPLY = DUE | ONEWAY | FUTURE
    if: (annotation.isSubtypeOf(Oneway)) then: {
      TOneway;
    } else: {
```

```
      if: (annotation.isSubtypeOf(DueT)) then: {
        makeTDue(annotation.timeout);
      } else: {
        if: (annotation.isSubtypeOf(Reply)) then: {
          TReply;
        } else: {
          raise: XIllegalAnnotation.new(
            "Illegal commLT annotation: " + annotation);
        }
      }
    };
  };
  def discLTTraitFor(annotation) {
    // LIFETIME = INSTANT | TRANSIENT | SUSTAINABLE
    if: (annotation.isSubtypeOf(Instant)) then: {
      ARImplModule.TInstant;
    } else: {
      if: (annotation.isSubtypeOf(TransientT)) then: {
        ARImplModule.makeTTransient(annotation.period)
      } else: {
        if: (annotation.isSubtypeOf(Sustain)) then: {
          ARImplModule.TSustain;
        } else: {
          raise: XIllegalAnnotation.new(
            "Illegal discLT annotation: " + annotation);
        }
      }
    };
  };
  def createHandler(msg, dfltArityType, dfltLifetimeType, dfltReplyType) {
    def ArityTrait := arityTraitFor((is: msg taggedAs: MsgArity).ifTrue: {
      Util.getAnnotationOfType(msg, MsgArity);
    } ifFalse: { dfltArityType });

    def CommLTTrait;
    def DiscLTTrait;

    // EXPIRABLE => LIFETIME = TRANSIENT AND REPLY = DUE
    if: (is: msg taggedAs: Expirable) then: {
      def ann := Util.getAnnotationOfType(msg, Expirable);
      DiscLTTrait := makeTExpires(ann.timeout);
      CommLTTrait := nil;
    } else: {
      DiscLTTrait := discLTTraitFor(
        (is: msg taggedAs: MsgDiscoveryLifetime).ifTrue: {
          Util.getAnnotationOfType(msg, MsgDiscoveryLifetime);
        } ifFalse: { dfltLifetimeType });
      CommLTTrait := commLTTraitFor(
        (is: msg taggedAs: MsgCommunicationLifetime).ifTrue: {
          Util.getAnnotationOfType(msg, MsgCommunicationLifetime);
        } ifFalse: { dfltReplyType });
    };
    // perform trait composition to construct the handler
    object: {
```

```
    import ArityTrait;
    import CommLTTrait;
    import DiscLTTrait;
  };
};
def TOneway := object: {
  def attachFuture(msg) { [nil,msg] };
  def discoveryStopped(msg) { /* do nothing */ };
};
def TReply := object: {
  def attachFuture(msg) {
    def [fut, res] := self.createFuture(); // dispatch TArity
    [fut, FuturesModule.futurize(msg, fut) ];
  };
  def discoveryStopped(msg) { /* do nothing */ };
};
def makeTDue(initDuePeriod) {
  extend: TReply with: {
    def timeLeftForReply(msg) { initDuePeriod };
    def discoveryStopped(msg) {
      when: self.timeLeftForReply(msg) elapsed: {
        (msg.handle.future)<-becomeResolved()
                                @[OneWayMessage,MetaMessage];
      };
    };
  };
};
def makeTExpires(expPeriod) {
  object: {
    import ARImplModule.makeTTransient(expPeriod);
    import makeTDue(expPeriod) exclude timeLeftForReply;

    def timeLeftForReply(msg) {
      self.transientPeriod - (now() - msg.handle.sendTime);
    };
  }
};

deftype AmbientReference;

def makeAmbientReference(typetagOrProtocol,
                         filter,
                         defaultArity,
                         defaultLifetime,
                         defaultReply) {

  def extendedMirror;
  def commonMirror := mirror: {
    import TEventualRef;

    // == provide required methods for TEventualRef trait ==

    def intercept(message) {
      def handler := createHandler(
```

```
        message,defaultArity,defaultLifetime,defaultReply);

      def [fut, newMsg] := handler.attachFuture(message);
      def registration;
      def handle := object: {
        def future := fut;
        def sendTime := now();
        def cancel() { registration.cancel() };
      };
      registration :=
        handler.dispatch(extendedMirror,
                         extendWithHandle(newMsg, handle));
      handle // value of ambient message send is always a handle
    };

    def toString() {
      "ambient:"+typetagOrProtocol.typeName;
    };

    def transportStrategy :=
      { |ARImplModuleCapsule,typetagOrProtocol,filter,defaultArity,
         defaultLifetime,defaultReply|
        // when arriving at the remote host, create a new local AR
        /.at.lang.ambientrefs(ARImplModuleCapsule).ambient:
                                typetagOrProtocol
                                where: filter
                                withArity: defaultArity
                                withLifetime: defaultLifetime
                                withReply: defaultReply };
  }; // end mirror
  extendedMirror := ARImplModule.extendAmbientReference(
    commonMirror, typetagOrProtocol, filter);
  object: {
  } taggedAs: [ AmbientReference ] mirroredBy: extendedMirror;
}; // end makeAmbientReference

def dfltFilter := script: { |o| true } carrying: [];
def dfltArity := One;
def dfltLifetime := Sustain;
def dfltReply := Oneway;

// the public interface of the AR language module
def PublicInterface := object: {
  def IllegalAnnotation := IllegalAnnotation;

  def One := One;
  def All := All;
  def Instant := Instant;
  def Transient := &Transient;
  def Sustain := Sustain;
  def Reply := Reply;
  def Due := &Due;
  def Oneway := Oneway;
  def Expires := &Expires;
```

