A Stream Redirection Architecture for Pervasive Computing Environments

by

Jorge Rafael Nogueras

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by . . . .
Hari Balakrishnan
Assistant Professor
Thesis Supervisor

Certified by . . . .
Stephen J. Garland
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Stream Redirection Architecture for Pervasive Computing Environments

by

Jorge Rafael Nogueras

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

We describe a framework for redirecting data streams to devices best equipped to handle them as users move around in a building. This is a useful capability for emerging pervasive computing environments such as MIT's Project Oxygen, as it allows a mobile user with a handheld device to easily control and benefit from specialized devices (e.g., speakers, large displays, etc.) available at various locations. For instance, as a user moves around, this system makes it possible for a sound or video stream to "follow" her, with the stream being played at whichever best-equipped, available output device is nearest to her at any point in time. The key challenges in building this system involve discovering resources identified by their location and in developing an architecture that achieves seamless stream redirection. We describe how our design and implementation meets these challenges.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Scientist

Acknowledgements

I would like to thank the following people who have helped me with my project and my thesis: Kalpak Kothari (for writing the Swing GUI for the prototype application and helping me with all things iPAQ), Bodhi Priyantha and Allen Miu (for taking the time to help me with Cricket), everyone in the Networks and Mobile Systems group, my family and friends (who have supported me in many ways), and of course my professors, Hari Balakrishan and Stephen Garland, for giving me their time, their expertise, and their keen eye for grammatical and typographical errors. May I make all of them proud.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

With every passing year, we find that smaller devices with greater computational power are becoming increasingly popular. It is now common to see people walking around with handheld computing devices that are small enough to carry everywhere. Chips performing computation and communication are being embedded into all sorts of appliances, enabling them to perform sophisticated operations and empowering them with network connectivity. Terms such as "pervasive computing" [5] have been coined to describe this tendency to integrate computing and communication into our daily lives.

An even more futuristic view is held by MIT's Project Oxygen [16], which believes that in the future, computation will be freely available everywhere, "like oxygen in the air we breathe." We will not need to carry personalized devices around with us. Instead, "anonymous" devices, either handheld or embedded in the environment, will bring computation to us, no matter where we are. These devices will personalize themselves in our presence by finding whatever information and software we need. We will not need to type or click; instead, we will communicate naturally, using speech, vision, and phrases that describe what we really want to do, leaving it to the computer to locate appropriate resources and carry out our intent.

No matter how useful the ubiquity of mobile devices may be, however, their practical utility is curtailed by the following two facts: first, small devices generally lack many features that larger (but fixed) devices boast (for example, a small handheld personal digital assistant (PDA) is usually not equipped with high-quality speakers or a screen large enough for conveniently viewing large documents or digital video). Second, software running on today's mobile devices generally lack one very important piece of information: *context*. Most of these devices have no concept of their environment, such as their location, the identity of the person using them, the existence of nearby devices that may be used to complement or augment their capabilities, etc. We feel that knowledge of environmental context will enable interesting new applications.

If we could somehow leverage the capabilities of larger devices while still benefitting from the practicality and portability of handheld devices, we could provide the user with a computing environment in which information is seamlessly directed to the device best-equipped to handle it.

We can thus conceive of a "coalition" of devices in which context-aware devices discover one another and make use of each other's capabilities to enhance the user's productivity. Imagine, for instance, walking into a room with a handheld device that lacks speakers: you notice, however, that there are computers with speakers in the room. In the current paradigm, you would simply have to forgo the use of your handheld device and somehow gain access to one of these computers, retrieve the desired media file and play it there. If you move to another room, the process would have to be repeated, and if the audio file was only played in part, you would have to manually advance the time counter to continue playing it from where it left off.

We see that to benefit from the resources of a fixed computing device, the user was forced to simply stop using the mobile handheld device. When the user moved from one room to another, she was unable to benefit from the handheld's inherent portability.

This thesis describes a new paradigm, based on a generic stream redirection framework.[1] Our framework describes several components that work together to allow a

[1] A *stream* is simply a sequence of bytes; when a file is being transferred from the server where it

user to use a handheld device to play media files using the resources of fixed devices in her surroundings.

1.2 Goals

Ideally, we would like to attain a balance between the better or more suitable resources usually offered by fixed devices with the portability and convenience of handheld devices. It is by achieving this synergism that we can truly benefit from both computational paradigms.

Specifically, we are interested in the possibility of using room resources for the playing of media files, using the user's handheld device as a "remote control" from where the user can specify which file should be played and where it should be played.

One important consideration is the limited network bandwidth usually available to portable devices. Handhelds usually use a wireless network connection that provides them with less network bandwidth than wired connections.

Also, handheld devices generally have limited permanent storage capacity, and some lack it completely. Media files tend to be as large as several megabytes, so it is unlikely that the handheld device could store many media files at a time.

For these reasons, it makes sense to store the media files not in the handheld device itself, but in some media file server that can be accessed through the wired network. Even if the handheld device *did* have enough permanent storage to maintain a collection of media files, since they are going to be played elsewhere, it does not seem prudent to have to transfer the media files over the wireless link to the fixed device where they will be played.

We can identify several goals that guided us in the design and implementation of our project:

1. First, we wanted to design a framework where handheld devices may be used to discover the existence of other devices based on their physical location, and be

resides to another host through the network, we can view the sequence of the bytes that make up said file as a *stream*.

able to use the resources of these devices to initiate the playing of media files. We were also interested in leveraging the mobility of these handheld devices by having the data stream "follow" the user to other appropriate devices as the user moves from room to room.

2. Another goal of our project was to make our framework platform-independent[2] and flexible.[3] We also wanted to offer a simple object-oriented application programming interface (API) that developers could use to write their own applications.[4]

3. The final goal of our project was to build a working prototype implementation that showcases the features and benefits of our framework.

1.3 System Overview

In this thesis we present and describe in detail an architecture for achieving application-level stream redirection that enables a user to begin playing a media file and have the data stream follow her as she moves from room to room. We also describe a working implementation that showcases our architecture's flexibility.

Our design uses two technologies to achieve the aforementioned requirements: it uses the Cricket location support system (see Section 2.2.1) so that the system can learn of the user's physical location wherever she goes, and it uses the Intentional Naming System (INS) network resource location system (see Section 2.3.1) to find out which speakers or displays are available for use in each room.

The user carries a handheld mobile device with network connectivity wherever she goes. This handheld device has attached to it a piece of hardware called a Cricket listener, which learns of its location by listening to signals coming from Cricket beacons

[2]Our implementation was done in the Java programming language [10, 4], so the framework can be deployed on any system for which a Java Virtual Machine (JVM) exists.

[3]By *flexible* we mean that it should be able to accommodate different file servers and output devices without having to modify any system component other than those in our framework.

[4]These applications may use stream redirection to make any other type of data other than media files follow a user, for instance.

installed in every room. Our software, running on the user's handheld device, learns of the user's location thanks to Cricket and asks INS if there are any available speakers in that room that may be used for playing music. If there are, the user can select which media file she wants to play and it will begin playing in a speaker in the current room.

As the user moves to another room, the software detects this change and asks INS for speakers in the user's new location; as soon as one is found, it instructs the new speaker to begin playing the media file from the point where the other one left off. As far as the user is concerned, the media file has automatically "followed her" from one room to another.

1.4 Contributions

Our main contribution is that our framework and implementation present an application-level and platform-independent solution for dynamic stream redirection. While other schemes for mobility [18, 20] have been devised, they often involve changing the end-points' protocol stacks (either at the network or at the transport level). Having to do this just so that an application will work can range from being an annoyance to being unfeasible (like modifying the protocol stack on a Windows machine). We argue that there is a way of making this application work without such operating system modifications and that this is a desirable property.

Finally, our framework requires that the stream redirection occur not because either of the end hosts has changed its network attachment point, but rather as an asynchronous event at the request of a third party that has moved in a physical (as opposed to network) sense. Having a third party make either end-point migrate a connection would require further modification on the mobility schemes we will discuss in Section 2.1 and could present serious security risks. Our framework, even if it is misused by a malicious user, could not do anything more than redirect a connection started by our own application and could not be used to allow a third party to redirect any connection in the host.

Chapter 2

Related Work

In this chapter we discuss several other projects that have concerned themselves with the concepts of redirecting data dynamically, giving devices a context of location, or network resource location. We will emphasize in particular *Cricket* and the *Intentional Naming System* since we will be using them in our project.

2.1 Mobility

Dynamic host mobility has been the topic of many research projects. We will discuss a few of them and see how they relate to our work.

2.1.1 Mobile IP

Mobile IP [18] is an enhancement to the Internet Protocol (IP) that allows transparent routing of IP datagrams to mobile nodes on the Internet. Each mobile node is always identified by its *home address*, regardless of its current point of attachment to the Internet. While away from its home, a mobile node is also associated with a *"care-of" address*, which provides information about its current point of attachment to the Internet. Mobile IP provides for registering the care-of address with a *home agent*. The home agent sends datagrams destined for the mobile node through a tunnel to the care-of address. After arriving at the end of the tunnel, each datagram is then delivered to the mobile node.

In Mobile IP parlance, there are the following components:

Mobile Node — This is the host that changes its point of attachment from one network to another. A mobile node may change its location without changing its IP address; it may continue to communicate with other nodes at any location using its (unchanging) IP address.

Home Agent — This is a router on a mobile node's home network which tunnels datagrams for delivery to the mobile node when it is away from home, and maintains current location information for the mobile node.

Foreign Agent — This is a router on a mobile node's "visited" network which provides routing services to the mobile node while registered. The foreign agent detunnels and delivers datagrams to the mobile node that were tunnelled by the mobile node's home agent. For datagrams sent by a registered mobile node, the foreign agent may serve as a default router.

A mobile node is given a long-term unchanging IP address on its home network. When away from its home network, a "care-of" address is associated with the mobile node and reflects the mobile node's current point of attachment. The mobile node normally uses its home address as the source address of all IP datagrams that it sends. Thus, this protocol describes a "triangular routing" scheme: packets meant for the mobile node are not sent directly to it, but rather to the mobile node's home agent, which then forwards them to the mobile node in the remote network.

As stated in [17], there are several performance problems in Mobile IP that should be mentioned. First, Mobile IP's tunnelling scheme creates the aforementioned triangle routing problem, causing packets to take a sub-optimal route. Second, packets in transit when a *handoff*[1] occurs are often lost because they are sent to the wrong address, due to out-of-date information. When the mobile host is changing its location frequently, this results in frequent handoffs; requiring a registration with a distant

[1]A *handoff* takes place when the mobile host moves from one coverage area to another.

home agent for each handoff causes higher overhead and further aggravates packet loss.

2.1.2 Migrate

Migrate [20] is an end-to-end architecture for Internet host mobility using dynamic updates to the Domain Name System (DNS) [14] to track host location. Existing TCP connections are retained using secure and efficient connection migration, enabling established connections to seamlessly negotiate a change in endpoint IP addresses without the need for a third party. The mobile host itself is in complete control of its mobility mode, so there is no need to have a home agent to receive packets on behalf of the mobile host, like in Mobile IP.

DNS is used as a level of indirection between a host's current location and an unchanging endpoint identifier. Migrate takes advantage of the fact that DNS is widely deployed, provides automatic hostname lookup for most applications, and supports secure dynamic updates. This means that when the mobile host detects that it has changed its point of attachment, it must immediately update the DNS mapping between its hostname and its IP address. To make sure this change is immediately perceived by other hosts trying to communicate with the mobile host, the DNS mapping update specifies a "time-to-live" (TTL) value of *zero*: this prevents the name entry from being cached in other hosts, which means that new connections to the same host must first initiate contact with the mobile host's name resolver to retrieve its current IP value.

Migrate proposes a new *Migrate* TCP option that serves to identify a connection as part of a previously established connection rather than a new one. This *Migrate* option includes a *token* that identifies a previously established connection to the same address and port. This token is used to identify the previous connection so that the host may retrieve its state and continue the connection from where it left off. The implementation of Migrate required the modification of the Linux 2.2.15 kernel, specifically, modifying the TCP stack to support Migrate options.

