RESOURCE-AWARE DEPLOYMENT, CONFIGURATION, AND ADAPTATION FOR FAULT-TOLERANT DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

By

Jaiganesh Balasubramanian

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December, 2009

Nashville, Tennessee

Approved:

Dr. Douglas C. Schmidt

Dr. Aniruddha Gokhale

Dr. Chenyang Lu

Dr. Christopher Gill

Dr. Gautam Biswas

Dr. Janos Sztipanovits

To Amma and Appa for their love and encouragement over the years

ACKNOWLEDGMENTS

My stay in graduate school at the University of California, Irvine, and the Vanderbilt University has been a long, and exciting journey that has made my emotions travel through many valleys, and a few peaks. I would not have been able to complete this journey without the help, support, and encouragement of the following individuals and I am ever grateful to them.

First and foremost, I would like to thank my advisors and mentors, Dr. Douglas C. Schmidt, and Dr. Aniruddha Gokhale, for providing me an opportunity to work with them at the Distributed Object Computing (DOC) group at Irvine, and later at Nashville. I thank Prof. Schmidt for providing me tremendous support, advice, and guidance for my graduate study, research, and career development. Prof. Schmidt has become my role model and has set a perfect example I can follow through my professinal career to work hard and become an insightful scholar and a successful leader as he is. I am deeply indebted to Dr. Gokhale for all the countless hours and tireless effort he has spent for me. More importantly, Dr. Gokhale taught me how to think independently and present my research in numerous occasions. All the walks around the Vanderbilt University that I have had with Dr. Gokhale have helped me in concretizing my research ideas, and those ideas have inturn translated to many conference and journal publications.

I would like to express my thanks to the rest of my committee members, Dr. Janos Sztipanovits, Dr. Gautam Biswas, Dr. Chenyang Lu, and Dr. Christopher Gill, for agreeing to serve on my dissertation committee, for their research collaborations, for their insightful suggestions, ideas, and questions, and for their timely feedback. I am especially grateful for the time Dr. Lu devoted to reviewing my research thoughts and his contructive feedbacks have greatly improved the quality of my dissertation.

Over the years, my graduate study has been supported by a variety of agencies and companies. I would like to take this opportunity to thank the following people: Richard Schantz

and Joe Loyall at BBN Technologies for providing the initial motivation on my work on fault-tolerance for real-time systems; Patrick Lardieri, Gautam Thaker, and Thomas Damiano at Lockheed Martin Advanced Technology Laboratories for supporting the work on dynamic resource management that emerged as the foundation blocks for my dissertation; Paul Werme at US Navy Dahlgren for funding the initial prototype of my research and providing many usecases and example scenarios; Balakrishnan Dasarathy at Telcordia Technologies for providing me with many insightful suggestions and feedback for my research; Dipa Suri and Adam Howell at Lockheed Martin Advanced Technology Center for collaborating with us on many projects and providing us with many interesting and challenging scenarios that are used as case studies in this dissertation.

My stay as a graduate student at Irvine was very enjoyable due to the following people: Maulik Oza, Akshay Verma, Shireesh Verma, Sundararajan Sowrirajan, Mayur Deshpande, Kartik Muktinutalapati, Manish Prasad, Priyanka Gontla, Mukesh Rajan, Krishna Raman, Mark Panahi, Raymond Klefstad. I would like to thank Ossama Othman, Carlos O'Ryan, Nanbor Wang, and Angelo Corsaro for their insights and constructive feedback to my ideas over the years. Special thanks are to Dr. Raymond Klefstad for guiding me and intellectually and financially supporting me through my graduate study at Irvine.

After my move to Nashville, I have been fortunate to interact with the following individuals: Hariharan Kannan, Swaminathan Ramkumar, Prakash Raghotamachar, Divya Prakash, Nagabhushan Mahadevan, Deepa Janakiraman, Abhishek Dubey, Aditya Agrawal, Sachin Majumdar, Sujata Majumdar, Manish Kushwaha, Amogh Kavimandan, Bharati Gokhale, Anantha Narayanan, Roopa Pundaleeka. I would like to thank Jeff Parsons for many enjoyable discussions and movie nights that we have had over the years. Balachandran Natarajan welcomed me at the airport, helped me settle down in Nashville, and has been a friend, guide, and an advisor over the years. Krishnakumar Balasubramanian and Arvind Krishna provided company for countless breaks that served as the home ground for endless critiques on anything from cricket to politics to movies. Sumant Tambe, Akshay Dabholkar,

and Nilabja Roy provided company for many "coffee sessions" that served as the breeding ground for creating graduate study jokes that alleviated the pains of the long and tiring PhD journey.

Finally, I would like to thank my entire family for providing a loving environment for me to pursue my graduate study. Most importantly, I would like to thank my parents for their endless love, understanding, and gratuitous support for me. To them, I dedicate this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Overview of the Problem Space

Distributed real-time and embedded (DRE) systems form the core of many mission-critical domains, such as shipboard computing environments [141], avionics mission computing [145], multi-satellite missions [155], intelligence, surveillance and reconnaissance missions [144], and smart buildings [146].

Such systems are composed of services and client applications that are deployed across networks that are normally the size of a local or metropolitan area network. Often these services and client applications are part of multiple end-to-end workflows that operate in environments that are constrained in the number of resources (*e.g.*, CPU, network bandwidth). These systems can predominantly be classified as being static/closed (*i.e.*, fixed system loads) or dynamic/open (*i.e.*, varying system loads), both of which may experience fluctuating resource availabilities (*e.g.* due to resource failures or overloads). It is in such operating conditions that each service within the workflows must process periodic events belonging to other services or clients while providing quality of service (QoS) assurances in the form of reliability and timeliness.

To enable DRE systems to support the QoS demands of their multiple application workflows, all of which contend for resources, there is a strong demand for techniques and mechanisms that can efficiently and effectively manage the limited number of resources, such as CPU and network, in the face of failures and workload changes. With the advent of low-cost high-speed processors, and large network bandwidth availabilities, it may appear conceptually simple to handle this problem by simply overprovisioning network bandwidth and CPU resources. In practice, however, the resource provisioning problem is more complex due to the need to differentiate applications and application flows based on

varied criteria including priorities, urgencies and mission-criticalities [100, 136]. A naive resource overprovisioning solution is not a viable option in cost- and resource-constrained environments in which DRE systems are often deployed.

I.2 Contemporary Mechanisms: QoS-enablers in Middleware

The responsibility of allocating resources to applications in a controlled manner has historically being delegated to the *middleware layer*, which acts as a bridge between the application and the underlying system resources. A significant amount of prior research exists in developing novel middleware mechanisms to ease the development of DRE systems. Earlier efforts in middleware research focused on providing location transparency, portability and interoperability to satisfy the needs of general-purpose DRE systems. These efforts gave rise to middleware, such as CORBA [115], Java [152], and DCOM [98]. Such middleware simplify the development of DRE systems by hiding complexities associated with low-level operating system and protocol-specific details of network programming.

Subsequent research efforts building on the successes of these middleware focused on providing missing capabilities, such as features to support the QoS needs of DRE systems. These efforts resulted in standards that have defined interfaces, services and strategies to enhance the timeliness and fault-tolerance capabilities of DRE systems. For example, RT-CORBA [113] and Distributed Real-time Java [64] provide capabilities to ensure predictable end-to-end behavior for remote object method invocations. Similarly, Fault-Tolerant CORBA (FT-CORBA) [110] and Continuous Availability API for J2EE [154] provide services and strategies to enhance the dependability of DRE applications.

Additional prior research efforts have also focused on middleware-based QoS management mechanisms including approaches that focus exclusively on timeliness assurances [39, 66, 78, 101, 123, 157, 167], or others focusing only on high availability assurances [11, 13, 48, 92].

Providing both high availability and soft real-time performance simultaneously for

DRE systems using the above described technologies and mechanisms is complex for the following reasons:

- Prior research on QoS mechanisms in middleware focus on addressing only *one* QoS dimension (*e.g.*, timeliness), but by no means *both* QoS dimensions (*e.g.*, timeliness and high availability) as expected by DRE systems. For example, fault-tolerance solutions are often not designed to honor timeliness while recovering from failures, whereas real-time solutions often do not account for failures and recovery times while ensuring predictable end-to-end behavior for remote object method invocations.

- It is not straightforward to expect both availability and timeliness assurances by simply combining one or more of the existing solutions (*e.g.*, FT-CORBA and RT-CORBA) due to the syntactic and semantic differences between the interfaces, and how the individual solutions are developed. Moreover, any solutions along this approach result in systems that are brittle and hard to maintain and upgrade.

I.3 Technical Gaps: Overview of Missing Middleware Capabilities

We are interested in the development of middleware-based mechanisms that provide both high availability and soft real-time performance simultaneously for both the open and closed types of DRE systems. For open systems, changing system loads and fluctuating system resource availabilities make the problem of assuring timeliness challenging because the timeliness properties of client applications are dependent on the performance characteristics of the hardware nodes hosting the server applications, which is continuously varying with time. Hence, sophisticated, adaptive resource management solutions at the middleware-level are required that adapt QoS by dynamically monitoring the performance characteristics at the hardware nodes in the system.

For closed system, all the properties associated with system workloads are invariant. As a result, any QoS solution made at design/deployment-time continues to be valid even at

runtime as long as that solution covers all possibilities of system evolution. Such *ahead-of-time* (*i.e.*, at deployment-time rather than at runtime) decisions to handle resource failures are essential since the highly resource-constrained nature of closed DRE offers very limited scope for any sophisticated runtime solutions, which were shown to be integral open DRE systems.

Given the diversity of the solution needs for assuring multiple QoS properties across a range of DRE systems, a *one-size-fits-all* approach that is prevalent with standards-based middleware [64, 110, 113, 154] will not suffice. Further, given the complexity of DRE systems and the market forces that require system development and maintenance costs to be kept low, it is not feasible to expect each and every application to develop their own proprietary solution to managing both the performance and fault-tolerance requirements of DRE systems.

Our goal is thus to address the issue of semantic differences between multiple QoS dimensions by enhancing existing standards-based middleware with novel features for designing, developing, deploying, and configuring DRE systems with both *deployment-time* as well as *runtime* QoS assurances. To realize these goals, there is a strong demand for algorithms, architectures, and mechanisms within middleware that overcomes the disadvantages of a *one-size-fits-all* solution yet holistically offers to:

- work for closed environments, where it can allocate CPU and network resources to contending applications at deployment-time and provide the required QoS subject to the resource constraints imposed by the closed systems, and
- work for open environments, where it can react to changing system loads and resource fluctuations at runtime and maintain the required QoS subject to the resource availabilities.

Supporting the vision of the middleware capabilities outlined above leads to three key

open research issues that are identified and resolved by this dissertation. These three open issues include:

1. **Deployment-time, Resource-aware Fault-tolerance for DRE Systems.**

   As noted earlier, DRE systems can benefit from middleware [7, 104, 120, 128, 174] that provides distributed software platforms for building fault-tolerant DRE systems. Server replication is a popular technique [61] adopted by such middleware to provide high availability assurances for DRE systems. ACTIVE and PASSIVE replication [61] are two common approaches for building fault-tolerant distributed applications that provide high availability and satisfactory response times for performance-sensitive distributed applications operating in dynamic environments.

   In ACTIVE replication [137], client requests are multicast and executed at all replicas. Failure recovery is fast because if any replicas fail, the remaining replicas can continue to provide the service to the clients. ACTIVE replication, however, imposes high communication and processing overheads, which may not be viable in resource-constrained environments. In contrast, in PASSIVE replication [20] only one replica—called the primary—handles all client requests, and backup replicas do not incur runtime overhead, except (in stateful applications) for receiving state updates from the primary. If the primary fails, a failover is triggered and one of the backups becomes the new primary. Due to its low runtime overhead, PASSIVE replication is appealing for applications that cannot afford the cost of maintaining active replicas.

   Although PASSIVE replication is desirable in resource-constrained systems, it is challenging to deliver soft real-time performance for applications based on PASSIVE replication. Specifically, the middleware [7, 104, 120, 128, 174] implementing PASSIVE replication schemes requires replica recovery decisions (such as per-replica failover targets) to be configured statically at deployment-time so that replica recovery from failure can be quick and appropriate. To configure the appropriate replica

recovery decisions, per-replica node allocation decisions need to be computed. Such per-replica node allocation decisions need to be computed in a *resource-aware* manner, as *ad hoc* mappings can deliver fault-tolerance, but may not deliver *resource-effective* fault-tolerance with acceptable response time and load.

Determining such per-replica node allocation decisions at runtime is expensive and time-consuming, particularly for closed DRE systems. Instead, the invariant, known, and fixed properties of closed DRE systems, such as the number of applications, their execution patterns, their timeliness and high availability requirements, and their resource constraints should be leveraged to determine such allocation decisions at *deployment-time* rather than at *runtime*. Further, when multiple replicas are hosted for each application and their replica node allocation decisions are computed to provide high availability for DRE systems, additional resources are inherently required.

For closed DRE systems, however, there is often a premium placed on the number of resources used, *e.g.*, $\sim$40% of a vehicle's cost is attributed to electronics [19]. It is therefore necessary to minimize the number of resources utilized while deriving the benefits of replication. Therefore, this dissertation identifies the need for developing efficient passively replicated real-time fault-tolerance solutions that are driven by replica allocation algorithms which incur low resource consumption overhead.

2. **Scalable QoS Provisioning, Deployment, and Configuration of Fault-Tolerant DRE Systems.**

   Although middleware-based fault-tolerance solutions [102, 128] are available for distributed systems, such solutions only deal with issues associated with the complexity of managing replication and failures at runtime. As described above, such solutions do not deal with the orthogonal issues associated with where the applications and their replicas are deployed (as described above, this decision dictates the failure recovery behavior), and how the application-specified CPU and network resources are

appropriately provisioned. In the past, significant research has been conducted in designing and developing general purpose, as well as domain specific, resource management algorithms and mechanisms for DRE systems.

Examples of general purpose resource management algorithms and mechanisms include network quality of service (QoS) mechanisms, such as integrated services (IntServ) [81] and differentiated services (DiffServ) [18], which support a range of network service levels for applications in DRE systems. Similarly, to configure required CPU resources for applications, prior work has focused on resource allocation algorithms [31, 57] that satisfy timing requirements of applications in a DRE system. Further, real-time fault-tolerant task allocation algorithms have focused on both active replication [24, 44, 54, 57] as well as passive replication [15, 53, 114, 125, 153, 171] to simultaneously provide both soft real-time performance and high availability assurances for DRE systems.

Although substantial number of results on QoS mechanisms have been achieved, there is still a significant question to be answered in how applications can avail of these mechanisms to satisfy their requirements. To provide end-to-end QoS for DRE systems, both CPU and network resources need to be provisioned. In the past, applications have conventionally used relatively low-level APIs provided by these QoS mechanisms to provision required resources. However, this forces frequent application source code changes, as different deployments of the same application might have different resource requirements.

Further, both CPU and network resources need to be provisioned together. For example, if a particular deployment of two applications across two different physical hosts do not satisfy the application's network resource requirements, the applications need to be deployed in different hosts. Addressing these limitations requires higher-level integrated CPU and network QoS provisioning technologies that decouple application source code from the variabilities (*e.g.*, different source and destination node

deployments, different QoS requirement specifications) associated with their QoS requirements.

This decoupling enhances application reuse across a wider range of deployment contexts (*e.g.*, different deployment instances each with different QoS requirements), thereby increasing deployment flexibility. Therefore, this dissertation identifies the following as an open issue, which deals with developing a deployment and configuration middleware that can deploy and configure QoS for applications in an integrated and non-intrusive manner.

3. **Resource-aware, Adaptive Fault-tolerance for Open DRE Systems.**

   Current middleware mechanisms [102, 128] configure PASSIVE replication recovery strategies in a static fashion, which allows timely client redirection. However, since the failover targets are chosen statically, and without knowledge of the current system resource availability, client failovers in dynamic environments could cause system pollution, where different and uncorrelated processor failures cause multiple clients to failover to the same processor. This could lead to cascading resource failures thereby seriously affecting the real-time and fault tolerance capabilities of the system.

   Further, current research [1, 56, 75] to provide adaptive fault tolerance do not focus on overload management techniques, which are required to reconfigure the system after a client failover in PASSIVE replication. Lack of such overload management techniques causes severe resource imbalance in the system which leads to inefficient resource usage. Additionally, when both real-time and fault tolerance must be satisfied within the same system, it is rather likely that trade-offs [103] are made during the composition.

   For example, in conditions where overloads cannot be controlled by migration, performance needs to be compromised by operating tasks with implementations which

consume less resources but deliver a performance lower than the possible capacity. However, current fault tolerance solutions do not provide support for applying algorithms for the automatic adaptation of the applications to the changing system conditions. Therefore, this dissertation identifies an open issue, which deals with developing efficient resource management algorithms that can adapt to transient load changes and fluctuating resource availabilities, and manage resources and failures in a passively replicated distributed system, so that application performance is not significantly affected before and after failures.

I.4 Research Approach and Contributions

To address the identified open issues related to the complexity of supporting performance-sensitive distributed applications based on PASSIVE replication, this dissertation develops a comprehensive and novel middleware-based solution. Our solution comprises resource management algorithms at both *deployment-time* (to support closed DRE systems) and at *runtime* (to support open DRE systems) in conjunction with adaptive, and configurable, architectures and mechanisms that together realize the design, deployment, configuration, and adaptation of fault-tolerant DRE systems.

In particular, this dissertation involves a combination of:

- ***Deployment-time Resource-aware Real-time Fault-tolerant Replica Allocation Framework***, which includes a replica allocation engine (DeCoRAM) that uses the timing and availability requirements of a closed DRE system to automatically determine allocation decisions for all applications and their replicas while honoring the real-time, resource minimization, and high availability requirements. This research provides real-time fault-tolerant allocation algorithms that are used to configure failure recovery (*e.g.*, per-replica failover targets) and management (*e.g.*, per-replica state synchronization frequency) behavior in real-time fault-tolerant middleware [22, 102, 128]. Chapter III describes DeCoRAM in detail.

- ***Scalable, Model-driven Deployment and Configuration Middleware***, which includes a domain specific model-driven [138] QoS provisioning engine (NetQoPE) that simplifies resource provisioning for applications by shielding application developers from the complexities of programming the lower-level CPU and network QoS mechanisms. NetQoPE provides mechanisms for application non-intrusive resource requirements specification, allocation, and enforcement. This research helps applications to implement and realize the QoS-specific decisions made by domain-specific resource allocation algorithms [31, 57]. Chapter IV describes NetQoPE in detail.

- ***Adaptive Real-time Fault-tolerant Middleware and Architecture***, which includes a adaptive fault-tolerant middleware (FLARe) with algorithms, architecture, and strategies for providing runtime resource-aware fault-tolerance for DRE applications, and a QoS-aware middleware (SwapCIAO) with application transparent mechanisms for in-place updating of component implementations. FLARe provides capabilities for managing applications and their replicas, and making dynamic fault-tolerance decisions that simultaneously support both performance and high availability requirements of applications. This research helps applications to react to changing system loads and system resource availabilities and maintains both soft real-time performance and high availability by recomputing the failure recovery and management decisions that were configured at deployment-time in middleware [22, 102, 128]. SwapCIAO provides capabilities for in-place updating of components which is useful for providing overload management when multiple implementations of a component that impose different loads on resources are available. Chapter V describes FLARe in detail while Chapter VI describes SwapCIAO.

I.5 Research Contributions

Our research on resource-aware fault-tolerance for DRE systems has resulted in QoS-aware middleware mechanisms that adaptively manage replicated resources in an application transparent manner. The key research contributions of this dissertation are summarized in Table 1.

Category	Benefits
Real-time Fault-tolerant Allocation Framework (DeCoRAM)	1. Provides a novel replica-node mapping algorithm that is (1) real-time aware, *i.e.*, honors application timing deadlines, (2) failure-aware, *i.e.*, handles a user-specified number of multiple processor failures by deploying multiple passive replicas such that each of those replicas can continue to meet client timing needs when processors fail, and (3) resource-aware, *i.e.*, minimizes the number of processors used for replication. 2. Provides a real-time fault-tolerance solution that incurs low (1) *resource consumption overhead*, where application replicas are deployed across processors in a resource-aware manner, and (2) *runtime processing overhead*, where failure recovery decisions are made at deployment-time.
Model-driven QoS Provisioning Engine (NetQoPE)	1. Provides a domain specific modeling language (DSML) for specifying per-application timeliness, network QoS, and high availability requirements. 2. Provides a middleware resource allocation framework that complements theoretical research on resource allocation and enables deployment and configuration of DRE systems 3. Provides a real-time fault-tolerance solution that incurs low *development overhead*, where application developers need not write application-specific code to obtain a real-time fault-tolerance solution.
Adaptive Real-time Fault-tolerant Middleware (FLARe and SwapCIAO)	1. Provides a Load-aware and Adaptive Failover (LAAF) strategy that adapts failover targets based on system load 2. Provides a Resource Overload Management Redirector (ROME) strategy that dynamically enforces CPU utilization bounds to maintain desired server delays in face of concurrent failures and load changes 3. Provides an efficient fault-tolerant middleware architecture that supports transparent failover to passive replicas 4. Provides an efficient QoS-aware component middleware that supports application transparent swapping of component implementations

Table 1: Summary Of Research Contributions

I.6 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter II describes the research related to our work on algorithms, architectures, and middleware mechanisms for providing timeliness and high availability assurances to DRE systems and points out the gap in existing research; Chapter III presents a deployment-time resource allocation framework that leverages the ahead-of-time known and invariant properties of closed distributed real-time and embedded (DRE) systems (such as the number of applications together with their timeliness and availability requirements) to ensure real-time and fault-tolerance while minimizing utilized resources; Chapter IV presents a model-driven middleware that shields application developers from the complexities of programming the lower-level CPU and network QoS mechanisms by simplifying activities related to requirements specification, resource allocation, and QoS enforcement, and provides a scalable QoS-aware deployment and configuration middleware for DRE systems; Chapter V presents a fault-tolerant load-aware real-time middleware that adjusts system fault-tolerance configurations at runtime in response to system load fluctuations and resource availability to provide both high availability and timeliness assurances for dynamic DRE systems; Chapter VI presents a lightweight middleware that supports dynamic updating of component implementations for automating the performance management (*e.g.*, overload management) of complex component-based DRE systems; Chapter VII provides a summary of the research contributions, presents concluding remarks and outlines future research work.

CHAPTER II

RELATED WORK

Section I.3 described the urgent need for algorithms, architectures, and mechanisms within middleware that can overcome the disadvantages of a *one-size-fits-all* solution for providing both high availability and soft real-time performance simultaneously for DRE systems. There are three main challenges involved in designing and developing a holistic middleware solution that works together for both closed as well as open DRE systems:

- **Deployment-time Resource-aware Fault-tolerance for DRE Systems.** The middleware should account for the invariant properties of closed DRE systems, such as the number of applications, their execution patterns, their timeliness and high availability requirements, and automatically determine how to configure the middleware real-time fault-tolerance properties (*e.g.*, replica-host mapping to honor timeliness properties and client failover order to honor fault-tolerance properties) to ensure that the required deployment-time assurances for both high availability and timeliness are provided for closed DRE systems.

- **Scalable QoS Provisioning, Deployment, and Configuration of Fault-Tolerant DRE Systems.** The middleware should shield application developers from the low-level complexities of accessing resource allocation algorithms, such as requirements specification, resource allocation, and QoS enforcement, so that application source code development is simple, can just focus on the business logic of the applications, but yet obtain access to resources to satisfy their QoS needs.

- **Resource-aware, Adaptive Fault-tolerance for Open DRE Systems.** The middleware should adapt to the unpredictability of the dynamic environments, obtain current performance characteristics from the system, perform runtime modification

to the allocation of resources to applications they made at deployment-time, and at runtime maintain the simultaneous QoS assurances they provided to applications at deployment-time.

Since our desired middleware solution needs to provide solutions for all the three challenges together, this chapter surveys alternate approaches to solutions for each of those challenges, and discusses limitations of existing approaches in providing these capabilities in a holistic manner.

II.1 Resource-aware Fault-tolerance by Design

Design and deployment mechanisms for fault-tolerance for performance-sensitive systems can be classified along the following dimensions.

Real-time fault-tolerance for transient failures. Prior research has focused on allocation algorithms that consider real-time and fault-tolerance together. For example, transient failures (failures that appear and disappear quickly) are handled in uniprocessor [3, 29, 86, 118, 122, 170] as well as multiprocessor [69, 85] systems. A common theme across all of these research approaches is that failure recovery is done using *time redundancy*, where extra time is reserved in the schedule for potential recovery operations, such as task re-execution within the same processor.

Real-time fault-tolerance for permanent failures. Prior research has focused on real-time fault-tolerant task allocation algorithms that handle permanent failures [24, 44, 54, 57]. All of these approaches have focused on active replication, whose resource consumption overhead is not suitable for certain classes of DRE systems. Prior research has also focused on passively replicated real-time fault-tolerant task allocation algorithms that deal with dynamic scheduling which exhibit extra overheads at runtime [2, 53, 91, 153, 171]. Prior research on static scheduling approaches has also focused on passively replicated real-time fault-tolerant task allocation algorithms that deal with only one processor failure [15, 114, 125].

Real-time fault-tolerant middleware systems. Fault-tolerant middleware has emerged as a core distributed software platform for developing closed DRE systems. For example, MEAD [37, 104, 105], AQUA [128], TMO [74, 75, 76], Delta-4/XPA [12, 96, 120], ARMADA [1, 65, 151, 173, 174], and MARS [77], are fault-tolerant middleware frameworks that provide replication management capabilities in a DRE system.

II.1.1 Unresolved Challenges

The advent of middleware that supports application-transparent passive replication [14, 22, 45] appears to simplify the development of fault-tolerant DRE systems. In practice, however, simultaneously meeting real-time and fault-tolerance requirements is hard due to the need to support fault-tolerance in a resource-aware manner that satisfies soft real-time application requirements [103]. In particular, the following problems must be addressed to deploy and configure (D&C) DRE systems which often become limitations of prior research in this area:

- Application developers must determine how to configure middleware fault-tolerance properties (*e.g.*, replica-host mapping and client failover order) to ensure that DRE system availability and performance requirements are met. *Ad hoc* fault-tolerance configurations can lead to unacceptable response times, overloads, and low-availability applications. Prior research on real-time-, failure- and resource-aware middleware does not address the automatic deployment and configuration of DRE systems with the replica-node mappings.

  In particular, existing solutions focus either on algorithms [24, 44, 57] or on deployment and runtime code generation [32, 58, 147], without considering fault-tolerance as a QoS parameter and automating application deployment and configuration.

- Even when algorithms focus on real-time and fault-tolerance together, often they do not focus on dealing with multiple processor failures while considering passive

replication [20], which is ideal for resource-constrained DRE systems as it reduces resource consumption when compared with active replication [20].

Chapter III describes our approach to provide a deployment-time real-time fault-tolerance solution that addresses these challenges.

II.2 Deployment and Configuration Mechanisms in Middleware

Deployment and configuration of a system is necessary to operationalize the system. Prior work in this area to support fault-tolerance for performance-sensitive systems can be classified along the following dimensions.

Model-based design tools. Model-based design tools provide an intuitive level of abstraction for designing large systems. PICML [10] enables DRE system developers to define component interfaces, their implementations, and assemblies, facilitating deployment of LwCCM-based applications. The *Embedded Systems Modeling Language* (ESML) [70] was developed at the Institute for Software Integrated Systems (ISIS) to provide a visual metamodeling language based on GME that captures multiple views of embedded systems, allowing a diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. Using these analyses, design decisions (such as component allocations to the target execution platform) can be performed.

VEST [148] and AIRES [60] analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provides automated allocation of components to processors. SysWeaver [33] supports design-time timing behavior verification of real-time systems and automatic code generation and weaving for multiple target platforms.

QoS management in middleware. Prior research has focused on adding various types of QoS capabilities to middleware. For example, [66] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [167] provides QoS to applications from varied domains using a component-based

runtime middleware. In addition, [30] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. Synergy [129] describes a distributed stream processing middleware that provides QoS to data streams in real time by efficient reuse of data streams and processing components.

Network QoS management in middleware. Prior work has focused on integrating network QoS mechanisms with middleware. Schantz et al. [126] show how priority- and reservation-based OS and network QoS management mechanisms can be coupled with standards-based middleware to better support distributed systems with stringent end-to-end requirements. Gendy et al. [40, 41] intercept application remote communications by adding middleware modules at the operating system kernel space and dynamically reserve network resources to provide network QoS for the application remote invocations.

Schantz et al. [135] intercept application remote communications by using middleware proxies and provide network QoS for application remote communications by using both DiffServ and IntServ network QoS mechanisms. Yemini et al. [166] focused on providing middleware APIs to shield applications from directly interacting with complex network QoS mechanism APIs. Middleware frameworks transparently converted the specified application QoS requirements into lower-level network QoS mechanism APIs and provided network QoS assurances.

Deployment-time resource allocation. Prior work has focused on deploying applications at appropriate nodes so that their QoS requirements can be met. For example, prior work [87, 149] has studied and analyzed application communication and access patterns to determine collocated placements of heavily communicating components. Other research [31, 57] has focused on intelligent component placement algorithms that maps components to nodes while satisfying their CPU requirements.

II.2.1 Unresolved Challenges

Although the network and CPU QoS deployment and configuration mechanisms described above are powerful, it is tedious and error-prone to develop applications that interact directly with low-level QoS mechanism APIs written imperatively in third-generation languages, such as C++ or Java. For example, applications must make multiple invocations on network QoS mechanisms to accomplish key network QoS activities, such as QoS mapping, admission control, and packet marking.

To address part of this problem, middleware-based network QoS provisioning solutions, that were discussed in Section II.2, have been developed that allow applications to specify their coordinates (source and destination IP and port addresses) and per-flow network QoS requirements via higher-level frameworks. The middleware frameworks—rather than the applications—are thus responsible for converting high-level specifications of QoS intent into low-level network QoS mechanism APIs.

Although middleware frameworks alleviate many accidental complexities of low-level network QoS mechanism APIs, they can still be hard to evolve and extend. In particular, the following challenges remain:

1. Application source code changes may be necessary whenever changes occur to the deployment contexts (*e.g.*, source and destination nodes of applications), per-flow requirements, IP packet identifiers, or middleware APIs. This limits application reusability as the same application source code could be used for many different deployment contexts, and each of those deployment contexts could have different network and CPU resource requirements.

2. Applications must explicitly determine the optimal source and destination nodes before they can obtain network performance assurances via the underlying network QoS mechanisms. Otherwise, network resource reservations might be made between two wrong pair of hosts, when applications could be deployed in some other pair of

source and destination nodes. Further, resource allocation backtracking cannot be made, when physical hosts in which the applications are deployed could be changed to try out newer set of network resource allocations to satisfy end-to-end application QoS requirements.

To address the limitations with current approaches described above, therefore, what is needed are higher-level integrated CPU and network QoS provisioning technologies that can completely decouple application source code from the variabilities (*e.g.*, different source and destination node deployments, different QoS requirement specifications) associated with their QoS provisioning needs. This decoupling enhances application reuse across a wider range of deployment contexts (*e.g.*, different instance deployments each with different QoS requirements), thereby increasing deployment flexibility. Chapter IV describes our approach to provide a model-driven QoS provisioning engine that addresses these challenges.

II.3 Resource-aware, Adaptive Fault-tolerance for Open DRE Systems

Prior research in adaptive fault tolerance solutions for open systems can be classified along the following dimensions.

CORBA-based fault-tolerant middleware systems. Prior research has focused on designing fault-tolerant middleware systems using CORBA. A survey of the different architectures, approaches, and strategies using which fault-tolerance capabilities can be provided to CORBA-based distributed applications is presented in [45]. [13] describes a CORBA portable interceptor-based fault-tolerant distributed system using passive replication and extends the interceptors to redirect clients with a static client failover strategy. MEAD [102], FTS [47] and IRL [11] use CORBA portable interceptors to provide fault-tolerance for CORBA-based distributed systems using active replication.

Scheduling algorithms. Fundamental ideas and challenges in combining real-time and fault tolerance are described in [160], where imprecise computations are used to provide

degraded QoS to applications operating in the presence of failures. [56] proposes adaptive fault tolerance mechanisms to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability. The Realize middleware [67] provides dynamic scheduling techniques that observes the execution times, slack, and resource requirements of applications to dynamically schedule tasks that are recovering from failure, and make sure that non-faulty tasks do not get affected by the recovering tasks.