```
// allow exporting an object together with a set of properties
def export: serviceObject as: typetagOrProtocol with: closure {
  def attributes := isolate: closure;
  ARImplModule.exportServiceObject(
    serviceObject, typetagOrProtocol, attributes);
};
// override export:as: to include a property object
def export: serviceObject as: type {
  export: serviceObject as: type with: {};
};
def export: svcObject {
  export: svcObject as: (ProtocolM.protocolOf: svcObject) with: {};
};
def export: svcObject with: clo {
  export: svcObject as: (ProtocolM.protocolOf: svcObject) with: clo;
};
// def [tFuture, discoverySubscription] := discover: Type
def discover: T {
  def [fut,res] := FuturesModule.makeFuture();
  def sub := when: T discovered: { |t|
    res.resolve(t);
  };
  [fut,sub];
};
// def [tFuture, discoverySubscription] :=
//   discover: Type where: { |t| filter(t) }
def discover: T where: filter {
  def [arFuture, arResolver] := FuturesModule.makeFuture();
  def sub := DiscoveryModule.whenever: T discovered: { |remoteRef|
    def props := isolate: { nil };
    if: (is: remoteRef taggedAs: PropertyObject) then: {
      remoteRef := remoteRef.service;
      props := remoteRef.properties;
    };
    if: (filter(props)) then: {
      sub.cancel();
      arResolver.resolve(remoteRef)
    };
  };
  [arFuture, sub];
};
def ambient: T {
  makeAmbientReference(
    T, dfltFilter, dfltArity, dfltLifetime, dfltReply);
};
def ambient: T withArity: A {
  makeAmbientReference(T, dfltFilter, A, dfltLifetime, dfltReply);
};
def ambient: T withLifetime: L {
  makeAmbientReference(T, dfltFilter, dfltArity, L, dfltReply);
};
def ambient: T withArity: A withLifetime: L {
  makeAmbientReference(T, dfltFilter, A, L, dfltReply);
```

```
    };
    def ambient: T withReply: R {
      makeAmbientReference(T, dfltFilter, dfltArity, dfltLifetime, R);
    };
    def ambient: T withArity: A withReply: R {
      makeAmbientReference(T, dfltFilter, A, dfltLifetime, R);
    };
    def ambient: T withLifetime: L withReply: R {
      makeAmbientReference(T, dfltFilter, dfltArity, L, R);
    };
    def ambient: T withArity: A withLifetime: L withReply: R {
      makeAmbientReference(T, dfltFilter, A, L, R);
    };
    def ambient: T where: filter {
      makeAmbientReference(T, filter, dfltArity, dfltLifetime, dfltReply);
    };
    def ambient: T where: filter withArity: A {
      makeAmbientReference(T, filter, A, dfltLifetime, dfltReply);
    };
    def ambient: T where: filter withLifetime: L {
      makeAmbientReference(T, filter, dfltArity, L, dfltReply);
    };
    def ambient: T where: filter withReply: R {
      makeAmbientReference(T, filter, dfltArity, dfltLifetime, R);
    };
    def ambient: T where: filter withArity: A withLifetime: L withReply:R{
      makeAmbientReference(T, filter, A, L, R);
    };
    def snapshot: ref {
      ref<-makeSnapshot()@[FuturesModule.FutureMessage,MetaMessage];
    };
    def snapshot: ref after: period {
      def [f,r] := FuturesModule.makeFuture();
      when: period elapsed: {
        r.resolve(snapshot: ref);
      };
      f;
    };
  }; // end anonymous module object
}; // end AmbientRefsModule function
```

A.2 Custom Eventual References Module

File `at/lang/firstclassrefs.at`

```
/* This module implements the TEventualRef trait which factors out the
 * common behaviour of first-class eventual reference objects in AT.
 */
def XIllegalOperation := /.at.exceptions.XIllegalOperation;
def Vector := /.at.collections.vector.Vector;

def FirstClassRefModule := object: {
```

```
// an annotation identifying messages destined for the mirror itself
deftype MetaMessage <: /.at.types.Message;

// sentinel used to end the asynchronous delegation chain
// between mirrors on eventual refs
def NilEventualRef := object: { };
/**
 * This trait is meant to be mixed into a mirror definition
 * to make the mirror behave as an eventual reference.
 *
 * Trait requires:
 *  // a closure describing how to pass the reference
 *  def transportStrategy;
 *  // a method defining how to intercept base-level messages
 *  def intercept(msg);
 *  // a method defining how to print the ref
 *  def toString();
 *
 * Optionally, a composite may redefine the 'delegate' field which
 * specifies the object to which metamessages should be forwarded if
 * they are not understood by the eventual ref itself
 */
def TEventualRef := object: {
  // disallow synchronous access to the object
  def invoke(slf, sel, args) {
    // the only legal operation on references is '=='
    if: (`(==) == sel) then: {
      // two eventual refs are equal iff their mirrors are equal
      self == (reflect: args[1])
    } else: {
      raise: XIllegalOperation.new("Cannot synchronously invoke " +
        sel + " on " + self.toString());
    }
  };
  def invokeField(slf, sel) { self.invoke(slf, sel, []) };

  def receive(msg) {
    if: (is: msg taggedAs: MetaMessage) then: {
      // check whether the *mirror* itself can respond to the selector
      if: ((reflect: self).respondsTo(msg.selector)) then: {
        self <+ msg; // process meta-message myself
      } else: {
        // forward meta-message to delegate if not understood
        self.delegate<-receive(msg)@MetaMessage;
      };
    } else: {
      self.intercept(msg); // provided by composite
    }
  };
  def delegate := NilEventualRef;

  // ensure that a reference object does not become a far reference when
  // parameter-passed. Rather, it implements its own by copy semantics
  def pass() {
```

```
      /.at.support.util.uponArrivalBecome: self.transportStrategy;
    };
    def print() {
      "<" + self.toString() + ">";
    };
    def isTaggedAs(typeTag) {
      typeTag.isSubtypeOf(/.at.types.FarReference).or: {
        (self.super).isTaggedAs(typeTag)
      }
    };
    def typeTags() {
      (self.super)^typeTags() + [/.at.types.FarReference];
    };
    // disallow meta-level operations on eventual references
    def clone() {
      raise: XIllegalOperation.new(
        "Cannot clone " + self.toString);
    };
    def newInstance(initargs) {
      raise: XIllegalOperation.new(
        "Cannot create new instance of " + self.toString);
    };
    def defineField(sym, obj) {
      raise: XIllegalOperation.new(
        "Cannot define field "+sym+" in " + self.toString);
    };
    def addField(fld) {
      raise: XIllegalOperation.new(
        "Cannot add field "+fld.name+" in " + self.toString);
    };
    def addMethod(mth) {
      raise: XIllegalOperation.new(
        "Cannot add method "+mth.name+" in " + self.toString);
    };
  }; // end TEventualRef
}; // end of FirstClassRefModule
```

A.3 Extensional Implementation Module

File `at/ambient/ar_extensional_impl.at`