Mobile IP and Migrate solve a different problem than our framework. While we are concerned with application-initiated, third-party requests for stream redirection in an asynchronous manner, Mobile IP and Migrate solve the problem of either end-host changing its network address while the communication is in progress. In our framework, no end-host changes its network address: both the server and the sink proxy are fixed hosts. It is the controller, a third party that is not an endpoint of the connection, that is mobile (in a physical, as opposed to network, sense) and instructs the stream to be redirected to another host.

2.2 Physical Location Support Systems

2.2.1 Cricket

Cricket [19] is a location support system for in-building, mobile, location-dependent applications developed at MIT's Networks and Mobile Systems (NMS) Group [15]. It relies on hardware components called "beacons" that broadcast their physical location and their counterparts, the "listeners," that receive the signals sent by the beacons to determine the closest beacon and thus the listener's physical location.

It was designed with the following goals in mind: user privacy, decentralized administration, network heterogeneity, and low cost. It does not explicitly track the location of the users; instead, Cricket aids them in figuring out their location and lets them decide whether or not to advertise this information and to whom. By not tracking users and services, user-privacy concerns are adequately met. Furthermore, Cricket does not rely on any centralized management or control, and there is no explicit coordination between beacons.

A Cricket deployment consists of a set of beacons that are installed across a building. Each beacon periodically transmits (using radio frequency and ultrasound signals) its location information. Devices that have an interest in learning about their location have Cricket listeners attached to them. By listening to the beacon advertisements, each listener determines its location, and informs the attached device of said location. Listeners use the inter-arrival time between the radio frequency and ultrasound signals to estimate the location of each beacon.

Note that there may very well be more than one beacon per room, so in fact the location being received by the listener might be more specific than simply a room name or number (the location could specify, for example, the north or south half of a room); however, for the sake of simplicity we will henceforth assume that there is only one beacon per room, so that "changing location" and "moving from one room to another" are interchangeable for our purposes.

2.2.2 Active Badges

The *Active Badges* project [1, 21] provides a means of locating individuals within a building by determining the location of their "Active Badge." This small device worn by the users transmits a unique infrared (IR) signal for approximately a tenth of a second every 15 seconds. Each office within a building is equipped with one or more networked sensors that detect these transmissions. The location of the badge (and hence its wearer) can thus be determined on the basis of information provided by these sensors. A master station, also connected to the network, is given the task of polling the sensors for badge 'sightings,' processing the data, and then presenting it in a useful visual form.

Pulse-width modulated IR signals were used for signalling between the badge and the sensor mainly because: IR emitters and detectors can be made very small and very cheaply, they can be made to operate with a 6-meter range, and the signals are reflected by partitions and therefore are not directional when used inside a small room. Also, the signals will not travel through walls like radio signals that can penetrate partitions found in office buildings.

However, since there is a centralized database that keeps track of each user's location, privacy becomes an issue. Also, infrared suffers from "dead-spots," which are places in a room where no signal is received; Cricket is immune to this problem because it uses ultrasound.

2.2.3 Global Positioning System

The Global Position System [9, 8] is a worldwide radio-navigation system formed from a constellation of 24 satellites and their ground stations. Each satellite has an atomic clock and emits radio frequency (RF) signals that include the time and a code telling its location. By analyzing signals from at least four of these satellites, a receiver on the surface of the Earth with a built-in microprocessor can display the location of the receiver (latitude, longitude, and altitude). Consumer receivers are approximately the size of a handheld calculator and provide a position accurate to 100 meters or so.

The receiver clock times the reception of each signal, then subtracts the emission time to determine the time lapse and hence how far the signal has travelled (at the speed of light). This is the distance the satellite was from the receiver when it emitted the signal. In effect, three spheres are constructed from these distances, one sphere centered on each of the three satellites; the receiver is located at the single point at which the three spheres intersect. Since the clock in the handheld receiver is not nearly so accurate as the atomic clocks carried in the satellites, the signal from a fourth satellite is employed to check the accuracy of the clock in the handheld receiver. This fourth signal enables the handheld receiver to process GPS signals as though it contained an atomic clock.

GPS can be very precise for outdoor use. However, there is a lot of RF noise, the signals have low power, and metallic objects can cause signal reflections; this can sometimes make GPS inappropriate for use inside buildings, where Cricket is a better solution.

2.3 Network Resource Discovery Systems

2.3.1 INS

Also developed at MIT's NMS group, the Intentional Naming System (INS) [3, 2, 13] is a resource discovery and service location system for dynamic and mobile networks of devices and computers. Mobile environments require a naming system that is:

expressive (to describe and make requests based on specific properties of services), responsive (to track changes due to mobility and performance), robust (to handle failures), and easily configurable.

Applications in an INS network are defined by their *intentional names*, which describe what they *are* or what they *do* (i.e., their *intent*). The naming language they use is based on (possibly hierarchical) attribute-value pairs that describe the application's intent. For instance, a color printer in the third-floor lounge might describe itself as follows:

```
[service =printer
        [type = color]
        [location = lounge
                 [floor = 3]]]
```

INS implements a *late binding* mechanism that integrates name resolution and message routing, enabling clients to continue communicating with end-nodes even if the name-to-address mappings change while a session is in progress. *Intentional Name Resolvers*, or *INRs*, self-configure to form an application-level overlay network, which they use to discover new services and perform late binding.

Using late binding, a client sends the packet payload through the INR overlay network; the packet is identified with the intentional name of the service where the packet should be sent. INRs then forward the packet to any entity that has announced the specified intentional name (this is what is called *intentional anycast*). Optionally, the client may specify that the packet should be sent to *all* the entities with the specified intentional name: this is what is called *intentional multicast*.

An alternative to late binding is *early binding*, where the INR resolver network is used to get the current IP addresses of the entities with the specified intentional name. This is useful when the entities are relatively static. In this fashion INS is used much like the DNS, simply providing a mapping between a high-level name and a network binding.

2.3.2 Jini

Jini [11, 6] is a distributed framework developed in Java by Sun Microsystems. It is designed for the creation and management of communities of "services" (pieces of code that perform some function); clients can find these services without previous knowledge of network topology and minimal configuration. Services are accessed through Java objects called *proxy objects*; these are pieces of code that get transferred to the client, which can perform local method calls on them that are carried out on the remote service object (using any pertinent means for remote method invocation).

In Jini's architecture there are central manager programs called *lookup services* where entities can both register the services they provide to the community and request the services they need. The most common way for clients to find these lookup services is using IP multicast in a process called *discovery*. This means that all lookup services in the network will reply to the discovery request with their own proxy objects (in other words, lookup services are Jini services themselves).

The normal operation cycle of a Jini community is as follows: the lookup services are started on some node(s) on the network (several lookup services can be used for redundancy and robustness). Services start appearing on the network: they perform discovery, find all lookup services in the network and register their proxy object with the lookup services. When a client that requires a service appears, it does discovery to find the lookup services and it performs a lookup call on it, asking it to return the proxy object of the service in question. After that, the client can interact with the proxy object directly.

As stated in [7], however, there are certain limitations with the Jini architecture:

1. The Jini infrastructure does not explicitly allow for service mobility: if a service changes its location in the network topology, all clients connecting to that service lose the connection and have to rediscover the service.

2. The discovery process explicitly relies on IP multicast.[2] This means that institu-

[2]Although there are provisions for *unicast discovery*, this requires previous knowledge of the network address of a service lookup.

tions whose networks do not allow multicast to propagate across the boundaries separating various network segments will be forced to set up their Jini communities manually or will have to make serious alterations to their networking infrastructure.

3. Jini *service templates* (which are the mechanism used by clients to describe the services they seek) are not always the easiest or most descriptive way of specifying the services needed.

Chapter 3

Design

3.1 Introduction

In this section we will explain in detail the design of our stream-redirection framework, the rationale that went into our design choices, and finally some advantages and disadvantages of the overall framework.

3.2 Design Challenges

Designing a framework that does dynamic stream redirection presents the following challenges:

Making the stream redirection be transparent — Traditionally, the stream of data being sent in a network file transfer can not be redirected automatically: one can set up a transfer between two hosts, but once it has begun the receiving host can not "tell" the stream to begin flowing to another host. Doing this actually entails breaking the first connection, establishing one to the next host, and informing the server from which byte offset it should resume the transfer. We would like to make all of this completely transparent to the user: as far as she is concerned, when she moves from one room to another, all that happens is that the media file she is listening to is "magically" moved from a speaker in the first room to a speaker in her new location. Our framework will be in charge of:

1. Knowing when to redirect the stream to a new location.
2. Finding a suitable new device.
3. Finding where (i.e., at which byte offset) in the stream the transfer must be resumed.
4. Closing down the old connection and re-starting the transfer in the new location at the point where the previous one left off.

Giving devices a context of location — To make this work we clearly need to endow the handheld devices and the fixed devices whose resources we will be using with the context of their *physical location*. We will need a *physical location discovery system* that can be interfaced with the devices so they may learn of their location and be informed if that location should change. We have chosen the *Cricket location support system* as our physical location discovery system (see Section 2.2.1).

Discovering resources indexed by their physical location — We also need to be able to find resources according to their physical location. How each device's physical location is actually described is arbitrary: we could think, for instance, of devising a hierarchical naming system, where we start by giving a name to the building, then identifying the floor, then the number for the room itself. An alternative is to identify each room with a "flat" namespace where each room has a number or a name (for instance, the name of the person who works there). In any case, what is important is that the naming scheme is well-known and shared by all devices. Having this uniform naming scheme enables us to use a *network resource discovery system* to have each device announce its existence and its location to the network and then being able to query said network later to find pertinent devices. We have chosen the *Intentional Naming System*, or *INS*, as our network resource discovery system (see Section 2.3.1).

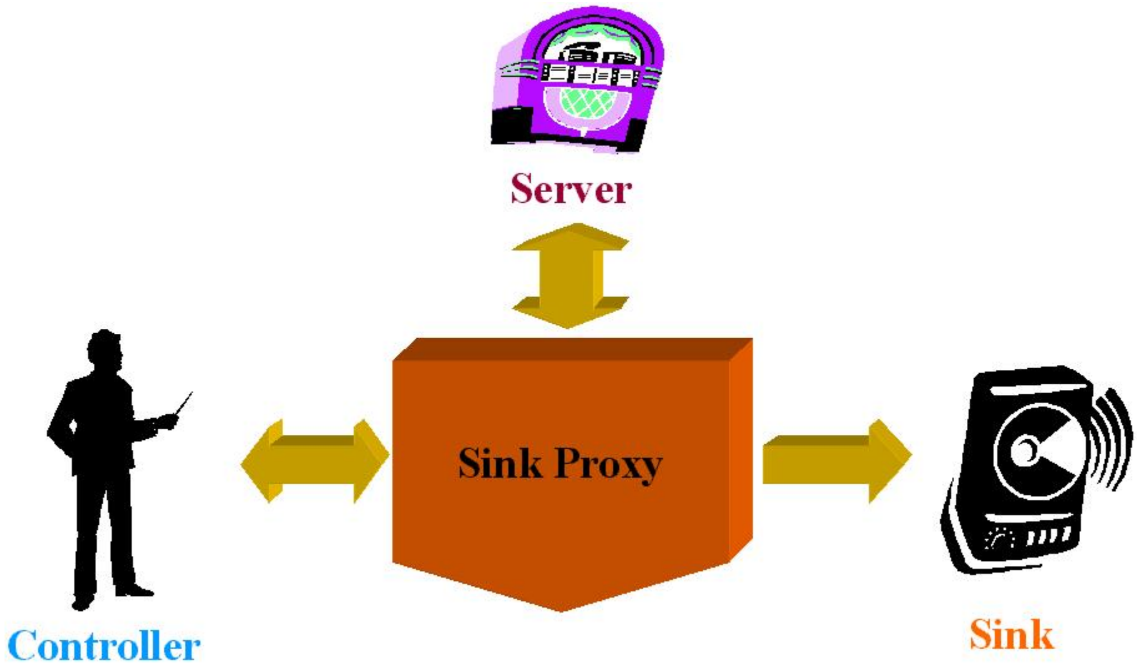

Figure 3-1: Framework Components

3.3 Design Overview

3.3.1 Components

Our framework is made up of four components that interact with each other through well-specified interfaces (as illustrated in Figure 3-1):

1. **Media Server**: This is the host where the media files that are to be played reside. Our system can interact with unmodified servers (such as web servers or FTP servers), so the media server is actually an external component of the framework. We can use existing servers as they are, as long as they support starting a transfer from any byte offset. No modification to the server is necessary to have it interact with the rest of our framework. We will go into more detail in Section 3.4.