Adaptive passive replication systems. Prior research has focused on adaptive passive replication to reduce delays incurred by conventional passive replication during fault detection, client failover, and fault recovery. For example, IFLOW [22] uses fault-prediction techniques to change the frequency of backup replica state synchronizations to minimize state synchronization during failure recovery. Similarly, MEAD [119] reduces fault detection and client failover time by determining the possibility of a primary replica failure using simple failure prediction mechanisms and redirects clients to alternate servers before failures occur. Other research [72] uses simulation models to analyze multiple checkpointing intervals and their effects on fault recovery in fault-tolerant distributed systems. [49] focuses on an adaptive dependability approach by mediating interactions between middleware and applications to resolve constraint consistencies while improving availability of distributed systems.

Load-aware adaptations of fault-tolerance configurations. Prior research has focused on run-time adaptations of fault-tolerance configurations [36]. For example, the DARX framework [93] provides fault-tolerance for multi-agent software platforms by focusing on dynamic adaptations of replication schemes as well as replication degree. Research performed in AQUA [80] dynamically adapts the number of replicas receiving a client request in an ACTIVE replication scheme so that slower replicas do not affect the response times received by clients. Eternal [68] dynamically changes the locations of active replicas by migrating soft real-time objects from heavily loaded processors to lightly loaded processors, thereby providing better response times for clients.

Quality of service using generic interception frameworks. Other projects have focused on interceptions above the middleware layer to add quality of service (QoS) for applications. For example, QuO [172] weaves in QoS aspects into applications at compile time by wrapping application stubs and skeletons with specialized delegates that can be used for intercepting application requests and replies. The ACT project [133] provides response to unanticipated behavior in applications by weaving adaptive code into ORBs at runtime and provides fine-grained adaptations by intercepting application requests and replies. CQoS [62] provides platform-dependent interceptors based on stubs and skeletons, and QoS-specific service components, that work with the interceptors to add QoS like fault-tolerance to applications.

Real-time fault-tolerant systems. Delta-4/XPA [120] provided real-time fault-tolerant solutions to distributed systems by using the semi-active replication model. MEAD [119] and its proactive recovery strategy for distributed CORBA applications can minimize the recovery time for DRE systems. The Time-triggered Message-triggered Objects (TMO) project [75] considers replication schemes such as the primary-shadow TMO replication (PSTR) scheme, for which recovery time bounds can be quantitatively established, and real-time fault tolerance guarantees can be provided to applications. AQUA [79] uses ACTIVE replication to provide both availability and timeliness capabilities for applications, and optimizes the response times for applications by dynamically deciding on the number of replicas executing the request. DARX [94] provides adaptive fault-tolerance for multi-agent software platforms by dynamically changing replication styles in response to changing resource availabilities and application performance.

II.3.1 Unresolved Challenges

Existing solutions for providing fault-tolerance in distributed systems demonstrate that PASSIVE replication has a much simpler programming model than ACTIVE replication, and

has much lesser resource consumption overhead than ACTIVE replication. Hence, PASSIVE replication is naturally suited for providing both performance and high availability for applications - especially those operating in resource-constrained environments.

Although PASSIVE replication is desirable in a resource-constrained environment, it is particularly challenging to support performance-sensitive distributed applications based on PASSIVE replication. Specifically, the following challenges in maintaining acceptable client response times and managing application utilizations while recovering from a failure remain unresolved:

1. After a failover, the client perceived response times will depend on the loads of the processor hosting the new primary. Incorrect client redirections could overload a processor thereby affecting the response time(s) for the redirected client(s) and other clients that were already invoking remote operations on targets hosted on that processor. If a client has multiple backup replica choices for failover, *load-aware failover target* decisions therefore must be made to determine the appropriate backup replica so that application performance is not affected after failure recovery.

2. Workload fluctuations (*e.g.*, deployment of new applications) and client failovers from (possibly multiple) processor failures might result in overloads. *Adaptive* overload management decisions must be made to recover from overloads so that application performance is not affected by fluctuating operating environments.

3. Fault-tolerant middleware should mask failures from clients and transparently redirect them to appropriate alternate servers while supporting the adaptive, load-aware failover and overload management capabilities. As described above, however, load-aware failover target selection and adaptive overload management decisions are needed to maintain high availability and acceptable performance for clients. Fault-tolerant middleware therefore needs to support adaptive failover and overload management decisions so that clients are shielded from failures and system load fluctuations.

Chapter V and Chapter VI describe our approach to provide an adaptive fault-tolerant real-time middleware that addresses these challenges.

CHAPTER III

DEPLOYMENT-TIME RESOURCE-AWARE FAULT-TOLERANCE FOR DRE SYSTEMS

Developing large-scale distributed real-time and embedded (DRE) systems is hard in part due to complex deployment and configuration issues involved in satisfying multiple quality for service (QoS) properties, such as real-timeliness and fault tolerance. Effective deployment requires developing and evaluating a range of task allocation algorithms that satisfy DRE QoS properties while reducing resources usage. Effective configuration requires automated tuning of middleware QoS mechanisms to avoid tedious and error-prone manual configuration.

This chapter makes three contributions to the study of deployment and configuration middleware for DRE systems that satisfy multiple QoS properties. First, it describes a novel task allocation algorithm for passively replicated DRE systems to meet their real-time and fault-tolerance QoS properties while consuming significantly less resources. Second, it presents the design of a strategizable allocation engine that enables application developers to evaluate different allocation algorithms. Third, it presents the design of a middleware-agnostic configuration framework that uses allocation decisions to deploy application components/replicas and configure the underlying middleware automatically on the chosen nodes. These contributions are realized in the DeCoRAM (Deployment and Configuration Reasoning and Analysis via Modeling) middleware. Empirical results on a distributed testbed demonstrate DeCoRAM's ability to handle multiple failures and provide efficient and predictable real-time performance.

The rest of this chapter is organized as follows. Section III.1 introduces the research problem and provides the motivation for our work; Section III.2 describes the fault model

and system model that underlies our work on DeCoRAM; Section III.3 describes the structure and functionality of DeCoRAM; Section III.4 empirically evaluates DeCoRAM in the context of distributed soft real-time applications with varying real-time and fault-tolerance deployment and configuration requirements; Finally, Section III.5 provides a summary of our contributions.

III.1 Introduction

Distributed real-time and embedded (DRE) systems operate in resource-constrained environments and are composed of tasks that must process events and provide soft real-time performance. Examples include shipboard computing environments; intelligence, surveillance and reconnaissance systems; and smart buildings. A second key quality of service (QoS) attribute of these DRE systems is *fault-tolerance* since system unavailability can degrade real-time performance and usability.

Fault-tolerant DRE systems are often built using *active* or *passive* replication [20, 120]. Due to its low resource consumption, passive replication is appealing for soft real-time applications that cannot afford the cost of maintaining active replicas and do not require hard real-time performance [45]. Despite improving availability, however, server replication invariably *increases* resource consumption, which is problematic for DRE systems that place a premium on minimizing the resources used [19].

To address these concerns, DRE systems require solutions that can exploit the benefits of replication, but share the available resources amongst the applications efficiently (*i.e.*, to minimize the number and capacities of utilized resources). These solutions must also provide both timeliness and high availability assurances for applications. For a class of DRE systems that are *closed* (*i.e.*, the number of tasks, their execution patterns, and their resource requirements are known ahead-of-time and are invariant), such solutions may be determined at design-time, which in turn can assure QoS properties at runtime.

The advent of middleware that supports application-transparent passive replication [104,

120, 128, 174] appears promising to provide such design-time QoS solutions for fault-tolerant DRE systems. Unfortunately, conventional passive replication schemes incur two challenges for resource-constrained DRE systems: (1) the middleware must generate the right replica-to-node mappings that meet both fault-tolerance and real-time requirements with a minimum number of nodes, and (2) the replica-to-node mapping decisions and QoS needs must be configured within the middleware. Developers must otherwise manually configure the middleware to host applications, which requires source code changes to applications whenever new allocation decisions are made or existing decisions change to handle new requirements. Due to differences in middleware architectures, these *ad hoc* and manual approaches are neither reusable nor reproducible, so this tedious and error-prone effort must be repeated.

To address the challenges associated with passive replication for DRE systems, this chapter presents a resource-aware deployment and configuration middleware for DRE systems called DeCoRAM (*Deployment and Configuration Reasoning and Analysis via Modeling*). DeCoRAM automatically deploys and configures DRE systems to meet the real-time and fault-tolerance requirements via the following novel capabilities:

- **A resource-aware task allocation algorithm** that improves the current state-of-the-art in integrated passive replication and real-time task allocation algorithms [15, 125, 153, 171] by providing a novel replica-node mapping algorithm called FERRARI (*FailurE, Real-time, and Resource Awareness Reconciliation Intelligence*). The novelty of this algorithm are its simultaneous (1) *real-time awareness*, which honors application timing deadlines, (2) *failure awareness*, which handles a user-specified number of multiple processor failures by deploying multiple passive replicas such that each of those replicas can continue to meet client timing needs when processors fail while also addressing state consistency requirements, and (3) *resource awareness*, which reduces the number of processors used for replication.

- **A strategizable allocation engine** that decouples the deployment of a DRE system

from a specific task allocation algorithm by providing a general framework that can be strategized by a variety of task allocation algorithms tailored to support different QoS properties of the DRE system. The novelty of DeCoRAM's allocation engine stems from its ability to vary the task allocation algorithm used from the feasibility test criteria.

- **A deployment and configuration (D&C) engine** that takes the decisions computed by the allocation algorithm and automatically deploys the tasks and their replicas in their appropriate nodes and configures the underlying middleware appropriately. The novelty of DeCoRAM's D&C engine stems from the design of the automated configuration capability, which is decoupled from the underlying middleware architecture.

DeCoRAM's allocation engine, and the deployment and configuration engine are implemented in ∼10,000 lines of C++. This chapter empirically evaluates the capabilities of DeCoRAM in a real-time Linux cluster to show how its real-time fault-tolerance middleware incurs low (1) *resource consumption overhead*, where application replicas are deployed across processors in a resource-aware manner using the FERRARI algorithm, (2) *runtime processing overhead*, where failure recovery decisions are made at deployment-time, and (3) *development overhead*, where application developers need not write application-specific code to obtain a real-time fault-tolerance solution.

III.2 Problem Definition and System Model

This section defines the problem definition for our work on DeCoRAM in the context of the task and fault system models used.

III.2.1 DRE System Model

Our research focuses on a class of DRE systems where the system workloads and the number of tasks are known *a priori*. Examples include system health monitoring applications found in the automotive domain (*e.g.*, periodic transmission of aggregated vehicle health to a garage) or in industrial automation (*e.g.*, periodic monitoring and relaying of health of physical devices to operator consoles), or resource management in the software infrastructure for shipboard computing. These systems also demonstrate stringent constraints on the resources that are available to support the expected workloads and tasks.

Task model. We consider a set of N long running soft real-time tasks (denoted as $S = \{T_1, T_2, ..., T_N\}$) deployed on a cluster of hardware nodes. Clients access these tasks periodically via remote operation requests: each application T_i is associated with its worst-case execution time (denoted as E_i), its period (denoted as P_i), and its relative deadline (which is equal to its period). On each processor, the rate monotonic scheduling algorithm (RMS) [84] is used to schedule each task and individual task priorities are determined based on their periods. We assume that the networks within this class of DRE systems provide bounded communication latencies for application communication and do not fail or partition.

Fault model. We focus on fail-stop processor failures within DRE systems that prevent clients from accessing the services provided by hosted applications. We use *passive replication* [20] to recover from fail-stop processor failures. In passive replication, only one replica—called the primary—handles all client requests when the application state maintained at the primary replica could change. Since backup replicas are not involved in processing client's requests, their application state must be synchronized with the state of the primary replica. We assume that the primary replica (which executes for worst-case execution time E_i) uses non-blocking remote operation invocation mechanisms, such as asynchronous messaging, to send state update propagations to the backup replica, while immediately returning the response to the client.

Each backup replica of a task T_i is associated with its worst-case execution time for

synchronizing state S_i, which significantly reduces the response times for clients, but supports only "best effort" guarantees for state synchronization. Replica consistency may be lost if the primary replica crashes after it responds to the client, but before it propagates its state update to the backup replicas. This design tradeoff is desirable in DRE systems where state can be reconstructed using subsequent (*e.g.*, sensor) data updates at the cost of transient degradation of services.

III.2.2 Problem Motivation and Research Challenges

The goal of DeCoRAM is to deploy and configure a passively replicated DRE system of N tasks that is tolerant to at most K fail-stop processor failures, while also ensuring that soft real-time requirements are met. To satisfy fault tolerance needs, no two replicas of the same task can be collocated. To satisfy real-time requirements, the system also must remain schedulable. These goals must be achieved while reducing resource utilization. To realize such a real-time fault-tolerant DRE system, a number of research questions arise, which we examine below via an example used throughout the paper.

Consider a sample task set with their individual periods, as shown in Table 2. Assuming

Task	E_i	S_i	P_i	Util
<A1,A2,A3>	20	0.2	50	40
<B1,B2,B3>	40	0.4	100	40
<C1,C2,C3>	50	0.5	200	25
<D1,D2,D3>	200	2	500	40
<E1,E2,E3>	250	2.5	1000	25

Table 2: Sample Ordered Task Set with Replicas

that the system being deployed must tolerate a maximum of two processor failures, two backup replicas of each task are needed as shown. The table also shows the execution times taken by the primary replica, the state synchronization times taken by the backup replicas, and the utilization of a primary replica.

Using bin packing algorithms [26, 34] (*e.g.*, based on first-fit allocation) and ensuring

that no two replicas of the same task are collocated, we can identify the lower and upper bounds on the number of processors required to host the system. For example, Figure 1 shows the placement of the tasks, indicating a lower bound on processors that is determined using a bin packing algorithm when no faults are considered. Figure 2 shows the upper

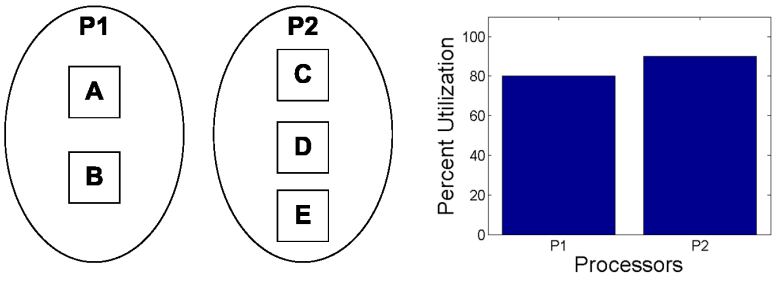

Figure 1: Lower Bound on Processors (No FT Case)

bound on processors needed when the system uses active replication. This case represents an upper bound because in active replication, all replicas contribute WCET. Passive repli-

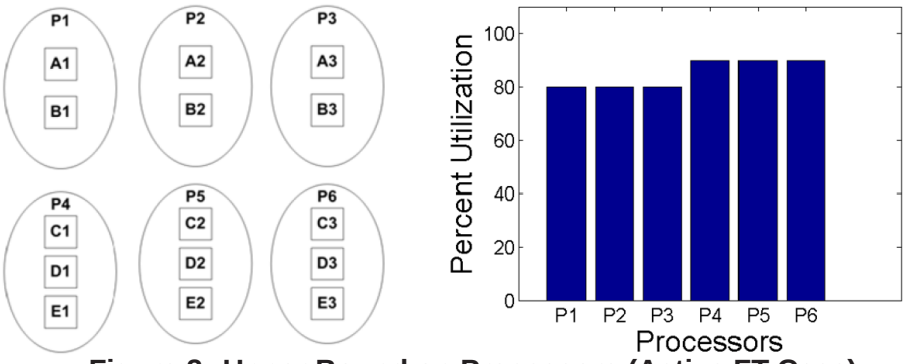

Figure 2: Upper Bound on Processors (Active FT Case)

cation can reduce the number of resources used because the backup replica in a passively replicated system only contributes to the state synchronization overhead. Naturally, the number of processors required for passive replication should be within the range identified above.

Researchers and developers must address the following questions when deploying and configuring DRE systems that must assure key QoS properties:

• How can developers accurately pinpoint the number of resources required?

• Does this number depend on the task allocation algorithm used?

• How can application developers experiment with different allocation algorithms and evaluate their pros and cons?

• How can the results of the allocations be integrated with the runtime infrastructures and how much effort is expended on the part of an application developer?

The three key challenges described below arise when addressing these questions.

Challenge 1: Reduction in resource needs. Since backups contribute to state synchronization overhead, a bin-packing algorithm can pack more replicas, thereby reducing the number of resources used. The resulting packing of replicas, however, is a valid deployment only in no-failure scenarios, which is unrealistic for DRE systems. On failures, some backups will be promoted to primaries (thereby contributing to WCET). Bin packing algorithm cannot identify which backups will get promoted, however, since failures are unpredictable and these decisions are made entirely at runtime. What is needed, therefore, is the ability to identify *a priori* the potential failures in the system and determine which backups will be promoted to primaries so as to determine the number of resources needed. Section III.3.1 describes an algorithm that uses the bounded and invariant properties of closed DRE systems to address this challenge in a design-time algorithm.

Challenge 2: Ability to evaluate different deployment algorithms. An algorithm for task allocation has limited benefit if there is no capability to integrate it with production systems where the algorithm can be executed for different DRE system requirements. Moreover, since different DRE systems may impose different QoS requirements, any one allocation algorithm is often limited in its applicability for a broader class of systems. What is needed, therefore, is a framework that can evaluate different task allocation algorithms

for a range of DRE systems. Section III.3.2 discusses how the DeCoRAM framework evaluates different task allocation algorithms.

Challenge 3: Automated configuration of applications on real-time fault-tolerant middleware. Even after the replica-to-node mappings are determined via task allocation algorithms, these decisions must be enforced within the runtime middleware infrastructure for DRE systems. Although developers often manually configure the middleware, differences in middleware architectures (*e.g.*, object-based vs. component-based vs. service-based) and mechanisms (*e.g.*, declarative vs. imperative) make manual configuration tedious and error-prone. What is needed, therefore, is a capability that can (1) decouple the configuration process from the middleware infrastructure and (2) seamlessly automate the configuration process. Section III.3.3 describes how the DeCoRAM configuration engine automates the configuration process.

III.3 The Structure and Functionality of DeCoRAM

This section presents the structure and functionality of DeCoRAM and shows how it resolves the three challenges described in Section III.2.2.

III.3.1 DeCoRAM's Resource-aware Task Allocation Algorithm

Challenge 1 described in Section III.2.2 is a well-known NP-hard problem [24, 26, 54]. Although this problem is similar to bin-packing problems [26], it is significantly harder due to the added burden of satisfying both fault-tolerance and real-time system constraints. We developed an algorithm called *FailurE, Real-time, and Resource Awareness Reconciliation Intelligence* (FERRARI) presented below to satisfy the real-time and fault-tolerance properties of DRE systems while reducing resource utilization. FERRARI is explained using the sample task set shown in Table 2.

III.3.1.1 Allocation Heuristic

Algorithm 1 describes the design of DeCoRAM's replica allocation algorithm called FERRARI. Line 3 replicates the original task set corresponding to the K fault tolerance requirements, and orders these tuples according to the task ordering strategy (Line 4). For example, to tolerate two processor failures, tasks could be ordered by RMS priorities and the resulting set could contain tasks arranged with tuples from highest priority to lowest as shown in a sample task set of Table 2.

Algorithm 1: Replica Allocation Algorithm

Input:
$T \leftarrow$ set of N tasks to be deployed (not including replicas),
$K \leftarrow$number of processor failures to tolerate,
Output:
Deployment plan $DP \leftarrow$ set of two tuples mapping a replica to a processor,
P_F: resulting set of processors used

```
1  begin
2    Intially, DP = {}, P_F = default set of one processor
3    Let T' ← {< t_ik >}, 1 ≤ i ≤ N, 1 ≤ k ≤ K // Replicate each tasks in T, K times so that T' contains set of N K-tuples
4    Task_Ordering(T') // Order the tasks and replicas
5    foreach tuple_i ∈ T', 1 ≤ i ≤ N do
6      for k = 1 to K do
7        // Allocate a task and all its K replicas before moving to the next
8        Proc_Select: Pick a candidate processor p_c from the set P_F not yet being evaluated for allocation
9        /* Check if allocation is feasible on this processor */
10       bool result = Test_Alloc_for_Feasibility(t_ik, k, p_c, K)
11       if result==false then // Infeasible allocation
12         GoTo Proc_Select for selecting the next candidate processor for this replica
13       else // Update the deployment plan
14         DP ← DP ⋃ {< t_ik, p_c >} // add this allocation
15       if no p_c from set P_F is a feasible allocation then
16         Add a new processor to P_F
17         GoTo Proc_Select// Attempt allocation again with the new set of candidate processors
18       end
19     end
20   end
21 end
```

Lines 5 and 6 show how FERRARI allocates a task and all of its K replicas before allocating the next task. For example, for the set of tasks in Table 2, first all replicas belonging to task A will be allocated followed by B and so on. To allocate each replica, FERRARI selects a candidate processor based on the configured bin-packing heuristic (Line 8). To satisfy fault-tolerance requirements, FERRARI ensures that the processor does not host another replica of the same application being allocated when selecting a candidate processor.

For the candidate processor, FERRARI runs a feasibility test using novel enhancements to the well-known time-demand analysis [84], which is used to test feasibility (see Section III.3.1.2). We chose the time-demand analysis for its accuracy in scheduling multiple tasks in a processor. Although the time-demand analysis method is computationally expensive, it is acceptable since DeCoRAM is a deployment-time solution.

The feasibility criteria evaluates if the replica could be allocated to the processor subject to the specified real-time and fault-tolerance constraints (Line 10). If the test fails for the current processor under consideration, a new candidate processor is chosen. For our sample task set, after deploying task sets A and B along with their replicas (as shown in Figure 3), the next step is to decide a processor for the primary replica of task C. Processor P1 is

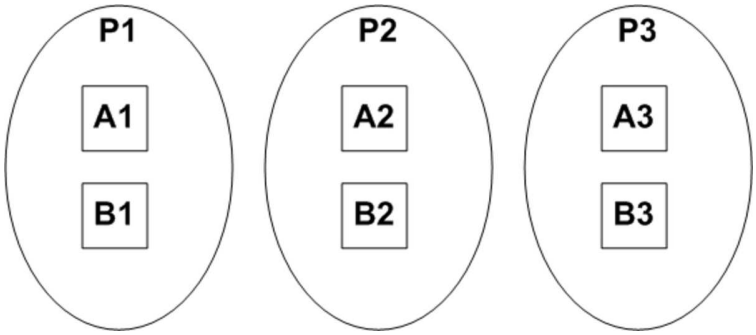

Figure 3: Allocation of Primary and Backup Replicas for Tasks A and B

determined an infeasible solution since the combined utilization on the processor would exceed 100% if C1 were allocated on P1 already hosting A1 and B1 (40+40+25=105).

If a feasible allocation is found, the output deployment plan set *DP* is updated (Line 14). If no candidate processor results in a feasible allocation, however, the set of candidate processors P_F is updated (Line 16) and the replica allocation is attempted again. As shown in Section III.3.1.2, C1 cannot be allocated to any of P1, P2 or P3, thereby requiring an additional processor (as shown in Figure 4). FERRARI completes after allocating all the tasks and its replicas.

III.3.1.2 Failure-Aware Look-Ahead Feasibility Algorithm

Challenge 1 implied exploring the state space for all possible failures in determining the feasible allocations. The time-demand analysis on its own cannot determine this state space. We therefore modify the well-known time-demand function $r_i(t)$ for task T_i in time-demand analysis [84] as follows:

$$r_i(t) = E_i + \left\{ \begin{array}{ll} \sum_{k=1}^{i-1} \lceil \frac{t}{P_k} \rceil E_k & \text{if k is primary} \\ \sum_{k=1}^{i-1} \lceil \frac{t}{P_k} \rceil S_k & \text{if k is backup} \end{array} \right\} for \quad 0 < t < P_i$$

where the tasks are sorted in non-increasing order of RMS priorities. This condition is checked for each task T_i at an instant called the *critical instant phasing* [84], which corresponds to the instant when the task is activated along with all the tasks that have a higher priority than T_i. The task set is feasible if all tasks can be scheduled under the critical instant phasing criteria.

Using this modified definition, we now enhance the feasibility test criteria using the following novel features:

(1) Necessary criteria: "lookahead" for failures. Section III.2.1 explained how a task being allocated can play the role of a primary (which consumes worst case execution time E) or a backup replica (which consumes worst case state synchronization time S). Due to failures, some backups on a processor will get promoted to primaries and because $E >> S$, the time-demand analysis method must consider failure scenarios so that the task allocation is determined feasible in both a non-failure and failure case. For our sample task set, this criteria implies that all possible failure scenarios must be explored for the snapshot shown in Figure 3 when allocating the primary replica for task C (*i.e.*, C1).

For any two processor failure combinations (*e.g.*, the failure of P1 and P2 or P1 and P3), the backups of tasks A and B will be promoted to being primaries. It is therefore no longer feasible to allocate C1 on either P2 or P3 (using the same reasoning that eliminated P1 as a choice). An enhancement to perform such a check must be made available in the

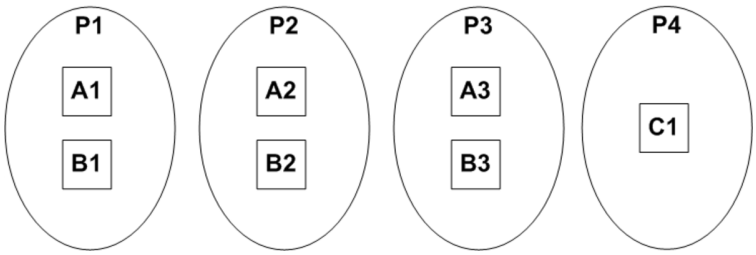

Figure 4: Feasible Allocation for Task C1

time-demand analysis, which then results in an extra processor to host C1, as shown in Figure 4.

(2) Relaxation criteria: assign "failover ordering" to minimize processors utilized. Clause 1 above helps determine the placement of newly considered primaries (*e.g.*, C1). We next address the allocation of backups. One approach is to allocate C2 and C3 on processors P5 and P6 (see Figure 2). This straightforward approach, however, requires the same number of resources used in active replication, which is contrary to the intuition that passive replication utilizes fewer resources.

Using Clause 1, P1 can be eliminated as a choice to host backup C2 since a failure of P4 will make C2 a primary on P1, which is an infeasible allocation. Clause 1 provides only limited information, however, on whether P2 and P3 are acceptable choices to host backups of C (and also those of D and E since they form a group according to the first-fit criteria). We show this case via our sample task set.

Consider a potential feasible allocation in a non-failure case that minimizes resources, as shown in Figure 5. Using Clause 1, we lookahead for any 2-processor failure combinations. If P1 and P2 fail, the allocation is still valid since only A3 and B3 on P3 will be promoted to primaries, whereas C1, D1 and E1 continue as primaries on P4. If P2 and P3 were to fail, the allocation will still be feasible since the existing primaries on P1 and P4 are not affected.

An interesting scenario occurs when P1 and P4 fail. There are two possibilities for how backups are promoted. If the fault management system promotes A2 and B2 on processor

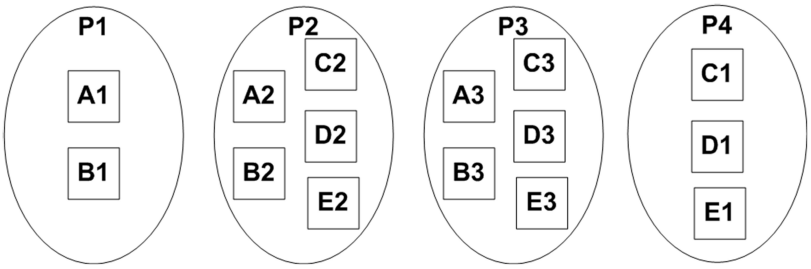

Figure 5: Determining Allocation of Backups of C, D and E

P2, and C3, D3 and E3 on processor P3 to primaries the allocation will still be feasible and there will be no correlation between the failures of individual tasks and/or processors. If the fault management system promotes all of A2, B2, C2, D2 and E2 to primaries on processor P2, however, an infeasible allocation will result. The unpredictable nature of failures and decisions made at runtime is the key limitation of Clause 1.

A potential solution is to have the runtime fault management system identify situations that lead to infeasible allocations and not enforce them. The drawback with this approach, however, is that the number of failure combinations increases exponentially, thereby making the runtime extremely complex and degrading performance as the system scale increases. A complex runtime scheme is unaffordable for closed DRE systems that place a premium on resources. Moreover, despite many properties of closed DRE systems being invariant, the runtime cannot leverage these properties to optimize the performance.

It is possible to overcome the limitation of Clause 1 if the runtime fault management system follows a specific order for failovers. Our algorithm therefore orders the failover of the replicas according to their suffixes, which eliminates the possibility of infeasible allocations at design-time. Naturally, the replica-to-node mapping and hence the time-demand analysis must be enhanced to follow this ordering.

Based on this intuition, even with K processor failures it is unlikely that backups on a live processor will be promoted all at once. In other words, only a subset of backups on a given processor will be promoted in the worst case, without causing an infeasible allocation. The rest of the backups will continue to contribute only S load, which enables

the overbooking of more backup replicas on a processor [53], thereby reducing the number of processors utilized.

These two criteria form the basis of the enhancements we made to the original time-demand analysis, which underpins the feasibility test in our task allocation algorithm FERRARI. Due to space considerations we do not show the feasibility test algorithm itself, but the details are available at www.isis.vanderbilt.edu/sites/default/files/-decoram_tr09.pdf.

Figure 6 shows a feasible allocation determined by FERRARI for the sample set of tasks and their replicas, which reduces the number of resources used and supports real-time performance even in the presence of up to two processor failures.

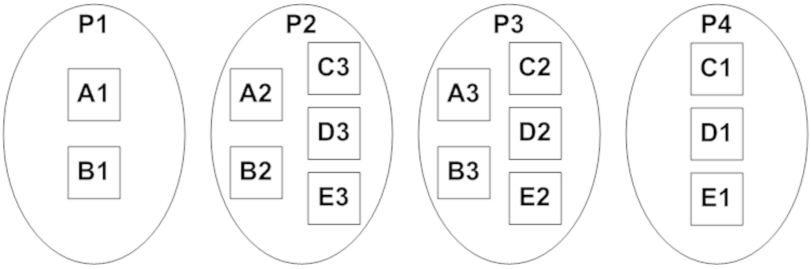

Figure 6: Allocation of Sample Task Set

III.3.1.3 DeCoRAM Algorithm Complexity

We now briefly discuss the complexity of FERRARI. The top-level algorithm (Algorithm 1) comprises an ordering step on Line 4, which results in $O(Nlog(N)$ for N tasks. Allocation decision must then be made for each of the N tasks, their K replicas, and upto M processors if the feasibility test fails for $M-1$ processors.

The overall complexity is thus $O(N*K*M*feasibility\_test)$, where feasibility_test is the failure-aware look-ahead feasibility algorithm described in Section III.3.1.2. Each execution of the feasibility test requires $(1+\binom{P_t}{K})$ executions of the enhanced time-demand

analysis [84]. Since the replica allocation algorithm allocates tasks according to non-increasing RMS priority order, however, the time-demand analysis is not overly costly and can be performed incrementally.

III.3.2 DeCoRAM Allocation Engine

The FERRARI algorithm presented in Section III.3.1 is one of many possible task allocation algorithms that target different QoS requirements of DRE systems. Moreover, it may be necessary to decouple an allocation algorithm from the feasibility test criteria. For example, FERRARI can leverage other schedulability testing mechanisms beyond time-demand analysis. To address these variabilities, Challenge 2 in Section III.2.2 highlighted the need for a framework to evaluate multiple different algorithms that can work with different feasibility criteria.