```
/* This module implements "extensional" ambient references
 * whose reach is represented as an explicit collection.
 */
def FuturesModule := /.at.lang.futures;
def MFuturesModule := /.at.lang.multifutures;
def Vector := /.at.collections.vector.Vector;
def OneWayMessage := FuturesModule.OneWayMessage;
def DiscoveryModule := /.at.ambient.discovery;
import /.at.support.timer;
import /.at.lang.firstclassrefs;
```

```
deftype DescriptorT;
def DescriptorObject := object: {
  def service;
  def attributes;
  def init(svcObject, attrs) {
    service := svcObject;
    attributes := attrs;
  };
  def ==(other) {
    (does: other respondTo: `service).and: { service == other.service };
  };
} taggedAs: [/.at.types.Isolate, DescriptorT];

def ARExtensionalImpl := object: {
  def capsule := /.at.support.util.uponArrivalBecome: {
    /.at.ambient.ar_extensional_impl
  };
  def extendAmbientReference(arMirror,
                             typetagOrProtocol,
                             filter) {
      extend: arMirror with: {
        def reach := Vector.new();
        def discoveryObservers := Vector.new();
        DiscoveryModule.whenever: typetagOrProtocol
                        discovered: { |descriptor|
          def service := descriptor;
          def attributes := isolate: { };
          if: (is: descriptor taggedAs: DescriptorT) then: {
            service := descriptor.service;
            attributes := descriptor.attributes;
          };

          if: filter(attributes) then: {
            // add the discovered service to the AR's reach
            reach.add(service);
            // notify discovery observers
            discoveryObservers.each: { |o| o<-apply([service]) };
          };
        };
        DiscoveryModule.whenever: typetagOrProtocol lost: { |descriptor|
          def service := descriptor;
          def attributes := isolate: { };
          if: (is: descriptor taggedAs: DescriptorT) then: {
            service := descriptor.service;
            attributes := descriptor.attributes;
          };
          if: reach.contains(service) then: {
            // remove the lost service from the AR's reach
            reach.remove(service);
          };
        };
        def addDiscoveryObserver: obs {
          discoveryObservers.add(obs);
          // return a subscription object
```

```
        object: { def cancel() { discoveryObservers.remove(obs) } };
      };
      def makeSnapshot() { reach.asTable() };
    }
}; // end extendAmbientReference

def TOne := object: {
  def createFuture() { FuturesModule.makeFuture() };
  def send: msg to: ar ifNoneInReach: closure {
    if: (ar.reach.isEmpty) then: closure else: {
      def receiver := ar.reach.random;
      receiver <+ msg;
    };
  };
  def sendAndRegister(ar, msg) {
    def registration := object: { def cancel() { false } };
    self.send: msg to: ar ifNoneInReach: {
      // dispatch TDiscoveryLT
      registration := self.registerMessageUntilReceiverFound(ar, msg);
    };
    registration
  };
};
def TAll := object: {
  def createFuture() { MFuturesModule.makeMultiFuture() };
  def send: msg to: ar ifNoneInReach: closure {
    if: (ar.reach.isEmpty) then: closure else: {
      ar.reach.each: { |receiver| receiver <+ msg };
    };
  };
  def sendAndRegister(ar, msg) {
    self.send: msg to: ar ifNoneInReach: { };
    // this method returns a registration object which is implicitly
    // passed on to sendAndRegister (allows early retraction of the msg)
    self.registerMessage(ar, msg); // dispatch TDiscoveryLT
  };
};
def TInstant := object: {
  def dispatch(ar, msg) {
    self.send: msg to: ar ifNoneInReach: { }; // dispatch TArity
    self.discoveryStopped(msg); // dispatch TCommLT
    // return publication object to cancel message delivery
    object: { def cancel() { false } };
  }
};
def TSustain := object: {
  def dispatch(ar, msg) {
    self.sendAndRegister(ar, msg) }; // dispatch TArity
  def registerMessageUntilReceiverFound(ar, msg) {
    def registration := ar.addDiscoveryObserver: { |rcvr|
      rcvr <+ msg;
      registration.cancel();
    };
    // returns publication that can be used to stop msg delivery
```

```
      def stopDiscovery() {
        self.discoveryStopped(msg) }; // dispatch TCommLT
      object: {
        def cancel() {
          registration.cancel();
          stopDiscovery();
        }
      };
    };
    def registerMessage(ar, msg) {
      def registration := ar.addDiscoveryObserver: { |rcvr| rcvr <+ msg };
      // returns publication that can be used to stop msg delivery
      def stopDiscovery() {
        self.discoveryStopped(msg) }; // dispatch TCommLT
      object: {
        def cancel() {
          registration.cancel();
          stopDiscovery();
        }
      };
    };
  };

  def makeTTransient(initTransientPeriod) {
    def makeAutoExpirable(originalRegistration) {
      def timerRegistration := when: initTransientPeriod elapsed: {
        originalRegistration.cancel();
      };
      object: {
        def cancel() { // signals a premature cancellation
          timerRegistration.cancel(); // stop the timer
          originalRegistration.cancel(); // cancel the registration now
        }
      }
    };
    extend: TSustain with: {
      def transientPeriod := initTransientPeriod;
      // override the methods defined by the sustainable lifetime trait
      def registerMessageUntilReceiverFound(@args) {
        def reg := super^registerMessageUntilReceiverFound(@args);
        makeAutoExpirable(reg);
      };
      def registerMessage(@args) {
        def reg := super^registerMessage(@args);
        makeAutoExpirable(reg);
      };
    };
  };
  def exportServiceObject(serviceObject, typetagOrProtocol, attrs) {
    DiscoveryModule.export: DescriptorObject.new(serviceObject, attrs)
                    as: typetagOrProtocol;
  };
}; // end of ARExtensionalImpl module
```

A.4 Intensional Implementation Module

File at/m2mi/ar_intensional_impl.at