2. **Media Sink**: This is where the media files are being played; in our previous example, the speakers are the media sink. We will go into more detail in Section 3.5.

3. **Media Sink Proxy**: Since actual media sinks will very likely lack computational abilities, we need a piece of software that can communicate with the rest of the system on behalf of the media sink. The media sink proxy is the piece of software that is the actual media sink's portal to the rest of the system. Its duties include communicating with the server to get the stream data and sending the data to the sink so the media file can be played (how this is actually done is sink-dependent but ultimately transparent to the rest of the framework). We will go into more detail in Section 3.6.

4. **Controller**: This is the piece of software running on the handheld device. Its duties include finding out its location, detecting motion from one location to another, and communication with a sink proxy to initiate a media file transfer. We will go into more detail in Section 3.7.

3.4 Media Server

3.4.1 Rationale

First, let us describe the rationale behind having a media server in our framework. We must start by realizing that most handheld devices in the market have very limited permanent disk storage. By contrast, the number of media files that a user may wish to play is virtually endless, and their size may surpass the permanent storage space that some handhelds provide.

Another reason for having a media server running on some other host different from the handheld device is limited bandwidth. Handhelds typically communicate with the network through a wireless network interface, whose bandwidth is considerably less than for wired networks. If the media files (whose sizes are usually several megabytes) resided on the handheld, they would have to be transferred through the wireless connection to the sink. Clearly it would be best to conserve the handheld's bandwidth and not use it unnecessarily for such large transfers.

The aforementioned reasons suggest the concept of an outside repository for the media files: this is what we call the *media server*.

3.4.2 Description

Although we include it as one of the components of our framework because it is a vital part of our framework's functioning, we consider it an *external component* because we have not written a specific server for our framework, opting instead to utilize existing servers. The only restriction placed on the types of media file servers used in our framework is that they support resumable downloads: in other words, that they accept a byte offset from which a file can be downloaded.

We need not modify an existing server to make it work with our framework; thus, it is perfectly acceptable to store the media files on a web server and to retrieve them using the HTTP protocol, or to store them on an FTP server, et cetera. We will explain how this flexibility is achieved in Section 3.8.4.

The sole responsibility of the media server is to listen for incoming file requests and to respond with the data of the requested file. How this is done is dependent on the type of server and the protocol it uses, but this is transparent to the rest of the framework. All that is known is that we will have at our disposal the stream of bytes that make up the media file so we are able to play it at the sink.

3.5 Media Sink

3.5.1 Rationale

The main purpose of our framework is to be able to utilize the resources of a room when the resources of our handheld are not satisfactory for playing media files. This means that there must be an appropriate sink, or recipient, of our media files that satisfies our needs. This is what we call the *media sink*.

Exactly what constitutes a media sink depends on the type of media file we are interested in playing. If we want to play an MP3 file, for instance, we would like to have some speakers in the room where we can hear the music; if we wish to play an

MPEG video file, on the other hand, we would like to have a monitor in the room where we could see the image.

3.5.2 Description

Like the *media server*, the *media sink* is actually a *external component* since the framework simply utilizes already-existing sinks, which require no modification.

This means that we do not require the use of "network-aware" sinks that can receive streaming data and play it automatically. We recognize that media sinks are typically attached to computationally-able devices such as desktop or laptop computers, which may be accessed through the network (how this knowledge is relevant will be discussed in Section 3.6.1).

The media sink's only responsibility to the framework is to allow the user to play or view the media file; again, how exactly the media sink works is hidden from the rest of the framework. For instance, the user only need know that there is a speaker in the room so she can play her MP3 files: what type of speaker it is, or how the media file's bytes are ultimately converted into sound, is of no interest to the rest of the framework.

3.6 Media Sink Proxy

3.6.1 Rationale

Although the framework's ultimate goal is to allow the user to play media files in some media sink resource available in the room, most media sinks (such as speakers or monitors) do not actually have computational ability and, as such, are unable to communicate with the rest of the framework by themselves.

It is thus necessary to have a piece of software with knowledge of the specifics of the sink that can communicate on its behalf with the rest of the framework. This software component of the framework is called the *media sink proxy*.

3.6.2 Description

The media sink proxy is co-located with the actual media sink, that is, it is running on the machine where the media sink physically resides. The sink proxy is thus in a position to interact with the rest of the framework and instruct the sink how to play the media file (by running a specific media player program, for instance).

Its basic responsibilities to the rest of the system consist of receiving commands from the *Controller* (Section 3.7), retrieving the stream for the media file from the media server, and instructing the sink how to play it.[1]

The sink proxy must also find its location (which may either be statically configured or be discovered through some physical location system, like *Cricket*), and announce its existence and its location to the network.

3.7 Controller

3.7.1 Rationale

To make the framework useful, there must be some piece of software that can detect the current location of the user as she moves around. This piece of software, called the *Controller*, runs in the handheld device that accompanies the user, and serves as her interface to the rest of the system.

3.7.2 Description

The controller has two main functions in the framework: the first is to present to the user a simple interface so she can select which media file she wants to play and allow her to play or stop said media file. The second function is to track the user's location: the controller must interface with a physical location system so it can determine its location and detect when said location has changed so the stream connection can be

[1]The framework makes no assumptions in terms of access control (that is, who has the "right" to use the sink at any particular time); this gives the application developer the flexibility to let her own sink proxy implementation utilize any admission scheme the application requires.

redirected to the appropriate new media sink.

Once the controller has determined the user's current location, it must find out if there are any available media sinks there. To do this, it must interface to a network resource location system, like the *Intentional Naming System*, or *INS*, to ask the network if there are in fact any media sink proxies running in the controller's current location.

3.8 System Interaction

Now that we have briefly introduced the components and the technologies that make up our framework, it is time to explain how they all fit together: how the pieces interact with each another and how the framework is flexible enough to accommodate specific implementations with different servers.

We will explain these interactions by going step-by-step and seeing the chain of events that are necessary to initiate the playing of a media file and the redirection of its stream from one sink to another as the user moves from one room to another.

We will also describe the interfaces that make up the skeleton of our framework. These interfaces, and the interactions between the objects that implement them, are at the heart of our design. Using these interfaces in the manner that we shall describe and knowing how to extend their functionality is what makes it easy to make good implementations using our design.

3.8.1 Physical Location Discovery

The first thing that will happen when the controller software is run is that the Cricket listener attached to it will inform it of its physical location (see Figure 3-2). Said location is *opaque* to the controller; this means that it is simply a string of characters that the controller receives but need not interpret.

In general, location information is simply used to figure out which sink proxies are running in the same location as the controller. It is unimportant exactly what this location string is or how it is constructed; it suffices only that the location string

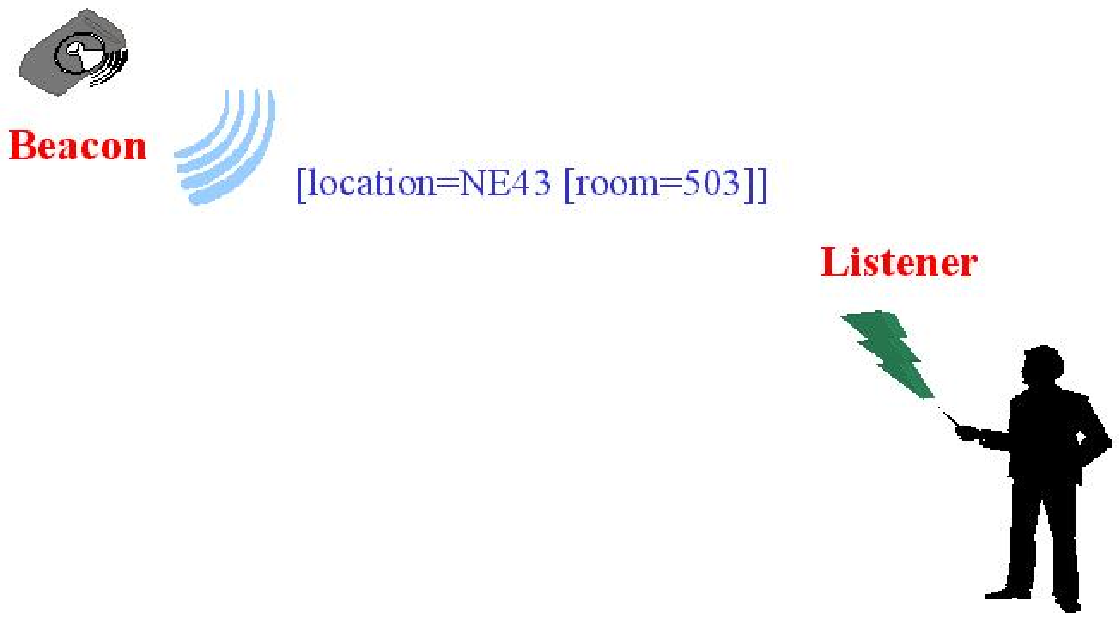

Figure 3-2: Physical Location Discovery Using Cricket

determined by the controller matches the one being announced by the sink proxy.

Sink proxies, thus, must also figure out their location information. In the case of a sink proxy, the location information (which is also opaque) may be given as a static, non-changing string (which may be useful if the sink proxy is running on a desktop computer whose location is known and fixed). However, it is also possible to interface a physical location system such as Cricket to the computer where the sink proxy is running so that its location may be given by the beacon in the room.

3.8.2 Announcement of Sink Resources

When the media sink proxy software is run, it must first figure out its location (as explained in Section 3.8.1). Once it does so, it must announce itself to the network so that it may be found by controllers interested in initiating a connection.

For this purpose, the sink proxy communicates with a network resource location system such as INS, sending it an announcement informing it of its existence and its location information (see Figure 3-3). Other information should be included in this network announcement; the sink proxy should reveal not only its physical location,

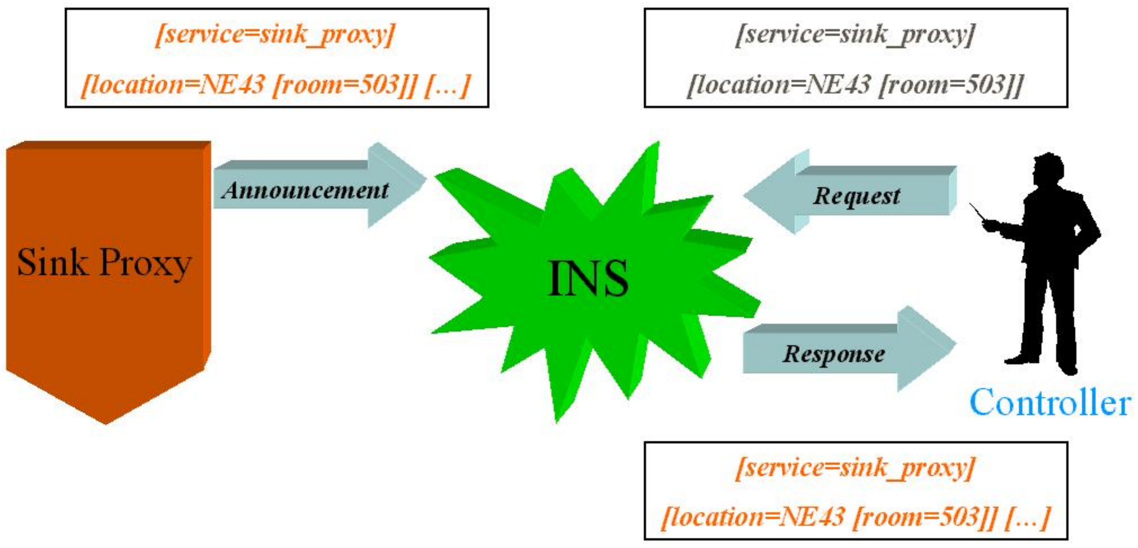

Figure 3-3: Announcement of Sink Resources Using INS

but also some information the controller may use to contact it (such as its network address and port), as well as which type of sink it is (if it can play audio or video, for example) and the transport type it accepts (*transports* are explained in Section 3.8.3).