The DeCoRAM Allocation Engine shown in Figure 7 provides such a framework comprising multiple components, each designed for a specific purpose. DeCoRAM's Alloca-

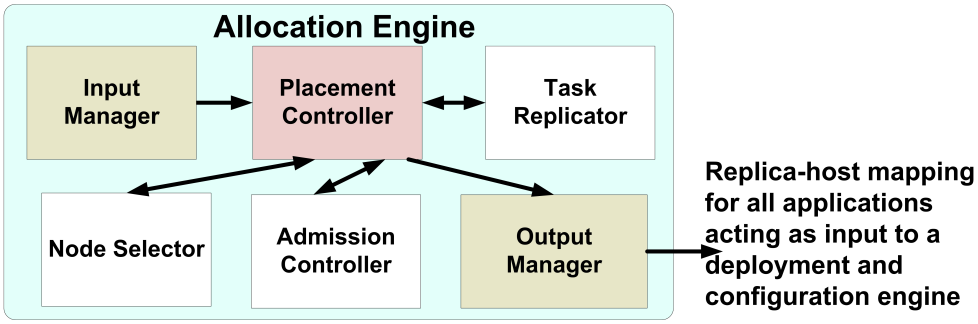

Figure 7: Architecture of the DeCoRAM Allocation Engine

tion Engine is implemented in ∼6,500 lines of C++ and provides a *placement controller component* that can be strategized with different allocation algorithms, including FERRARI (see Section III.3.1). This component coordinates its activities with the following other DeCoRAM components:

1. Input manager. DRE system developers who need to deploy a system with a set of real-time and fault-tolerance constraints express these requirements via QoS specifications that include: (1) the name of each task in the DRE system, (2) the period, worst-case execution time, and worst-case state synchronization time of each task, and (3) the number of processor failures to tolerate. Any technique for gathering these QoS requirements can be used as long as DeCoRAM can understand the information format. For the examples in this paper, we used our CoSMIC modeling tool (`www.dre.vanderbilt.edu/cosmic`), which supplies information to DeCoRAM as XML metadata. The input manager component parses this XML metadata into an in-memory data structure to start the replica allocation process.

2. Node selector. To attempt a replica allocation, the allocation algorithm must select a candidate node, *e.g.*, using efficient processor selection heuristics based on bin-packing [26]. The *node selector component* can be configured to select suitable processors based on first-fit and best-fit bin packing heuristics [88] that reduce the total number of processors used, though other strategies can also be configured.

3. Admission controller. Feasibility checks are required to allocate a replica to a processor. As described above, the goal of DeCoRAM's allocation algorithm is to ensure both real-time and fault-tolerance requirements are satisfied when allocating a replica to a processor. The *admission controller component* can be strategized by a feasibility testing strategy, such as our enhanced time-demand analysis algorithm (see Section III.3.1.2).

4. Task replicator. The *task replicator component* adds a set of K replicas for each task in the input task set and sorts the resultant task set according to a task ordering strategy to facilitate applying the feasibility algorithm by the admission controller component. Since FERRARI uses time-demand analysis [84] for its feasibility criteria, the chosen task ordering strategy is RMS prioritization, with the tasks sorted from highest to lowest rate to facilitate easy application of the feasibility algorithm. Other task ordering criteria also can be used by the task replicator component.

For the closed DRE systems that we focus on in this paper, the output from the DeCoRAM Allocation Engine framework is (1) the replica-to-node mapping decisions for all the tasks and their replicas in the system, and (2) the RMS priorities in which the primary and backup replicas need to operate in each processor. This output format may change depending on the type of algorithm and feasibility criteria used. The output serves as input to the deployment and configuration (D&C) engine (described in Section III.3.3). This staged approach helps automate the entire D&C process for closed DRE systems.

III.3.3 DeCoRAM Deployment and Configuration (D&C) Engine

The replica-to-node mapping decisions must be configured within the middleware, which provides the runtime infrastructure for fault management in DRE systems. Challenge 3 in Section III.2.2 highlighted the need for a deployment and configuration capability that is decoupled from the underlying middleware. This capability improves reuse and decouples the task allocation algorithms from the middleware infrastructure.

The DeCoRAM D&C Engine automatically deploys tasks and replicas in their appropriate nodes and configures the underlying middleware using ∼3,500 lines of C++. Figure 8 shows how this D&C engine is designed using the Bridge pattern [51], which decouples the interface of the DeCoRAM D&C engine from the implementation so that the latter can vary. In our case, any real-time fault-tolerant component middleware can serve as the implementation. By using a common interface, DeCoRAM can operate using various component middleware, such as [104, 128].

The building blocks of DeCoRAM's D&C engine are described below:

• **XML parser.** The *XML parser component* converts the allocation decisions captured in the deployment plan (which is the output of the allocation engine) into in-memory data structures used by the underlying middleware.

• **Middleware deployer.** The *middleware deployer component* instantiates middleware-specific entities on behalf of application developers, including essential building blocks of

any fault tolerance solution, such as the *replication manager*, which manages the replicas; a *per-process monitor*, which checks liveness of a host; and *state transfer agent*, which synchronizes state of primary with backups.

• **Middleware configurator.** The *middleware configurator component* configures the QoS policies of the real-time fault-tolerant middleware to prepare the required operating environment for the tasks that will be deployed. Examples of these QoS policies include thread pools that are configured with appropriate threads and priorities, *e.g.*, RMS priorities for periodic tasks.

• **Application installer.** The *application installer component* installs and registers tasks with the real-time fault-tolerant middleware, *e.g.*, it registers the created object references for the tasks with the real-time fault-tolerant middleware. Often these references are maintained by middleware entities, such as the replication manager and fault detectors. Client applications also may be transparently notified of these object references.

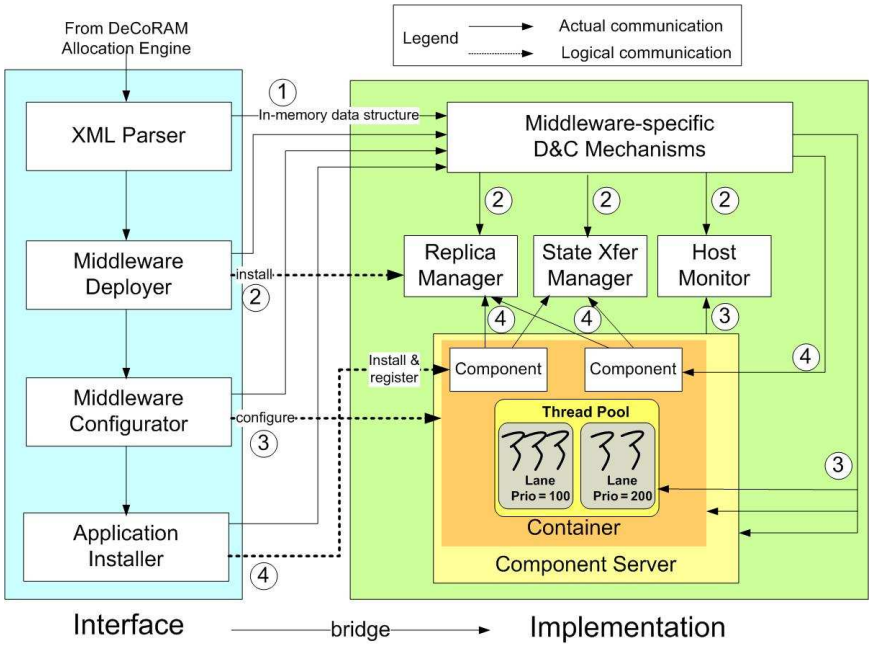

Figure 8: Architecture of the DeCoRAM D&C Engine

DeCoRAM's D&C engine provides two key capabilities: (1) application developers

need not write code to achieve fault-tolerance, as DeCoRAM automates this task for the application developer, and (2) applications need not be restricted to any particular fault-tolerant middleware; for every different backend, DeCoRAM is required to support the implementation of the bridge. This cost is acceptable since the benefits can be amortized over the number of DRE systems that can benefit from the automation.

III.4 Evaluation of DeCoRAM

This section empirically evaluates DeCoRAM along several dimensions by varying the synthetic workloads and the number of tasks/replicas.

III.4.1 Effectiveness of the DeCoRAM Allocation Heuristic

By executing FERRARI on a range of DRE system tasks and QoS requirements, we demonstrate the effectiveness of DeCoRAM's allocation heuristic in terms of reducing the number of processors utilized.

Variation in input parameters.

We randomly generated task sets of different sizes N, where $N = \{10, 20, 40, 80, 160\}$. We also varied the number of failures we tolerated, K, where $K = \{1, 2, 3, 4\}$. DRE systems often consist of hundreds of applications, while passively replicated systems often use 3 replicas, which make these input parameters reflect real-world systems. For each run of the allocation engine, we varied a parameter called *max load*, which is the maximum utilization load of any task in the experiment. Our experiments varied *max load* between 10%, 15%, 20%, and 25%.

For each task in our experiments, we chose task periods that were uniformly distributed with a minimum period of 1 msec and a maximum period of 1,000 msec. After the task period was obtained, each task load was picked at random from a uniformly distributed collection with a minimum task load of 0% up to the specified maximum task load, which determines the worst-case execution times of each task.

We applied a similar methodology to pick the worst-case state synchronization times for all tasks between 1% and 2% of the worst-case execution times of each task. The deadline of each task was set to be equal to its period. Our objective in varying these parameters as outlined above was to understand how effectively DeCoRAM reduces resources and how each input parameter impacts the result.

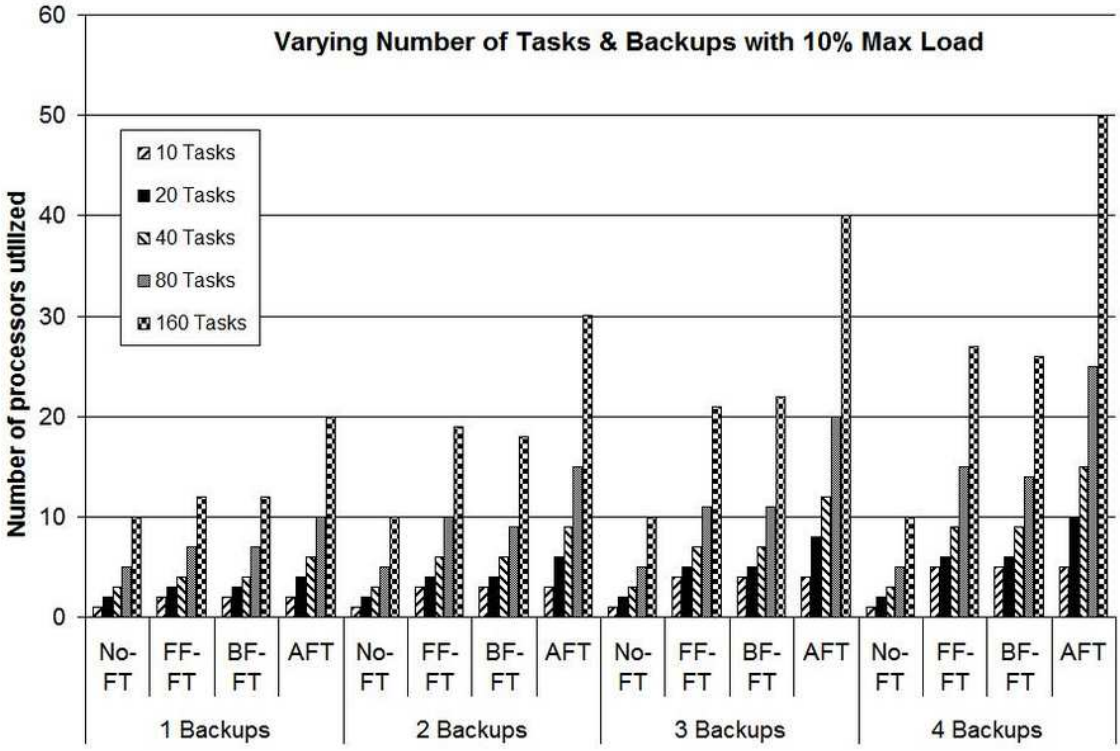

Figure 9: Varying number of tasks with 10% max load

Evaluation criteria. To determine how many resources FERRARI was able to save, we defined two baseline bounds:

• *Lower bound*, where FERRARI determined the lower bound on processors needed by implementing the allocation heuristic [35] that is known to allocate tasks without fault tolerance (No-FT) in the minimal number of processors.

• *Upper bound*, where FERRARI determined the upper bound on number of processors

needed by allocating K replicas for each task using the same heuristic [35] (we make sure that no two replicas of a task are in the same processor). This configuration represents active replication fault-tolerance (AFT) with the minimal number of processors used.

We then strategized FERRARI to use the first-fit (FF-FT) and best-fit (BF-FT) allocation techniques, and computed the number of processors needed. Section III.3.2 showed how the node selector component in the DeCoRAM Allocation Engine can be strategized with these techniques.

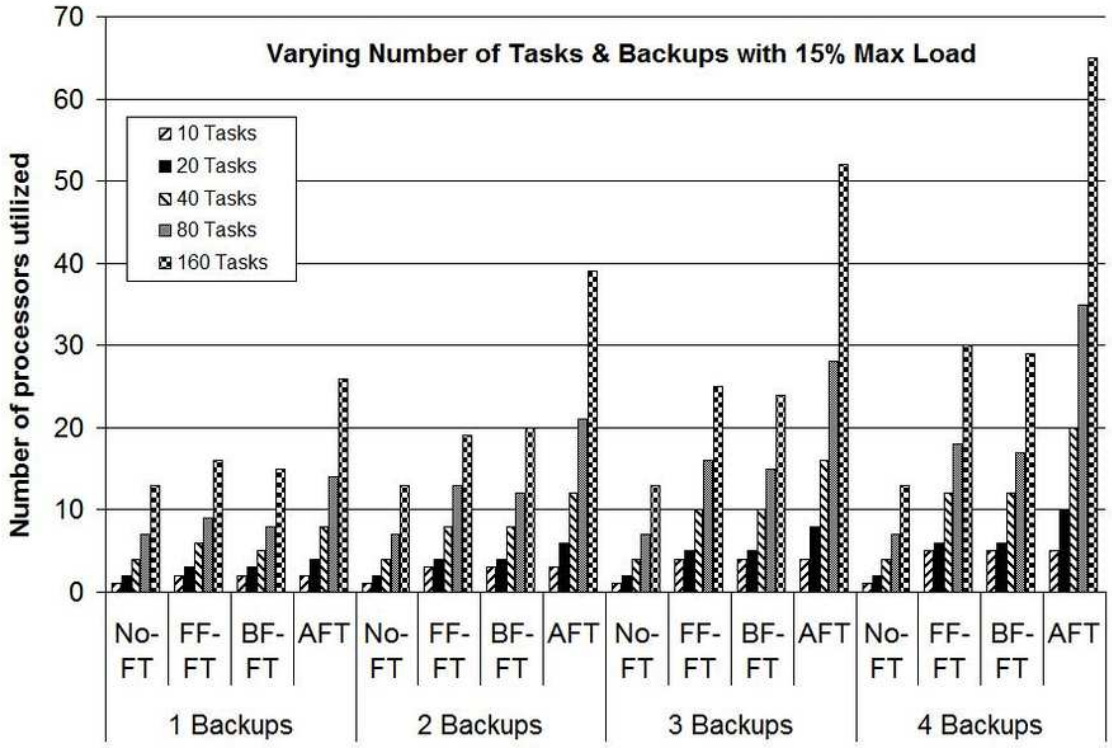

Figure 10: Varying number of tasks with 15% max load

Analysis of results. Figures 9, 10, 11, and 12 show the number of processors used when each of the allocation heuristics attempts to allocate varying number of tasks with varying *max load* for a task set. As N and K increase, the number of processors used also increased exponentially for *AFT*. This exponential increase in processors is due to the behavior of the

active replication scheme, which executes all the replicas to provide fast failure recovery on a processor failure.

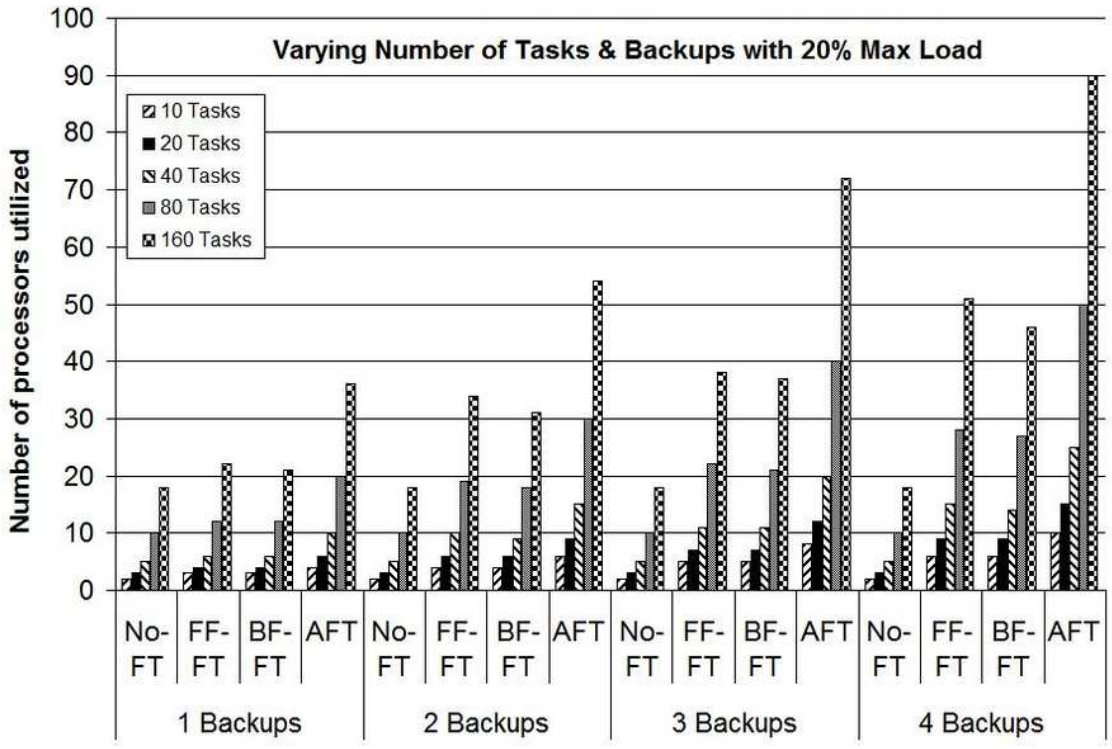

Figure 11: Varying number of tasks with 20% max load

In contrast, when DeCoRAM uses the *FF-FT* or the *BF-FT* allocation heuristics, the rate of increase in number of processors used in comparison with the *No-FT* allocation heuristic is slower compared to *AFT*. For example, when *K* is equal to 1, the number of processors used by both the *FF-FT* and *BF-FT* allocation heuristics is only slightly larger than those used by the *No-FT* allocation heuristics.

As the number of tasks and processor failures to tolerate increases, the ratio of the number of processors used by the *FF-FT* and the *BF-FT* allocation heuristics to those used by the *No-FT* allocation heuristic increases, but at a rate much slower than the increase in the case of *AFT*. Particularly for large *N* as well as *K* (for example, see Figure 12, 160 tasks

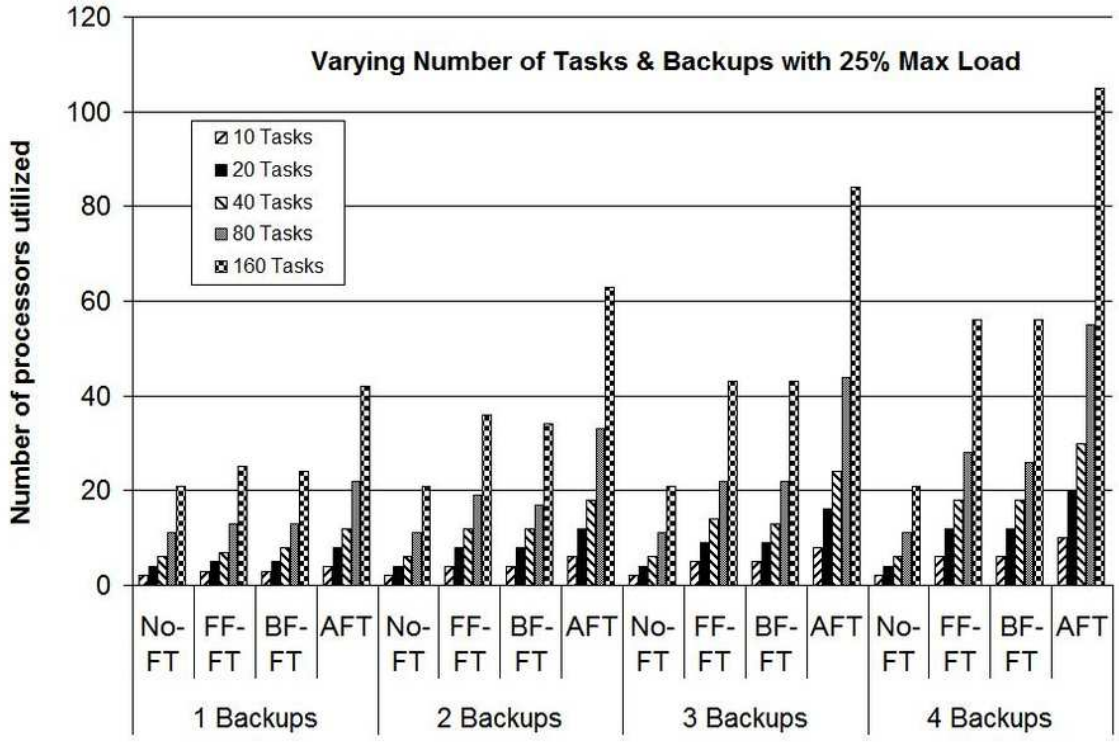

Figure 12: Varying number of tasks with 25% max load

and 4 backups for each task), the number of processors used by the *FF-FT* and the *BF-FT* allocation heuristics is only half the number of processors used by *AFT*.

This result is a direct consequence of the relaxation criteria described in Section III.3.1.2. As the number of tasks to allocate and number of backup replicas increases, the look ahead step finds more opportunities for passive overbooking of backups on a processor for FF-FT and BF-FT allocation heuristics.

III.4.2 Validation of Real-time Performance

We now empirically validate the real-time and fault-tolerance properties of an experimental DRE system task set deployed and configured using DeCoRAM. The experiment was conducted in the ISISlab testbed (`www.dre.vanderbilt.edu/ISISlab`) using 10 blades (each with two 2.8 GHz CPUs, 1GB memory, and a 40 GB disk) and running the

Fedora Core 6 Linux distribution with real-time preemption patches (www.kernel.org-/pub/linux/kernel/projects/rt) for the kernel. Our experiments used one CPU per blade and the blades were connected via a CISCO 3750G switch to a 1 Gbps LAN.

The experimental setup and task allocation follows the model presented in Figure 6 and Table 2. For our experiment we implemented the Bridge pattern [51] in the DeCoRAM D&C engine for our FLARe middleware [7]. Clients of each of the 5 tasks are hosted in 5 separate blades. FLARe's middleware replication manager ran in the remaining blade.

The experiment ran for 300 seconds. We introduced 2 processor failures (processors P1 and P2 in Figure 6) 100 and 200 seconds, respectively, after the experiment was started. We used a fault injection mechanism where server tasks call the *exit()* system call (crashing the process hosting the server tasks) while the clients CLIENT-A or CLIENT-B make invocations on server tasks. The clients receive COMM_FAILURE exceptions and then failover to replicas according to the order chosen by DeCoRAM.

Figure 13 shows the response times observed by the clients despite the failures of 2 processors. As shown by the label A in Figure 13, at 100 seconds when replica A1 fails

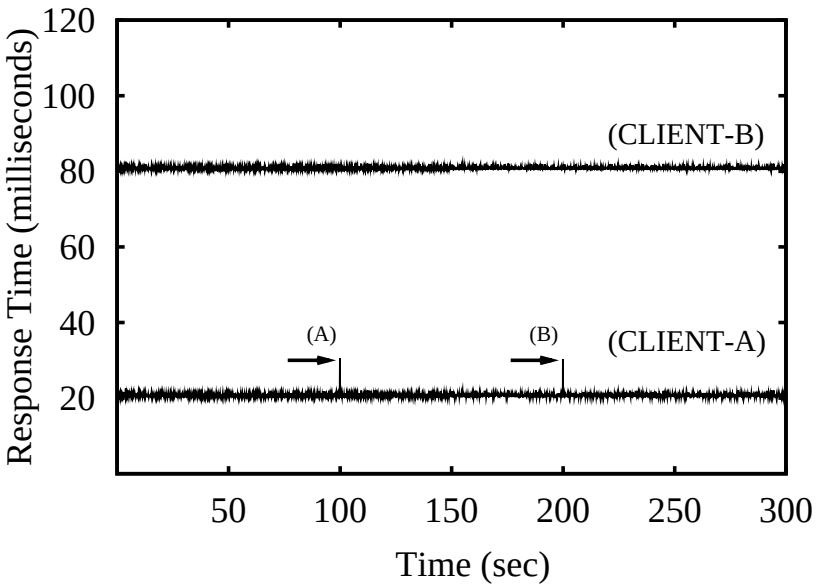

Figure 13: DeCoRAM Empirical Validation

(processor P1 fails, thereby failing B1 as well), client CLIENT-A experiences a momentary increase of 10.6 milliseconds in its end-to-end response time, which is the combined time for failure detection and subsequent failover but stabilizes immediately, thereby ensuring soft real-time requirements. The same behavior is also observed at 200 seconds (see label B) when P2 fails.

These results demonstrate that irrespective of the overbooking of the passive replicas, DeCoRAM can still assure real-time and fault-tolerance for applications.

III.4.3 Evaluating DeCoRAM's Automation Capabilities

We now define a metric that counts the number of steps per deployment and configuration activity to provide a qualitative evaluation of developer effort saved using DeCoRAM. Assuming *N* number of tasks, *K* number of failures to tolerate, and *M* processors needed to host the tasks, Table 3 shows the efforts expended by the developer in conventional approaches versus using DeCoRAM (we assume the use of our FLARe [7] real-time fault-tolerant middleware).

Activity	Effort (Steps Required)	
	Manual	DeCoRAM
Specification	N	N
Allocation	N*(K+1)	0
XML Parsing	1	0
Middleware Deployment	1 + N + 2*M	0
Middleware Configuration	M	0
Application Installation	2*N*(K+1)	0

Table 3: Effort Comparison

The contents of the table are explained below. For *N* tasks, both the conventional and DeCoRAM approaches require developers to specify the QoS requirements. All steps in DeCoRAM are then automated and hence no effort is expended by developers. In contrast, in a manual approach, developers must determine the allocation for $K+1$ replicas (primary and *K* backups) of the *N* tasks followed by one step in parsing the XML output.

Middleware deployment requires one step in deploying the FLARe middleware replication manager, N steps to install the FLARe client request interceptors on the N clients of the servers, and 2 steps each to deploy the FLARe monitor and FLARe state transfer agent on each of the M processors. One step is then necessary to configure the underlying middleware (*e.g.*, setting up thread pools with priorities) on M processors for a total of M steps. Finally, installation of each task requires two steps to register a task with the FLARe middleware replication manager and FLARe state transfer agent for the N tasks with $K+1$ replicas each.

III.5 Concluding Remarks

This paper describes the structure, functionality, and performance of the DeCoRAM deployment and configuration framework, which provides a novel replica allocation algorithm called FERRARI that provides real-time and fault-tolerance to closed DRE systems while significantly reducing resource utilization. DeCoRAM also provides a strategizable allocation engine that is used to evaluate FERRARI's ability to reduce the resources required in passively replicated closed DRE systems. Based on the decisions made by FERRARI, DeCoRAM's deployment and configuration engine automatically deploys application components/replicas and configures the middleware in the appropriate nodes, thereby eliminating manual tasks needed to implement replica allocation decisions. The results from our experiments demonstrate how DeCoRAM provides cost-effective replication solutions for resource-constrained, closed DRE systems.

Below is a summary of lessons learned from our work developing and empirically evaluating DeCoRAM:

- DeCoRAM requires a small number of additional processors to provide fault-tolerance, particularly for smaller number of processor failures to tolerate, *i.e.*, smaller values of K.

- As loads contributed by individual tasks increases, the gains in processor reduction increases when compared with active replication since DeCoRAM exploits the failover order of backup replicas to overbook multiple backup replicas whose ranks are high and whose lower ranked replicas are deployed across different processors.
- The gains seen by FERRARI hold when the state synchronization overhead is a small fraction of the worst case execution time. As the state synchronization overhead approaches 50% or more of the WCET, the reduction seen in processors consumed is no longer attractive, which indicates that such DRE systems may benefit from using active replication.

DeCoRAM is available in open-source format at `www.dre.vanderbilt.edu/~jaiDeCoRAM`.

CHAPTER IV

SCALABLE QOS PROVISIONING, DEPLOYMENT, AND CONFIGURATION OF FAULT-TOLERANT DRE SYSTEMS

Coordinated allocation of both CPU as well as network resources are required by many DRE systems to satisfy their end-to-end QoS requirements. Although CPU QoS mechanisms, such as bin-packing algorithms, and network QoS mechanisms, such as differentiated services (DiffServ), can manage a single resource in isolation, relatively little work has been done on QoS-aware mechanisms for managing multiple heterogeneous resources in a coordinated, integrated, and non-invasive manner to support end-to-end application QoS requirements.

In this chapter, we present two contributions to the study of middleware that supports QoS-aware deployment and configuration of applications in DRE systems. First, we present a model-driven component middleware framework called NetQoPE and describe how it shields applications from the complexities of lower-level CPU and network QoS mechanisms by simplifying (1) the specification of per-application CPU and per-flow network QoS requirements, (2) resource allocation and validation decisions (such as admission control), and (3) the enforcement of per-flow network QoS at runtime. Second, we empirically evaluate how NetQoPE provides QoS assurance for applications in distributed real-time and embedded (DRE) systems. Our results demonstrate that NetQoPE provides flexible and non-invasive QoS configuration and provisioning capabilities by leveraging CPU and network QoS mechanisms without modifying application source code.

The rest of this chapter is organized as follows. Section IV.1 introduces the research problem and provides the motivation for our work; Section IV.2 describes a case study that motivates common requirements associated with provisioning QoS for DRE applications;

Section IV.3 explains how NetQoPE addresses those requirements via its multistage model-driven middleware framework; Section IV.4 empirically evaluates the capabilities provided by NetQoPE in the context of a representative DRE application case study; Finally, Section IV.5 provides a summary of our contributions.

IV.1 Introduction

Emerging trends and limitations. Distributed real-time and embedded systems (DRE), such as smart buildings, high confidence medical devices and systems, and traffic control and safety systems consist of applications that participate in multiple end-to-end application flows, operate in resource-constrained environments, and have varying quality-of-service (QoS) requirements driven by the dynamics of the physical environment in which they operate. For example, smart buildings can host different types of applications with diverse (1) CPU QoS requirements (*e.g.*, personal desktop applications versus fire sensor data analyzers), and (2) network QoS requirements (*e.g.*, transport of e-mails versus transport of security-related information). In such systems, there is a need to allocate CPU and network resources to contending applications subject to the constraints on resources imposed by the physical phenomena (*e.g.*, a fire may partition a set of resources requiring rerouting of network flows).

The QoS provisioning problem is complex due to the need to differentiate applications and application flows at the processors and the underlying network elements, respectively, so that mission-critical applications receive better performance than non-critical applications [100, 136]. Overprovisioning is often not a viable option in cost- and resource-constrained environments where DRE applications deployed, *e.g.* in emerging markets that cannot afford the expense of overprovisioning. DRE application developers must therefore seek effective resource management mechanisms that can efficiently provision CPU and network resources, and address the following two limitations in current research:

Limitation 1: Need for physics-aware integrated allocation of multiple resources.

Prior work has focused predominantly on allocating and scheduling CPU [31, 57] or network resources [18, 81] in isolation. While single resource QoS mechanisms have been studied extensively, little work has focused on coordinated mechanisms that allocate multiple resources, particularly for DRE applications where the coordinated resource management must be aware of the physical dynamics. In the absence of such mechanisms, DRE applications systems may not meet their QoS goals. For example, an application CPU allocation algorithm [31, 158], could dictate multiple placement choices for application(s), but not all placement choices may provide the network *and* CPU QoS because physical limitations may not permit certain allocations (*e.g.*, the placement of a fire sensor impacts its wireless network connectivity to nearby access points). Coordinated mechanisms are therefore needed to allocate CPU and network resources in an integrated manner.