```
/* This module implements "intensional" ambient references
 * whose reach is represented using an M2MI omnihandle.
 */
def FuturesModule := /.at.lang.futures;
def MFuturesModule := /.at.lang.multifutures;
def Vector := /.at.collections.vector.Vector;
def ProtocolM := /.at.lang.structuraltypes;
def OneWayMessage := FuturesModule.OneWayMessage;

import /.at.support.timer;
import /.at.lang.firstclassrefs;
import /.at.m2mi.api;

deftype ReplyHandlerT;
deftype ProtocolAmbientRefType;

def RECALL_PERIOD := seconds(5);
def DISCOVERY_RATE := seconds(2);
def ANYCAST_TIMEOUT := millisec(500);

def min(x,y) { (x < y).ifTrue: { x } ifFalse: { y } };

def makeLeasedIdEntry(id,originalTTL) {
  object: {
    def timeToLive := originalTTL;
    def id := id;
    def lapse(duration) { timeToLive := timeToLive - duration };
    def renew(duration) {
      timeToLive := min(originalTTL, timeToLive + duration) };
    def expired() { timeToLive < seconds(0) };
  }
};
// create a table of leased entries
def makeLeasedEntryTable() {
  def entries := Vector.new();
  def leaseRevoker;

  def activateLeaseRevoker() {
    leaseRevoker := whenever: RECALL_PERIOD elapsed: {
      entries.removeAll: { |entry|
        entry.lapse(RECALL_PERIOD);
        entry.expired(); // if true, the entry will be removed
      };
      if: (entries.isEmpty) then: {
        leaseRevoker.cancel(); // all entries deleted, can stop checking
        leaseRevoker := nil;
      };
    };
  };
  object: {
```

```
    def containsEntry(id) {
      entries.contains(id, { |elt,entry| elt == entry.id })
    };
    def addEntry(id, ttl) {
      entries.add(makeLeasedIdEntry(id, ttl));
      if: (leaseRevoker == nil) then: {
        activateLeaseRevoker();
      };
    };
    def renewEntry(id, ttl) {
      def idx := (entries.find: { |entry| entry.id == id });
      entries.at(idx).renew(ttl);
    };
    def deactivateLeaseRevoker() {
      (nil == leaseRevoker).ifFalse: {
        leaseRevoker.cancel();
      };
      entries := nil;
    };
  }
};

deftype AmbientReference;
deftype AmbientMessageHandler;

def generateMessageId(msg) { (print: msg.selector) + now().intValue };

def ARIntensionalImpl := object: {
  def capsule := /.at.support.util.uponArrivalBecome: {
    /.at.m2mi.ar_intensional_impl
  };
  def extendAmbientReference(arMirror, typetagOrProtocol, filter) {
    extend: arMirror with: {
      def reach;
      def scope;

      if: (is: typetagOrProtocol taggedAs: /.at.types.TypeTag) then: {
        // convert type tag directly into Java interface
        reach := omnireference: typetagOrProtocol;
        scope := filter;
      } else: {
        // use a dummy type tag and test the protocol at discovery time
        reach := omnireference: ProtocolAmbientRefType;
        scope := script: { |service|
          (/.at.lang.structuraltypes.does: service.protocol
                                    match: typetagOrProtocol).and: {
          filter(service) };
        } carrying: `[typetagOrProtocol,filter];
      };
      def makeSnapshot() {
        def [fut,res] := FuturesModule.makeFuture();
        def snapshot := [];
        def replyhandler := unireference: ReplyHandlerT for: (object: {
          def reply(receiver) { snapshot := snapshot + [receiver] };
```

```
      });

      self.performAnycast(replyhandler);
      when: ANYCAST_TIMEOUT elapsed: {
        detachUniRef: replyhandler;
        res.resolve(snapshot);
      };
      fut;
    };
    def performAnycast(replyHandler) {
      reach<-anycast(scope, replyHandler);
    };
    def performBroadcast(msg, id, ttl) {
      reach<-broadcast(scope, msg, id, ttl);
    };
  }
}; // end extendAmbientReference

def exportServiceObject(obj, typetagOrProtocol, attributes) {
  def attributes.protocol := ProtocolM.protocolOf: obj;
  def alreadyReceivedMessages := makeLeasedEntryTable();
  def unicastMessageHandler :=
    unireference: AmbientMessageHandler for: obj;
  def broadcastMessageHandler := object: {
    def anycast(scope, replyHandler) {
      if: scope(attributes) then: {
        replyHandler<-reply(unicastMessageHandler);
      }
    };
    def broadcast(scope, msg, id, ttl) {
      if: scope(attributes) then: {
        if: !alreadyReceivedMessages.containsEntry(id) then: {
          if: !(ttl == 0) then: {
            alreadyReceivedMessages.addEntry(id, ttl);
          };
          obj <+ msg; // deliver the message
        } else: {
          // msg already previously received, update lease time
          // to make sure the object does not receive it twice
          alreadyReceivedMessages.renewEntry(id, ttl);
        }
      }
    };
  };
  def pub := export: broadcastMessageHandler asTypeTag:
    ((is: typetagOrProtocol taggedAs: /.at.types.TypeTag).ifTrue: {
      typetagOrProtocol;
    } ifFalse: {
      ProtocolAmbientRefType;
    });
  object: {
    def unexport() {
      pub.unexport();
      detachUniRef: unicastMessageHandler;
```