3.8.3 Controller-Sink Proxy Communication

Now that the sink proxy has announced itself to the network, controllers interested in starting a connection need only request the network resource locator system to inform it of any entities whose announced name includes the controller's current location (see Figure 3-3). If there are, the controller will receive a response with their information and the controller can use that information to determine with which sink proxy it should establish a connection.

We thought it would be a useful property if the communication between the controller and the sink proxies could be carried out in any one of different ways and not restricted to just one; this is why we have the concept of a *transport*.

Transports

Transports specify how the communication between the controller and a sink proxy occurs. For instance, the two might communicate by establishing a TCP connection, or by sending UDP datagrams, or perhaps even by using INS's late binding feature (as explained in Section 2.3.1).

The way this works is as follows: our architecture defines two interfaces that must be implemented to enable communication between the controller and the sink proxy: Transport and ListeningTransport. The controller must use an object that implements the Transport interface, which defines the methods that may be used to communicate with a sink proxy (we will see these commands in Section ref.

Who must care about about the Transport and ListeningTransport interfaces and why are they important? It is the application developer that decides how the controller and the sink proxies must communicate in the way that is most convenient for her specific application.[2] Transports are important because they allow the application writer to have the controller and the sink proxy communicate in the manner that best suits the application; for instance, a transport that provides some type of authentication scheme and/or data encryption could be implemented and seamlessly integrated with the rest of the framework.

The concrete class that implements the Transport interface must make sure that each of these methods conveys the corresponding message to the sink proxy: how this is done (through an established TCP connection, by sending a UDP datagram, et cetera) is totally dependent on the concrete implementation of the transport pair.

Of course, the sink proxy must in turn use an object that implements the ListeningTransport interface: the job of this object is to listen for incoming messages from a controller and relay them to the sink proxy.

The concrete implementation of the Transport interface used by the controller must be the counterpart of said ListeningTransport. In other words, if the sink proxy has a ListeningTransport that is listening for TCP connections, the controller should connect to it using a Transport implementation that uses TCP.

How does the controller know which transports the sink proxy it wants to connect to has available? The sink proxy must include a "transport_type" attribute/value

[2]We have written a transport pair implementation that uses TCP sockets, which are ready to use and may be good enough for many applications.

pair in its announcement that the controller can retrieve to determine how to connect to it (we will learn more about this attribute in Section 4.3.1).

3.8.4 Establishing a Connection

Once the method of communication (*transport*) between the controller and the sink proxy has been determined, the controller is now ready to send the necessary commands to initiate the playing of a media file or to manipulate (e.g., stop) an already-existing connection. Later we will go into the methods available to the Transport interface, which are the commands that the controller may call in the sink proxy, but first we shall discuss three concepts related to controller-sink proxy connections: leases, ServerConnection objects, and Command objects.

leases A *lease* specifies a period of time that the controller can use the sink proxy. If the controller goes away (e.g., if the handheld device is shut off), the sink proxy will close the connection to that controller after some period of time. It is thus the controller's responsibility to *renew* the lease before it expires by signalling to the sink proxy that it is still running and that it still wants to use the sink proxy .

ServerConnection We have already stated that our framework allows the media file to reside in any type of (unmodified) server. For this to work, however, the sink proxy must know how to communicate with each type of server and how to retrieve the desired media file. Since it is impossible for the sink proxy to know beforehand about *every* server that may be utilized and which protocols they speak, an abstract and dynamic approach is required. For this purpose our framework describes an interface called ServerConnection that defines the methods necessary to connect to a server and to retrieve a media file (we will explain each of these methods later in this section). A concrete implementation of this interface knows how to connect to a specific type of server and how to retrieve a media file from it. For instance, for communicating with a web server, a HTTPServerConnection concrete implementation exists that knows how to connect to a web server and how to send correct HTTP requests to it. The specific ServerConnection object necessary to retrieve a particular media file is determined at run-time by the controller and sent to the sink proxy upon connection establishment (as will be explained later in this section).

Command The *Command* interface describes a command that may be sent between the controller and the sink proxy. In reality there are no specific methods that describe a Command: it is merely what is called a *tagging interface* since it serves simply to *tag* or *group* objects that share the same abstract functionality (in this case, objects that embody a command that can be sent to the sink proxy). There are two types of Command objects: ServerCommand objects and SinkCommand objects. As the name suggests, ServerCommand objects are those meant to be sent to the media server and SinkCommand objects are those meant to be sent to the sink itself (*how* they are sent will be explained later). When the sink proxy receives a Command object, it can determine by its type whether it is meant for the media server or the sink, and forward it to the appropriate one (we will see how this is done when we describe the ServerConnection and SinkConnection interfaces later in this section).

Let us now examine the methods that describe the Transport interface to discover how the controller communicates with the sink proxy in an abstract way:

***ping*()** The `ping()` method is used to verify the existence of a corresponding ListeningTransport instance. When the controller finds a sink proxy on the network, it uses this method to verify that the sink proxy has an actual ListeningTransport ready to listen for incoming connections before informing the user that such a proxy was found. How the existence of the corresponding ListeningTransport is actually verified is left up to the actual concrete implementation of the transport.

***establishConnection*()** This is the method that actually informs the sink proxy that this controller wants to initiate a media file transfer on behalf of the user.

As a parameter to this call, the controller sends an object that implements the ServerConnection interface and knows how to interact with the server from where the media files are to be retrieved. In other words, this ServerConnection object knows to *which* server it should connect and *how* to communicate with it in order to retrieve the media files that will later be requested (a more detailed explanation on the ServerConnection interface follows below). The sink proxy will keep a reference to this ServerConnection object and will create some state for the connection in an internal table (a more complete explanation of these details follows in Section 4.3). The sink proxy will return a long integer value (called a *cookie*) that uniquely identifies the connection just established: the controller will use it in all future commands regarding that same connection so the sink proxy knows to which connection those commands apply.

***sendCommand*()** After the connection has been established, the controller may now send commands to the sink proxy to retrieve a particular media file and to play it on the sink. A *command* object is a concrete implementation of the Command interface; it is an opaque object that is sent to the sink proxy to be processed by the corresponding ServerConnection or SinkConnection object,[3] depending on its type. That is, when the sink proxy receives a concrete Server-Command object from the controller, it forwards it to the ServerConnection's `sendCommand()` method (explained below); when the sink proxy receives a concrete SinkCommand object from the controller, it forwards it to the SinkConnection's `sendCommand()` method (explained below). These methods, in turn, know how to use that Command object to send a command to the media server or the sink. For instance, if we are talking about a web server, the HTTPCommand object sent encapsulates an HTTP request for a specified media file: the HTTPServerConnection object knows how to convert an HTTPCommand object into an actual HTTP request that the web server can understand.

***renewLease*()** As explained before, the controller must keep renewing its *lease* on

[3]The SinkConnection interface is explained later in this section

the connection it has established; for this purpose, it must call the `renewLease()` method periodically, before the time of the lease runs out.

***endConnection*()** When the controller is no longer interested in sending commands to the sink proxy with which it is connected, it should call the `endConnection()` method to signal to the sink proxy that it should close the connection between the media server and the sink and that it may release any state information it has been keeping. The `endConnection()` method is also called when the controller has detected that the user has changed her location; the controller will end the connection with the sink in the previous location and then resume the connection in the new sink from where the previous one left off. The `endConnection()` method returns a Command object that the controller can later use to resume the connection in a new sink proxy from where it left off (we will see how this works in Section 3.8.5).

Communicating with Different Media Servers

As explained previously, the way our framework achieves connectivity with different types of media servers is through the use of ServerConnection objects. The controller decides at run-time which concrete implementation of ServerConnection to send to the sink proxy when a connection to a specific server is desired.

Note that the controller should know all the information relevant to playing a specific media file, namely, the server where the media files reside (network address and port), the media type of the file (if it is an MP3 music file or an MPEG video file, for instance), and the type of server where it resides. Using this knowledge the controller is in a position to decide which concrete implementation of ServerConnection it should send to the sink proxy upon connection establishment.

However, we have yet to describe how this object is used by the sink proxy to actually communicate with the media server. Let us now go through the methods available in the ServerConnection interface and how they are used by the sink proxy.

***establishServerConnection*()** This is the method the sink proxy calls in order to connect to the media server. Note that it is expected that the concrete implementation of the ServerConnection object has embodied within it the appropriate information to connect to the server (this may include, for instance, the network address and port for the server). The sink proxy, thus, does not know (and does not care about) the process by which this connection is established or what it entails: it is simply understood that all the sink proxy needs to do to establish a connection to the media server is to call this method on the appropriate ServerConnection object.

***sendCommand*()** This method is passed a ServerCommand object that describes the command that must be sent to the server; this command is then written out to the media server (using the previously-established connection) using the server's own protocol. The `sendCommand()` method is called by the sink proxy whenever the sink proxy's ListeningTransport receives the `sendCommand()` call from the controller. Note that the sink proxy does not need to examine or understand the command it received before it can forward it to the ServerConnection object: the ServerCommand object is thus said to be *opaque*. If for instance we are communicating with a web server, an example of a ServerCommand that might be sent would be a HTTPGetCommand object: the HTTPServerConnection object's `sendCommand()` method will take the HTTP request embodied in this HTTPGetCommand object and write it out to the web server (in this case, initiating the retrieval of the specified resource on the web server).

***setOutputStream*()** We have talked about retrieving the data that makes up the specified media file, but we have said nothing of what is being done with said data. Where does it go? This is where the `setOutputStream()` method comes into play: it specifies the output stream where the data being received from the server should be written. An *output stream* is simply an object to which stream data is written. Note that it is the ServerConnection object's responsibility to connect to the media server and initiate the transfer of the media file, but it does not know what to do with the stream of data being sent by the server.

This is why the sink proxy tells the ServerConnection object where it should direct its stream of data using the `setOutputStream()` method: after all, it is the sink proxy that ultimately knows what to do with the stream.

`getResumeCommand()` The ServerConnection object is burdened with the responsibility of keeping track of how far the media file transfer has gone. In other words, it should be able to respond, at any moment, with information regarding the offset of the last byte read from the media server, and it should be able to construct a ServerCommand object that allows the transfer to be resumed from that offset. Concretely, the `getResumeCommand()` method will be called by the sink proxy when the controller calls the `endConnection()` method, and it should return a ServerCommand object that embodies the knowledge of resuming the connection from the appropriate byte offset. If for instance we are connecting to a web server, this method would return an HTTPResumeCommand, which is simply a HTTPGetCommand with an HTTP range-request header stating from which byte offset the file should be retrieved. Note however that the sink proxy does not concern itself with analyzing this ServerCommand object; it simply calls this method to get the resume ServerCommand and returns it to the controller as-is.

`closeServerConnection()` This method closes the connection that was previously established with the media server. It is called when the controller has signalled that the transfer of the media file should be terminated.

Thus far we have said nothing of how the media file being retrieved from the server is actually being played by the sink. What does the sink proxy do with this stream being retrieved from the media server?

The answer is that the framework is very flexible in what is done with this stream. Just as there is a ServerConnection interface that describes the interaction with the media server, there is a SinkConnection interface that describes the interaction with the sink.

Sink proxies have a default SinkConnection object that knows how to interact with the actual sink. Also, upon connection establishment, the controller *may* optionally send a SinkConnection object to the sink proxy, specifying exactly what to do with the stream it receives from the server. In most cases, however, the sink proxy's default SinkConnection object will be used for all connections.