Limitation 2: Need for a non-invasive application-level resource management framework. Even if an integrated, physics-aware multi-resource management framework existed for DRE applications, developers would still incur accidental complexities in using the low-level APIs of the framework. Moreover, application source code changes may be needed whenever changes occur to the deployment contexts (*e.g.*, source and destination nodes of applications), per-flow network resource requirements, per-application CPU resource requirements, or IP packet identifiers.

Middleware frameworks that perform CPU [39, 78, 101, 123, 157] or network [30, 40, 136, 166] QoS provisioning often shield application developers from these accidental complexities. Despite these benefits, DRE applications can still be hard to evolve and extend when the APIs change and middleware evolve. Addressing these limitations requires higher-level integrated CPU and network QoS provisioning technologies that decouple application source code from the variabilities (*e.g.*, different source and destination node deployments, different QoS requirement specifications) associated with their QoS

requirements. This decoupling enhances application reuse across a wider range of deployment contexts (*e.g.*, different deployment instances each with different QoS requirements), thereby increasing deployment flexibility.

Solution approach → Model-driven deployment and configuration middleware for DRE applications. To simplify the development of DRE applications, we developed a multistage, model-driven deployment and configuration framework called *Network QoS Provisioning Engine* (NetQoPE) that integrates CPU and network QoS provisioning via declarative domain-specific modeling languages (DSML) [97]. NetQoPE leverages the strengths of middleware while simultaneously shielding developers from specific middleware APIs. This design allows system engineers and software developers to perform *reusable* deployment-time analysis (such as schedulability analysis [59]) of non-functional system properties (such as CPU and network QoS assurances for end-to-end application flows). The result is enhanced deployment-time assurance that the QoS requirements of DRE applications will be satisfied.

IV.2 Motivating NetQoPE's QoS Provisioning Capabilities

This section presents a case study of a representative DRE application from the domain of smart office environments. We use this case study throughout the chapter to motivate and evaluate NetQoPE's model-driven, middleware-guided CPU and network QoS provisioning capabilities.

IV.2.1 Smart Office Environment Case Study

Smart offices belong to a domain of systems called *Smart Buildings* [146] and showcase state-of-the-art computing and communication infrastructure in its offices and meeting rooms, as shown in Figure 14. Sensors and actuators pervade across a smart office enterprise, and control different functionality within the enterprise.

For example, ventilation and air conditioning systems are controlled by sensors that

monitor and send current room temperatures to an air conditioning service in the command and operations center using the communication infrastructures of the smart office enterprise. The air conditioning service analyzes the sensory data and automatically configures the actuators in response to control room temperatures. In addition to the network traffic associated with the sensors, actuators, and other related embedded systems, the communication infrastructure of a smart office enterprise is also shared by the network traffic associated with the day-to-day enterprise operations of the employees (*e.g.*, e-mail, video conferencing).

Below we describe the cyber physical traits of the smart office environment, focusing on the development and deployment challenges DRE application developers face when ensuring the integration between the cyber and physical aspects of the system.

• *Fire and smoke management.* Detectors are placed in different rooms to send periodic sensory information to a fire and smoke management service. In the event of a fire, this service should activate the sprinkler system in the right places, activate the public address system announcing the right evacuation paths for occupants of the building, and notify external entities, such as fire stations and hospitals of the incident with the right details.

While designing and deploying this capability, developers must ensure the delivery of sensory data to the management service—and the outgoing traffic from this service—is *high priority, i.e.*, it should always obtain the desired CPU and network resources, even though the emergency mode operation (*e.g.*, in the event of a fire) of this service is infrequent. Moreover, sensory and actuation traffic must be reliable. The service should also adapt its policies of routing information to other resources when the current set of resources become unavailable, *e.g.*, due to fire or other adverse event.

• *Security surveillance.* This service uses a feed from cameras and audio sensors in different rooms and performs appropriate audio and video processing to sense physical movements and other intrusions. To notify the security control room, developers must ensure that the input feed from these sensors obtain high bandwidth for their multimedia

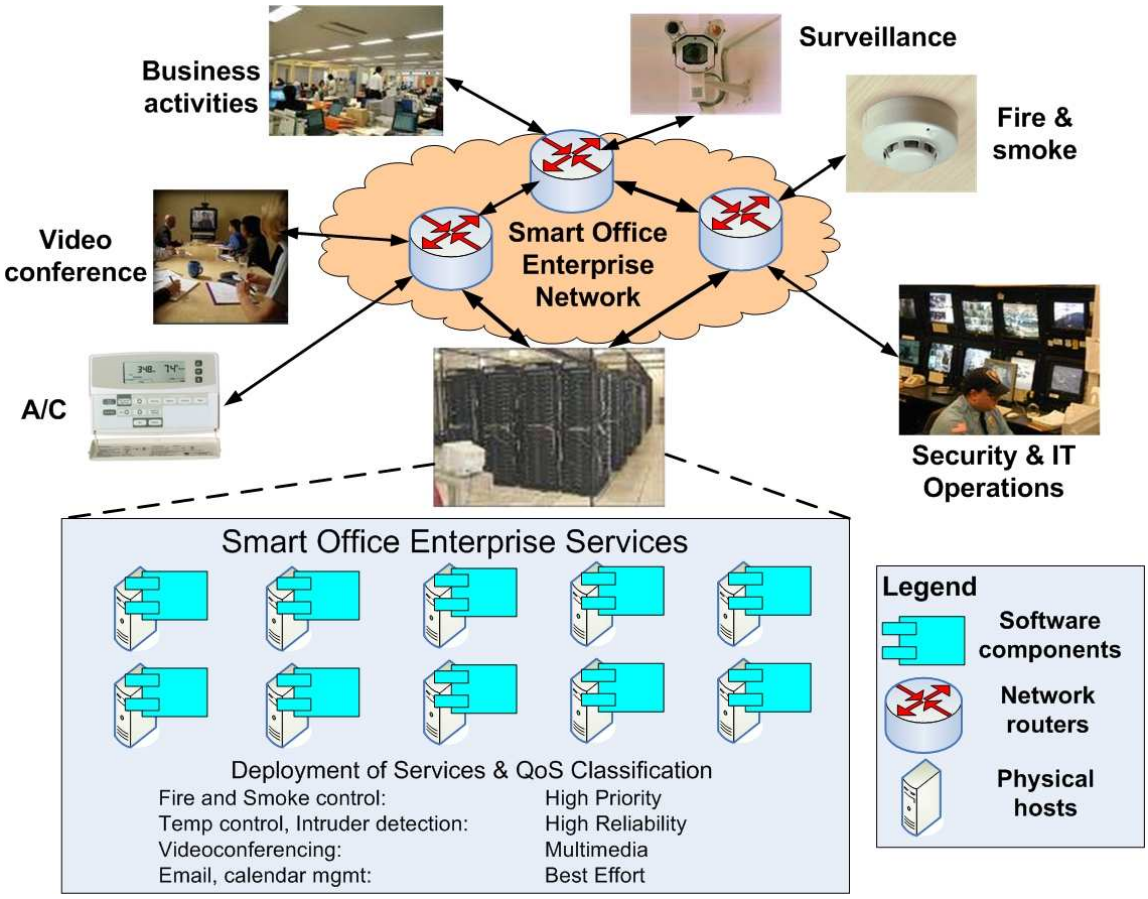

Figure 14: Network Configuration in a Smart Office Environment

traffic, while the outgoing alert notifications and activation of door controls are provided high priority. The image processing task must also be allocated its required CPU resources to perform intrusion detection.

• *Air conditioning and lighting control.* The air conditioning and lighting control service maintains appropriate ambient temperatures and lighting, respectively, in different parts of a building, including business offices, conference rooms and server rooms. It also turns off lights when rooms are not occupied to save energy. This service receives sensory data from thermostats and motion sensors, and controls the air conditioning vents and light switches. This service must be assured reliable transmission of information, though it does not necessarily require high priority.

• *Multimedia video and teleconferencing.* Offices often provide several multimedia-enabled conference rooms to conduct meetings simultaneously. These multimedia conferences require high bandwidth provisioning. A moderator of each meeting submits a request for bandwidth to this service, which must be reliably transmitted to the service. The service in turn must provision the appropriate bandwidth for the multimedia traffic. This service may also need to actuate a public address system informing people of a meeting. Since resources are finite, developers must make tradeoffs and assign this category of public address announcements to the best effort class of traffic, though that announcements about evacuations must be treated with high priority.

• *Email and other web traffic.* Offices also involve a number of other kinds of traffic including email, calendar management, and web traffic. This service must manage these best effort class of traffic on behalf of the people.

IV.2.2 Challenges in Provisioning and Managing QoS in the Smart Office

We now describe the challenges encountered when implementing the QoS provisioning and managing steps described above in the DRE applications that comprise our case study:

• **Challenge 1: Physics-aware QoS requirements specification.** Manually modifying application source code to specify both CPU and network QoS requirements is tedious, error-prone, and non-scalable. In particular, applications could have different resource requirements depending on the physical context in which they are deployed. For example, in our smart office case study, fire sensors have different importance levels (*e.g.*, fire sensors deployed in the parking lot have lower importance than those in the server room). The sensor to monitor flows thus have different network QoS requirements, even though the software controllers managing the fire sensor and the monitor are reusable units of functionality. It may be hard to envision at development time all the contexts in which source code will be deployed; if such information is readily available, application source code can be modified to specify resource requirements for each of those contexts.

The need to know source and destination addresses of an application—coupled with the fact that multiple choices are possible for deploying applications—makes changing application source code to specify resource requirements inflexible and non-scalable. Section IV.3.1 describes how NetQoPE provides a solution to this challenge by providing a domain-specific modeling language (DSML) to support design-time application non-invasive specification of per-application network and CPU QoS requirements.

• **Challenge 2: Application resource allocation.** Manual modifications to source code to reserve resources tightly couple application components with a network QoS mechanism API (*e.g.*, Telcordia's Bandwidth Broker [28]). This coupling complicates deploying the same application component with resources reserved using a different network QoS mechanism API (*e.g.*, GARA Bandwidth Broker [46]). Similarily, source code modifications are also required when the same application is deployed with different network QoS requirements (*e.g.*, requesting more bandwidth on its application flows).

Moreover, network QoS mechanism APIs that allocate network resources require IP addresses for hosts where the resources are allocated. Components that require network QoS must therefore know the physical node placement of the components with which they communicate. This component deployment information may be unknown at development time since deployments are often not finalized until CPU allocation algorithms decide them. Maintaining such deployment information at the source code level or querying it at runtime is unnecessarily complex.

Ideally, network resources should be allocated without modifying application source code and should handle complexities associated with specifying application source and destination nodes, which could vary depending on the deployment context. Section IV.3.2 describes how NetQoPE provides a solution to this challenge by providing a resource allocator framework that supports resource reservation for each application and all its application flows in a non-invasive and transparent manner.

• **Challenge 3: Application QoS configuration.** Application developers have historically written code that instructs the middleware to provide the appropriate runtime services, *e.g.*, DSCP markings in IP packets [136]. Since applications can be deployed in different contexts, modifying application code to instruct the middleware to add network QoS settings is tedious, error-prone, and non-scalable.

Application-transparent mechanisms are therefore needed to configure the middleware to add these network QoS settings depending on the application deployment context. Section IV.3.3 describes how NetQoPE provides a solution to this challenge by providing a network QoS configurator that provides deployment-time configuration of component middleware containers to automatically add flow-specific identifiers to support router layer QoS differentiations.

IV.3 NetQoPE's Multistage Network QoS Provisioning Architecture

This section describes how NetQoPE addresses the challenges from Section IV.2.2 associated with allocating and providing network and CPU QoS in tandem to DRE applications. NetQoPE deploys and configures component middleware-based DRE applications and enforces their network and CPU QoS requirements using the multistage (*i.e.*, design-, pre-deployment-, deployment-, and run-time) architecture shown in Figure 15. NetQoPE's multistage architecture consists of the following elements in the workflow, which automates the task of QoS provisioning for DRE applications.

• The **Network QoS specification language** (NetQoS), which is a DSML that supports design-time specification of per-application CPU resource requirements, as well as per-flow network QoS requirements, such as bandwidth and delay across a flow. NetQoPE uses NetQoS to resolve *Challenge 1* of Section IV.2.2, as described in Section IV.3.1.

• The **Network Resource Allocation Framework** (NetRAF), which is a middleware-based resource allocator framework that uses the network QoS requirements captured by

NetQoS as input at pre-deployment time to help guide QoS provisioning requests on the underlying network and CPU QoS mechanisms at deployment time. NetQoPE uses NetRAF to resolve *Challenge 2* of Section IV.2.2, as described in Section IV.3.2.

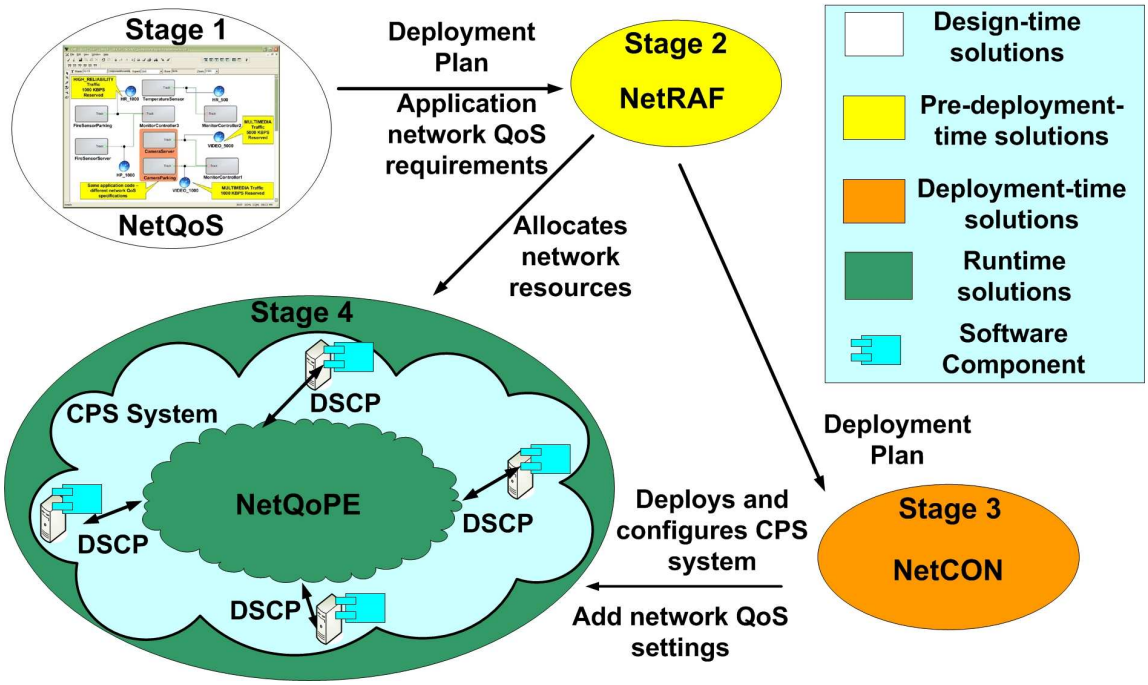

Figure 15: NetQoPE's Multistage Architecture

• The **Network QoS Configurator** (NetCON), which is a middleware-based network QoS configurator that provides deployment-time configuration of component middleware containers. NetCON adds flow-specific identifiers (*e.g.*, DSCPs) to IP packets at runtime when applications invoke remote operations. NetQoPE uses NetCON to resolve *Challenge 3* of Section IV.2.2, as described in Section IV.3.3.

NetQoPE implementation technologies. We developed a prototype of the smart office environment case study using the Lightweight CORBA Component Model [165]. We also used a Bandwidth Broker [28] to allocate per-application-flow network resources using DiffServ network QoS mechanisms. In addition, we used the Generic Modeling Environment (GME) [71] to create domain-specific modeling languages (DSMLs) [10] that

simplify the development and deployment of smart office environment applications (see Appendix A for an overview of all these technologies).

The remainder of this section describes each element in the NetQoPE's multistage architecture and explains how they provide the functionality required to meet the end-to-end QoS requirements of DRE applications. Although the case study in this chapter leverages LwCCM and DiffServ, NetQoPE can be used with other network QoS mechanisms (*e.g.*, IntServ) and component middleware technologies (*e.g.*, J2EE).

IV.3.1 NetQoS: Supporting Physics-aware CPU and Network QoS Requirements Specification

To resolve *Challenge 1* of Section IV.2.2, NetQoPE enables DRE application developers to specify their resource requirements at application deployment-time using a DSML called the *Network QoS Specification Language* (NetQoS). NetQoS is built using the Generic Modeling Environment (GME) [71] and works in concert with the *Platform Independent Component Modeling Language* (PICML) [10]. NetQoS provides applications with an application-independent, declarative (as opposed to application-intrusive [30], middleware-dependent [39], and OS-dependent [95]) mechanism to specify multi-resource requirements simultaneously that can account for the physical context in which the system is deployed.

NetQoS also allows specifying resource requirements as applications are deployed and configured in the target environment. Its declarative mechanisms (1) decouple this responsibility from application source code, and (2) specialize the process of specifying resource requirements for the particular deployment and usecase. Below we describe the steps in using NetQoS' capabilities.

1. Declarative specification of resource requirements. DRE applications developers can use NetQoS to (1) model application elements, such as interfaces, components, connections, and component assemblies, (2) specify CPU utilization of components, and (3)

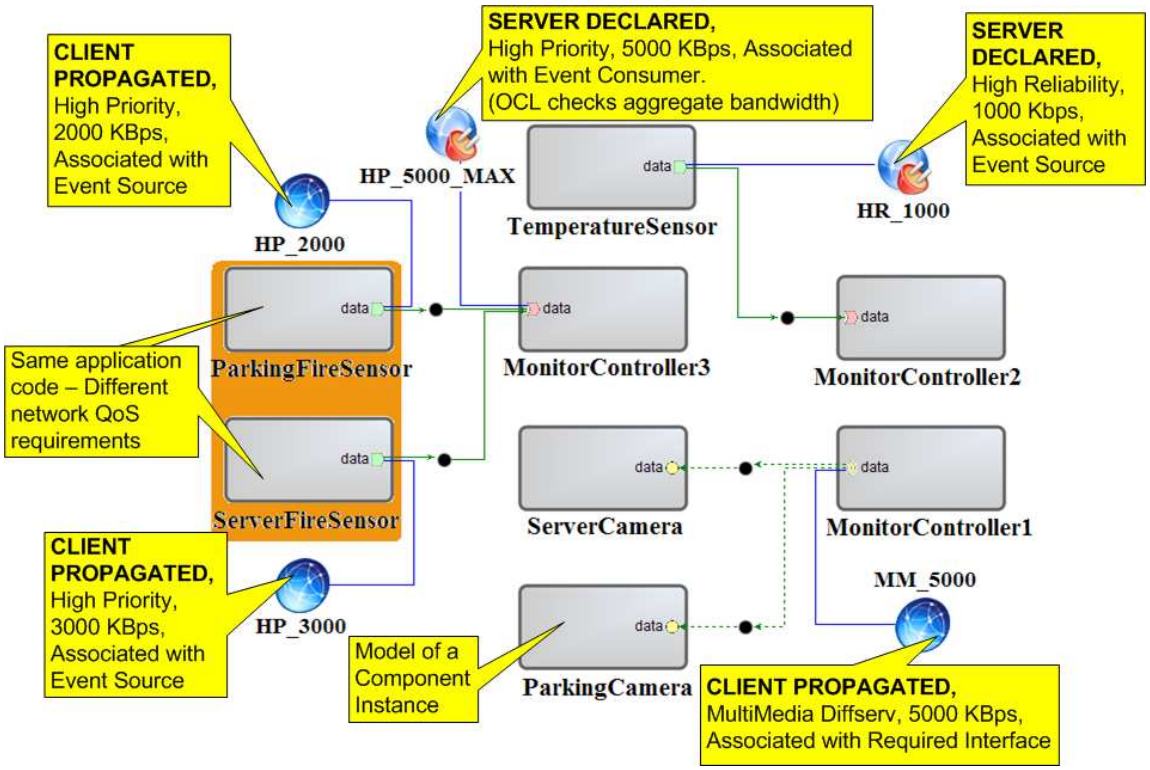

Figure 16: Applying NetQoS Capabilities to the Case Study

specify the network QoS classes, such as HIGH PRIORITY (HP), HIGH RELIABILITY (HR), MULTIMEDIA (MM), and BEST EFFORT (BE), bi-directional bandwidth requirements on the modeled application elements.[1] NetQoS's network QoS classes correspond to the DiffServ levels supported by an underlying network-level resource allocator, such as the Bandwidth Broker [28] we used in our case study.[2] For example, the HP class represents the highest importance and lowest latency traffic (*e.g.*, fire detection reporting in the server room) whereas the HR class represents traffic with low drop rate (*e.g.*, surveillance data). Figure 16 show how NetQoS was used to model the QoS requirements of our case study.

2. Flexible enforcement of network QoS. In certain application flows in the smart office

[1] Middleware such as the Lightweight CORBA Component Model allow components to communicate using *ports* that provide application-level communication endpoints. NetQoS provides capabilities to annotate communication ports with the network QoS requirement specification capabilities.

[2] NetQoS's DSML capabilities can also be extended to provide requirements specification conforming to other network QoS mechanisms, such as IntServ.

case study, (*e.g.*, a monitor requesting location coordinates from a fire sensor) clients control the network priorities at which requests/replies are sent. In other application flows (*e.g.*, a temperature sensor sending temperature sensory information to monitors), the servers control the reception and processing of client requests. If such *design intents* are not captured, applications could potentially misuse network resources at runtime, and also affect the performance of other applications that share the network.

To support both models of communication (*i.e.*, whether clients or servers control network QoS for a flow), NetQoS supports annotating each bi-directional flow using either: (1) the CLIENT_PROPAGATED network priority model, which allows clients to request real-time network QoS assurance even in the presence of network congestion, or (2) the SERVER_DECLARED network priority model, which allows servers to dictate the service that they wish to provide to the clients to prevent clients from wasting network resources on non-critical communication.

NetQoS initiates the allocation of CPU and network resources on behalf of applications by triggering the next stage of the workflow. Section IV.3.3 describes how NetQoPE uses component middleware frameworks at runtime to *realize* the design intent captured by NetQoS and *enforce* network QoS for applications.

3. Early detection of QoS specification errors. Defining network and CPU QoS specifications in source code or through NetQoS is a human-intensive process. Errors in these specifications may remain undetected until later lifecycle stages (such as deployment and runtime) when they are more costly to identify and fix. To identify common errors in network QoS requirement specification early in the development phase, NetQoS uses built-in constraints specified via the OMG Object Constraint Language (OCL) that check the application model annotated with network and CPU priority models.

For example, NetQoS detects and flags specification network resource specification errors, such as negative or zero bandwidth. It also enforces the semantics of network priority models via syntactic constraints in its DSML. For example, the CLIENT_PROPAGATED

model can be associated with ports in the client role only (*e.g.*, required interfaces), whereas the SERVER_DECLARED model can be associated with ports in the server role only (*e.g.*, provided interfaces). Figure 17 shows other examples of network priority models supports by NetQoS.

Network Priority Models of NetQoS		SERVER DECLARED	CLIENT PROPAGATED	Semantics Enforced Using OCL
Application Modeling Elements (ports)	Provided Interface	Allowed	Disallowed	Yes
	Required Interface	Disallowed	Allowed	Yes
	Event Source	Allowed	Disallowed	Yes
	Event Consumer	Disallowed	Allowed	Yes
Network Priority Model Options	Ingress and Egress Bandwidth	Non-zero, positive KB/sec	Non-zero, positive KB/sec	Yes
	Network Level QoS (aggregate checking)	Allowed	Allowed	Yes
	Best Effort QoS (no aggregate checking)	Allowed	Allowed	Yes

Figure 17: Network QoS Models Supported by NetQoS

4. Preparation for allocating CPU and network resources. After a model has been created and checked for type violations using NetQoS's built-in constraints, network resources must be allocated using a network QoS mechanism [28, 46]. As described in Section IV.2.2, this process requires determination of source and destination IP addresses of the applications.

NetQoS allows the specification of CPU utilization requirements of each component and also the target environment where components are deployed. NetQoS's model interpreter traverses CPU requirements of each application component and generates a set of

feasible deployment plans algorithms, such as *first fit*, *best fit*, and *worst fit*, as well as *max* and *decreasing* variants of these algorithms. NetQoS can be used to choose the desired CPU allocation algorithm and to generate the appropriate deployment plans automatically, thereby shielding developers from tedious and error-prone manual component-to-node allocations.

To perform network resource allocations (see Section IV.3.2), NetQoS's model interpreter captures the details about (1) the components, (2) their deployment locations (determined by the CPU allocation algorithms), and (3) the network QoS requirements for each application flow in which the components participate.

Application to the case study. Figure 16 shows a NetQoS model that highlights many capabilities described above. In this model, multiple instances of the same reusable application components (*e.g.*, FireSensorParking and FireSensorServer components) are annotated with different QoS attributes using drag-and-drop.

Our case study has scores of application flows with different client- and server-dictated network QoS specifications, which are modeled using CLIENT_PROPAGATED and SERVER_DECLARED network priority models, respectively. The well-formedness of these specifications are checked using NetQoS's built-in constraints. In addition, the same QoS attribute (*e.g.*, HR_1000 in Figure 16) can be reused across multiple connections, which increases the scalability of expressing requirements for a number of connections prevalent in large-scale DRE applications, such as our smart office environment case study.

NetQoS's ability to plug-in different bin-packing algorithms to determine CPU allocations also decouples applications from the responsibility of manually specifying all possible allocations to allocate network resources. This feature—coupled with NetQoS's declarative mechanisms to specify resource requirements—shields applications (and hence modifications to their source code) from the complexities of QoS specification and allocation. Section IV.4.2 empirically evaluates these capabilities provided by NetQoS.

IV.3.2 NetRAF: Alleviating Complexities in Network Resource Allocation and Configuration

NetQoPE's *Network Resource Allocator Framework* (NetRAF) is a resource allocator engine that allocates network resources for DRE applications using DiffServ network QoS mechanisms, which resolves *Challenge 2* described in Section IV.2.2.. NetRAF allocates network resources for application flows on behalf of the applications (recall how NetQoS invokes NetRAF on behalf of the applications as part of their workflow) and shields applications from interacting with complex network QoS mechanism APIs. To ensure compatibility with different implementations of network QoS mechanisms (*e.g.*, multiple DiffServ Bandwidth Broker implementations [28, 46]), NetRAF uses XML descriptors that capture CPU and network resource requirement specifications (which were specified using NetQoS in the previous stage) in *QoS-independent* manner. These specifications are then mapped to *QoS-specific* parameters depending on the chosen network QoS mechanism. The task of enforcing those QoS specifications are then left to the underlying network QoS mechanism, such as DiffServ, IntServ, and RSVP.

NetRAF provides a clean separation of functionality between resource reservation (provided by NetRAF) and QoS enforcement (done by underlying network elements), as described in the following steps:

1. Network resource allocations. Figure 18 shows how NetRAF's *Network Resource Allocator Manager* accepts application QoS requests at pre-deployment-time. It processes these requests in conjunction with a *DiffServ Allocator*, using deployment specific information (*e.g.*, source and destination nodes) of components and per-flow network QoS requirements embedded in the deployment plan created by NetQoS. This capability shields applications from interacting directly with complex APIs of network QoS mechanisms thereby enhancing the flexibility NetQoPE for a range of deployment contexts. Moreover, since NetRAF provides the capability to request network resource allocations on behalf of components, developers need not write source code to request network resource allocations

for all applications flows, which simplifies the creation and evolution of application logic (see Section IV.4.2).

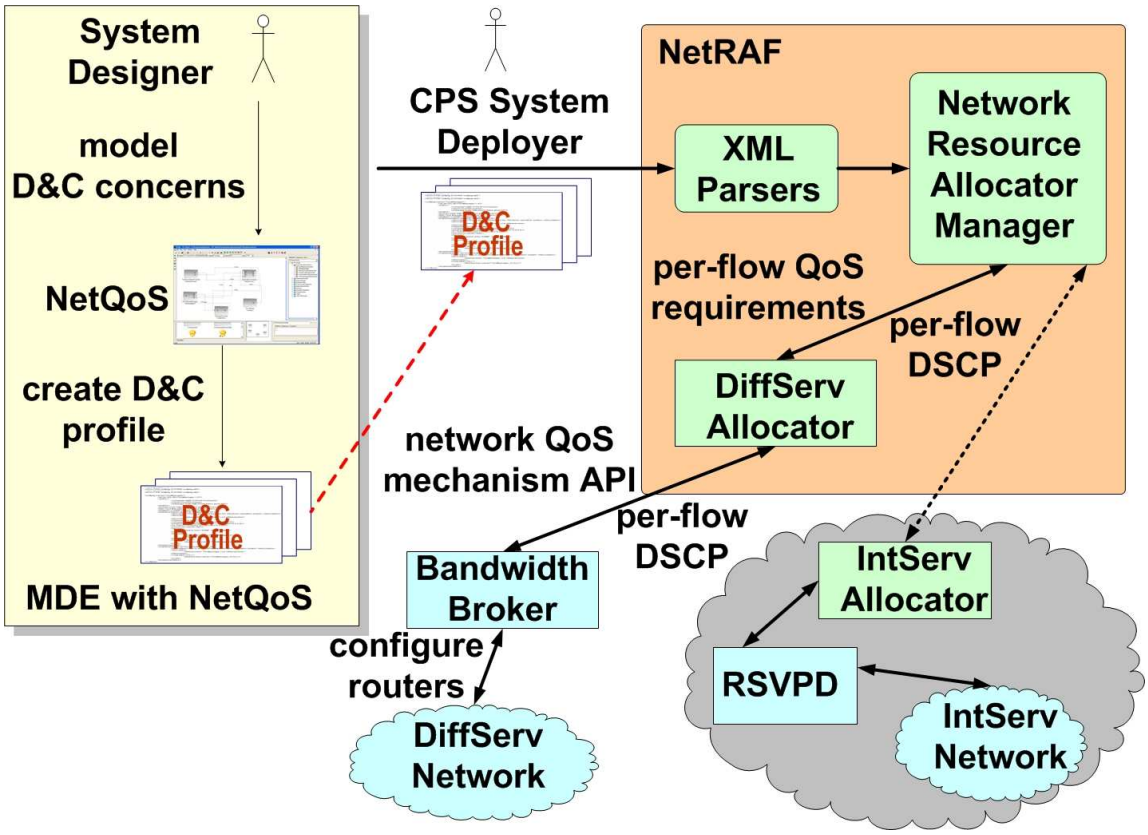

Figure 18: NetRAF's Network Resource Allocation Capabilities

2. Integrated CPU and network QoS provisioning. While interacting with network QoS mechanism specific allocators (*e.g.*, a Bandwidth Broker), NetRAF's Network Resource Allocator Manager may need to handle exceptional conditions, such as infeasible resource allocation errors. Although NetQoS checks the well-formedness of network requirement specifications at application level, it cannot identify every situation that may lead to scenarios with infeasible resource allocations, since these depend on the dynamics of the physical environment.

To handle such scenarios, NetRAF provides hints to regenerate CPU allocations for

components using the CPU allocation algorithm selected by application developers using NetQoS. For example, if network resource allocations fails for a pair of components deployed in a particular source and destination node, NetRAF requests revised CPU allocations by adding a constraint to not deploy the components in the same source and destination nodes. After the revised CPU allocations are computed, NetRAF will (re)attempt to allocate network resources for the components.

NetRAF automates the network resource allocation process by iterating over the set of deployment plans until a deployment plan is found that satisfies both types of requirements (*i.e.*, both the CPU and network resource requirements) thereby simplifying system deployment via the following two-phase protocol: (1) it invokes the API of the QoS mechanism-specific allocator, providing it one flow at a time without actually reserving network resources, and (2) it commits the network resources if and only if the first phase is completely successful and resources for all the flows can be successfully reserved.

This protocol prevents the delay that would otherwise be incurred if resources allocated for a subset of flows must be released due to failures occurring at a later allocation stage. If no deployment plan yields a successful resource allocation, the network QoS requirements of component flows must be reduced using NetQoS.