```
      alreadyReceivedMessages.deactivateLeaseRevoker();
    }
  }
};

def TInstant := object: {
  def timeLeft(forMsg) { seconds(0) };
  def stillValid(msg) { false };
};
def makeTTransient(timeout) {
  object: {
    def transientPeriod := timeout;
    def timeLeft(msg) { (msg.handle.sendTime + timeout) - now() };
    def stillValid(msg) { self.timeLeft(msg) > seconds(0) };
  }
};
def TSustain := object: {
  // RECALL_PERIOD is max amount of time receiver wants to recall msg
  def timeLeft(forMsg) { RECALL_PERIOD };
  def stillValid(msg) { true };
};
def TAll := object: {
  def createFuture() { MFuturesModule.makeMultiFuture() };
  def dispatch(ar, msg) {
    def continuation;
    def id := generateMessageId(msg);
    def sendOnce() {
      def ttl := self.timeLeft(msg); // dispatch to DiscLT

      ar.performBroadcast(msg, id, ttl);
      continuation := when: BROADCAST_RATE elapsed: {
        if: self.stillValid(msg) then: { // dispatch TDiscoveryLT
          sendOnce(); // recursive call to start new broadcast
        } else: { // stop broadcasting
          self.discoveryStopped(msg); // dispatch TCommLT
        }
      }
    };
    sendOnce(); // start broadcasting
    def stopDiscovery() {
      continuation.cancel();
      self.discoveryStopped(msg); // DISPATCH to CommLT
    };
    object: { def cancel() {  stopDiscovery() } }
  };
};
def TOne := object: {
  def createFuture() { FuturesModule.makeFuture() };
  def dispatch(ar, msg) {
    def continuation;
    def sendOnce() {
      def receivers := [];
      def replyHandler := unireference: ReplyHandlerT for: (object: {
        def reply(rcvr) { receivers := receivers + [rcvr] };
```

```
        });

        ar.performAnycast(replyHandler);
        continuation := when: ANYCAST_TIMEOUT elapsed: {
          if: !receivers.isEmpty then: {
            receivers[(1 ?? receivers.length).round] <+ msg;
            detachUniRef: replyHandler;
            self.discoveryStopped(msg); // dispatch TCommunicationLT
          } else: {
            continuation := when: BROADCAST_RATE elapsed: {
              if: self.stillValid(msg) then: { // dispatch TDiscoveryLT
                sendOnce(); // recursive call to start new anycast
              } else: { // stop the anycast protocol
                self.discoveryStopped(msg); // dispatch TCommunicationLT
              }
            }
          }
        }
      };
      sendOnce();