What could be the use of having the controller send a SinkConnection object to the sink proxy? This SinkConnection object could specify that the stream retrieved from the server should be written to a file or even to a remote socket: the framework is not restrictive. However, we remark that it is *not* necessary to specify any SinkConnection object (as it is to specify a ServerConnection object) upon connection establishment: the framework will use the sink proxy's default SinkConnection object without any problems.

The SinkConnection interface is very similar to that of the ServerConnection: we will therefore describe it briefly, pointing out the main differences between the two:

***establishSinkConnection*()** This is the method the sink proxy calls in order to connect to the sink. For instance, if our sink is a pair of speakers physically attached to the computer and there is an audio device file (e.g. */dev/audio*), the sink proxy may connect to (or open) this device file and write the media file's contents to it, and the audio device (i.e. the speakers) would automatically play the media file. However, note that in many cases it may be impossible to "connect" with the sink in any meaningful way (if, for instance, in order to play a media file an external player must be started). This would indicate that perhaps a SinkConnection is not necessary for that specific implementation: in fact, our standard library includes a NullSinkConnection that is simply a concrete implementation of the SinkConnection interface that does nothing (we will see how a framework implementation like this works in Section 4.3.3).

***sendCommand*()** Much as a media server might receive commands from the controller, our framework allows for sinks to receive commands as well. Whatever SinkCommand object is sent to the sink proxy will be forwarded to the sink through SinkConnection's `sendCommand()` method. How the SinkCommand is interpreted by the SinkConnection depends on the actual implementation: the sink proxy knows nothing about the commands themselves.

`setInputStream()` The same way we need to tell the ServerConnection object where to write the stream it receives from the media server (by calling the `setOutput-Stream()` method), we must instruct the actual sink from where it should read the stream data. With this method we specify to the sink the *input stream* it should use for reading.

`closeSinkConnection()` This method closes the connection that was previously established with the sink when the controller asks that the connection be closed.

3.8.5 Example of Stream Migration

Now that we have defined the different components of the system and their interaction, we are ready to explain how the framework handles stream redirection and migrates a specific connection from one sink to another. We shall do so with an example:

1. When a sink proxy is started, it finds out its location (either through a physical location discovery system or by having it statically defined) and it announces its existence (along with its physical location and other parameters explained in Section 4.3.1) to the network.

2. A user comes into the room, and the controller software running on her hand-held device discovers its current location through a physical location discovery system. It uses the network resource discovery system to find out which sinks are available for playing media files.

3. The user decides she wants to play a media file and tells the controller the name of the resource and on which media server it resides (more information on this in Section 4.2.3).

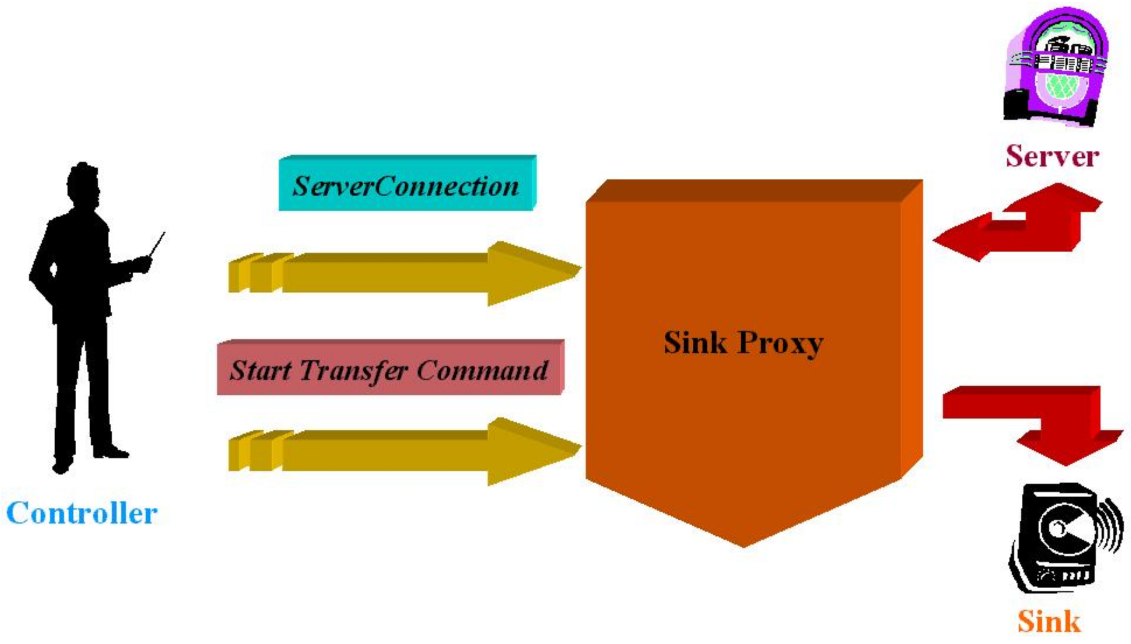

Figure 3-4: Connection Establishment

4. The controller will contact a sink proxy (which may be a *local sink*, that is, one in the user's current location, or possibly any other sink) and establish a connection with it. As explained before, a ServerConnection object of the appropriate concrete type (which depends on the media server from which the file resides) will be sent to the sink proxy (see Figure 3-4).

5. The controller creates a ServerCommand object that embodies the information about the name of the media file the user specified. This ServerCommand object specifying that the transfer should be started will be sent to the sink proxy through the `sendCommand()` method (see Figure 3-4).

6. The sink proxy will pass this ServerCommand object to the ServerConnection object, which will write it out to the media server, initiating the transfer of the media file.[4] The data stream is sent to the sink so it can play the media file.

7. Now let us suppose that the controller detects that the user has changed her location while this transfer is ongoing. The controller will query the network

[4]As stated before, the ServerConnection must keep track of the number of bytes that have been sent by the media server.

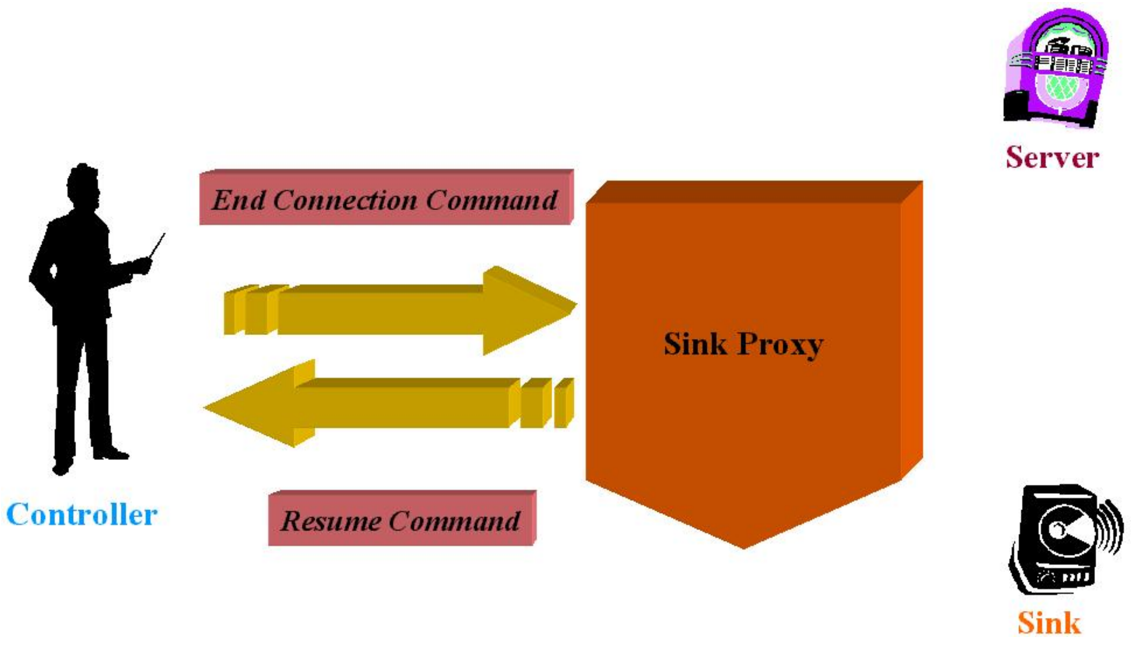

Figure 3-5: Ending a Connection

resource location system for any sink proxies active in the controller's new location.

8. When it finds a suitable new sink proxy, the controller will send the original sink proxy a `endConnection()` command and wait for its response.

9. The sink proxy will close the connection to the media server and to the sink by calling the `closeServerConnection()` and `closeSinkConnection()` on the ServerConnection and SinkConnection objects, respectively. It will also call the `getResumeCommand()` in the ServerConnection object to get the ServerCommand object that can be used to resume the connection: it sends this object to the controller as the return value to the `endConnection()` command (see Figure 3-5).

10. The controller establishes a connection to the new sink proxy as in Step 4.

11. Now, however, instead of generating a ServerCommand object, the controller uses the resume ServerCommand object it received from the old sink proxy to instruct the new sink proxy which media file to retrieve, from which server, and

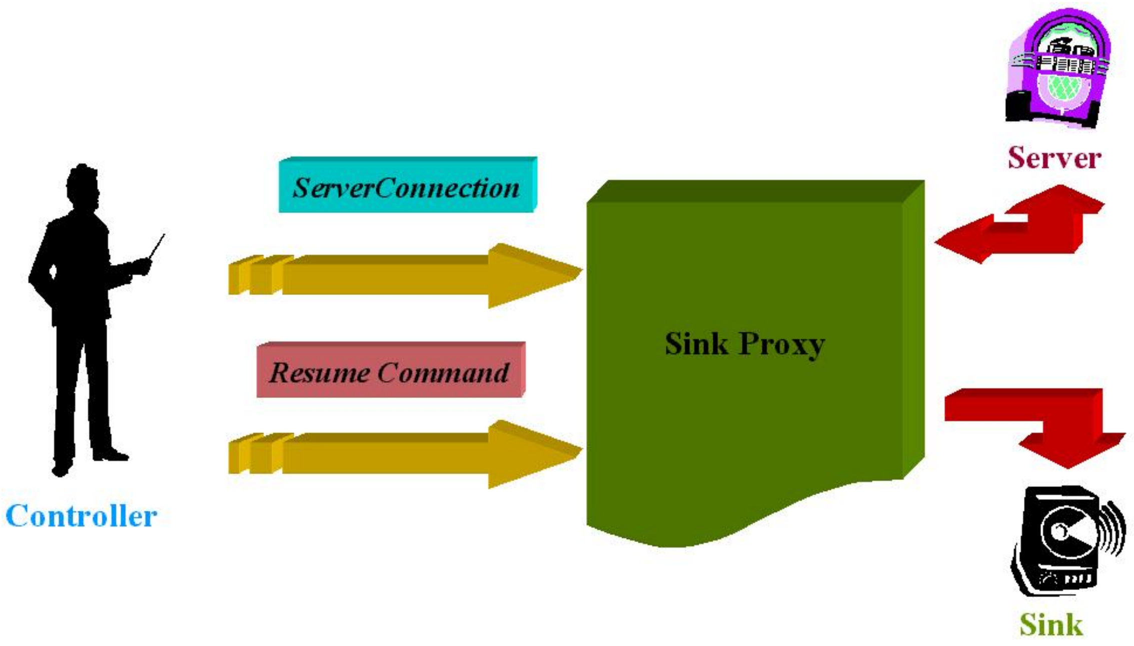

Figure 3-6: Resuming a Connection

more importantly, starting at which byte offset. This way the media file will begin playing from where it left off in the previous sink (see Figure 3-6).

3.9 Summary

In this chapter we have seen the main components of our framework, how they fit together and interact with one another, and we have seen the main classes that make the framework flexible by being configurable at run-time.

In the next chapter we will see in more detail how we have implemented some of these components and we will describe a specific prototype implementation we have written using our framework.

Chapter 4

Implementation

Now that we have described the overall design of our framework we can go into more detail about its implementation. We also describe a prototype implementation that uses our framework to allow a user to play media files residing on a web server.

4.1 Introduction

As mentioned in Chapter 1, we implemented our object-oriented design in the Java programming language. We have taken advantage of Java's dynamic class loading mechanisms to make our implementation not only flexible, but also very lightweight (more information on how this works will be given in Section 5.3.1).