Application to the case study. Since our case study is based on DiffServ, NetRAF uses its *DiffServ Allocator* to allocate network resources, which in turn invokes the Bandwidth Broker's admission control capabilities [28] by feeding it one application flow at a time. NetRAF's DiffServ Allocator instructs the Bandwidth Broker to reserve bi-directional resources in the specified network QoS classes, as described in Section IV.3.1. The Bandwidth Broker determines the bi-directional DSCPs and NetRAF encodes those values as connection attributes in the deployment plan. This chapter assumes the underlying network QoS mechanism (*e.g.*, the Bandwidth Broker) is responsible for configuring the routers to provide the per-hop behavior [28].

IV.3.3 NetCON: Alleviating Complexities in Network QoS Settings Configuration

NetQoPE's *Network QoS Configurator* (NetCON) resolves *Challenge 3* described in Section IV.2.2 by enabling the auto-configuration of component middleware containers, which provide a hosting environment for application component functionality. Through NetCON auto-configuration, containers can add DSCPs to IP packets when applications invoke remote operations. The current version of NetCON is developed for the LwCCM component middleware and is shown in Figure 19.

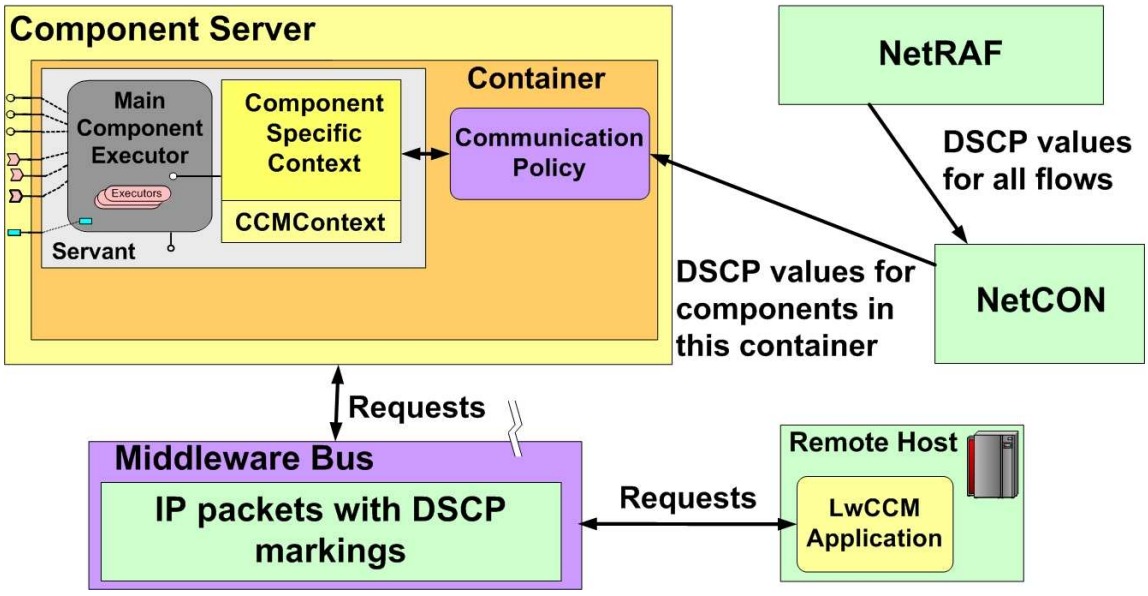

Figure 19: NetCON's Container Auto-configurations

During deployment, NetCON parses the deployment plan (which now includes both the CPU allocations and network DSCP tags for the connections) to determine (1) source and destination components, (2) the network priority model to use for their communication, (3) the bi-directional DSCP values (obtained via NetRAF), and (4) the target nodes on which the components are deployed. NetCON deploys the components on their respective containers and creates the associated object references for use by clients in a remote invocation.

NetCON's container programming model can transparently add DSCPs and enforce the network priority models (see Figure 16). To support the SERVER_DECLARED network priority model, NetCON encodes a SERVER_DECLARED policy and the associated request/-reply DSCPs on the server's object reference. When a client invokes a remote operation with this object reference, the client-side middleware checks the policy on the object reference, decodes the request DSCP, and includes it in the request IP packets. Before sending the reply, the server-side middleware checks the policy again and the reply DSCP is added to the associated IP packets.

To support the CLIENT_PROPAGATED network priority model, NetCON configures the containers to apply a CLIENT_PROPAGATED policy at the point of binding an object reference with the client. In contrast to the SERVER_DECLARED policy, the CLIENT_PROPAGATED policy allows clients to control the network priorities with which their requests and replies traverse the underlying network and different clients can access the servers with different network priorities. When the source component invokes a remote operation using the policy-applied object reference, NetCON adds the associated forward and reverse DSCP markings on the IP packets, thereby providing network QoS to the application flow. A NetQoPE-enabled container can therefore transparently add both forward and reverse DSCP values when components invoke remote operations using the container services.

Application to the case study. In our case study shown in Figure 16, the FireSensor software controller component is deployed in two different instances to control the operation of the fire sensors in the parking lot and the server room. There is a single MonitorController software component (MonitorController3 in Figure 17) that communicates with the deployed FireSensor components. Due to differences in importance of the FireSensor components deployed, however, the MonitorController software component uses CLIENT_PROPAGATED network priority model to communicate with the FireSensor components with different network QoS requirements.

After the first two stages of NetQoPE, NetCON configures the *container* hosting the

MonitorController3 component with the CLIENT_PROPAGATED policy, which corresponds to the CLIENT_PROPAGATED network priority model defined on the component by NetQoS. This capability is provided automatically by containers to ensure that appropriate DSCP values are added at runtime to both forward and reverse communication paths when the MonitorController3 component communicates with either the FireSensorParking or FireSensorServer component. Communication between the MonitorController3 and the FireSensorParking or FireSensorServer components thus receives the required network QoS since NetRAF configures the routers between the MonitorController3 and FireSensorParking components with the source IP address, destination IP address, and DSCP tuple.

NetCON thus allows developers of DRE applications to focus on their application component logic (*e.g.*, the MonitorController component in the case study), rather than wrestling with low-level mechanisms for provisioning network QoS. Moreover, NetCON provides these capabilities without modifying application code, and minimizing runtime overhead thereby simplifying resource provisioning as validated in Section IV.4.4.

IV.4 Empirical Evaluation of NetQoPE

This section empirically evaluates NetQoPE's capabilities to provide CPU and network QoS assurance to end-to-end application flows. We first demonstrate how NetQoPE's model-driven QoS provisioning capabilities can significantly reduce application development effort compared with conventional approaches. We then validate that NetQoPE's automated model-driven approach can provide differentiated network performance for a variety of DRE applications, such as our case study in Section IV.2.

IV.4.1 Evaluation Scenario

Hardware and software testbed. Our empirical evaluation of NetQoPE was conducted on ISISlab (`www.dre.vanderbilt.edu/ISISlab`), which consists of (1) 56 dual-CPU blades running 2.8 GHz XEONs with 1 GB memory, 40 GB disks, and 4 NICs per

blade, and (2) 6 Cisco 3750G switches with 24 10/100/1000 MPS ports per switch. Our experiments were conducted on 15 of dual CPU blades in ISISlab, where (1) 7 blades (A, B, D, E, F, G, and H) hosted our smart office enterprise case study software components (*e.g.*, a fire sensor software controller) and (2) 8 other blades (P, Q, R, S, T, U, V, and W) hosted Linux router software. Figure 20 depicts these details.

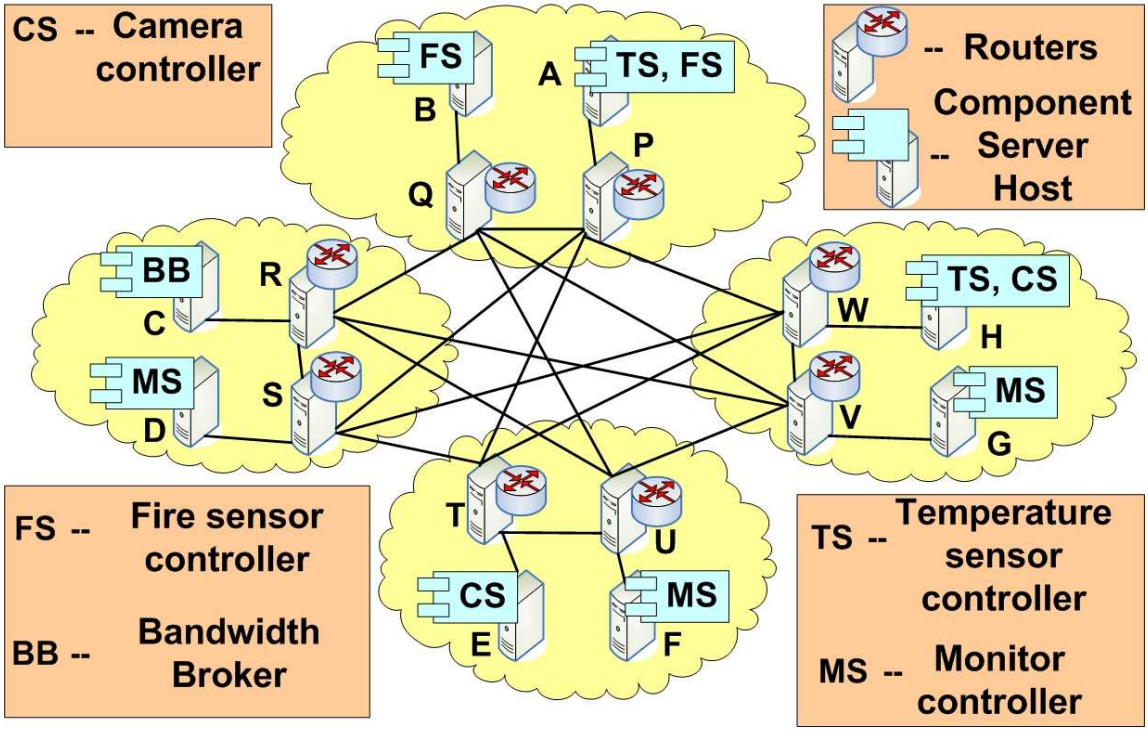

Figure 20: Experimental Setup

The software controller components were developed using the CIAO middleware, which is an open-source LwCCM implementation developed atop the TAO real-time CORBA object request broker [123]. Our evaluations used DiffServ QoS and the associated Bandwidth Broker [28] software was hosted on blade *C*. All blades ran Fedora Core 4 Linux distribution configured using the real-time scheduling class. The blades were connected over a 1 Gbps LAN via virtual 100 Mbps links.

Evaluation scenario. In this scenario six sensory and imagery software controllers sent

their monitored information to three monitor controllers so that appropriate control actions could be performed by enterprise supervisors monitoring abnormal events. For example, Figure 20 shows two *fire sensor controller* components deployed on hosts A and B. These components sent their monitored information to *monitor controller* components deployed on hosts D and F. Each of these software controller components have their own CPU resource requirements and the physical node allocations for those components were determined by the CPU allocation algorithms employed by NetQoS. Further, communication between these software controllers used one of the traffic classes (*e.g.*, HIGH PRIORITY (HP)) defined in Section IV.3.1 with the following capacities on all links: HP = 20 Mbps, HR = 30 Mbps, and MM = 30 Mbps. The BE class used the remaining available bandwidth in the network.

To emulate the CPU and network behavior of the software controllers when different QoS requirements are provisioned, we created the `TestNetQoPE` performance benchmark suite.[3] We used `TestNetQoPE` to evaluate the flexibility, overhead, and performance of using NetQoPE to provide CPU and network QoS assurance to end-to-end application flows. In particular, we used `TestNetQoPE` to specify and measure diverse CPU and network QoS requirements of the different software components that were deployed via NetQoPE, such as the application flow between the *fire sensor controller* component on host A and the *monitor controller* component on host D. These tests create a session for component-to-component communication with configurable bandwidth consumption (components also consume a configurable percentage of CPU resource on their hosted processors). High-resolution timer probes were used to measure roundtrip latency accurately for each client invocation.

[3] `TestNetQoPE` can be downloaded as part of the CIAO open-source middleware available at (`www.dre.vanderbilt.edu/CIAO`).

IV.4.2 Evaluating NetQoPE's Model-driven QoS Provisioning Capabilities

Rationale. This experiment evaluates the effort application developers spend using NetQoPE to (re)deploy applications and provision QoS and compares this effort against the effort needed to provision QoS for applications via conventional approaches.

Methodology. We first identified four flows from Figure 20 whose network QoS requirements are described as follows:

• A fire sensor controller component on host A uses the high reliability (HR) class to send potential fire alarms in the parking lot to the monitor controller component on host D.

• A fire sensor controller component on host B uses the high priority (HP) class to send potential fire alarms in the server room to the monitor controller component on host F.

• A camera controller component on host E uses the multimedia (MM) class and sends imagery information from the break room to the monitor controller component on host G.

• A temperature sensor controller component on host A uses the best effort (BE) class and sends temperature readings to the monitor controller component on host F.

The clients dictated the network priority for requests and replies in all flows *except* for the temperature sensor and monitor controller component flow, where the server dictated the priority. TCP was used as the transport protocol and 20 Mbps of forward and reverse bandwidth was requested for each type of network QoS traffic.

To evaluate the effort saved using NetQoPE, we developed a taxonomy of technologies that provide CPU and network QoS assurances to end-to-end DRE application flows. This taxonomy is used to compare NetQoPE's methodology of provisioning integrated network and CPU QoS for these flows with conventional approaches, including (1) object-oriented [40, 136, 166], (2) aspect-oriented [38], and (3) component middleware-based [30, 144] approaches.

Below we describe how each approach provides the following functionality needed to leverage network QoS mechanism capabilities:

• **QoS Requirements specification**. In conventional approaches applications use (1)

middleware-based APIs [40, 166], (2) contract definition languages [136], (3) runtime aspects [38], or (4) specialized component middleware container interfaces [30] to specify QoS requirements. These approaches do not, however, provide capabilities to specify both CPU and network requirements and assume that physical node placement for all components are decided (*i.e.*, applications are already deployed in appropriate hosts) before the network resource allocations are requested using the appropriate APIs. This assumption allows those applications to specify the source and destination IP addresses of the applications when requesting network resources for an end-to-end application flow.

In such approaches, application source code must change whenever the deployment context (*e.g.*, different physical node allocations, component deployment for a different usecase) and the associated QoS requirements (*e.g.*, CPU or network resource requirements) change, which limits reusability. In contrast, NetQoS provides domain-specific, declarative techniques that increase reusability across different deployment contexts and alleviate the need to specify QoS requirements programmatically, as described in Section IV.3.1.

• **Resource allocation**. Conventional approaches require application deployment before their per-flow network resource requirements can be provisioned by network QoS mechanisms. Recall that appropriate hosts for each application is determined by intelligent CPU allocation algorithms [31] before their per-flow network resource requirements can be provisioned by network QoS mechanisms. If the required network resources cannot be allocated for these applications after a CPU allocation decision is made, however, the following steps occur: (1) the applications must be stopped, (2) their source code must be modified to specify new resource requirements (*e.g.*, either source and destination nodes of the components can be changed, forcing application re-deployments as well or for the same pair of source and destination nodes the network resource requirements could be changed, and (3) the resource reservation process must be restarted.

This approach is tedious since applications may be deployed and re-deployed multiple

times, potentially on different nodes. In contrast, NetRAF handles deployment changes via NetQoS models (see Section IV.3.2) at pre-deployment, *i.e.*, *before* applications have been deployed, thereby reducing the effort needed to change deployment topology or application QoS requirements.

• **Network QoS enforcement**. Conventional approaches modify application source code [136] or programming model [30] to instruct the middleware to enforce runtime QoS for their remote invocations. Applications must therefore be designed to handle two different usecases—to enforce QoS and when no QoS is required—thereby limiting application reusability. In contrast, NetCON uses a container programming model that transparently enforces runtime QoS for applications without changing their source code or programming model, as described in Section IV.3.3.

Based on this taxonomy, we now compare the effort required to provision end-to-end QoS to the 4 end-to-end application flows described above using conventional manual approaches vs. the NetQoPE model-driven approach. We decompose this effort across the following general steps: (1) *implementation*, where software developers write code to specify resource requirements and allocate needed resources, (2) *deployment*, where system deployers map (or stop) application components on their target nodes, and (3) *modeling tool use*, where application developers use NetQoPE to model a DRE application structure, specify per-application CPU resource and per-flow network resource requirements, and allocate needed CPU and network resources.

To compare NetQoPE with other conventional efforts, we devised a realistic scenario for the 4 end-to-end application flows described above. In this scenario, three sets of experiments were conducted with the following deployment variants:[4]

• **Baseline deployment**. This variant configured all 4 end-to-end application flows with the CPU and network QoS requirements as described above. The manual effort required using conventional approaches for the baseline deployment involved 10 steps: (1) modify

[4]In each of the experiment variants, we kept the same per-application CPU resource requirements, but varied the network resource requirements for the application flows.

source code for each of the 8 components to specify their QoS requirements (8 implementation steps – note that CPU allocation algorithms were used to determine the appropriate physical node allocations for the applications before network resources were requested for each application flow), (2) deploy all components (1 deployment step), and (3) shutdown all components (1 deployment step).

In contrast, the effort required using NetQoPE involved the following 4 steps: (1) model the DRE application structure of all 4 end-to-end application flows using NetQoS (1 modeling step), (2) annotate QoS specifications on each application and each end-to-end application flow (1 modeling step), (3) deploy all components (1 deployment step – this step also involved allocation of both CPU and network resources for applications using NetRAF's two step allocation process described in Section IV.3.2), and (4) shutdown all components (1 deployment step).

• **QoS modification deployment**. This variant demonstrated the effect of changes in QoS requirements on manual efforts by modifying the bandwidth requirements from 20 Mbps to 12 Mbps for each end-to-end flow. As with the baseline variant above, the effort required using a conventional approach for the second deployment was 10 steps since source code modifications were needed as the deployment contexts changed (in this case the bandwidth requirements changed across 4 different deployment contexts – however, the CPU resource requirements did not change, and hence the application physical node allocations did not change as well).

In contrast, the effort required using NetQoPE involved 3 steps: (1) annotate QoS specifications on each end-to-end application flow (1 modeling step), (2) deploy all components (1 deployment step), and (3) shutdown all components (1 deployment step). Application developers also reused NetQoS' application structure model created for the initial deployment, which helped reduce the required efforts by a step.

• **Resource (re)reservation deployment**. This variant demonstrated the effect of changes in QoS requirements and resource (re)reservations taken together on manual efforts. We

modified bandwidth requirements of all flows from 12 Mbps to 16 Mbps. We also changed the temperature sensor controller component to use the high reliability (HR) class instead of the best effort BE class. Finally, we increased the background HR class traffic across the hosts so that the resource reservation request for the flow between temperature sensor and monitor controller components fails. In response, deployment contexts (*e.g.*, bandwidth requirements, source and destination nodes) were changed and resource re-reservation was performed.

The effort required using a conventional approach for the third deployment involved 13 steps: (1) modify source code for each of the 8 components to specify their QoS requirements (8 implementation steps), (2) deploy all components (1 deployment step), (3) shutdown the temperature sensor component (1 deployment step – note that the resource allocation failed for the component), (4) modify source code of temperature sensor component back to use BE network QoS class (deployment context change) (1 implementation step), (5) redeploy the temperature sensor component (1 deployment step – note that the CPU allocation algorithms were rerun to change physical node allocations), and (6) shutdown all components (1 deployment step).

In contrast, the effort required using NetQoPE for the third deployment involved 4 steps: (1) annotate QoS specifications on each end-to-end application flow (1 modeling step), (2) begin deployment of all components, though NetRAF's pre-deployment-time allocation capabilities determined the resource allocation failure and prompted the NetQoPE application developer to change the QoS requirements (1 pre-deployment step), (3) re-annotate QoS requirements for the temperature sensor component flow (1 modeling step) (4) deploy all components (1 deployment step), and (5) shutdown all components (1 deployment step).

Table 4 summarizes the step-by-step analysis described above. These results show that conventional approaches incurred roughly an order of magnitude more effort than NetQoPE

Approaches	# Steps in Experiment Variants		
	First	Second	Third
NetQoPE	4	3	5
Conventional	10	10	13

Table 4: Comparison of Manual Efforts Incurred in Conventional and NetQoPE Approaches

to provide CPU and network QoS assurance for end-to-end application flows. Closer examination shows that in conventional approaches, application developers spend substantially more effort developing software that can work across different deployment contexts. Moreover, this process must be repeated when deployment contexts and their associated QoS requirements change. In addition, conventional implementations are complex since the requirements are specified directly using middleware [166] and/or network QoS mechanism APIs [81].

Application (re)deployments are also required whenever reservation requests fail. In this experiment only 1 flow required re-reservation and that incurred additional effort of 3 steps. If there are large number of flows—and DRE systems like our case study often have scores of flows—conventional approaches require significantly more effort.

In contrast, NetQoPE's ability to "write once, deploy multiple times for different QoS requirements" increases deployment flexibility and extensibility in environments that deploy many reusable software components. To provide this flexibility, NetQoS generates XML-based deployment descriptors that capture context-specific QoS requirements of applications. For our experiment, communication between fire sensor and monitor controllers was deployed in multiple deployment contexts, *i.e.*, with bandwidth reservations of 20 Mbps, 12 Mbps, and 16 Mbps. In DRE applications such as our case study, however, the same communication patterns between components could occur in many deployment contexts.

For example, the same communication patterns could use any of the four network QoS

Number of communications	Deployment contexts			
	2	5	10	20
1	23	50	95	185
5	47	110	215	425
10	77	185	365	725
20	137	335	665	1325

Table 5: Generated Lines of XML Code

classes (HP, HR, MM, and BE). The communication patterns that use the same network QoS class could make different forward and reverse bandwidth reservations (*e.g.*, 4, 8, or 10 Mbps). As shown in Table 5, NetQoS auto-generates over 1,300 lines of XML code for these scenarios, which would otherwise be handcrafted by application developers. These results demonstrate that NetQoPE's model-driven CPU and network QoS provisioning capabilities significantly reduce application development effort compared with conventional approaches. Moreover, NetQoPE also provides increased flexibility when deploying and provisioning multiple application end-to-end flows in multiple deployment and diverse QoS contexts.

IV.4.3 Evaluating NetQoPE's QoS Customization Capabilities

Rationale. This experiment empirically evaluates the benefits of the the flexibility and decoupling resulting from NetQoPE's multi stage architecture, and whether the DRE applications indeed obtain their required QoS.

Methodology. From Figure 20, the four flows that were described in Section IV.4.2 were modeled with the same set of network and CPU QoS requirements using NetQoS. The CLIENT_PROPAGATED network policy was used for all flows, except for the temperature sensor and monitor controller component flow, which used the SERVER_DECLARED network policy.

We executed two variants of this experiment. The first variant used TCP as the transport protocol and requested 20 Mbps of forward and reverse bandwidth for each type of QoS

traffic. `TestNetQoPE` configured each application flow to generate a load of 20 Mbps and the average roundtrip latency over 200,000 iterations was calculated. The second variant used UDP as the transport protocol and `TestNetQoPE` was configured to make *oneway* invocations with a payload of 500 bytes for 100,000 iterations. We used high-resolution timer probes to measure the network delay for each invocation on the receiver side of the communication.

At the end of the second experiment we recorded 100,000 network delay values (in milliseconds) for each network QoS class. Those network delay values were then sorted in increasing order and every value was subtracted from the minimum value in the whole sample, *i.e.*, they were normalized with respect to the respective class minimum latency. The samples were divided into fourteen buckets based on their resulting values. For example, the 1 ms bucket contained only samples that are <= to 1 ms in their resultant value, the 2 ms bucket contained only samples whose resultant values were <= 2 ms but > 1 ms, etc.

Traffic Type	Background Traffic in Mbps			
	BE	HP	HR	MM
BE (TS - MS)	85 to 100			
HP (FS - MS)	30 to 40		28 to 33	28 to 33
HR (FS - MS)	30 to 40	12 to 20	14 to 15	30 to 31
MM (CS - MS)	30 to 40	12 to 20	14 to 15	30 to 31

Table 6: Application Background Traffic

To evaluate application performance in the presence of background network loads, several other applications were run in both experiments, as described in Table 6 (in this table TS stands for “temperature sensor controller,” MS stands for “monitor controller”, FS stands for “fire sensor controller,” and CS stands for “camera controller”). NetRAF determined the DSCP values which were then enforced in each outgoing packet through the container auto-configuration effected by NetCON.

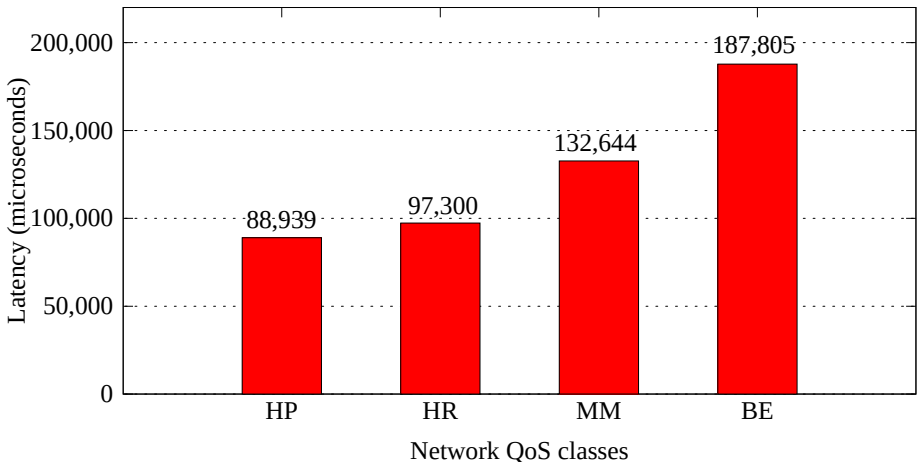

Figure 21: Average Latency under Different Network QoS Classes

Analysis of results. Figure 21 shows the results of experiments when the deployed applications were configured with different network QoS classes and sent TCP traffic. This figure shows that irrespective of the heavy background traffic, the average latency experienced by the fire sensor controller component using the HP network QoS class is lower than the average latency experienced by all other components. In contrast, the traffic from the BE class is not differentiated from the competing background traffic and thus incurs a high latency (*i.e.*, throughput is very low). Moreover, the latency increases while using the HR and MM classes when compared to the HP class.

Figure 22 shows the (1) cardinality of the network delay groupings for different network QoS classes under different ms buckets and (2) losses incurred by each network QoS class. These results show that the jitter values experienced by the application using the BE class are spread across all the buckets, *i.e.*, are highly unpredictable. When combined with packet or invocation losses, this property is undesirable in DRE applications. In contrast, the predictability and loss-ratio improves when using the HP class, as evidenced by the spread of network delays across just two buckets. The application's jitter is almost constant and is not affected by heavy background traffic.

The results in Figure 22 also show that the application using the MM class experienced more predictable latency than applications using BE and HR class. Approximately 94%

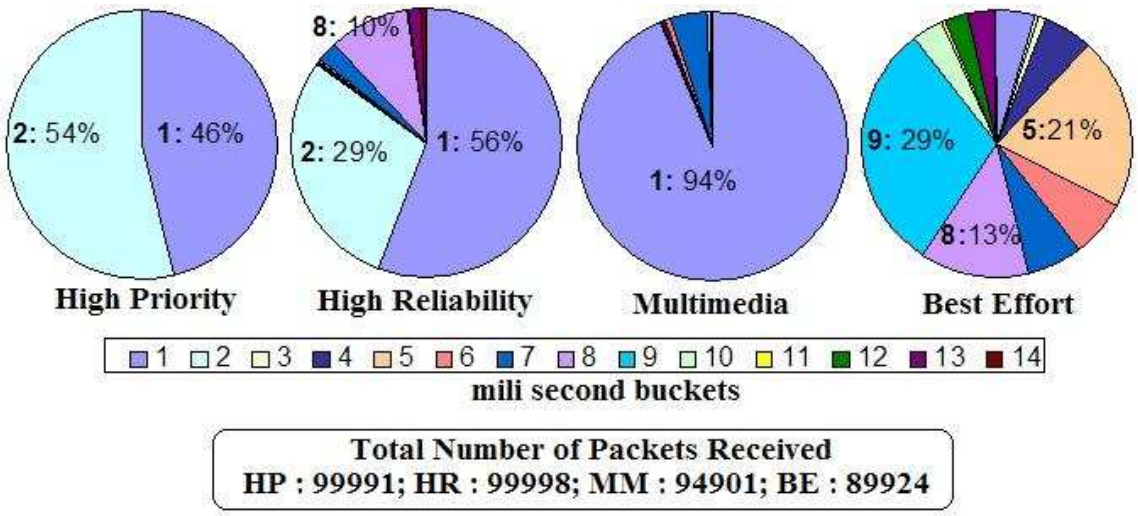

Figure 22: Jitter Distribution under Different Network QoS Classes

of the MM class invocations had their normalized delays within 1 ms. This result occurs because the queue size at the routers is smaller for the MM class than the queue size for the HR class, so UDP packets sent by the invocations do not experience as much queuing delay in the core routers as packets belonging to the HR class. The HR class provides better loss-ratio, however, because the queue sizes at the routers are large enough to hold more packets when the network is congested.

These results demonstrate that NetQoPE can provide significant flexibility and customizability, while ensuring that applications obtain their required QoS.

IV.4.4 Evaluating the Overhead of NetQoPE for Normal Operations

Rationale. This experiment evaluates the runtime performance overhead of using NetQoPE to enforce network QoS.

Methodology. NetCON and NetRAF are design-/deployment-time capabilities that incur no runtime overhead. In contrast, NetCON configures component middleware containers at post-deployment-time by adding DSCP markings to IP packets when applications invoke remote operations (see Section IV.3.3). NetCON may therefore incur runtime overhead,

e.g., when containers apply a network policy models to provide the source application with an object reference to the destination application.

To measure NetCON's overhead, we conducted an experiment to determine the runtime overhead of the container when it performs extra work to apply the policies that add DSCPs to IP packets. This experiment had the following variants: (1) the client container was not configured by NetCON (no network QoS required), (2) the client container was configured by NetCON to apply the CLIENT_PROPAGATED network policy, and (3) the client container was configured by NetCON to apply the SERVER_DECLARED network policy. This experiment had no background network load to isolate the effects of each variant.

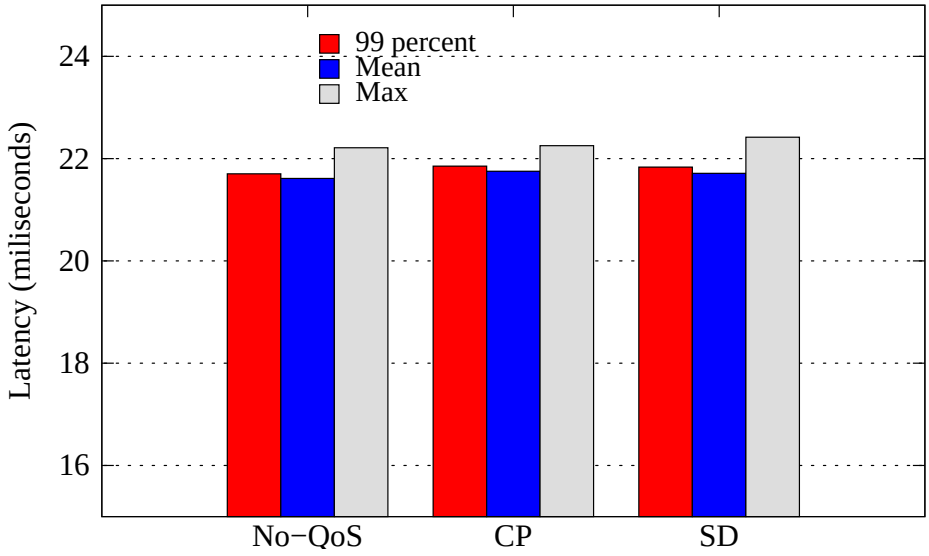

Figure 23: Overhead of NetQoPE's Policy Framework

Our experiment had no network congestion, so QoS support was thus not needed.[5] The network priority models were therefore configured with DSCP values of 0 for both the forward and reverse direction flows. `TestNetQoPE` was configured to make 200,000 invocations that generated a load of 6 Mbps and average roundtrip latency was calculated

[5]Our experimentation goal was to measure the runtime overhead of using NetQoPE middleware to enforce network QoS. So we wanted to remove other effects in the experiment such as network congestion.

for each experiment variant. The routers were not configured to perform DiffServ processing (provide routing behavior based on the DSCP markings), so no edge router processing overhead was incurred. We configured the experiment to pinpoint only the overhead of the container no other entities in the path of client remote communications.