      def stopDiscovery() {
        continuation.cancel();
        self.discoveryStopped(msg); // DISPATCH to CommLT
      };
      object: { def cancel() {  stopDiscovery() } }
    };
  };
}; // end of ARIntensionalImpl module
```

Bibliography

[ABC+00] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölszle, John Maloney, Randall Smith, David Ungar, and Mario Wolczko. The SELF 4.1 programmer's reference manual, 2000.

[AC93] Gul Agha and C. J. Callsen. Actorspace: An open distributed programming paradigm. In *Proceedings of the 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, pages 23–32, 1993.

[ADH+98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *Higher Order Symbol. Comput.*, 11(1):7–105, 1998.

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[AH87] Gul Agha and Carl Hewitt. Concurrent programming using actors. *Object-oriented concurrent programming*, pages 37–53, 1987.

[AR98] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. In *ACM Computing Surveys*, volume 30, pages 330–373, 1998.

[Arn99] Ken Arnold. The jini architecture: Dynamic services in a flexible network. In *36th Annual Conference on Design Automation (DAC'99)*, pages 157–162, 1999.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 2006.

[Ben86] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[BH77] Henry G. Baker Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages*, volume 8 of *ACM Sigplan Notices*, pages 55–59, 1977.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86. ACM Press, 1986.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[Bri88] J.-P. Briot. From objects to actors: study of a limited symbiosis in smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 69–72, New York, NY, USA, 1988. ACM Press.

[Bro95] Frederick Brooks Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–343, 2004.

[CA94] C. J. Callsen and G. Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, 1994.

[Car89] Denis Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–18, November–December 1989.

[Car93] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 286–297. ACM Press, 1995.

[Car99] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.

[CBM+02] Licia Capra, Gordon S. Blair, Cecilia Mascolo, Wolfgang Emmerich, and Paul Grace. Exploiting reflection in mobile computing middleware. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):34–44, 2002.

[CD96] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *The VLDB Journal*, pages 3–14, 1996.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th edition)*. Addison-Wesley, 2005.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, Berlin Germany, 1998.

[CJ02] Gianpaolo Cugola and H.-Arno Jacobsen. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):25–33, 2002.

[CJF01] Harry Chen, Anupam Joshi, and Timothy Finin. Dynamic Service Discovery for Mobile Computing: Intelligent Agents meet Jini in the Aether. *Cluster Computing*, 4(4):343–354, Oct 2001.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, Jul/Aug 2000.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[DCYAD05] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 299–318, 2005.

[DD03] Theo D'Hondt and Wolfgang De Meuter. Of first-class methods and dynamic scope. *RSTI - L'objet no. 9/ 2003. LMO 2003*, pages 137–149, 2003.

[DDD04] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Pico: Scheme for mere mortals. In Jacques Malenfant and Bjarte M. Østvold, editors, *ECOOP Workshops*, volume 3344 of *Lecture Notes in Computer Science*. Springer, 2004.

[De 04] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, 2004.

[Ded06] Jessie Dedecker. *Ambient-Oriented Programming*. PhD thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, May 2006.

[DGM+07] Jessie Dedecker, Elisa Gonzalez Boix, Stijn Mostinckx, Stijn Timbermont, Jorge Vallejos, and Tom Van Cutsem. The Ambienttalk/2 Tutorial, 2007. `http://prog.vub.ac.be/amop/at/tutorial/tutorial` (captured in March 2008).

[D'H96] Theo D'Hondt. The pico programming project. 1996. `http://pico.vub.ac.be` (captured in March 2008).

[DTM+05] Wolfgang De Meuter, Eric Tanter, Stijn Mostinckx, Tom Van Cutsem, and Jessie Dedecker. Flexible object encapsulation for Ambient-Oriented Programming. In *Dynamic Languages Symposium at OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.

[DV04] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. In L. Yang, M. Guo, J. Gao, and N. Jha, editors, *International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, August 2004.

[DVM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.

[DVM+06a] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[DVM+06b] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, , Wolfgang De Meuter, and Theo D'Hondt. Ambienttalk: Language support for mobile computing. In *International Workshop on System Support for Future Mobile Computing Applicatons (FUMCA)*. IEEE Computer Society Press, 2006.

[EAC98] S.O. Ehmety, I. Attali, and D. Caromel. About the automatic continuations in the Eiffel// model. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 219–225, 1998.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[EG01] Patrick Th. Eugster and Rachid Guerraoui. Content-based publish/subscribe with structural reflection. In *COOTS'01: Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems*, pages 10–10, Berkeley, CA, USA, 2001. USENIX Association.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 254–269, New York, NY, USA, 2001. ACM Press.

[EGH05] Patrick Eugster, Benoît Garbinato, and Adrian Holzer. Location-based publish/subscribe. *Fourth IEEE International Symposium on Network Computing and Applications*, pages 279–282, 2005.

[EGH06] Patrick Eugster, Benoît Garbinato, and Adrian Holzer. Pervaho: A development & test platform for mobile ad hoc applications. In *Third annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, pages 1–5, July 2006.

[EGS00] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 252–276, London, UK, 2000. Springer-Verlag.

[Esp95] David Espinosa. *Semantic Lego*. PhD thesis, Department of Computer Science, Columbia University, 1995.

[Eug07] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.

[Fet05] Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., October 2005.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 348–355, New York, NY, USA, 1984. ACM.

[GC89] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[GDL+04] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.

[GF99] Rachid Guerraoui and Mohammed E. Fayad. OO Distributed Programming is *Not* Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, 1999.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification third edition*. Addison-Wesley Professional, 2005.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[GR03] B. Garbinato and P. Rupp. From ad hoc networks to ad hoc applications. In *Proceedings of the 7th International Conference on Telecommunications*, pages 145–149, 2003.

[Gro03] IST Advisory Group. Ambient intelligence: from vision to reality, September 2003.

[GVDD07] Elisa Gonzalez Boix, Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Language support for leasing in mobile ad hoc networks. Technical Report VUB-PROG-TR-07-08, Vrije Universiteit Brussel, 2007.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Computer Languages, Systems & Structures*, 32(2-3):109–124, 2006.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

[Hoa73] C. A. R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford University, 1973.

[Hoh06] Gregor Hohpe. Programming without a call stack – event-driven architectures, 2006. `www.enterpriseintegrationpatterns.com/docs/EDA.pdf` (captured in March 2008).

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523, 1993.

[HRBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, 2008.

[Int05] International Telecommunication Union. *ITU Internet Reports 2005: The Internet of Things, 7th edition*. Geneva, Switzerland, 2005.

[JdT+95] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 156–171, Colorado, December 1995.