We have strived to provide classes that abstract away many of the inner workings of the system, so developers wanting to write their own application (perhaps providing their own controller or sink proxy implementations) can do so using our API.

Let us start by describing the classes used to implement the controller and the API that developers may use to interface with our framework.

4.2 Controller Internals

The main class for the controller implementation is, aptly enough, named Controller. It interfaces with Cricket to determine its location, and it allows others to register their interest in finding out when this location changes through the LocationUpdateListener interface (Section 4.2.1). It also provides the methods necessary to find sink proxies, in the current location or anywhere else on the network.

First we describe how location awareness is managed, and then we describe the methods provided by the Controller class API that a developer may use to interface with the rest of the network.

4.2.1 Location Awareness

The application that is using the Controller class will probably be interested in being notified when the current location changes. Said application need only implement the LocationUpdateListener interface, which simply contains a method called `updateLocation()` that will get called whenever Cricket detects a change in location.[1] Clearly, what should happen when a location change occurs is application-dependent, and it is the application that should decide what should be done. For instance, what happens if there are several sinks in the new location: which one should be chosen? Or what should happen if there are no sinks in the new location, but perhaps there is one somewhere else? The application should use its own policies to decide where the stream should be redirected: the Controller class should only make it easy for the application to redirect the stream wherever it wants.

4.2.2 The Controller Library Class

We now go over the interface provided to the application through the Controller class:

***getAllSinks*()** This method returns a list of *all* the active sink proxies it has found on the network. By *active* we mean that we have called its `ping()` method to verify that it is actually running.

[1]Note that the Controller class does not automatically redirect the stream to a new sink: it simply notifies the application that a location change has occurred, and the application can in turn easily request the connection to be moved (as we will see in Section 4.2.2).

getLocalSinks `()` This method returns a list of all the active *local* sinks, that is, only those that are in the same location as the controller.

getAnyLocalSink `()` Basically a convenience method, it will get the list of the active local sinks and return any of those (it should be used if any sink will do for the application's purpose).[2]

establishConnection `()` This method is used to establish a connection to a specific sink proxy, specifying the server from where the media files will be retrieved and the sink proxy we are connecting to. It returns a long integer number called a *cookie* that uniquely identifies the connection just established.

sendCommand `()` Send the specified Command object to the specified sink proxy. This method, like the next two, uses the *cookie* value returned by the `establish-Connection()` method to identify the connection where this command should be sent.

endConnection `()` This method specifies that the connection to a specific sink proxy should be ended; it returns a ServerCommand object that allows the connection to be resumed later where it left off.[3]

moveConnection `()` This method takes care of all the details necessary to move an existing connection to another sink proxy. It only needs to know the *cookie* of the connection that needs to be moved and the new sink's information; it returns the *cookie* of the connection established with the new sink.

This API provides great functionality and a high level of abstraction while still affording flexibility to the application.[4]

[2]A basic application may only need this method, and not even concern itself with what the current location is.

[3]Note that this is provided just to allow the application more flexibility: it is simpler to use the `moveConnection()` method described next to redirect the stream from one sink proxy to another.

[4]Note how the notion of *transports* is totally transparent to the application, as is the notion of physical location awareness. In fact, the application does not need to concern itself with what the current location is: it need only be informed *when* the location has changed and request that the connection be moved to a sink proxy in the new current location.

4.2.3 Controller Prototype Implementation

We now briefly describe the prototype implementation we have written that uses the Controller class to allow the user to start the playing of a media file using a graphical user interface (GUI).

Our controller GUI was written using the Swing graphic libraries contained in Sun's Java Development Kit (JDK) 1.3. It gives the user the flexibility of discovering all the sink proxies in the network (not only those in the user's current location) and letting her choose where the media file should be played.

It also makes it easier for the user to play her favorite media files by allowing her to save the server and filename information in a local database. The user can later browse this collection of media file resources and play them without having to re-enter the server or file name information. The information that should be stored in this database for every resource is the following: the type of server where it resides (HTTP, FTP, etc.), the host name and port of the server, the full path to the resource and its type (*media types* are explained in Section 4.3.3). Optional information that can be stored for convenience are the size (in bytes) of the resource and a textual description.

Aside from the code to allow for the creation and maintenance of this resource database and the code for creating the actual GUI, the application is very small and has to do very little. It uses the Controller library class described in Section 4.2.2, which informs the application when the user's location has changed. At that point, the application asks for any sink in the user's new location and asks that the connection be moved there (until it finds one, the media file keeps playing at the old sink, unless it is explicitly stopped by the user).

The user can also request to see a list of all the sink proxies that were discovered on the network and request that the media file begin playing in any one of them (not just the one in the user's current location), and the user can also request that the stream be redirected to any arbitrary sink even when her location has not changed. This functionality serves to showcase the flexibility afforded by our API: the application is free to do much more than simply have a stream follow a user.

4.3 Sink Proxy Internals

Let us now describe the implementation details of the sink proxy. Much as there is a library class for the controller, there is a library class, called SinkProxy, that abstracts away all the command processing operations from the sink proxy executable. The SinkProxy object is basically in charge of processing the commands received by the ListeningTransport: it maintains a table of connections (indexed by each connection's *cookie*) where it stores the appropriate ServerConnection and SinkConnection objects for each connection.

This implies that there can be more than one connection to the same sink proxy from different controllers. In fact, the framework *does* allow connections from different controllers, since the application developer might want to have this flexibility. For instance, one sink proxy might allow the playing of sound files through some speakers while at the same time allowing video files to be played on a monitor. Clearly, such behavior is implementation-dependent, and as such, it is not constrained by our library classes.

4.3.1 Sink Proxy Intentional Names

Let us now describe how sink proxies identify themselves to the network in our prototype implementation. As mentioned in Chapter 1, the network resource discovery system we employ in our framework is *INS*.

In INS every entity has an *intentional name* that describes what it is or what it does: the specific metalanguage in which this description is written is encapsulated in an INS library class called NameSpecifier. A NameSpecifier object is initialized with the string representation of the intentional name and can be used to parse the attribute-value pairs contained therein.

We now describe the different attribute-value pairs that are expected from *all* sink proxies:

1. [**service = sink_proxy**] — This identifies the entity as a sink proxy: this will differentiate a sink proxy from other INS services that may coexist on the same network.

2. [**location =** ***location_description***] — Here is where the sink proxy's location is defined. As explained in Chapter 2, the exact format of the location is application-dependent and may be hierarchical or flat: the only important detail is that both the controller and the sink proxy agree on the location's format, since otherwise the controller will be unable to find any sink proxies. In our own prototype implementation we used the following hierarchical naming scheme:

   [**location =** ***name_of_place***
           [**building =** ***name_of_building***
                   [**floor =** ***floor_number***
                           [**room =** ***room_number_or_name***]]]]

3. [**transport_type =** ***name_of_transport***] — The name of the transport defines how the controller can communicate with the sink proxy. It is important that the controller and sink proxy agree on the name of the transport they are using. Furthermore, the controller must have a class file that implements that transport, and is named by pre-pending the transport's name to the string "Transport," and the sink proxy must have a listening counterpart named by pre-pending the transport's name to the string "'Listening-Transport." For instance, in our prototype implementation we implemented a TCP transport: this means that the sink proxy must announce the attribute-value pair "[**transport_type = TCP**]," the controller must have a class file called "TCPTransport.class," and the sink proxy must have its counterpart, "TCPListeningTransport.class." This naming convention should always be observed since our library classes use Java's *reflection* mechanism[5] to load classes at run-time.

[5] *Reflection* allows, among other things, to load classes based on their names and to inspect the data members or methods belonging to a class at run-time.

4.3.2 The SinkProxy Library Class

We now describe the methods contained in the SinkProxy library class, which relieves the application writer from having to worry about responding to commands sent by the controller:

***establishConnection*()** This method generates a new, unique *cookie* value, establishes connections with the server and the sink, and binds the ServerConnection object's output stream to the SinkConnection object's input stream so that the stream data read from the server gets written to the sink. The state pertinent to this connection (including a *lease timer* that expires after some time has passed without a lease renewal) is stored, and the newly-generated *cookie* value is returned.[6]

***sendCommand*()** This method checks the type of the Command object that was sent, and if it was a ServerCommand we forward it to the ServerConnection object's `sendCommand()` method and if it was a SinkCommand we forward it to the SinkConnection object's `sendCommand()` method.

***renewLease*()** This method renews the controller's lease for a specific connection with a sink proxy. As explained in Section 3.8.4, the controller must continuously renew its lease on all the connections it has established: if a lease expires, the connection will automatically be closed since it is assumed that the controller has either crashed or lost network connectivity and is thus unable to end the connection by itself at a future time.

***endConnection*()** This method closes the connections to both the server and the sink and asks the ServerConnection object for the resume ServerCommand: said object is then returned to the controller.

A sink proxy implementation written using our API need not concern itself with any of the previously-described sink proxy methods. The responsibilities of the sink

[6]This method, like the next ones, gets called when a command by the same name is received from the controller.

proxy implementation are simply the following:

1. Create a ListeningTransport object (this is left up to the sink proxy developer, who may choose to use a transport protocol such as TCP, UDP, et cetera).

2. Create a SinkProxy object, specifying a default SinkConnection object and a reference to the ListeningTransport object that will be receiving the commands from the controller. This object will take care of processing all the commands received from the controller.

3. Announce its existence using the network resource location system (in our case, INS). The exact information contained in this announcement is to some degree application-dependent, but the basic information it must contain was described earlier in Section 4.3.1.

4.3.3 Sink Proxy Prototype Implementation

Now that we have discussed the API provided by the SinkProxy object, we describe the prototype sink proxy executable we have written. Our prototype sink proxy implementation allows media files to be played using external applications we call *players* that are able to use the HTTP protocol to retrieve the media files. During the rest of this section we discuss the different pieces that make up our prototype implementation and how they interact.

Players and Media Types

Players are external applications that may be used to play media files (the *WinAmp* and *xmms* MP3 players are a couple of examples). This abstraction of a player allows the sink proxy to be run on different platforms, since we can use whatever player is appropriate for each platform.

A media player executable used for our prototype sink proxy implementation must fulfill two requirements:

1. It must be able to use the HTTP protocol to retrieve the media files it plays (remember that in our prototype implementation we give support only to web servers), taking the Uniform Resource Location (URL) of the desired media file as a command-line parameter.

2. It must be able to do so using a *web proxy*. A *web proxy* is an intermediary between the player (which works as an HTTP client since it is requesting a file using the HTTP protocol) and a web server. The client makes the file request to the web proxy, which forwards it to the web server; the data stream received by the web proxy from the server is then sent back to the originating client.

We will explain these requirements in greater detail later in this section.

We also have the concept of *media types* which describe the media file that is being played. We use the *Multimedia Internet Mail Extensions*, or *MIME types*, already standardized to describe the type of the media files. For instance, to identify the resource as an MP3 audio file, we use the appropriate MIME type, "audio/mpeg"; if it is a WAV audio file, then its type would be "audio/x-wav," and so on.

Our prototype implementation has a PlayerManager class whose job it is to read a player configuration file which associates MIME types with an executable player. This is how we tell the sink proxy how to actually play the media files depending on their types (since it is possible to have different audio types that need to be played by different programs).

The PlayerManager creates a table mapping each MIME type to its external player; when a media file must be played, this table will be queried and the appropriate player will be started.

The PlayerManager also compiles the list of all MIME types supported by the sink proxy for a very important reason. As we will see later in this section, our prototype sink proxy augments its announcement information with all the MIME types it supports; that way a controller can choose one sink proxy over another if it supports the MIME type of the media file being requested but the other one does not.

Prototype-Application Specific Attributes

Aside from the basic attributes that every sink proxy must announce described in Section 4.3.1, our prototype application also includes other attribute-value pairs to further describe our sink proxy implementation:

1. [**hostname = *host_name***] and [**port = *port***] — Since our prototype implementation uses a TCP transport, we must specify the host name and the port where the sink proxy is listening. There is no "well-known" sink proxy port: the TCPListeningTransport class binds to a random port when it is started, and it is this random port that is announced.