Analysis of results. Figure 23 shows the average roundtrip latencies experienced by clients in the three experiment variants (in this figure CP is the CLIENT_PROPAGATED network priority model and SD is the SERVER_DECLARED model). To honor the network policy models, the NetQoPE middleware added the request/reply DSCPs to the IP packets. The latency results shown in Figure 23 are all similar, which shows that NetCON is efficient and adds negligible overhead to applications. If another variant of the experiment was run with background network loads, network resources will be allocated and the appropriate DSCP values used for those application flows. The NetCON runtime overhead will remain the same, however, since the same middleware infrastructure is used, only with different DSCP values.

IV.5 Summary

This chapter described the design and evaluation of NetQoPE, which is a model-driven middleware framework that manages CPU and network QoS for DRE applications. The lessons we learned developing NetQoPE and applying it to a representative DRE application case study thus far include:

- NetQoPE's domain-specific modeling languages (*e.g.*, NetQoS) help capture per-deployment QoS requirements of applications so that CPU and network resources can be allocated appropriately. Application business logic consequently need not be modified to specify deployment-specific QoS requirements, thereby increasing software reuse and flexibility across a range of deployment contexts, as shown in Section IV.3.1.

- Programming network QoS mechanisms directly in application code requires the deployment and execution of applications before they can determine if the required network

resources are available to meet QoS needs. Conversely, providing these capabilities via NetQoPE's model-driven, middleware framework helps guide resource allocation strategies *before* application deployment, thereby simplifying validation and adaptation decisions, as shown in Section IV.3.2.

• NetQoPE's model-driven deployment and configuration tools help configure the underlying component middleware transparently on behalf of applications to add context-specific network QoS settings. These settings can be enforced by NetQoPE's runtime middleware framework without modifying the programming model used by applications. Applications therefore need not change how they communicate at runtime since network QoS settings can be added transparently, as shown in Section IV.3.3.

• NetQoPE's strategy of allocating network resources before deployment may be too limiting for certain types of DRE applications. In particular, because of the physical nature of the systems, faults might occur at runtime, and applications might not consume all their resource allotment at runtime. Similarily, applications in open systems might require dynamic provisioning of resources based on application demand. Our future work is therefore extending NetQoPE to overprovision resources for applications on the assumption that not all applications will use their allotment.

NetQoPE's model-driven middleware platforms and tools described in this chapter and used in the experiments are available in open-source format from `www.dre.vanderbilt.edu/cosmic` and in the CIAO component middleware available at `www.dre.vanderbilt.edu`. The Bandwidth Broker is a product licensed by Telcordia.

CHAPTER V

RESOURCE-AWARE ADAPTIVE FAULT-TOLERANCE IN DISTRIBUTED SYSTEMS

Supporting uninterrupted services for distributed soft real-time applications is hard in resource-constrained and dynamic environments, where processor or process failures and system workload changes are common. Fault-tolerant middleware for these applications must achieve high service availability and satisfactory response times for client applications. Although passive replication is a promising fault tolerance strategy for resource-constrained systems, conventional client failover approaches are non-adaptive and load-agnostic, which can cause system overloads and significantly increase response times after failure recovery.

In this chapter, we present four contributions to the study of passive replication for distributed soft real-time applications. First, we describe how our Fault-tolerant Load-aware and Adaptive middlewaRe (FLARe) dynamically adjusts failover targets at runtime in response to system load fluctuations and resource availability. Second, we describe how FLARe's overload management strategy proactively enforces desired CPU utilization bounds by redirecting clients from overloaded processors. Third, we present the design and implementation of FLARe's lightweight middleware architecture that manages failures and overloads transparently to clients. Finally, we present experimental results on a distributed Linux testbed that demonstrate how FLARe adaptively maintains soft real-time performance for clients operating in the presence of failures and overloads with negligible runtime overhead.

The rest of this chapter is organized as follows. Section V.1 introduces the research problem and provides the motivation for our work; Section V.2 describes the system and

fault models that form the basis for our work on FLARe; Section V.3 describes the structure and functionality of FLARe; Section V.4 empirically evaluates FLARe in the context of distributed soft real-time applications with dynamic application arrivals and failures; Finally, Section V.5 provides a summary of our contributions.

V.1 Introduction

Distributed real-time middleware, such as Real-time CORBA [113] and Distributed Real-time Java [64], has been used to develop a range of distributed soft real-time applications, such as online stock trading systems and supervisory control and data acquisition (SCADA) systems. Such applications operate in dynamic environments where system loads and resource availabilities fluctuate significantly at runtime due to service request arrivals and processor failures. In such environments, it is important for applications to maintain both system availability and desired soft real-time performance. For example, in SCADA systems for power grid monitoring, remote terminal units must continue to process updates from sensors monitoring power grid failures, even when load fluctuations and failures occur.

ACTIVE and PASSIVE replication [61] are two common approaches for building fault-tolerant distributed applications. In ACTIVE replication [137], client requests are multicast and executed at all replicas. Failure recovery is fast because if any replicas fail, the remaining replicas can continue to provide the service to the clients. ACTIVE replication imposes high communication and processing overhead, however, which may not be viable in resource-constrained systems [22].

In PASSIVE replication [21] only one replica—called the primary—handles all client requests, and backup replicas do not incur runtime overhead, except for receiving state updates from the primary. If the primary fails, a failover is triggered and one of the backups becomes the new primary. Due to its low resource consumption, PASSIVE replication is

appealing for soft real-time applications that cannot afford the cost of maintaining active replicas and need not assure hard real-time performance.

Although PASSIVE replication is desirable in resource-constrained systems, it is challenging to deliver soft real-time performance for applications based on PASSIVE replication. In particular, conventional client failover solutions [13, 45] in PASSIVE replication are non-adaptive and load-agnostic, which can cause post-recovery system overloads and significantly increase response times for clients. Moreover, the middleware must dynamically handle overload conditions caused by workload fluctuations and concurrent failures. Therefore, a lightweight middleware architecture is needed that can handle failures and overloads transparently from the applications.

To address this need, we have developed the *Fault-tolerant, Load-aware and Adaptive middlewaRe (FLARe)* which maintains service availability and soft real-time performance in dynamic environments. This chapter evaluates the following contributions to developing distributed soft real-time applications:

- A **Load-aware Adaptive Failover (LAAF) strategy**, which dynamically adjusts failover targets in response to load fluctuations and processor/process failures based on current CPU utilization.
- A **Resource Overload Management rEdirector (ROME) strategy**, which dynamically enforces schedulable utilization bounds by proactively redirecting clients from overloaded processors.
- A **lightweight adaptive middleware architecture**, which handles failures and overloads transparently from applications.

FLARe has been implemented atop the TAO Real-time CORBA middleware [63, 142]

and evaluated empirically in the ISISlab testbed (www.dre.vanderbilt.edu/ISIS-ab). The experimental results reported in this chapter demonstrate how FLARe can dynamically maintain both system availability and desired soft real-time performance for clients, while incurring negligible run-time overhead.

V.2 System and Fault Models

FLARe supports distributed systems where application servers provide multiple long-running services on a cluster of computing nodes. The services in a system are invoked by clients periodically via remote operation requests. Further, these types of systems experience *dynamic* workloads when clients start and stop services at runtime. Clients demand both soft real-time performance as well as system availability despite workload fluctuations and processor and process failures.

The end-to-end delay of a remote operation request comprises delays on the server, the client, and the network. FLARe is designed to bound server latencies, which often dominate in distributed real-time systems (*e.g.*, SCADA systems) equipped with high-speed networks. To meet desired server latencies FLARe allows users to specify a utilization bound for each CPU on the servers. The utilization bound can be set to below the schedulable utilization bound of the real-time scheduling policy (*e.g.*, rate monotonic) supported by the middleware scheduling service. At run time FLARe maintains desired server latencies by dynamically enforcing the utilization bounds on the servers[1].

Processors and processes may experience fail-stop [137] failures and concurrent failures in multiple processors or processes can occur. To provide lightweight fault-tolerance, FLARe employs PASSIVE replication [20], where services are replicated and deployed across multiple processors. We assume that networks provide bounded communication latencies and do not fail or partition. This assumption is reasonable for many soft real-time systems, such as SCADA systems, where nodes are connected by highly redundant

[1]FLARe is targeted at *soft* real-time applications and does not provide hard guarantees on meeting every deadline

high-speed networks. Relaxing this assumption through integration of our middleware with network-level fault tolerance and QoS management techniques [5] is an area of future work.

V.3 Design and Implementation of FLARe

This section describes the design and implementation of FLARe. The key design goals of FLARe are to (1) mask clients from processor and process failures through transparent client failover, (2) alleviate post recovery overload through load-aware failover target selection, and (3) maintain desired soft real-time performance by dynamically enforcing suitable CPU utilization bounds on the servers through overload management.

V.3.1 FLARe Middleware Architecture

FLARe's architecture, shown in Figure 24, has four main components: the *middleware replication manager*, the *client failover manager* for each client process, the *monitor* on each processor hosting servers, and the *state transfer agent* on each process hosting servers. FLARe achieves fault-tolerance through PASSIVE replication of CORBA objects, where the primary and backup replicas are deployed across different processors in the distributed system.

Middleware replication manager. FLARe's *middleware replication manager* (MRM) allows server objects to provide information about (1) the processors and processes in which their primaries and backups are hosted, (2) the CPU utilization that they will require to serve client requests should they become primary, and (3) their interoperable object reference (IOR) so that clients can invoke remote operations on them when the server objects are added to the system. To manage the primary and backup replicas—and to make adaptive failover target decisions—FLARe's MRM uses a *monitor* on each processor to track failures and CPU utilizations of all processors hosting the primary and backup replicas of each server object.

As highlighted by label A in Figure 24, FLARe's MRM employs a *Load-Aware and Adaptive Failover* (LAAF) target selection algorithm (described in Section V.3.2) to prepare an rank-ordered list of failover targets for each *primary* object in the system. The rank list includes multiple failover targets in order to handle multiple failures of the same server object. In some situations the current *primary* replica can become overloaded, *e.g.*, due to sudden workload fluctuations and multiple failures. FLARe's MRM employs the *Resource Overload Management rEdirector* (ROME) algorithm (described in Section V.3.3) to redirect clients from overload processors to maintain the desired soft real-time performance. The LAAF and ROME strategies are detailed in Section V.3.2 and Section V.3.3, respectively. Finally, MRM could be co-located with server objects (*i.e.*, Host 1 or Host 2 in Figure 24) as the computation load of the LAAF and ROME algorithms implemented in MRM is relatively low compared to that of the server objects.

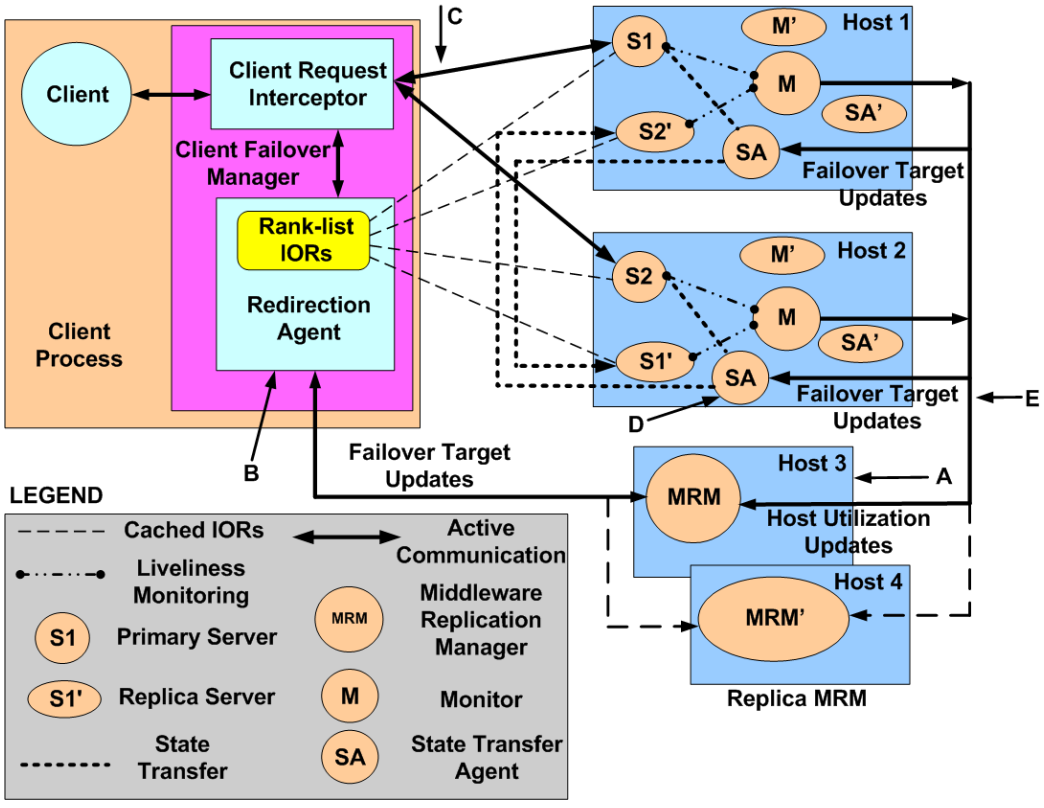

Figure 24: The FLARe Middleware Architecture

Monitors. The liveliness of the processes hosting the server objects and CPU utilization of the hosts is probed by monitors co-located with the server objects. Failures of processes, if any, are communicated instantaneously to the MRM whereas the CPU utilization is communicated at a configurable sampling rate. We do not, however, require fine-grained time synchronization since the sampling period is typically longer than the task periods. For instance, the task periods in the experiments described in Section V.4 vary from one second to one-tenth of a second whereas the monitor sampling period is greater than one second.

Client failover manager. As highlighted by label B in Figure 24, FLARe's *client failover manager* contains a *redirection agent* that is updated with failover and redirection targets so clients can recover transparently from failures and overloads, respectively. To handle failures, as highlighted by label C in Figure 24, FLARe's *client request interceptor* catches failure exceptions and instead of propagating the exception to the client application, the client request interceptor redirects the client invocation to the appropriate failover target provided by the redirection agent.

State transfer agent. As highlighted by label D in Figure 24, FLARe's *state transfer agent* allows server objects to inform it about changes to application states. The *state transfer agent* is updated with per-server-object failover targets by FLARe's MRM. When a *primary* replica in a process informs it about application state change, the *state transfer agent* utilizes interfaces provided by the server object to obtain the new state. The *state transfer agent* synchronizes the state of the *backup* replicas with the new state, by making remote invocations on the *backup* replicas using the provided failover target references as highlighted by label E in Figure 24.

FLARe schedules state update propagations from the primary replica to the backup replicas using remote operation requests, from the state transfer agent on the primary replica to one of the backup replicas. The period of the state update task is equal to the

period of the primary task. In the current implementation, each state update task is scheduled on the processor hosting the backup replicas at the priority determined by the rate-monotonic scheduling algorithm.

To support distributed soft real-time applications in FLARe, the *primary* replica updates the states of its *backup* replicas *after* it sends its response to the client. This design choice significantly reduces the response times for clients, but supports only "best effort" guarantees for state synchronization. Replica consistency may be lost if the *primary* replica crashes after it responds to the client, but before it propagates its state update to the *backup* replicas. This design tradeoff is desirable in many distributed soft real-time applications where state can be reconstructed using subsequent (*e.g.*, sensor) data updates at the cost of transient degradation of services.

V.3.2 Load-aware and Adaptive Failover

As described in Section V.3.1, FLARe's MRM collects periodic measurement updates from the monitors about CPU utilizations and liveness of processors/processes. FLARe provides a *load-aware, adaptive failover (LAAF)* target selection algorithm that uses these measurements to select per-object failover targets. LAAF uses the following inputs: (1) the list of processors and the list of processes in each processor, (2) the list of primary object replicas operating in each process, (3) the list of backup replicas for each primary object replica and the processors hosting those replicas, and (4) the current CPU utilizations of all processors in the system. This algorithm is executed whenever there is a change in the CPU utilization by a *threshold* (*e.g.*, $\pm$ 10%) in any of the processors in the system since FLARe must react to such dynamic changes.

The output of LAAF is a ranked list of failover targets for each primary object replica in the system. To deal with concurrent failures, FLARe maintains an ordered list of failover targets, instead of only the first one. When both the primary replica and some of its backup replicas fail concurrently, the client can failover to the first backup replica in the list that is

still alive. LAAF estimates the post-failover CPU utilizations of processors hosting backup replicas for a primary object, assuming the primary object fails. The backup replicas are then ordered based on the estimated CPU utilizations of the processors hosting them, and the backup replica whose host has the lowest estimated CPU utilization is the first failover target of the replica. To balance the load after a processor failure, LAAF redirects the clients of different primary objects located on the same processor to replicas on different processors. Finally, the references (IORs) to those replicas are collected in a list and provided to the redirection agents for use during failure recovery. To reduce the failover delay, MRM *proactively* updates a client whenever its failover target list changes.

Algorithm 2: LAAF Target Selection Algorithm

Input:
$P_i \leftarrow$ Set of processes on processor i
$O_j \leftarrow$ Set of primary replica objects in process j
$R_k \leftarrow$ list of processors hosting backup replicas for a primary object k
$cu_i \leftarrow$ current utilization of processor i
$eu_i \leftarrow$ expected utilization of processor i after failovers
$l_k \leftarrow$ CPU utilization attributed to primary object k

```
1  begin
2   for every processor i do
3    eu_i = cu_i // reset expected utilization
4    for every process j in P_i do
5     for every primary object k in O_j do
6      // sort R_k in increasing order of expected CPU utilization
7      // eu_x += l_k, where processor x is the head of the sorted list R_k
8     end
9    end
10  end
11 end
```

Algorithm 2 depicts the steps in the LAAF target selection algorithm. For every processor in the system (line 2), LAAF iterates through all hosted processes (line 4), and the primary replicas that are hosted in those processes (line 5). For every primary replica, the algorithm determines the processors hosting its backup replicas and the least loaded of

those processors (line 6). The algorithm then adds the load of the primary object replica (known to FLARe's MRM because of the registration process as explained in Section V.3.1) to the load of least loaded processor and defines that as the *expected utilization* of that processor (line 12) were such a failover to occur.

The algorithm repeats the process described above for every other primary replica object hosted in the same process (Lines 5–7). The least loaded failover processor is determined by considering the expected utilizations of the processors (line 6). This decision allows the algorithm to consider the failover of co-located primary replica objects within a processor while determining the failover targets of other primary replica objects hosted in the same processor. The failover target selection algorithm therefore makes decisions not only based on the dynamic load conditions in the system (which are determined by the monitors), but also based on load additions that may be caused by failovers of co-located primary objects. The failover targets are then used for redirecting a client if any failure occurs before the next time LAAF is run.

LAAF is optimized for multiple process failures or single processor failures. It may result in suboptimal failover targets, however, when multiple processors fail concurrently. In this case, clients of objects located on different failed processors may failover to a same processor, thereby overloading it. Similarly, LAAF may also result in suboptimal failover targets when process/processor failures and workload fluctuation occur concurrently, *i.e.*, before FLARe's MRM receives the updated CPU utilization from the monitors. To handle such overload situations FLARe employs the ROME algorithm (described next in Section V.3.3) to redirect clients of overloaded processors, proactively to less loaded processors.

• **Analysis of the LAAF target selection algorithm**. The failover targets determined by the LAAF algorithm could be incorrect under certain circumstances. For example, those circumstances could be:

New resource additions and dynamic workloads. As described in Section V.3.2, the

LAAF algorithm is executed when the monitor in any of the processors senses a load change or a failure. However, if a failure occurs, before the MRM could adapt to the change, clients would failover using targets that were determined before accounting the new change. Such a failover could affect the response times clients receive after a failover, as the clients could have potentially failed over to a processor that is overloaded or that gets overloaded after a failover.

For example, if the change involves a dynamic workload addition (*e.g.*, deployment of a new service), and the subsequent failover is to the same processor, then the processor could get overloaded - client response times are also affected after the failover. Such an overload needs to be immediately handled to restore satisfactory response times of all the affected clients. Similarly, if the change involves addition of a new processor or a rejuvenation of a failed processor, then new replicas (both primary and backup replicas) are added into the system, which could be potentially used as failover targets. However, since the failover occurred before the replication manager could utilize those new replicas, clients could potentially be wasting a chance to utilize replicas that are deployed in processors with much lesser CPU utilization than the processor they are currently making remote invocations on. Such a CPU utilization imbalance between processors hosting replicas of the same *type* should be quickly detected, so that clients could be redirected to appropriate replicas.

Simultaneous multiple processor failures. Note that, in line 3 of the LAAF algorithm, the expected utilization of all the processors is reset to the utilization that is determined by the monitors. This means that when failover target decisions are made for a processor, the algorithm does not account for the expected CPU utilization increases that could arise because of the failure of another processor in the system. This decision leads to a possibility of two or more services picking the same processor as the destination of their failover target replicas.

For example, let us assume P_i and P_j be the *primary* replicas of two different services and C_i and C_j be their respective clients. The rank list maintained by client C_i's redirection

agent is represented by the tuple: $\langle R_{i1}, R_{i2}, \cdots, R_{iN} \rangle$; where R_{i1} represents the most appropriate failover target and R_{iN} represents the least appropriate failover target determined by the LAAF algorithm, and N represents the number of replicas of the service. Client C_j's redirection agent also maintains a similar rank list for the *primary* replica P_j.

Now assume that R_{i1} operates in the least loaded processor among all the processors hosting replicas of the *primary* replica P_i. Similarly, R_{j1} operates in the least loaded processor amongst all the processors hosting replicas of the *primary* replica P_j. If R_{i1} and R_{j1} operate in the same processor, then the failover target selection algorithm would have picked that processor as the failover target for both the *primary* replicas P_i and P_j. In other words, the effective utilization increments are not considered for two or more processors in the LAAF algorithm. At runtime, if both *primary* replicas P_i and P_j fail together, the clients of the failed services will be redirected to the same backup processor. These multiple redirections can cause an overload on the backup processor degrading response times for the clients.

The above observation does not mean that LAAF cannot support simultaneous multiple processor failures. In the above example, it so happened that R_{i1} and R_{j1} were located in the same processor, and were also the least loaded targets. If the replicas were located in different nodes, then LAAF would have been able to handle simultaneous multiple processor failures. We assume that, such kind of application placements and configurations are rare, and hence we did not accommodate handling such kind of failures in our LAAF algorithm. In the next section, we describe our adaptive client redirection strategy that can handle the overloads and processor CPU utilization imbalances illustrated above.

V.3.3 Resource Overload Management and Redirection

FLARe's MRM employs the *Resource Overload Management and rEdirection (ROME)* algorithm to enforce desired CPU utilization and service delay bounds. FLARe allows users to specify a per-processor *utilization bound* based on the schedulable utilization

bound of the real-time scheduling policy (*e.g.*, rate monotonic) supported by the middleware scheduling service. A processor whose CPU utilization exceeds the *utilization bound* is considered overloaded.

ROME is needed to resolve processor overload since CPU saturation may cause system failure due to kernel starvation [89]. Distributed soft real-time applications can use ROME to specify the overload threshold based on the suitable schedulable utilization bounds [143] needed to achieve satisfactory response times. ROME also allows users to specify a per-object *migration threshold* to redirect clients of primary objects hosted in current heavily loaded (but not overloaded) processor to the least loaded processor hosting a replica of that object. Balancing processor CPU utilization helps reduce the response times and avoid overload on a subset of processors in the system. FLARe thus uses ROME to handle CPU overload and load imbalance as special cases of failures for distributed soft real-time applications.

In the case of failures, the clients are redirected to appropriate failover targets based on decisions made by LAAF, as described in Section V.3.2. In the case of overloads, clients of the current primary replicas are redirected automatically to the chosen new backup replicas. We refer to this load redistribution mechanism as *lightweight migration* since we migrate the *loads* (through client redirection) of objects as opposed to the less efficient alternative of migrating the *objects* themselves. Moreover, ROME leverages existing replicas and effectively utilizes them for maintaining satisfactory response times for clients.

Algorithm 3 depicts the steps ROME uses to handle CPU overload and load imbalance, respectively.

Handling overloads. When the CPU utilization at any of the processor crosses the *utilization bound*, FLARe's MRM triggers ROME to react to the overloads. FLARe determines the primary objects whose clients need to be redirected, and their target hosts, using ROME. Given an overloaded processor (*i.e.*, whose CPU utilization exceeds the *utilization bound*),

Algorithm 3: Determine Load-redistributing Targets

Input:
$O_i \leftarrow$ list of primary objects in an overloaded processor i
$R_j \leftarrow$ list of processors hosting object j's replicas
$cu_i \leftarrow$ current utilization of processor i
$eu_i \leftarrow$ expected utilization of processor i after migrations
$l_j \leftarrow$ CPU utilization of primary object j
$t_i \leftarrow$ upper bound for processor i's CPU utilization
$eu_i = cu_i$ for every processor i

```
1  begin
2   for every overloaded processor i do
3    sort O_i in decreasing order of their CPU utilizations
4    for every object j in the sorted list O_i do
5     min : processor i in R_j with lowest CPU utilization
6     if (l_j + eu_min) < t_min then
7      migrate the load of object j to j's replica in min
8      eu_min += l_j
9      eu_i -= l_j
10    end
11    if eu_i < t_i then
12     processor i is no longer overloaded; stop
13    else
14     migrate another primary object j in the processor i
15   end
16  end
17 end
```

ROME considers the primary objects on the processor in decreasing order of CPU utilization (line 3), and attempts to migrate the load generated by those objects to the least-loaded processor hosting their backup replicas (lines 5 through 9). The attempt fails if the least-loaded processor of the backup replicas would exceed the *utilization bound* if the migration occurs. ROME attempts migrations until (1) the processor is no longer overloaded or (2) all clients of primary objects in the overloaded processor have been considered for redirection.

Similar to LAAF, ROME also uses the *expected CPU utilization* to spread the load of multiple objects on an overloaded processor to different hosts. The expected CPU utilization accounts for the load change due to the redirection decisions affecting the overloaded

processor. After new reconfigurations are identified, redirection agents are updated to redirect existing clients from the current primary replica to the selected backup replica at the start of the next remote invocation. Clients are thus redirected to new targets with less perturbations.

V.3.4 Implementation of FLARe

FLARe has been implemented atop the TAO Real-time CORBA middleware. It is implemented in ∼9,000 lines of C++ source code (excluding the code in TAO). Below we highlight several key aspects of the FLARe implementation (a more detailed description of FLARe appears in [6]).

Monitoring CPU utilization and processor failures. On Linux, FLARe's *monitor* process uses the `/proc/stat` file to estimate the CPU utilization (*i.e.*, the fraction of time when the CPU is not idle) in each sampling period. We chose to measure the CPU utilization online, rather than relying on the estimated CPU utilization provided by users to account for estimation errors and for other activities in the middleware and OS kernel.

To detect the failure of a process quickly, each application process on a processor opens up a passive POSIX local socket (also known as a UNIX domain socket) and registers the port number with the monitor. The monitor connects to the socket and performs a blocking read. If an application process crashes, the socket and the opened port will be invalidated, in which case the monitor receives an invalid read error on the socket that indicates the process crash. Fault tolerance of the monitor processes is also achieved through passive replication. If the *primary* monitor replica fails to send updated information or to respond to FLARe's *middleware replication manager* (described below) within a timeout period, FLARe suspects that the processor has crashed.

Middleware replication manager. FLARe's *middleware replication manager* is designed using the Active Object pattern [140] to decouple the reporting of a load change or a failure from the process. This decoupling allows several monitors to register with FLARe's

middleware replication manager while allowing synchronized access to its internal data structures. Moreover, FLARe can be configured with the LAAF and ROME algorithms via the Strategy pattern [51]. FLARe's *middleware replication manager* is replicated using SEMI_ACTIVE replication [55] (provided by the TAO middleware), with regular state updates to the backup replicas.

Client failover manager. As shown in Figure 24, the client's failover manager comprises a CORBA portable interceptor-based *client request interceptor* [164] and a redirection agent, which together coordinate to handle failures in a manner transparent to the client application logic. Whenever a primary fails, the interceptor catches the CORBA COMM_FAILURE exception. Since portable interceptors are not remotely invocable objects, it was not feasible for an external entity (such as a MRM) to send the rank list information to the interceptor, which is necessary to determine the next failover target. The redirection agent is therefore a CORBA object that runs in a separate thread from the interceptor thread. The interceptor consults the redirection agent for the failover target from the rank list it maintains. The interceptor will then reissue the request to the new target. The rank list is propagated to the redirection agent *proactively* by FLARe's MRM whenever the failover target list changes.

V.4 Empirical Evaluation of FLARe

We empirically evaluated FLARe in ISISlab (`www.dre.vanderbilt.edu/ISIS-lab`) on a testbed of 14 blades. Each blade has two 2.8 GHz CPUs, 1GB memory, a 40 GB disk, and runs the Fedora Core 4 Linux distribution. Our experiments used one CPU per blade and the blades were connected via a CISCO 3750G switch into a 1 Gbps LAN. 12 of the blades ran Real-time CORBA applications on FLARe. FLARe's MRM and its backup replicas ran in the other 2 blades. To emulate distributed soft real-time applications, the clients in these experiments used threads running in the Linux real-time scheduling class to invoke operations on server objects at periodic intervals. All operations and state updates

on the servers were executed according to the rate monotonic scheduling policy supported by the TAO scheduling service.

V.4.1 Evaluating LAAF

The first experiment was designed to evaluate FLARe's LAAF algorithm (described in Section V.3.2) and compare it with the optimal *static* client failover strategy. In the static client failover strategy, the client middleware is initialized with a *static* list of IORs of the backup replicas, ranked based on the CPU utilization of their processors at *deployment time*. The list is not updated at run-time based on the current CPU utilizations in the system (the failover targets are optimal at deployment time, but any *static* failover target can become suboptimal at run-time in face of dynamic workloads). In contrast, LAAF dynamically recomputes failover targets whenever there is a change in the CPU utilization by a *threshold* (*e.g.*, $\pm$ 10%) in any of the processors in the system.

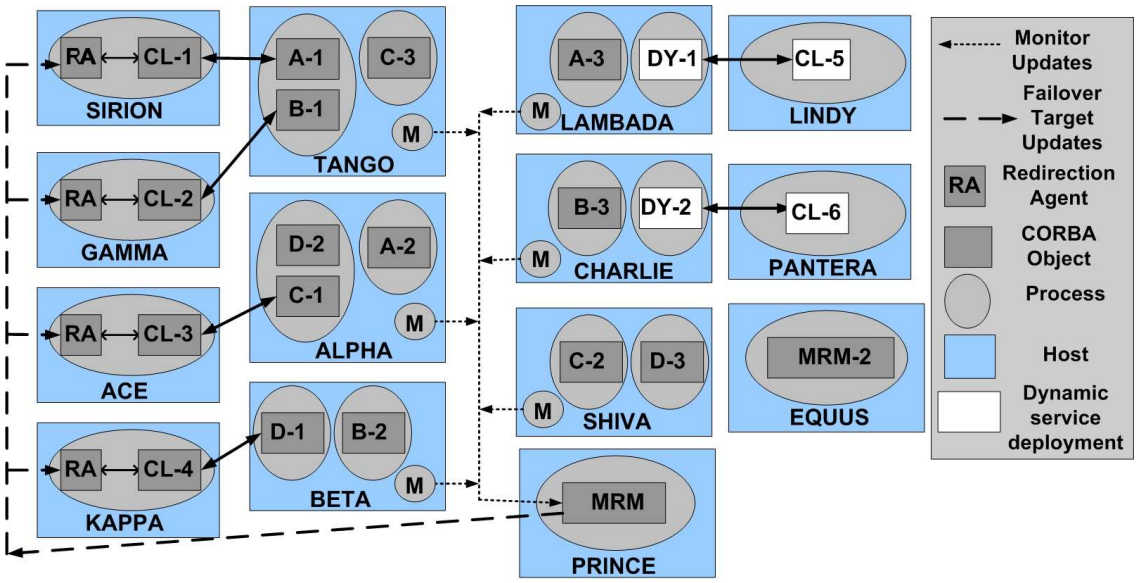

Figure 25: Load-aware Failover Experiment Setup

Experiment setup. Figure 25 and Table 7 illustrate our experimental setup. The experiment ran for 300 seconds. To evaluate FLARe in the presence of *dynamic workload*

changes, at 50 seconds after the experiment was started, we introduced dynamic invocations on two server objects DY-1 and DY-2, using client objects, CL-5, and CL-6, respectively. The *static* failover strategy selects failover targets that are optimal at deployment time, as follows: if A-1 fails, contact A-3 followed by A-2; if B-1 fails, contact B-3 followed by B-2.