[Jef85] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.

[KB02] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Onward! Track*, New York, NY, USA, 2002. ACM Press.

[KC03] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 1(36):41–50, 2003.

[KK04] Wooyoung Kim and Alan H. Karp. Customizable description and dynamic discovery for web services. In *EC '04: Proceedings of the 5th ACM conference on Electronic commerce*, pages 142–151, New York, NY, USA, 2004. ACM.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KP96] Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. Tutorial slides and notes, Software Design Area, Xerox Corporation, 1996. `http://www.parc.xerox.com/csl/groups/sda/publications`.

[KRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[KS90] Kenneth Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint (logic) programming. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 57–66, New York, NY, USA, 1990. ACM Press.

[KSMA06] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1297–1300, New York, NY, USA, 2006. ACM.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 126–134, New York, NY, USA, 1990. ACM.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.

[Lie87] Henry Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications Of The ACM*, 31(3):300–312, 1988.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto, CA, USA, February 1997.

[LM07] Ralf Lämmel and Eric Meijer. Revealing the X/O impedance mismatch (Changing lead into gold). In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, LNCS. Springer-Verlag, 06 June 2007.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[Mac86] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Oxford University Press, USA, December 1986.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[Mai04] Christian Maihöfer. A survey of geocast routing protocols. *IEEE Communications Surveys & Tutorials*, 6(2):32–42, 2004.

[Mar06] Pierre Martin. A reflective approach to building extensible distributed actor languages. Master's thesis, Vrije Universiteit Brussel, 2006.

[MC03] René Meier and Vinny Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, 2003.

[McA95] Jeff McAffer. Meta-level programming with coda. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214, London, UK, 1995. Springer-Verlag.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag New York, Inc., 2002.

[McG00] Robert E. McGrath. Discovery and its discontents: Discovery protocols for ubiquitous computing. Technical Report UIUCDCS-R-99-2132, Department of Computer Science University of Illinois Urbana-Champaign, 2000.

[MCNC05] René Meier, Vinny Cahill, Andronikos Nedos, and Siobhán Clarke. Proximity-based service discovery in mobile ad hoc networks. In *Distributed Applications and Interoperable Systems*, pages 115–129. Springer, 2005.

[Mei02] René Meier. Communication paradigms for mobile computing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):56–58, 2002.

[Mey93] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, March 2000.

[MHS02] R. Sharma J. Fialli M. Hapner, R. Burridge and K. Stout. Java message service specification version 1.1. Technical report, Sun Microsystems, Inc., 2002.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[Mil06] Mark Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, John Hopkins University, Baltimore, Maryland, USA, May 2006.

[MK88] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. *SIGPLAN Not. Special issue: 'OOPSLA 88 Conference Proceedings*, 23(11):276–284, 1988.

[MMF00] Mark Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proceedings of Financial Cryptography*. springer-Verlag, 2000.

[MMH05] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. Emma: Epidemic messaging middleware for ad hoc networks. *Personal Ubiquitous Comput.*, 10(1):28–36, 2005.

[MMY96] H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In *Proceedings of Reflection Symposium '96*, pages 79–91, April 1996.

[MPR01] A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

[MRV98] A. L. Murphy, G.-C. Roman, and G. Varghese. An exercise in formal reasoning about mobile communications. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 25, Washington, DC, USA, 1998. IEEE Computer Society.

[MS03] Mark Miller and Jonathan Shapiro. Paradigm regained : Abstraction mechanisms for access control. In *Proceedings of Eighth Asian Computing Science Conference*, December 2003.

[MTS05] Mark Miller, E. Dean. Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in E as plan coordination. In R. De Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[MVTT07] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the Dynamic Languages Symposium - OOPSLA'07: Companion of the 22st annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications.*, pages 222–248. ACM Press, 2007.

[MWN02] Scott McLean, Kim Williams, and James Naftel. *Microsoft .Net Remoting*. Microsoft Press, Redmond, WA, USA, 2002.

[MYS03] Mark Miller, Ka Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, Combex, Inc., 2003.

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 263, Washington, DC, USA, 2004. IEEE Computer Society.

[Nid01] Michael Nidd. Service discovery in DEAPspace. *IEEE Pers. Commun.*, 8(4):39–45, Aug 2001.

[NKSI05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 249–260, New York, NY, USA, 2005. ACM.

[Obj02] Object Management Group. Common Object Request Broker Architecture: Core specification, 2002. `http://www.omg.org`.

[OIT92] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the Workshop on New Models for Software Architecture*, November 1992.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes), 1996. Presentation given at the 1996 Usenix Annual Technical Conference, January 1996. `http://www.softpanorama.org/People/Ousterhout/Threads` (captured in March 2008).

[Pae93] Andreas Paepcke. User-level language crafting: Introducing the CLOS metaobject protocol. In *Object-oriented programming: the CLOS perspective*, pages 65–99. MIT Press, Cambridge, MA, USA, 1993.

[Per82] Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, Sept. 1982.

[PMS07] Aleš Plšek, Philippe Merle, and Lionel Seinturier. Ambient-oriented programming in fractal. In *3rd Intl. Workshop on Object Technology for Ambient Intelligence and Pervasive Systems (OT4AmI), co-located with ECOOP '07*, 2007.

[Qui07] John Quigley. The white programming language, 2007. CODE Group, Illinois Institute of Technology. `http://dijkstra.cs.iit.edu/code/white`.

[Ram08] Victor Ramiro. Proximity is in the eye of the beholder: a conceptual framework. Master's thesis, Universidad de Chile, Santiago de Chile, 2008. To Appear.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.

[SG01] U. Saif and D.J. Greaves. Communication primitives for ubiquitous systems or rpc considered harmful. pages 240–245, 2001.

[SHM07] David Svensson, Gorel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services. In *IEEE International Conference on Pervasive Services*, pages 301–307, July 2007.

[Smi84] Brian Cantwell Smith. Reflection and semantics in LISP. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, New York, NY, USA, 1984. ACM Press.

[SS78] Guy L Steele and Gerald J Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). Technical report, Cambridge, MA, USA, 1978.

[Ste94a] Guy L. Steele, Jr. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[Ste94b] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, 1994.

[Sun98] Sun Microsystems. Java remote method invocation specification, 1998. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html` (captured in March 2008).

[TMHK95] E. Dean Tribble, Mark S. Miller, Norm Hardy, and David Krieger. Joule: Distributed application foundations. Technical Report ADd03.4P, Agorics Inc., December 1995. `www.agorics.com/Library/joule.html`.

[TMY94] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pages 275–292. American Mathematical Society, 1994.