2. [**MIME_types = *number_of_MIME_types MIME_type_list***] — With this attribute-value pair we announce which MIME types are supported by our sink: first we specify how many MIME types we support and then we include the list of MIME types supported in attribute-value pairs, like so:

   [**MIME_types = 2**

   [**0 = audio/mpeg**]

   [**1 = audio/x-wav**]]

   This list can then be parsed by our controller, which can then decide whether or not the sink proxy can play the media file the user wants to play. If the controller receives a list of several local sinks, it can use the supported media types list to select which one is the most appropriate for the media file the user wants to play (naturally, if all local sinks support the same media types, then the choice becomes arbitrary since all of them are equally well-equipped to play the media file).

Web Proxy

We have thus far described how our prototype sink proxy implementation knows how to play a media file and how to announce to the network which media files it can

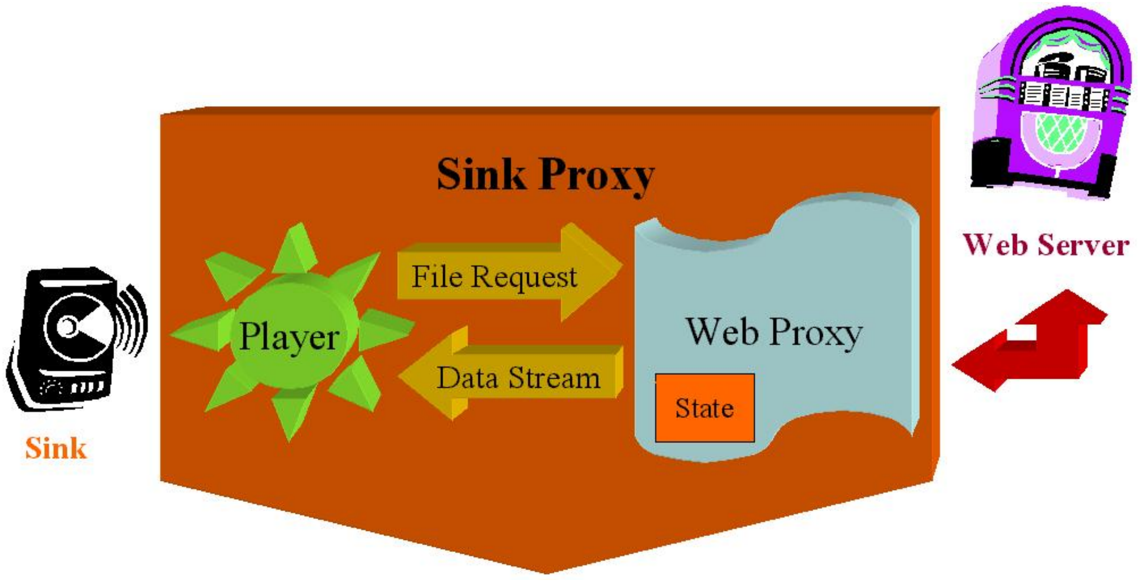

Figure 4-1: Use of Web Proxy

play. However, we have not yet described how the media file itself is retrieved from the web server and ultimately played.

To understand our prototype implementation design, let us remember one important detail of the overall framework: there is a component of the framework, namely the ServerConnection object, that is charged with the responsibility of keeping track of how many bytes have been played so that if the connection is ended before the media file has finished playing, we may return the byte offset where it left off to the controller so it can resume the connection on another sink.

How do we go about this in our prototype implementation? We make use of a *web proxy* (see Figure 4-1). As explained earlier in this section, our prototype implementation requires that the media players used be able to retrieve the media files using the HTTP protocol and be able to do so using a web proxy.

This is why we have implemented a lightweight web proxy ourselves in a class called WebProxyManager: this class listens on a well-known port (which is 8080 by default) for HTTP connections and forwards these requests to the specified web server. The media player program used must be manually configured to use the local machine as its web proxy, specifying the port where our local web proxy is listening for connections.

In its overall behavior, our web proxy behaves as any other web proxy (although it only supports the "GET" HTTP method, since it is the only one needed for our purposes). The special functionality provided by the WebProxyManager class is that for every connection it keeps some state, namely, how many bytes have been sent to the requesting player (more on this later).

When the sink proxy receives the command to retrieve (and play) a specific media file, it must start the appropriate external media player as explained before. The trick here is that when the player is started, it is given as a command-line parameter the URL that allows it to retrieve the media file from the appropriate web server using the HTTP protocol. However, since we have previously configured the media player to use our own web proxy running on the local machine, all the media player's requests for the media file are first "intercepted" by our own web proxy, which then requests the media file's data from the appropriate web server and sends its data stream back to the player (see Figure 4-1). This allows the WebProxyManager to be "in the middle" of the connection and be in the correct place to keep track of the number of bytes sent to the media player.

Now, how can the WebProxyManager keep track of which state belongs to what connection? When the external media player is started with a URL as its command-line parameter, we pre-pend the connection's *cookie* value to the actual path of the resource. For instance, if the controller instructs our sink proxy to retrieve and play the media file "/pub/mp3/music.mp3" from the web server "www.myhost.com" and the value of the established connection's cookie is "438502349538794," we would start the media player with the following command-line parameter:

```
http://www.myhost.com/438502349538794/pub/mp3/music.mp3
```

Our web proxy strips away the cookie value from the URL before connecting to the actual web server and uses this cookie value to index the state for that specific connection. As it sends data to the media player, it updates the connection's state information to reflect the latest byte offset sent.

For our prototype implementation we have created the WebProxyServerConnection class that knows how to interact with the web proxy to query it for a connection's last byte offset: this subclass of ServerConnection is the one that is sent to our prototype sink proxy upon connection. Concretely, when our sink proxy receives an `endConnection()` command from the controller, it calls the WebProxyServerConnection object's `getResumeCommand()` method. This method queries the WebProxyManager for the offset of the last byte sent to the player on a specific connection and constructs a resume command using said offset.

4.4 Summary

In this chapter we have gone into more detail regarding the implementation of the library classes described in Chapter 3 and we have seen the actual prototype implementation we have written using our framework. We note that ours was just a particular implementation, written as a proof of concept just to make use of our design and to show that it works. More sophisticated and feature-rich implementations are certainly possible.

Chapter 5

Results and Discussion

5.1 Introduction

We have thus far presented the design of our framework, given details into its object-oriented implementation and described a prototype implementation we have written using our framework, giving some insight into the decisions that were made.

We will now discuss a few salient points of our framework, like design and implementation issues, how flexibility is attained, some numerical results obtained from our prototype implementation, and finally some conclusions.

5.2 Design and Implementation Issues

In this section we will discuss some interesting issues that concern our design and our implementation.

5.2.1 Resource Access Control

What happens when two users try to use the same sink at the same time? What happens if a user moves into a room where the sink is already in use? These and similar questions refer to the issue of *resource access control*. Who has the "right" to use a specific resource at a specific time? How is this access restricted to only those that should be able to use it?

As mentioned in Section 3.6.2, our framework makes no assumptions in terms of resource access control. That is, it is up to the specific implementation of the sink proxy to enforce any kind of access control it deems necessary.

In the simplest case, the sink proxy may not need to do any explicit access control. One may assume that there is only one user in the system, in which case contention for resources will never happen; in other cases, the sink proxy may simply let the last user always gain access to the sink, or it may determine that the first user to ask for the sink should get it and the others must wait.

In any case, the developer is free to make the sink proxy implement any policy that is appropriate for the specific application.

5.2.2 Authentication and Security

In our prototype implementation, all users are "equal" in the sense that they all have the same permissions to use any sink proxy: they need not be distinguished from one another. Other implementations, however, may require the use of *authentication*.

Authentication refers to the process of identifying the user making a request: systems in which there are several users, each with her own level of permissions, must have a scheme for identifying each user and assigning her a set of permissions. For instance, there may exist the concept of the "owner" of a sink, and perhaps that user should have preferential access to that sink over other users.

Our framework allows such a scheme to be devised: the developer may subclass the Transport and ListeningTransport objects to send some extra information identifying the user making the request, and the sink proxy implementation may verify the user's permissions in whatever way the developer deems necessary.

In the prototype implementation, the connection between the controller and the sink proxy and the connection between the media server and the sink proxy all occur in *cleartext*, that is, without any kind of encryption, enabling anyone to examine network packets in transit. We felt no compelling reason why these communications should be encrypted; however, if a specific implementation requires a higher level of security, the developer may create her own subclasses of Transport and ListeningTransport that instead of sending commands over a regular socket employ instead secure sockets that do encryption.

In summary, both authentication and security can be easily integrated into our framework without modifying any of the existing functionality and without having to change any of the core library classes: subclassing of Transport and ListeningTransport may be all that is necessary to gain the extra functionality.

5.2.3 Media Player Issues

As mentioned in Chapter 4, our prototype implementation uses *media players*, which are external programs that play the media file on the sink. The concept of a media player makes the prototype implementation quite flexible because we can start any player that is appropriate for the specific operating system where the sink proxy is running, without having to tie ourselves to playing the files in a platform-specific way. The media player abstraction also allows many different types of media files to be played without much configuration: if we want to play MPEG video files, we only need to specify which player we want to use to play that media type, and the rest of the system need not change.

However, using such a scheme does present its drawbacks, which we will now discuss. First of all, there is a latency in starting up the media player (this latency is greater the first time around, when the player is not yet loaded in memory, but depending on the processor speed it may cause some small delay even after it is loaded once). Also, most players that retrieve media files using the HTTP protocol do some internal buffering before they start playing; this introduces a little latency before the media file starts to play and, more noticeably, lets the media file continue playing a few seconds after the user has ordered the playing to stop (this is because there is data still in the media player's buffer that it must play out).

A solution to this problem includes making the internal buffer as small as possible; if it is made too small, however, playback may become "jumpy" or "jittery," so a balance must be struck between a shorter latency and a smoother playback. It helps if the media server from where the media files are retrieved exists relatively “close” to the sink proxy itself, since network delays will be less and we could get away with a smaller buffer in the media player.

5.3 Achieving Flexibility

We will now discuss how we have achieved one of the main goals stated in Chapter 1: *flexibility*. By *flexibility* we mean that our framework should be able to accommodate different file servers and output devices without having to modify any system component other than those in our framework. It also includes making this as easy to deploy as possible; not much is gained by making the framework very flexible at the expense of making each component very complicated to deploy.

5.3.1 Media Server Independence

We have said how the framework allows the flexibility of permitting different types of media file servers where media files can be retrieved; we have described how a ServerConnection object is sent to the sink proxy upon connection establishment, and how this ServerConnection object “knows” how to communicate with the appropriate server.

Object Serialization

However, how do we physically get this object from the controller to the sink proxy? We use Java’s *object serialization* mechanism to create a ServerConnection object in the controller, configure it with the appropriate information (like the media server’s network address and port, for instance), convert said object to bytes that can then be sent over the network and reconstruct it at the sink proxy. This works by having the Transport object in the controller write the ServerConnection object to a specialized output stream that converts the objects to bytes and sends them across the network: these bytes are later reconstituted in the ListeningTransport.

A detail to note here, however, is that Java's serialization mechanism does not send the bytecode[1] of the actual class file being serialized; rather, it converts to bytes the *state* of all the data members of the object, so that when it is de-serialized, a new object can be initialized with the same values the original object had. Since it does *not* serialize the bytecode that describes the class itself along with the state of its data members, the bytecode for the class *must* be available at the JVM where the object is being de-serialized. This may be a problem if the object is being de-serialized in a remote JVM that does not have access to the class path of the JVM that originally serialized the object.

Dynamic Class Loading

The situation describes previously implies that if the controller is serializing an object of type WebProxyServerConnection, for instance, the sink proxy must *also* have the bytecode for this class in order to be able to correctly de-serialize the object (otherwise a run-time error will occur and the WebProxyServerConnection object will not be available for the sink proxy to use).