Client Object	Server Object	Invocation Rate (Hz)	Server Object Utilization
Static Loads			
CL-1	A-1	10	40%
CL-2	B-1	5	30%
CL-3	C-1	2	20%
CL-4	D-1	1	10%
Dynamic Loads			
CL-5	DY-1	5	50%
CL-6	DY-2	10	50%

Table 7: Experiment setup for LAAF

We emulated a process failure 150 seconds after the experiment started. We used a fault injection mechanism, where when clients CL-1 or CL-2 make invocations on server objects A-1 or B-1, respectively, the server objects calls the *exit (1)* command, crashing the process hosting server objects A-1 and B-1 on processor TANGO. The clients receive COMM_FAILURE exceptions, and then failover to replicas chosen by the failover strategy.

Analysis of results. Figure 26 shows the CPU utilizations at all the processors, when clients used the static client failover strategy. At 50 seconds, servers DY-1 and DY-2 were invoked by clients CL-5 and CL-6 causing the CPU utilizations at processors LAMBADA and CHARLIE to increase from 0% to 50%.

At 150 seconds when process hosting both A-1 and B-1 fails on the processor TANGO, clients CL-1 and CL-2 failover to the statically configured replicas A-3 at processor LAMBADA and B-3 at processor CHARLIE respectively. As a result, the CPU utilizations at

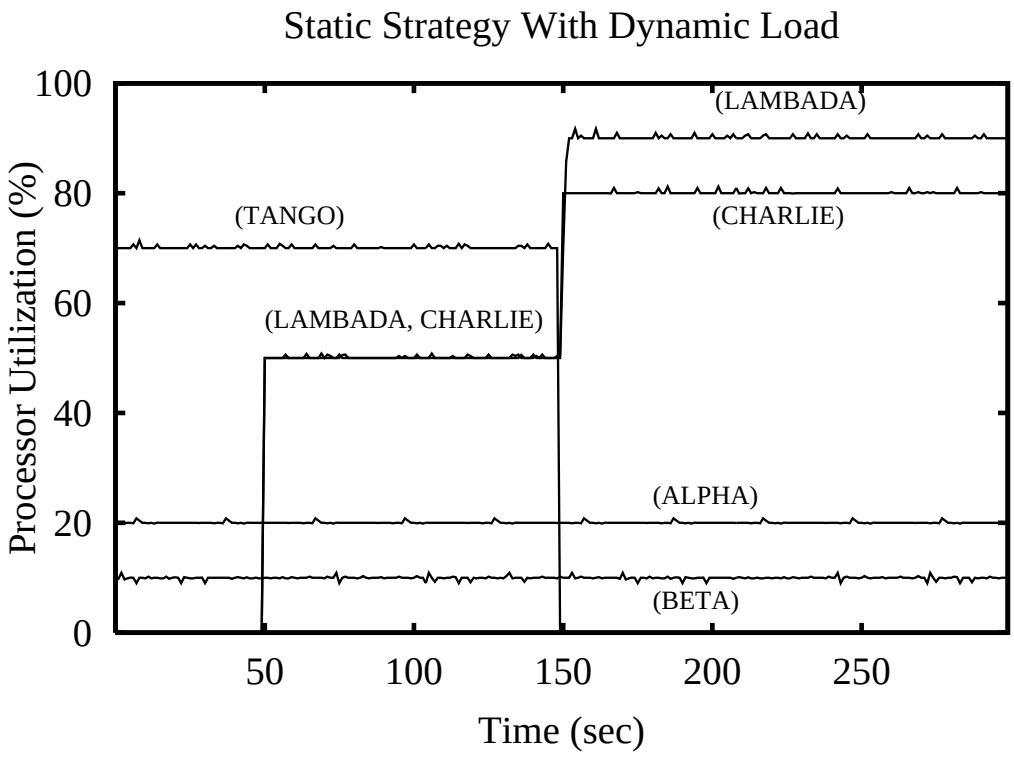

Figure 26: Utilization with Static Failover Strategy

processors LAMBADA and CHARLIE increase to 90% and 80% respectively. Note that 90% CPU utilization is highly undesirable in middleware systems because it is close to saturating the CPU which may result in kernel starvation and system crash [89]. The high CPU utilizations on processors CHARLIE and LAMBADA occur, because the *static* client failover strategy did not account for *dynamic* system loads while determining client failover targets.

In contrast, FLARe's MRM triggers LAAF to recompute the failover targets in response to load changes. At 50 seconds, LAAF changed the failover target of the primary replica A-1 from A-3 to A-2, in response to the load increase on processor LAMBADA (host of A-3). Similarly, LAAF also changed the failover target of B-1 from B-3 to B-2 in response to the load increase on processor CHARLIE (host of B-3). At 150 seconds, clients CL-1 and CL-2 failover to backup replicas A-2 and B-2 respectively. As shown in Figure 27, none of the processor utilizations is greater than 60% after the failover of clients CL-1 and

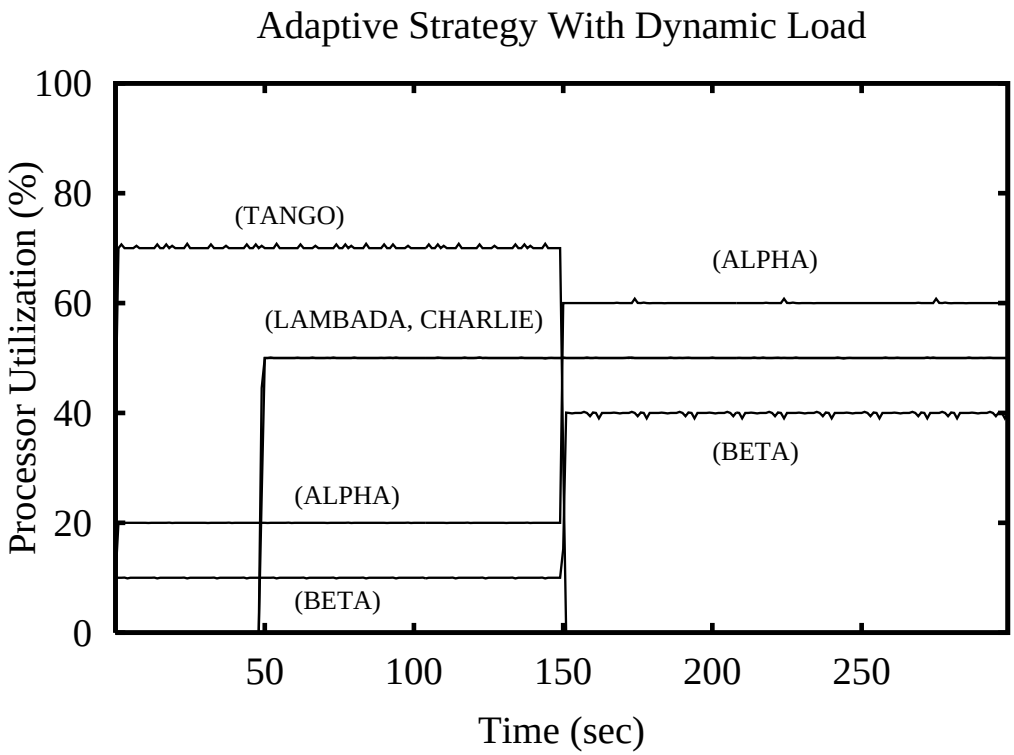

Figure 27: Utilization with Adaptive Failover Strategy

CL-2. This result shows that LAAF effectively alleviates processor overloads after failure recovery, due to its adaptive and load-aware failover strategy.

V.4.2 Evaluating ROME

We designed two more experiments to evaluate the ROME algorithm described in Section V.3.3. We stress-tested ROME under overloads caused by dynamic workload changes and multiple failures.

Experiment setup. Figure 28 and Table 8 show the experimental setup. The utilization bound on every processor was set to 70%, which is below the schedulable utilization bound (based on the number of tasks) for the rate monotonic policy supported by the middleware scheduling service. The required server delay for each task equalled its invocation period.

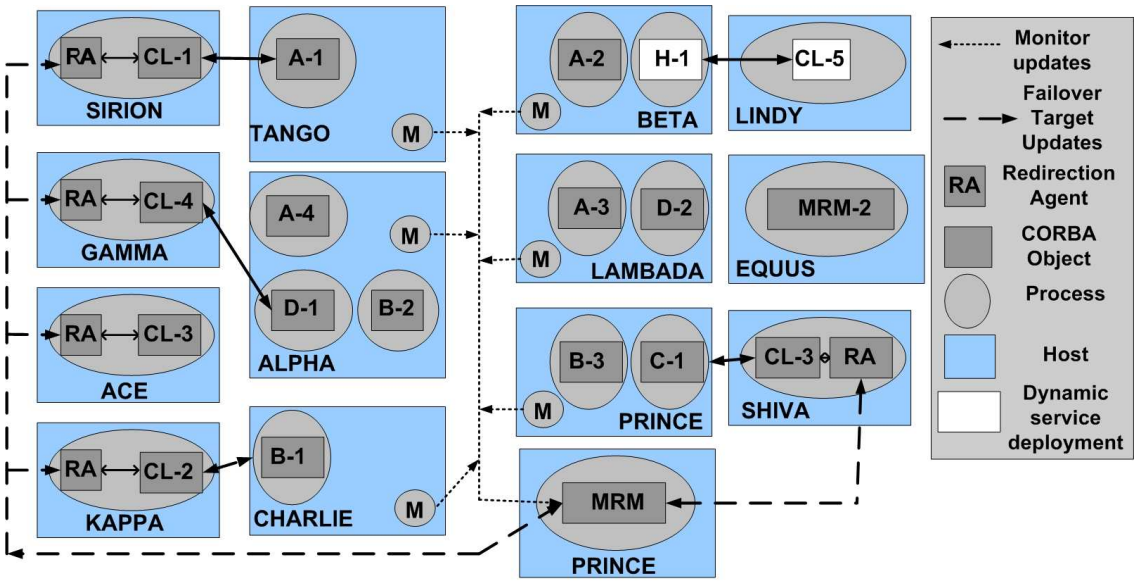

Figure 28: Overload Redirection Experiment Setup

Client Object	Server Object	Invocation Rate (Hz)	Server Object Utilization
Static Loads			
CL-1	A-1	10	40%
CL-2	B-1	5	30%
CL-3	C-1	2	30%
CL-4	D-1	1	10%
Dynamic Loads			
CL-5	H-1	10	50%

Table 8: Experiment setup for ROME

Concurrent Workload Change and Process Failure. We emulated a failure 50 seconds after the experiment started. We used a fault injection mechanism, where when client CL-1 makes invocations on server object A-1, the server object calls the *exit (1)* command, crashing the process hosting server object A-1 on the processor TANGO. The client CL-1 receives a COMM_FAILURE exception due to the failure of A-1, and then consults its rank list to make a failover decision, which is A-2. At the same time, a client CL-5 starts making invocations on a new service H-1.

As a result of the concurrent failure and workload change, the load on the processor BETA rises to 90% (highlighted by point A in the Figure 29), which exceeds the specified utilization bound (70%) and consequently triggers ROME. ROME then performs a lightweight migration of the clients of A-2 and redirects all of its clients to A-3, which is hosted in the least loaded of all the processors hosting a replica of A-1. Within 1 second, the utilization of processor BETA decreases to 50%, while the utilization of processor LAMBADA increases to 40% due to A-3 becoming the new primary replica.

At this stage, the CPU utilizations of all processors are below 70%. We also plot the measured end-to-end response times perceived by the clients in Figure 30. After ROME redirected the client's requests, the end-to-end response times of all the clients drop below the required server delays, indicating that every server object achieved its required server delay (which is a part of the corresponding end-to-end response times). This result demonstrated that ROME can handle overload effectively and efficiently.

Concurrent Failures. We then stress-tested ROME further with concurrent failures. Since the CPU utilizations in the system have changed dynamically, FLARe's MRM also employs LAAF to redetermine the failover targets for all the primary objects in the system. The recomputed failover targets are as follows: (1) for A-1, it is $\langle$A-4, A-2$\rangle$ (2) for B-1, it is $\langle$B-2, B-3$\rangle$, and (3) for D-1, it is $\langle$D-2$\rangle$

We emulated a failure 150 seconds after the experiment started. We used a fault injection mechanism, where when clients CL-1 and CL-2 make invocations on server objects A-3 and B-1, respectively, the server objects call the *exit (1)* command, crashing the process hosting server objects A-3 on processor LAMBADA and B-1 on processor CHARLIE. The clients receive COMM_FAILURE exceptions, and then fail over to replicas chosen by the failover strategy. Using the failover targets computed by LAAF, client CL-1 fails over to A-4 while client CL-2 fails over to B-2, both of which end up starting on the same processor ALPHA, which is already hosting a primary D-1.

As a result, the CPU utilization of the processor ALPHA jumps to 80% (as highlighted

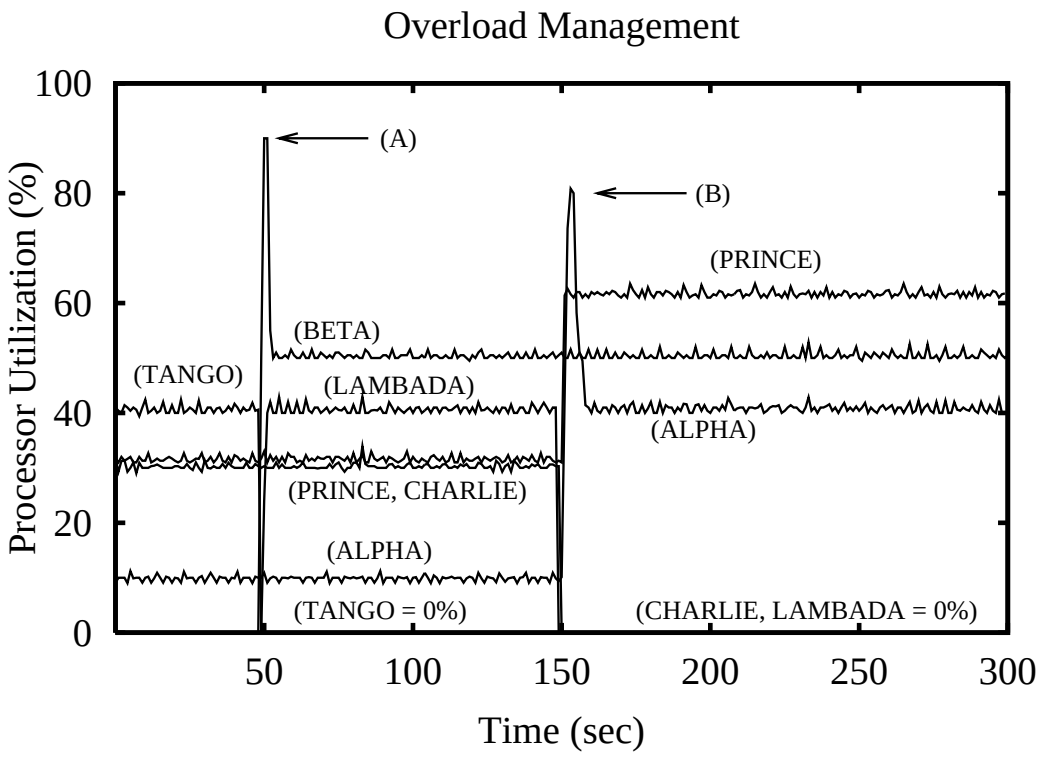

Figure 29: Utilizations with ROME Overload Management

by point B in Figure 29), while the clients CL-1, CL-2, and CL-4 see an increase in response times (as shown in Figure 30). FLARe's MRM triggers ROME once again to resolve the overload, starting with the most heavily loaded service, A-4, but clients of A-4 cannot be moved, as that would again overload the processor BETA. Hence, ROME redirects all clients of B-2 (which is the next most heavily loaded object) to its replica B-3 on processor PRINCE. As a result, the CPU utilizations of all the processors settle below 70% as shown by point (B in Figure 29), while the end-to-end response times (and hence the server delays) drop below the required server delays.

This experiment demonstrates that ROME can effectively enforce the specified utilization bound and server delays by dynamically handling overloads caused by concurrent failures and workload changes.

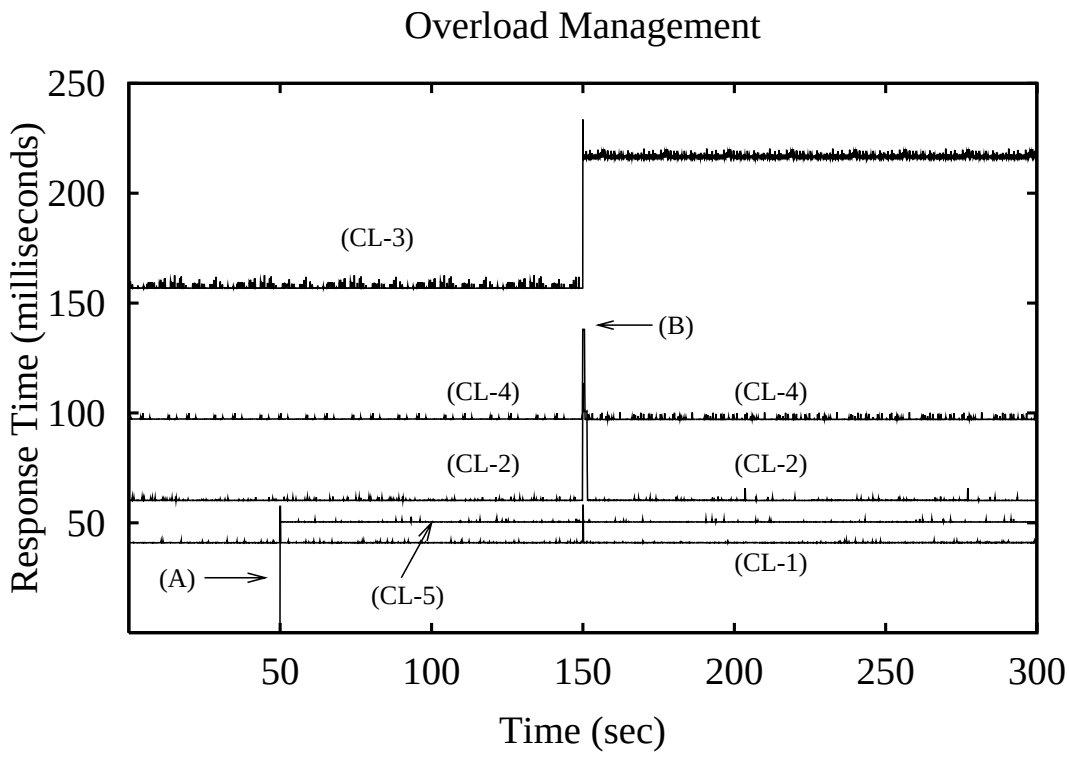

Figure 30: Client Response Times with ROME Overload Management

V.4.3 Failover Delay

To empirically evaluate the failover delays under the *static* and the *adaptive* failover strategies, we ran an experiment with client CL-1 invoking 10,000 requests on server object A-1. No other processes operated in the processor hosting A-1, so that the response time will equal the execution time of the server. A fault was injected to kill the server while executing the $5,001^{st}$ request. The clients then failover to backup server objects A-2, which execute the remaining 5,000 requests (including the one experiencing the failure).

The left side of Figure 31 shows the different response times perceived by client C-1 in the presence of server object failures. The failover delays for the *static* and *adaptive* failover strategies are similar because under the static strategy the client knows the failover decision *a priori*, while under the LAAF strategy, FLARe's MRM proactively sends the updated failover targets to the client so they are also readily available when a failover

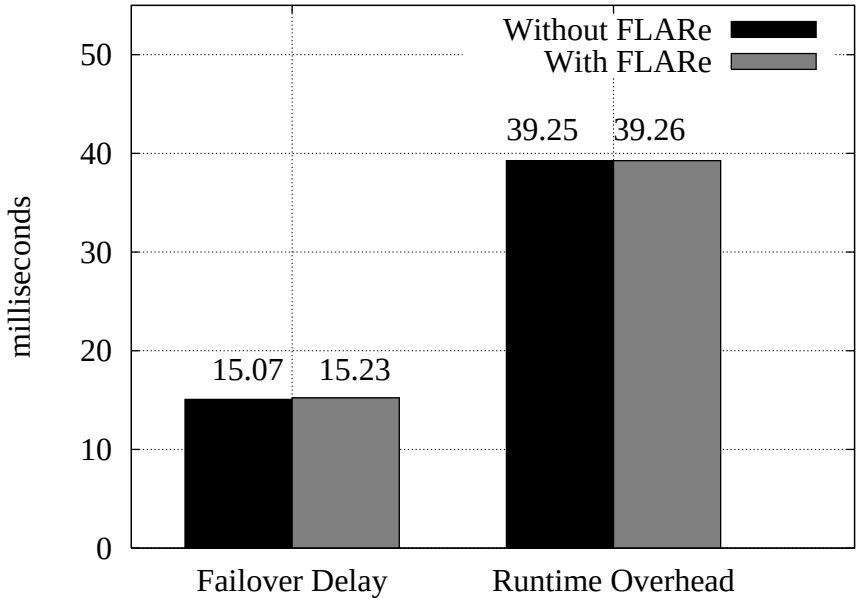

Figure 31: Failover delay and run-time overhead

occurs. Our results indicate that FLARe's proactive failover strategy achieves fast failover with a failover delay comparable to the static strategy.

V.4.4 Overhead under Fault-Free Conditions

FLARe uses a CORBA client request interceptor to catch COMM_FAILURE exceptions and transparently redirect clients to suitable failover targets. To evaluate the runtime overhead of these per-request interceptions during normal failure free conditions, we ran a simple experiment with client CL-1 making invocations on server object A-1 with and without client request interceptors.

We ran this experiment for 50,000 iterations and measured the average response time perceived by CL-1. The right side of Figure 31 shows that the average response time perceived by CL-1 increased by only 8 microseconds when using the client request interceptor. This result shows that interceptors add negligible overhead to the normal operations of an application.

V.5 Summary

This chapter presents the Fault-tolerant Load-aware and Adaptive middlewaRe (FLARe) for distributed soft real-time applications. FLARe features (1) the Load-aware and Adaptive Failover (LAAF) strategy that adapts failover targets based on system load; (2) the Resource Overload Management Redirector (ROME) strategy that dynamically enforces CPU utilization bounds to maintain desired server delays in face of concurrent failures and load changes; and (3) an efficient fault-tolerant middleware architecture that supports transparent failover to passive replicas. FLARe has been implemented on top of the TAO RT-CORBA middleware as open-source software. Empirical evaluation on a distributed testbed demonstrates FLARe's capability to maintain system availability and soft real-time performance in the face of dynamic workload and failures while introducing only negligible run-time overhead.

It is possible to conceive of overload management schemes by virtue of in-place replacement of component implementations wherein performance can be traded off by replacing a resource-intensive implementation with an implementation that consumes less resources but demonstrates degraded performance. In Chapter VI we describe this capability.

CHAPTER VI

MIDDLEWARE MECHANISMS FOR OVERLOAD MANAGEMENT IN DISTRIBUTED SYSTEMS

Component technologies are increasingly being used to develop and deploy distributed real-time and embedded (DRE) systems. To enhance flexibility and performance, developers of DRE systems need middleware mechanisms that decouple component logic from the binding of a component to an application, i.e., they need support for dynamic updating of component implementations in response to changing modes and operational contexts. This chapter presents three contributions to R&D on dynamic component updating. First, it describes an inventory tracking system (ITS) as a representative DRE system case study to motivate the challenges and requirements of updating component implementations dynamically. Second, it describes how our SwapCIAO middleware supports dynamic updating of component implementations via extensions to the server portion of the Lightweight CORBA Component Model. Third, it presents the results of experiments that systematically evaluate the performance of SwapCIAO in the context of our ITS case study. Our results show that SwapCIAO improves the flexibility and performance of DRE systems, without affecting the client programming model or client/server interoperability.

The rest of this chapter is organized as follows. Section VI.1 introduces the research problem and provides the motivation for our work; Section VI.2 describes the structure and functionality of an inventory tracking system, which is a DRE system case study that motivates the need for dynamic component implementation updating; Section VI.2.2 describes the key design challenges in provisioning the dynamic component implementation updating capability in QoS-enabled component middleware systems; Section VI.3 describes the design of SwapCIAO, which provides dynamic component implementation updating capability for Lightweight CCM [111]; Section VI.4 analyzes the results from experiments that

systematically evaluate the performance of SwapCIAO for various types of DRE applications in our ITS case study; Finally, Section VI.5 provides a summary of our contributions.

VI.1 Introduction

Component middleware is increasingly being used to develop and deploy next-generation distributed real-time and embedded (DRE) systems, such as shipboard computing environments [141], inventory tracking systems [106], avionics mission computing systems [145], and intelligence, surveillance and reconnaissance systems [144]. These DRE systems must adapt to changing modes, operational contexts, and resource availabilities to sustain the execution of critical missions. However, conventional middleware platforms, such as J2EE, CCM, and .NET, are not yet well-suited for these types of DRE systems since they do not facilitate the separation of quality of service (QoS) policies from application functionality [161].

To address limitations of conventional middleware, *QoS-enabled component middleware*, such as CIAO [165], Qedo [131], and PRiSm [132], explicitly separates QoS aspects from application functionality, thereby yielding systems that are less brittle and costly to develop, maintain, and extend [165]. Our earlier work on QoS-enabled component middleware has focused on (1) identifying patterns for composing component-based middleware [8, 140], (2) applying reflective middleware [163] techniques to enable mechanisms within the component-based middleware to support different QoS aspects [162], (3) configuring real-time aspects [165] within component middleware to support DRE systems, and (4) developing domain-specific modeling languages that provide design-time capabilities to deploy and configure component middleware applications [9]. This chapter extends our prior work by *evaluating middleware techniques for updating component implementations dynamically and transparently (i.e., without incurring system downtime) to optimize system behavior under diverse operating contexts and mode changes.*

A component [156] is a unit of composition with well-defined provided and required

interfaces and interactions between components happen using connectors that bind required interfaces to provided interfaces. Traditional objects of the conventional middleware can access components using the standard interfaces provided by the components. The key forces associated with providing dynamic implementation update capabilities in a QoS-enabled component middleware involve wrestling with challenges unseen in conventional middleware such as handling component-connections with external non-component and component clients to provide capabilities to upgrade components in a transparent manner without incurring a system downtime.

Our dynamic component updating techniques have been integrated into *SwapCIAO*, which is a QoS-enabled component middleware framework that enables application developers to create multiple implementations of a component and update (*i.e.* "swap") them dynamically. SwapCIAO extends CIAO, which is an open-source[1] implementation of the OMG Lightweight CCM [108], Deployment and Configuration (D&C) [109], and Real-time CORBA [112] specifications (see Appendix A for an overview of these technologies).

The key capabilities that SwapCIAO adds to CIAO include (1) mechanisms for updating component implementations dynamically without incurring system downtime and (2) mechanisms that transparently redirect clients of an existing component to the new updated component implementation. As discussed in this chapter, key technical challenges associated with providing these capabilities involve updating component implementations without incurring significant overhead or losing invocations that are waiting for or being processed by the component.

VI.2 Case Study to Motivate Dynamic Component Updating Requirements

To examine SwapCIAO's capabilities in the context of a representative DRE system, we developed an *inventory tracking system* (ITS), which is a warehouse management infrastructure that monitors and controls the flow of goods and assets within a storage facility.

[1] SwapCIAO and CIAO are available from `www.dre.vanderbilt.edu/CIAO`.

Users of an ITS include couriers (such as UPS, DHL, and Fedex), airport baggage handling systems, and retailers (such as Walmart and Target). This section describes (1) the structure/functionality of our ITS case study and (2) the key requirements that SwapCIAO dynamic component updating framework had to address. Naturally, SwapCIAO's capabilities can be applied to many DRE systems – we focus on the ITS case study in this chapter to make our design discussions and performance experiments concrete.

VI.2.1 Overview of ITS

An ITS provides mechanisms for managing the storage and movement of goods in a timely and reliable manner. For example, an ITS should enable human operators to configure warehouse storage organization criteria, maintain the inventory throughout a highly distributed system (which may span organizational and national boundaries), and track warehouse assets using decentralized operator consoles. In conjunction with colleagues at Siemens [107], we have developed the ITS shown in Figure 32 using SwapCIAO. This figure shows how our ITS consists of the following three subsystems:

- **Warehouse management**, whose high-level functionality and decision-making components calculate the destination locations of goods and delegate the remaining details to other ITS subsystems. In particular, the warehouse management subsystem does not provide capabilities like route calculation for transportation or reservation of intermediate storage units.

- **Material flow control**, which handles all the details (such as route calculation, transportation facility reservation, and intermediate storage reservation) needed to transport goods to their destinations. The primary task of this subsystem is to execute the high-level decisions calculated by the warehouse management subsystem.

- **Warehouse hardware**, which deals with physical devices (such as sensors) and transportation units (such as conveyor belts, forklifts, and cranes).

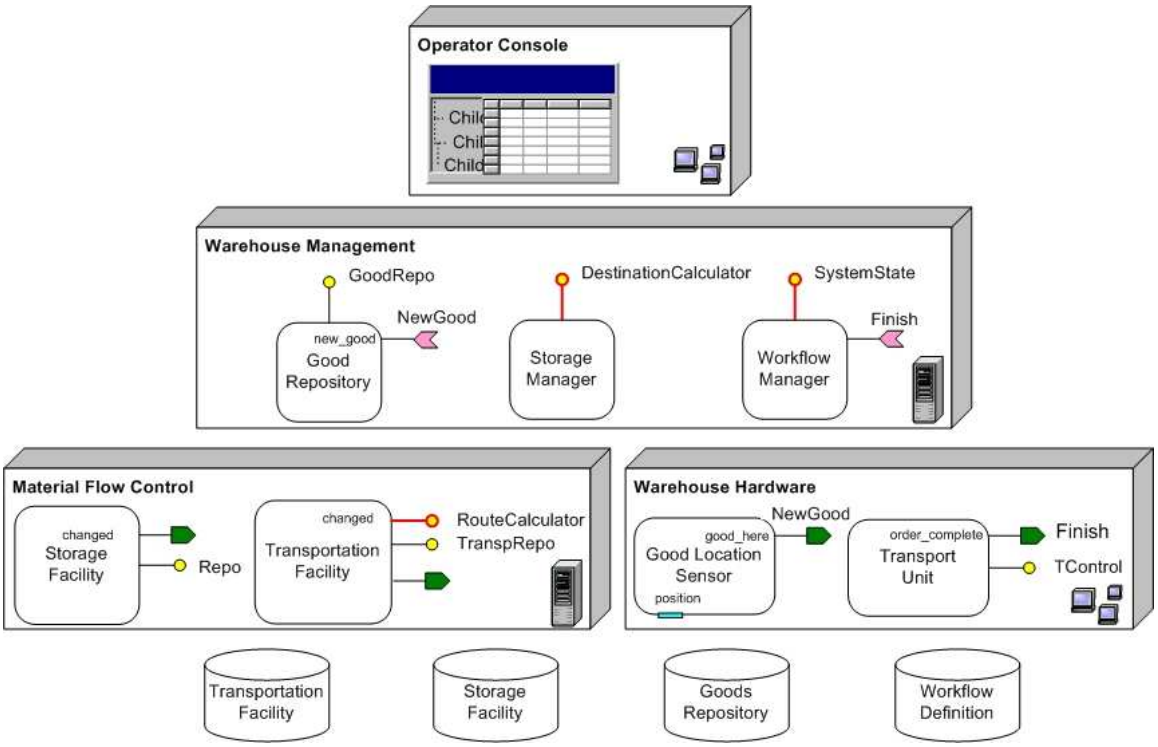

Figure 32: Key Components in ITS

VI.2.2 Requirements for Dynamic Component Updates

Throughout the lifetime of an ITS, new physical devices may be added to support the activities in the warehouse. Likewise, new models of existing physical devices may be added to the warehouse, as shown in Figure 33.