[TPST98] D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer. The case for non-transparent replication: Examples from Bayou. *IEEE Data Engineering Bulletin*, 21(4):12–20, december 1998.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, 1991.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[Van06] Tom Van Cutsem. A Modular Mixin-based Implementation of Ambient References. Technical Report VUB-PROG-TR-06-07, Vrije Universiteit Brussel, 2006.

[VDD07] Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Object-oriented coordination in mobile ad hoc networks. In *9th International Conference on Coordination Models and Languages (COORDINATION)*, volume 4467 of *Lecture Notes in Computer Science*, pages 231–248, Heidelberg, June 2007. Springer-Verlag.

[VDM+06] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez, Theo D'Hondt, and Wolfgang De Meuter. Ambient references: addressing objects in mobile networks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 986–997, New York, NY, USA, 2006. ACM Press.

[VDMD05] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, and Wolfgang De Meuter. Abstractions for Context-aware Object References. In *2nd Workshop on Building Software for Pervasive Computing, OOPSLA 05*, 2005.

[VMD07] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between actors and threads. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, ACM Digital Library, August 2007.

[VMD08] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic Symbiosis between Actors and Threads. *Computer Languages, Systems and Structures*, 1(35), September 2008. To Appear.

[VVG+07] Jorge Vallejos Vargas, Tom Van Cutsem, Elisa Gonzalez Boix, Stijn Mostinckx, and Wolfgang De Meuter. The message-oriented mobility model. *Journal of Object Technology. Special Issue: TOOLS EUROPE 2007*, 6(9):363–382, October 2007.

[Wal99] Jim Waldo. The Jini Architecture for Network-centric Computing. *Commun. ACM*, 42(7):76–82, 1999.

[Wal01] Jim Waldo. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 9, 2001.

[WBB06] Torben Weis, Christian Becker, and Alexander Brändle. Towards a programming paradigm for pervasive applications based on the ambient calculus. In *International Workshop on Combining Theory and Systems Building in Pervasive Computing (CTSB), co-located with Pervasive 2006*, 2006.

[WD05] Marcel Weiher and Stéphane Ducasse. Higher order messaging. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 23–34, New York, NY, USA, 2005. ACM.

[Wei91] Mark Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.

[WWWK96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1996.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 306–315. ACM Press, 1988.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. MIT Press, 1990.

[Zad65] L.A. Zadeh. Fuzzy sets. *Information and Control*, (8):338–353, 1965.

Index

ABCL, 35, 94
Active Object, 19, 23
ActorNet, 37
Actors, 37, 93
    in AmbientTalk, 66, 67
ActorSpace, 38, 179
Ad hoc
    Application, 16
    Network, 14
`All`, 140
Ambient Acquaintance Management, 18, 21, 34
Ambient Actor Model, 19, 57
Ambient Bindings, 239
Ambient Message, 130
Ambient References, 128
    in AmbientTalk/1, 22
Ambient-oriented Programming, 13, 17
AmbientTalk/1, 19
Annotations, 78
Anonymous
    Communication, 31
    Service, 18
Anonymous Far References, 156
Argus, 36, 94
Arity, 32
    of Ambient References, 131, 138
    One-to-many, 132
    Point-to-point, 131
Arity Decoupling, 32, 120, 167
Aspect-oriented Programming, 241
Asynchronous Message, 18
At Least Once, 152
At Most Once, 152
Attribute Object, 137
Attributes, 137

Best Effort, 152
Block Closures, 60

Call Stack, 79
Callbacks, 70
Capabilities, 6, 88
Cardinality, 173
Carrier, 130
Client-server, 18
Collocation, 16
Communication, 27
    Decoupled, 30
    history, 24
    trace, 18, 20
Communication Lifetime, 131, 140
    Bounded two-way, 132
    One-way, 132
    Unbounded two-way, 132
Communication Range, 129
Conceptual Integrity, 58
Conditional Synchronisation
    in AmbientTalk, 73
Connection-oriented, 52, 155, 169
Containment, 19, 23
CORBA, 35

Deadlock, 18, 66
Decoupling
    in arity, 32, 49
    in space, 31, 49, 165
    in synchronisation, 32, 49, 166
    in time, 31, 49
Deep Copy, 23
Deferrables, 94
Delegation, 59, 64, 89
Delivery Handles, 127, 142
Delivery Policies, 130
Designation, 169
Disconnected Operation, 41
Discovery, 21, 27
    Decentralised, 29, 49, 163
    in Jini, 41
Discovery Event Handler, 76
Discovery Lifetime, 131, 144
    Instant, 131

Sustained, 131, 146
Transient, 131, 145
Distributed Asynchronous Collections, 183
Distributed Naming, 31
Domain-specific Language, 5
Duck Test, 135
`Due`, 78, 143
Dynamic Inheritance, 59, 63

E, 36, 93
Elasticity, 172
Embedding, 113
Emerald, 35
EMMA, 43
Encapsulation, 62
Epidemic Routing, 44
Erlang, 36
Event Loops, 6, 53, 58, 66
Event-driven Computation, 18
Eventual References, 72
Exactly Once, 152
Exceptions
in AmbientTalk, 71
Exclusive State Access, 67
Expirable Ambient Messages, 148
`Expires`, 148
Exporting Objects, 76, 82

Failure Event Handler, 75
Failure Handling, 28, 85
Connection-independent, 33, 50, 167
in WAN Languages, 36
Failure handling
in AmbientTalk, 75, 78
Far References, 67, 72, 75
Anonymous, 156
Filters, 135
Fractal, 239
Futures, 22, 70, 94
and Anonymous Far References, 158
of Ambient Messages, 141
ruining, 71
Fuzzy Sets, 242

Garbage Collection, 24, 215
Distributed, 24, 86, 92
Geocast, 240

Handover, 118
Higher-order Messages, 98

Idempotent, 154
Impedance Mismatch, 6
Object-event, 50
`Instant`, 144
Instantiation, 59
Intercession, 102
Inversion of Control, 54, 79
Isolates, 69, 88

J2ME, 90
Janus, 36
Java
JMS, 43
RMI, 35
Jini, 41, 94

Language Mixin, 22, 25
Leasing, 85, 87, 94
in EMMA, 43
in Jini, 42
LIME, 39
Linda, 39
Linguistic Symbiosis, 108
Listeners, 79
Little Language, 5
Lookup Service, 29
LPS, 42, 95

M2MI, 44, 178, 217
as Implementation Library, 209
Mail Address, 37, 177
Mailbox, 20, 21, 37, 92
MARS, 40
Marshalling, 214
Message Handler, 186
`MetaMessage`, 104
Metaobject Protocol, 21
Mirages, 102
Mirrors, 99
Mobile Ad Hoc Network, 14
Mobile Ambients, 38
Multifutures, 142, 149
Multilisp, 94
Multireferences, 159
Musical Match Maker, 28, 81

Nomadic Network, 16, 40
Nominal Type, 65
Non-blocking Communication, 18, 34, 66

Object Nesting, 62

Obliq, 35
`One`, 139
One-to-many Messages, 139
One-way Messages, 140
One.world, 45, 165, 180
`Oneway`, 141
Open Implementations, 238
Ownership by actors, 67
Oz, 36

Paradigm, 17
Paradigm Leak, 6
Passive Object, 19, 23
Point-to-point Messages, 138
Potential Receivers, 130
ProActive, 94, 240
Promise Pipelining, 94
Promises, 36, 70, 94
Protocols, 134
Provisional Services, 124
Proximity, 240
    in STEAM, 43
Publish/Subscribe, 42, 95

QRPC, 41
Quasi-quoting, 100

Reach, 129
Recall Period, 153
Referential Transparency, 6
Reflection, 20, 99
    Inter-language, 109
Reification, 20
Remote Object Reference, 20, 52
Remote Procedure Call, 35
Replication, 22
`Reply`, 141
Request/reply, 53
RFID, 14, 124
Roaming, 117, 139
Rover, 41
RPC, 35
    Queued, 41

Salsa, 36, 240
Scope
    Extensional Implementation, 188
    Intensional Implementation, 198
    Lexical, 61
    Object, 61
    of Ambient References, 129, 133
Scoping
    Dynamic, 25
    Spatial, 43
Security, 88
Serial Execution, 66
Service Discovery
    and Security, 90
    in AmbientTalk, 76, 83, 87
    in AmbientTalk/1, 24
    versus Service Lookup, 29
Service Objects, 136
Session Types, 242
Snapshots, 159
Space Decoupling, 31
SpatialViews, 37
Stateful, 52, 169
STEAM, 43
    Proximities, 88
Stratification, 25, 108
`Sustain`, 146
Synchronisation, 28
Synchronisation Decoupling, 32

Tag, 21
Temporal Scope, 144
Time Decoupling, 31
Timeouts, 78
TOTA, 40
Traits, 63
Transaction, 18
Transient
    Disconnection, 15
`Transient`, 145
Tuple Spaces, 39, 176
Twisted, 94
Two-way Messages, 141
Type Tags, 65, 76

Ubiquitous Computing, 1
Uniform Access Principle, 59

Vats, 66
Volatile Connections, 14, 31, 80

Wait by necessity, 70
`when:becomes:`, 70
White, 241
Wireless Sensor Network, 36

Zero Infrastructure, 15, 81