This fact implies that one of two things must be done:

1. The class files for *every conceivable* ServerConnection *subclass* (and necessary support files) must be given to all sink proxies so they can be found upon de-serialization.[2]

2. Sink proxies must be able to do *dynamic class loading* to find the bytecode of the serialized objects and use it to do the de-serialization.

The first alternative would make the framework harder to deploy: it would mean that new ServerConnection types could not be created after an initial deployment without having to go to every already-existing sink proxy and copying the new class files. This would not only be unwieldy in situations where there are many sink proxies

[1]The *bytecode* of a Java class is the sequence of bytes that makes up its class definition, or in other words, the binary description of the class' data members and its methods.

[2]By this we mean that the class files for all these new classes must be copied to the class path of each sink proxy.

already widely deployed, but it also means that sink proxies should keep class files that may or may not ever be used.[3]

Clearly the second option is better: if a new ServerConnection subclass is introduced, its class file should only have to exist at one location (the controller, which is the one that has to be aware of its existence in the first place), and its bytecode should be dynamically sent to the sink proxy if it needs it. We have opted to use this mechanism for deployment of new ServerConnection subclasses.

Web Server

The way we chose to make dynamic class loading work is as follows: the controller application is also running a lightweight *web server*, which is listening for connections on a random port. This web server will only respond to "GET" commands and will only serve a JAR file[4] containing the classes that may be needed by sink proxies running in remote JVMs.

At this point we must mention that there is a system property that may be set in a JVM called "`java.rmi.server.codebase`."[5] For dynamic class loading to work, this property must be set to reflect the local host name and the random port where the local web server is running as well as the name of the JAR file that contains the classes a remote JVM may need.[6] For example, if the local host name is "myhost.com," the local web server is running on port 4562, and the name of the JAR file being served is "streamredirector.jar," the "`java.rmi.server.codebase`" property would be set to the following:

```
http://myhost.com:4562/streamredirector.jar
```

[3]Note that not only would the new ServerConnection subclasses need to be distributed to all sink proxies, but all support classes for the new server type (like new server-specific Command objects) must be distributed as well.

[4]A *Java ARchive*, or *JAR* file is basically a zip file that archives and optionally compresses several files, specifically Java class files.

[5]A *codebase* is a URL from where class files may be loaded.

[6]Note that starting the local web server and proper initialization of the "`java.rmi.server.codebase`" property is all done programmatically and without intervention from the developer; our API already takes care of all these details automatically so dynamic class loading works off the bat.

When an object is serialized, said object is "annotated" with the value of the "`java.rmi.server.codebase`" system property. When the remote JVM (where the sink proxy is running) attempts to de-serialize the object, it will try to find the object's bytecode from its local class path. If it it can not find it there, it will check the "`java.rmi.server.codebase`" property the object has been annotated with and it will attempt to retrieve the bytecode from the specified codebase.

Since that codebase is a URL, it will connect to the web server being run by the controller and retrieve the JAR file containing the new class files and load them from there. Deployment of new ServerConnection types (and all necessary support classes) is as simple as adding them to this JAR file: the next time a sink proxy tries to de-serialize an object it has never seen before, it will request the JAR file from the controller and deployment will be automatic.

5.3.2 Transport Independence

Our framework also provides another important feature that enhances its flexibility: *transport independence*. By this we mean that the way in which commands are sent and data is transferred between the controller and the sink proxy is defined only abstractly in the Transport and ListeningTransport interfaces, and that how this communication actually takes place may be implemented by the application developer as she sees fit.

For the purposes of our prototype implementation, for instance, we needed nothing too sophisticated, so we implemented a TCP-based transport that basically opens a socket connection between the sink proxy and the controller and sends the commands, arguments and return values over this socket. This transport may very well be all that a particular developer's application needs, and in that case she may simply use our own TCPTransport and TCPListeningTransport classes as they are.

However, what if the developer wants her application to communicate over a secure channel, encrypting the commands being sent? Or what if encryption is not necessary, but the application would benefit from an authentication scheme where a controller has to identify itself before a sink proxy will allow it to play a media file there?

These and many other schemes are made possible by the abstract definition of a transport: developers are free to implement the Transport and ListeningTransport interfaces as necessary to fit their own application's needs.

5.4 Results

Let us finally discuss the results we obtained with our working prototype implementation.

We tested our prototype implementation with the following setup:

- The controller was running on an iPAQ (running Linux).
- There were two sink proxies, both running on ThinkPad laptops running Windows 98. They were both in the same room, but they were placed more than 3 feet apart and beside each one there was a beacon announcing a different room location.[7] They each had a Cricket listener attached to their serial ports so they could learn they physical location dynamically.
- The media server was an unaltered Apache web server that was running on a separate machine.
- The Domain Space Resolver (DSR) and Intentional Name Resolver (INR) for INS were running on that same machine (which we will call the server host). There is no requirement, and no limitation, for where the web sever and the DSR and INR must run. For convenience, we ran them all on the same machine, but they could all be in separate hosts if so desired.

The tests we ran followed the following procedure:

1. The web server, DSR and INR were started on the server host.

[7]A distance of more than 3 feet is sufficient so that listeners can distinguish one beacon's signals from the other's.

2. One sink proxy was started on each laptop. This caused them both to learn of their respective locations through Cricket and begin announcing their existence to the INS network.

3. The controller application was started on the iPAQ; it learned of its location using Cricket. It discovered that both sinks were on the network, but picked the one closest to it as the local sink.

4. On the controller, a media file was selected and we instructed that it should begin playing.

5. We moved the iPAQ running the controller closest to the laptop announcing a different location. We measured the time from when we brought the iPAQ closer to the "new location" until the music began playing on that laptop. This time is what we call the *redirection latency*.

Notice that we are simulating walking from one room to another by bringing the controller nearer to another beacon just a few feet away; we do this because it makes it easier to measure the redirection latency. If we were walking from one room to another, for instance, many other factors would come into play, like the placement of the beacons in both rooms, the relative signal power of each beacon as we move into the next room, et cetera. It would be unclear when we should begin timing the redirection latency, so the results would be more inaccurate and would vary more from one experiment to the next.

Our initial experiments show a redirection latency of around 8 to 10 seconds. This latency is made up of the following delays in which the system incurs for stream redirection:

1. **Delays imposed by Cricket** — The Cricket software takes around 5 to 6 seconds (worst-case) in realizing that the user has left one location and is now in another one. This is because its algorithms take several measurements to be sure it has received sufficient data to announce the location has changed and informing the application about it.

2. **Delays imposed by INS** — The INS software takes less than 2 seconds on average to return the list of entities that match a particular intentional name. We have observed, however, situations in which it takes closer to 5 seconds to get the list of active sink proxies; for this reason we have opted to take INS out of the critical execution path. We will request *all* active sink proxies periodically and use that list to find the one(s) in the user's new location instead of querying INS every time the user moves (in other words, we will be caching the list of all the proxies in the system and constantly refreshing it). By employing this technique we have made the new sink lookup time almost negligible.

3. **Delays imposed by system** — The delay imposed by our system is around 3 or 4 seconds: the greatest source of latency is the starting of the external media player (which is greatest when the media player has never been loaded in memory, but is considerably less after the player has already been loaded).

Note that some of these latencies are caused by our particular prototype implementation, which has not been completely tuned yet. We will continue our work and strive to make the prototype implementation even faster by writing our own media player in Java. Instead of relying on external players (with which we have a limited interface), we will write a player using the Java Media Framework (JMF) [12]. This will allow us to directly query how many bytes of the media file have been played and we can request the playing to stop immediately, even if there is data on the buffer. This way we can ameliorate response time and make stopping a media file more instantaneous. Fortunately, the abstract treatment of players in our framework will make integration with a native player as easy as writing its code.

We conclude that if we can decrease the time it takes Cricket to detect that the user changed her location, we can have a more reasonable redirection latency, as Cricket is the bottleneck of the system. Since Cricket is itself a project in progress, we are confident that its response time can be improved upon, and that this will immediately benefit the redirection latency of our project.

5.5 Conclusions

In summary, we were successful in the goals we set out for our project:

1. **Design a framework allowing the discovery of other devices based on physical location, and being able to use them to initiate the playing of media files** — We described in detail the design of this framework in Chapter 3 and we showed how we use a physical location discovery system to give our components the context of location and how we utilize a network resource discovery system to utilize this location context to find an appropriate device to use.

2. **Make the framework flexible and configurable, and offer a simple, feature-rich object-oriented API** — We also described in Chapter 3 what features of our design make our framework flexible and configurable and easy to use by developers. We explained how the framework is agnostic in terms of the server used to retrieve the media files, the way the media files are played in the sink, and the way the communication takes place between the controller and the sink proxy. We discussed how the ServerConnection object sent to the sink proxy contains all the knowledge necessary to contact the media server and retrieve the desired media file, while requiring no changes to the sink proxy. We also saw how the sink proxy decides how to play each media file, abstracting this process away from the rest of the framework. We discussed how the Transport and ListeningTransport classes define abstractly the way communication between the controller and the sink proxy is performed, allowing a developer to implement any type of scheme at this point (such as authentication, encryption, et cetera). Finally, we showed the basic API provided by the Controller and SinkProxy classes and how a developer need not concern herself with any of the details of how the framework works at its deepest level, thus enabling her to concentrate on the details pertaining to her specific implementation's needs.

3. **Build a working prototype implementation using our framework** — In

Chapter 4 we described the actual implementation of our core library classes and a working prototype implementation that uses our framework to allow a user to choose a media file residing on a web server and then allows her to play it in any available sink. We discussed the specific design decisions made, and we described in great detail the techniques employed to get the implementation to work.

In conclusion, we have seen how we can design systems that greatly augment the functionality of handheld devices, and how by giving them the context of their location we can empower them to utilize resources in nearby devices that better serve the user's needs.

Bibliography

[1] The Active Badge Location System Homepage. http://www.uk.research.att.com/ab.html, 2001.

[2] W. Adjie-Winoto. A Self-Configuring Resolver Architecture for Resource Discovery and Routing in Device Networks. Master's thesis, Massachusetts Institute of Technology, May 2000.

[3] Adjie-Winoto, W., Schwartz, E. and Balakrishnan, H. and Lilley, J. The design and implementation of an intentional naming system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, December 1999.

[4] M. Campione and K. Walrath. *The Java Tutorial*. Addison-Wesley, Reading, MA, 1996.

[5] M. Dertouzos. The Future of Computing. *Scientific American*, August 1999. Available from http://www.sciam.com/1999/0899issue/0899dertouzos.html.

[6] W. K. Edwards. *Core Jini*. Prentice-Hall, Upper Saddle River, NJ, 1999.

[7] K. Gajos and J. R. Nogueras. *Improving Jini Discovery Mechanisms Using INS*, December 1999. http://web.mit.edu/˜rafaeln/Public/JiniINS/JiniINS.pdf.

[8] Global Positioning System. http://www.eftaylor.com/pub/projecta.pdf, 2000.

[9] B. Hoffmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System: Theory and Practice, Fourth Edition*. Springer-Verlag, 1997.

[10] Java (TM) Programming Language. http://java.sun.com/, 2001.

[11] Jini. http://java.sun.com/products/jini/, 1998.

[12] Java Media Framework. http://java.sun.com/products/java-media/jmf/index.html, 2001.

[13] J. Lilley. Scalability in an Intentional Naming System. Master's thesis, Massachusetts Institute of Technology, May 2000.

[14] P. Mockapetris. *Domain Names - Implementation And Specification*, November 1987. RFC 1035 (http://www.ietf.org/rfc/rfc1035.txt).

[15] Networks and Mobile Systems (NMS) Group. http://nms.lcs.mit.edu/, 2001.

[16] Oxygen home page. http://oxygen.lcs.mit.edu/.

[17] C. Perkins and K. Wang. Optimized smooth handoffs in mobile ip, 1999.

[18] C. E. Perkins. *IP Mobility Support*, June 1996. RFC 2002 (http://www.ietf.org/rfc/rfc2002.txt).

[19] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Proc. 6th ACM MOBICOM Conf.*, Boston, MA, August 2000.

[20] A. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proc. 6th ACM MOBICOM Conference*, August 2000.

[21] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.