This figure shows the addition of a new conveyor belt that handles heavier goods in a warehouse. The ITS contains many software controllers, which collectively manage the entire system. For example, a software controller component manages each physical device controlled by the warehouse hardware subsystem. When a new device is introduced, a new component implementation must be loaded dynamically into the ITS. Likewise, when a new version of a physical device arrives, the component that controls this device should be updated so the software can manage the new version. ITS vendors are responsible for providing these new implementations.

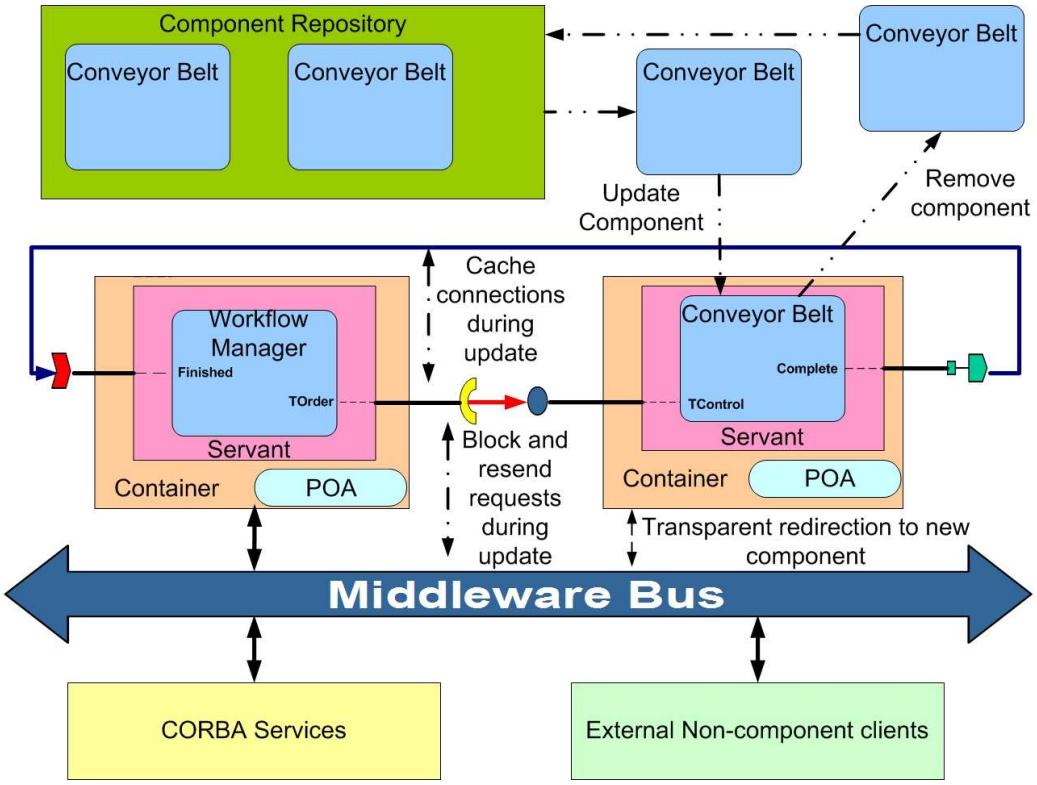

Figure 33: Component Updating Scenario in ITS

As shown in Figure 33, a workflow manager component is connected to a conveyor belt component using a facet/receptacle pair and an event source/sink pair. To support this scenario, the ITS needs middleware that can satisfy the following three requirements:

1. Consistent and uninterrupted updates to clients. As part of the dynamic update process, a component's implementation is deactivated, removed, and updated. To ensure that the ITS remains consistent and uninterrupted during this process, the middleware must ensure that (1) ongoing invocations between a component and a client are completed and (2) new invocations from clients to a component are blocked until its implementation has been updated. Figure 33 shows that when a conveyor belt's component implementation is updated, pending requests from the workflow manager to the conveyor belt component to move a new good to a storage system should be available for processing after the implementation is updated. Section VI.3.1 explains how SwapCIAO supports this requirement.

2. Efficient client-transparent dynamic component updates. After a component is updated, the blocked invocations from clients should be redirected to the new component implementation. This reconfiguration should be transparent to clients, *i.e.*, they should not need to know when the change occurred, nor should they incur any programming effort or runtime overhead to communicate with the new component implementation. Figure 33 shows how a client accessing an ITS component should be redirected to the updated component transparently when dynamic reconfiguration occurs. Section VI.3.2 explains how SwapCIAO supports this requirement.

3. Efficient (re)connections of components. Components being updated may have connections to other components through the ports they expose. The connected components and the component being updated share a requires/provides relationship by exchanging invocations through the ports. In Lightweight CCM, these connections are established at deployment time using data provided to the deployment framework in the form of XML descriptors. During dynamic reconfiguration, therefore, it is necessary to cache these connections so they can be restored immediately after reconfiguration. Figure 33 shows how, during the update of a conveyor belt component, its connections to the workflow manager component must be restored immediately after the new updated conveyor belt component implementation is started. Section VI.3.3 explains how SwapCIAO supports this requirement.

VI.3 The SwapCIAO Dynamic Component Updating Framework

This section describes the design of SwapCIAO, which is a C++ framework that extends CIAO to support dynamic component updates. Figure 34 shows the following key elements in the SwapCIAO framework:

- SwapCIAO's *component implementation language definition* (CIDL) compiler supports the *updatable* option, which triggers generation of "glue code" that (1) defines a factory interface to create new component implementations, (2) provides hooks

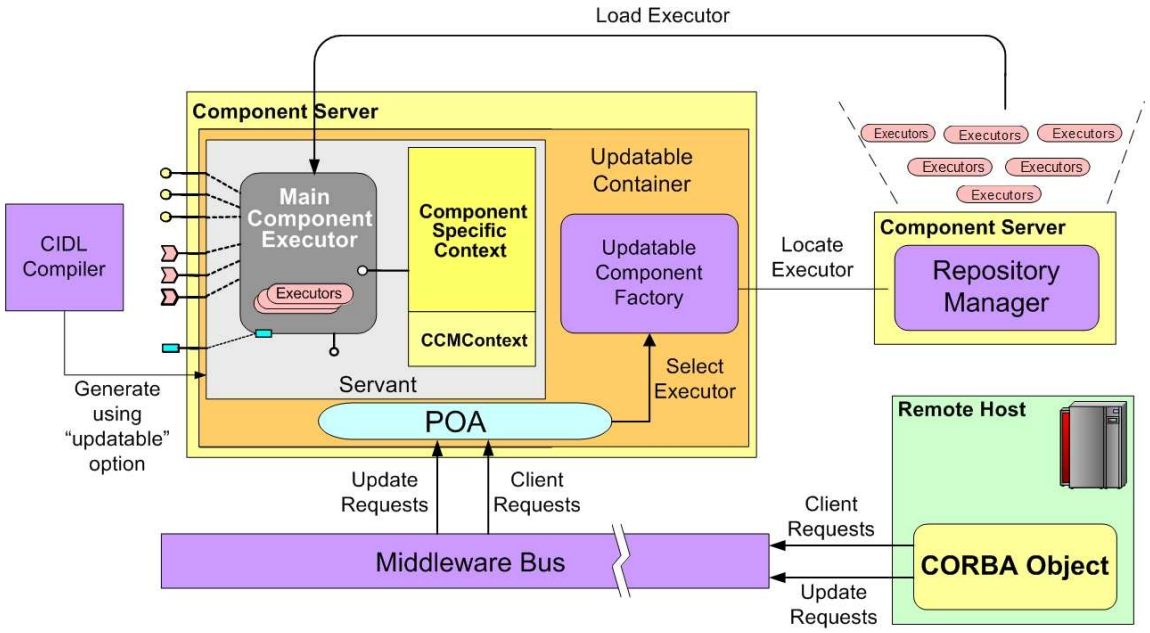

Figure 34: Dynamic Interactions in the SwapCIAO framework

for server application developers to choose which component implementation to deploy, (3) creates, installs, and activates components within a portable object adapter (POA) [63, 112] chosen by an application, and (4) manages the port connections of an updatable component.

- The *updatable container* provides an execution environment in which component implementations can be instantiated, removed, updated, and (re)executed. An updatable container enhances the standard Lightweight CCM *session container* [139] to support additional mechanisms through which component creation and activation can be controlled by server application developers.

- The *updatable component factory* creates components and implements a wrapper facade [140] that provides a portable interface used to implement the Component Configurator pattern [140], which SwapCIAO uses to open and load dynamic link libraries (DLLs) on heterogeneous run-time platforms.

- The *repository manager* stores component implementations. SwapCIAO's updatable component factory uses the repository manager to search DLLs and locate component implementations that require updating.

The remainder of this section describes how the SwapCIAO components in Figure 34 address the requirements presented in Section VI.2.2.

VI.3.1 Providing Consistent and Uninterrupted Updates to Clients

Problem. Dynamic updates of component implementations can occur while interactions are ongoing between components and their clients. For example, during the component update process, clients can initiate new invocations on a component – there may also be ongoing interactions between components. If these scenarios are not handled properly by the middleware some computations can be lost, yielding state inconsistencies.

Solution → Reference counting operation invocations. In SwapCIAO, all operation invocations on a component are dispatched by the standard Lightweight CCM portable object adapter (POA), which maintains a *dispatching table* that tracks how many requests are being processed by each component in a thread. SwapCIAO uses standard POA reference counting and deactivation mechanisms [124] to keep track of the number of clients making invocations on a component. After a server thread finishes processing the invocation, it decrements the reference count in the dispatching table.

When a component is about to be removed during a dynamic update, the POA does not deactivate the component until its reference count becomes zero, *i.e.*, until the last invocation on the component is processed. To prevent new invocations from arriving at the component while it is being updated, SwapCIAO's updatable container blocks new invocations for this component in the server ORB using standard CORBA portable interceptors [164].

Applying the solution to ITS. In the ITS case study, when the conveyor belt component implementation is being updated, the warehouse hardware system could be issuing requests to the conveyor belt component to move goods. The updatable container (which runs in the

same host as the conveyor belt component) instructs the SwapCIAO middleware to block those requests. After the requests are blocked by SwapCIAO, the updatable container's POA deactivates the conveyor belt component only when all requests it is processing are completed, *i.e*, when its reference count drops to zero.

VI.3.2 Ensuring Efficient Client-transparent Dynamic Component Updates

Problem. As shown in the Figure 34, many clients can access a component whose implementation is undergoing updates during the dynamic reconfiguration process. In Lightweight CCM, a client holds an object reference to a component. After a component implementation is updated, old object references are no longer valid. The dynamic reconfiguration of components needs to be transparent to clients, however, so that clients using old references to access updated component do not receive "invalid reference" exceptions. Such exceptions would complicate client application programming and increase latency by incurring additional round-trip messages, which could unduly perturb the QoS of component-based DRE systems.

Solution → Use servant activators to redirect clients to update components transparently. Figure 35 shows how SwapCIAO redirects clients transparently to an updated component implementation. During the component updating process, the old component implementation is removed. When a client makes a request on the old object reference after a component has been removed, the POA associated with the updatable container intercepts the request via a *servant activator*. This activator is a special type of intercepter that can dynamically create a component implementation if it is not yet available to handle the request. Since the component has been removed, the POA's active object map will have no corresponding entry, so the servant activator will create a new component implementation dynamically.

SwapCIAO stores information in the POA's active object map to handle client requests efficiently. It also uses CORBA-compliant mechanisms to activate servants via unique user

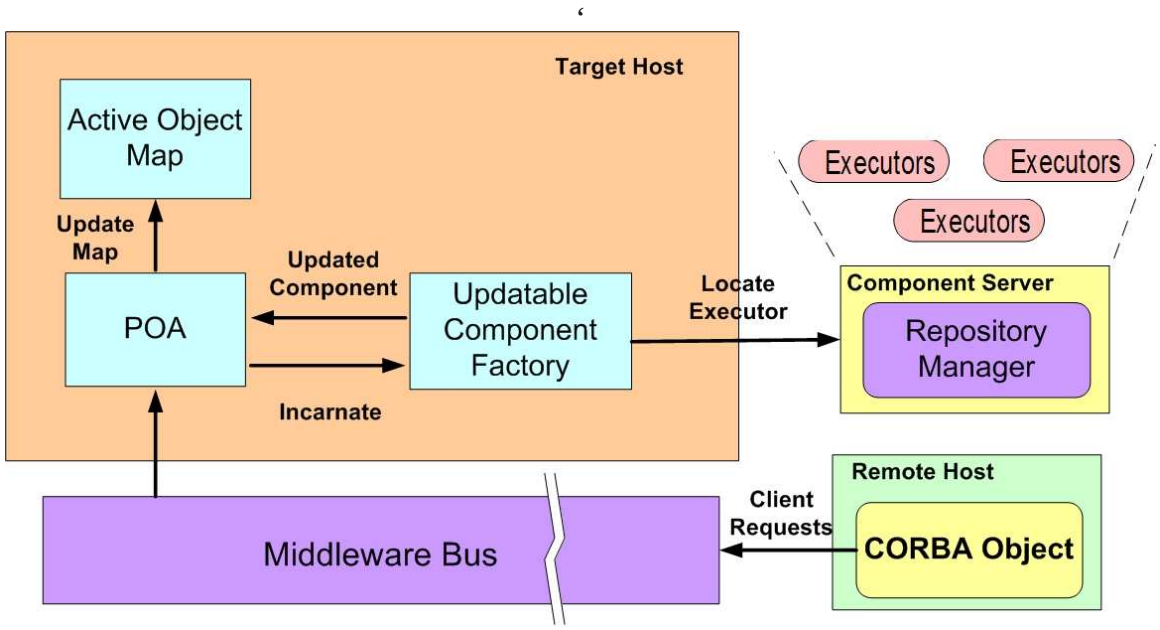

Figure 35: Transparent Component Object Reference Update in SwapCIAO

id's that circumvent informing clients of the updated implementation. This design prevents extra network round-trips to inform clients about an updated component's implementation.

Applying the solution to ITS. In the ITS case study, when the conveyor belt component implementation is being updated, the warehouse hardware system could be issuing requests to the conveyor belt component to move goods. After the current conveyor belt component is removed, the servant activator in the updatable container's POA intercepts requests from the warehouse hardware subsystem clients to the conveyor belt component. The servant activator then activates a new conveyor belt component implementation and transparently redirects the requests from the warehouse hardware subsystem to this updated implementation. SwapCIAO uses these standard CORBA mechanisms to enable different component implementations to handle the requests from warehouse hardware subsystem clients transparently, without incurring extra round-trip overhead or programming effort by the clients.

VI.3.3 Enabling (Re)connections of Components

Problem. As discussed in Section A.1, Lightweight CCM applications use the standard OMG Deployment and Configuration (D&C) [109] framework to parse XML assembly descriptors and deployment plans, extract connection information from them, and establish connections between component ports. This connection process typically occurs during DRE system initialization. When component implementations are updated, it is therefore necessary to record each component's connections to its peer components since their XML descriptors may not be available to establish the connections again. Even if the XML is available, reestablishing connections can incur extra round-trip message exchanges across the network.

Solution → Caching component connections Figure 36 shows how SwapCIAO handles component connections during the component update process. During the component up-

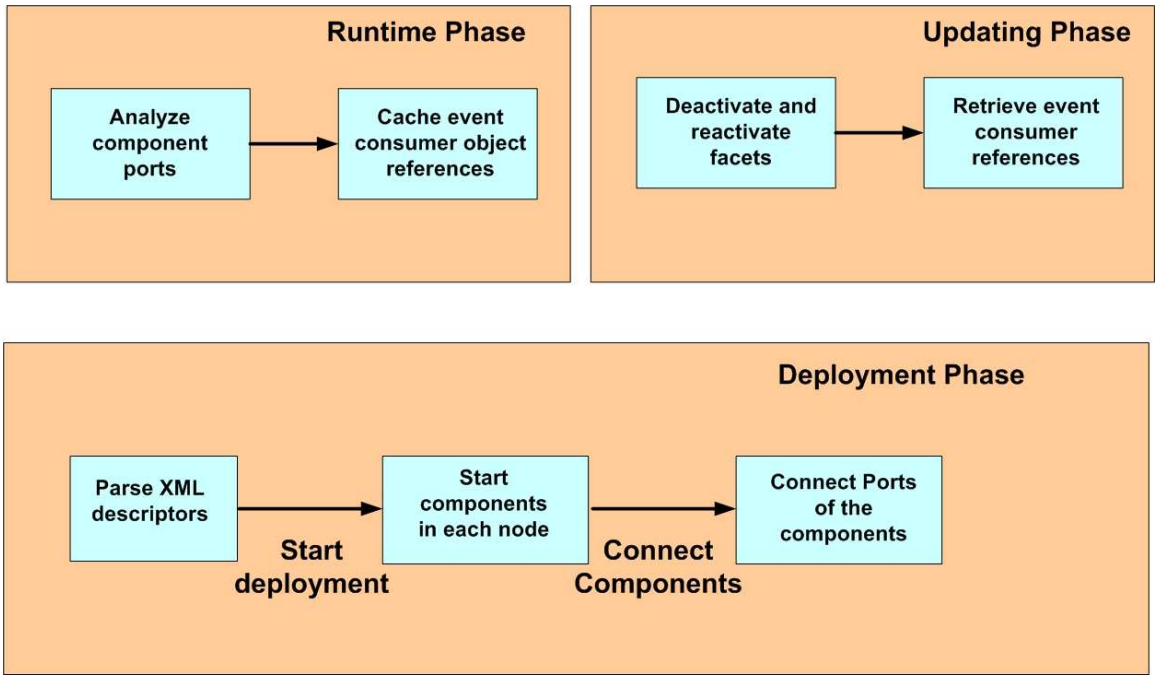

Figure 36: Enabling (Re)connections of Components in SwapCIAO

dating process, SwapCIAO caches component connections to any of its peer component

ports. SwapCIAO automatically handles the case where the updated component is a facet and the connected component is a receptacle. Since the receptacle could make requests on the facet while the component implementation is being updated, SwapCIAO uses the mechanisms described in Section VI.3.1 to deactivate the facets properly, so that no invocations are dispatched to the component. When the new component is activated, the facets are reactivated using the SwapCIAO's POA servant activator mechanism discussed in Section VI.3.2. For event source and event sinks, if the component being updated is the publisher, SwapCIAO caches the connections of all the connected consumers. When the updated component implementation is reactivated, its connections are restored from the cache. As a result, communication can be started immediately, without requiring extra network overhead.

Applying the solution to ITS. In the ITS, a conveyor belt component in the warehouse hardware subsystem is connected to many sensors that assist the conveyor belt in tracking goods until they reach a storage system. When a conveyor belt component is updated, its connections to sensor components are cached before deactivation. When the updated conveyor belt component implementation is reactivated, the cached connections are restored and communication with the sensors can start immediately and all requests blocked during the update process will then be handled.

VI.4 Empirical Results

This section presents the design and results of experiments that empirically evaluate how well SwapCIAO's dynamic component updating framework described in Section VI.3 addresses the requirements discussed in Section VI.2.2. We focus on the performance and predictability of SwapCIAO's component updating mechanisms provided by version 0.4.6 of SwapCIAO. All experiments used a single 850 MHz CPU Intel Pentium III with 512 MB RAM, running the RedHat Linux 7.1 distribution, which supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. The benchmarks ran in the

POSIX real-time thread scheduling class [73] to increase the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

Figure 37 shows key component interactions in the ITS case study shown in Figure 32 that motivated the design of these benchmarks using SwapCIAO.

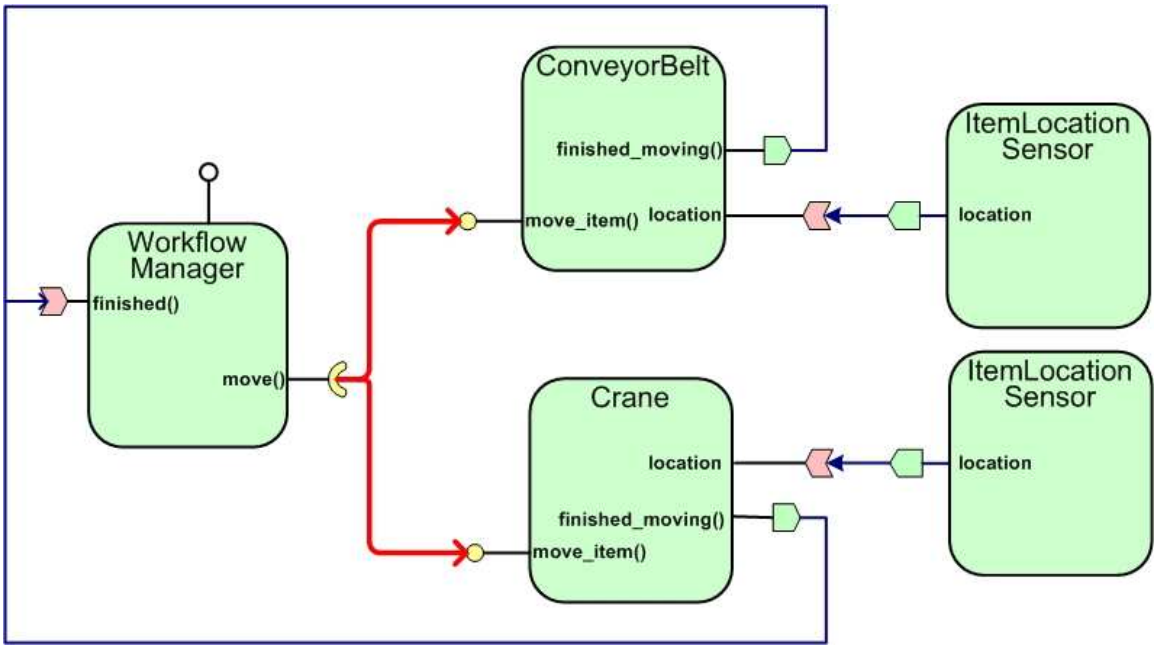

Figure 37: Component Interaction in the ITS

As shown in this figure, the workflow manager component of the material flow control subsystem is connected to the conveyor belt and forklift transportation units of the warehouse hardware subsystem. We focus on the scenario where the workflow manager contacts the conveyor belt component using the `move_item()` operation to instruct the conveyor belt component to move an item from a *source* (such as a loading dock) to a *destination* (such as a warehouse storage location). The `move_item()` operation takes source and destination locations as its input arguments. When the item is moved to its destination successfully, the conveyor belt component informs the workflow manager using the

`finished_moving()` event operation. The conveyor belt component is also connected to various sensor components, which determine if items fall off the conveyor belt. It is essential that the conveyor belt component not lose connections to these sensor components when component implementation updates occur.

During the component updating process, workflow manager clients experience some delay. Our benchmarks reported below measure the delay and jitter (which is the variation of the delay) that workflow manager clients experience when invoking operations on conveyor belt component during the component update process. They also measure how much of the total delay is incurred by the various activities that SwapCIAO performs when updating a component implementation. In our experiments, all components were deployed on the same machine to alleviate the impact of network overhead in our experimental results.

The core CORBA benchmarking software is based on the single-threaded version of the "`TestSwapCIAO`" performance test distributed with CIAO.[2] This benchmark creates a session for a single client to communicate with a single component by invoking a configurable number of `move_item()` operations. The conveyor belt component is connected to the sensor components using event source/sink ports.

Section VI.3.3 describes how caching and reestablishing connections to peer components are important steps in the component updating process. We therefore measured the scalability of SwapCIAO when an updated component has upto 16 peer components using event source/sink ports. The tests can be configured to use either the standard Lightweight CCM session containers or SwapCIAO's updatable containers (described in Section VI.3). `TestSwapCIAO` uses the default configuration of TAO, which uses a reactive concurrency model to collect replies.

[2]The source code for `TestSwapCIAO` is available at `www.dre.vanderbilt.edu/~jai/TAO/CIAO/performance-tests/SwapCIAO`.

VI.4.1 Measuring SwapCIAO's Updatable Container Overhead for Normal Operations

Rationale. Section VI.3 described how SwapCIAO extends Lightweight CCM and CIAO to support dynamic component updates. DRE systems do not always require dynamic component updating, however. It is therefore useful to compare the overhead of SwapCIAO's updatable container versus the standard Lightweight CCM session container under *normal operations* (*i.e.*, without any updates) to evaluate the tradeoffs associated with this feature.

Methodology. This experiment was run with two variants: one using the SwapCIAO updatable container and the other using the standard CIAO session container. In both experiemnts, we used high-resolution timer probes to measure the latency of `move_item()` operation from the workflow manager component to the conveyor belt component. Since SwapCIAO caches and restores a component's connections to its peer components, we varied the number of sensor components connected to the conveyor belt and then collected latency data with 2, 4, 8, and 16 ports to determine whether SwapCIAO incurred any overhead with additional ports during normal operating mode. The `TestSwapCIAO` client made 200,000 invocations of `move_item()` operation to collect the data shown in Figure 38.

Analysis of results. Figure 38 shows the comparitive latencies experienced by the workflow manager client when making invocations on conveyor belt component created with the session container versus the updatable container. These results indicate that no appreciable overhead is incurred by SwapCIAO's updatable container for normal operations that do not involve dynamic swapping.

The remainder of this section uses the results in Figure 38 as the *baseline processing delay* to evaluate the delay experienced by workflow manager clients when dynamic updating of a conveyor belt component occurs.

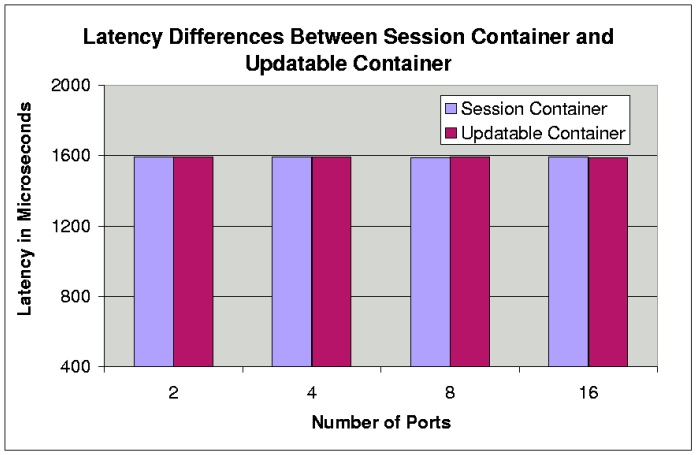

Figure 38: Overhead of SwapCIAO's Updatable Container

VI.4.2 Measuring SwapCIAO's Updatable Container Overhead for Updating Operations

Rationale. Evaluating the efficiency, scalability, and predictability of SwapCIAO's component updating mechanisms described in Section VI.3.2 and Section VI.3.3 is essential to understand the tradeoffs associated with updatable containers. SwapCIAO's *component update time* includes (1) the *removal time*, which is the time SwapCIAO needs to remove the existing component from service, (2) the *creation time*, which is the time SwapCIAO needs to create and install a new component, and (3) the *reconnect time*, which is the time SwapCIAO needs to restore a component's port connections to its peer components.

Methodology. Since the number of port connections a component has affects how quickly it can be removed and installed, we evaluated SwapCIAO's component update time by varying the number of ports and measuring the component's:

- *Removal time*, which was measured by adding timer probes to SwapCIAO's `CCM_Object::remove()` operation, which deactivates the component servant, disassociates the executor from the servant, and calls `ccm_passivate()` on the component.

- *Creation time*, which was measured by adding timer probes to SwapCIAO's `PortableServer::ServantActivator::incarnate()` operation, which creates and installs a new component, as described in Section VI.3.2.

- *Reconnect time*, which was measured by adding timer probes to `CCM_Object::ccm_activate()`, which establishes connections to ports.

We measured the times outlined above whenever a component update occurs during a `move_item()` call for 200,000 iterations and then calculated the results presented below.

Analysis of creation time. Figure 39 shows the minimum, average, and maximum latencies, as well as the 99% latency percentile, incurred by SwapCIAO's servant activator to create a new component, as the number of ports vary from 2, 4, 8, and 16. This figure shows that latency grows linearly as the number of ports initialized by `PortableServer::ServantActivator::incarnate()` increases. It also shows that SwapCIAO's servant activator spends a uniform amount of time creating a component and does not incur significant overhead when this process is repeated 200,000 times. SwapCIAO's creation mechanisms described in Section VI.3.2 are therefore efficient, predictable, and scalable in *ensuring efficient client-transparent dynamic component updates*.

Analysis of reconnect time. Figure 40 shows the minimum, average, and maximum latencies, as well as 99% latency percentile, incurred by SwapCIAO's reconnect mechanisms to restore a new component's connections, as the number of ports vary from 2, 4, 8, and 16. As shown in the figure, the reconnect time increases linearly with the number of ports per component. These results indicate that SwapCIAO's reconnect mechanisms described

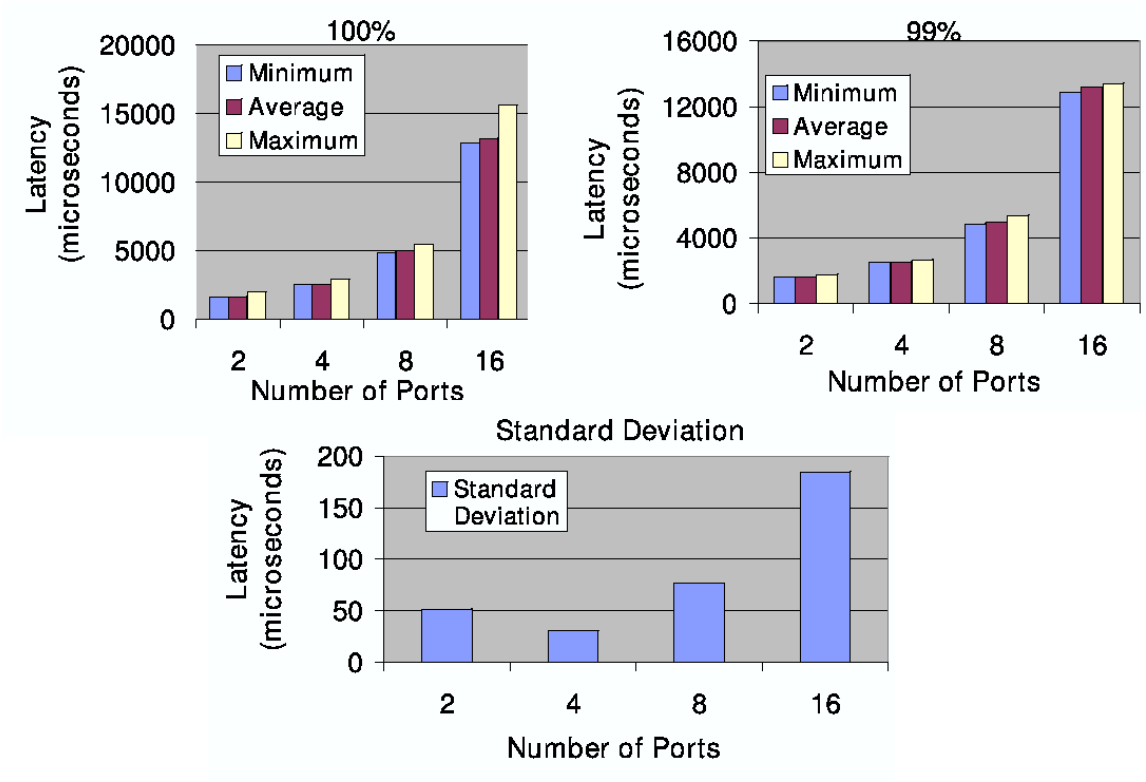

Figure 39: Latency Measurements for Component Creation

in Section VI.3.3 provide *efficient (re)connection of components* and do not incur any additional roundtrip delays by propagating exceptions or sending GIOP LOCATE_FORWARD messages to restore connections to components.

Analysis of removal time. Figure 41 shows the time used by SwapCIAO's removal mechanisms to cache a component's connections and remove the component from service, as a function of the number of its connected ports. This removal time increases linearly with the number of ports, which indicates that SwapCIAO performs a constant amount of work to manage the connection information for each port. SwapCIAO's removal mechanisms described in Section VI.3.1 are therefore able to *provide consistent and uninterrupted updates to clients*.

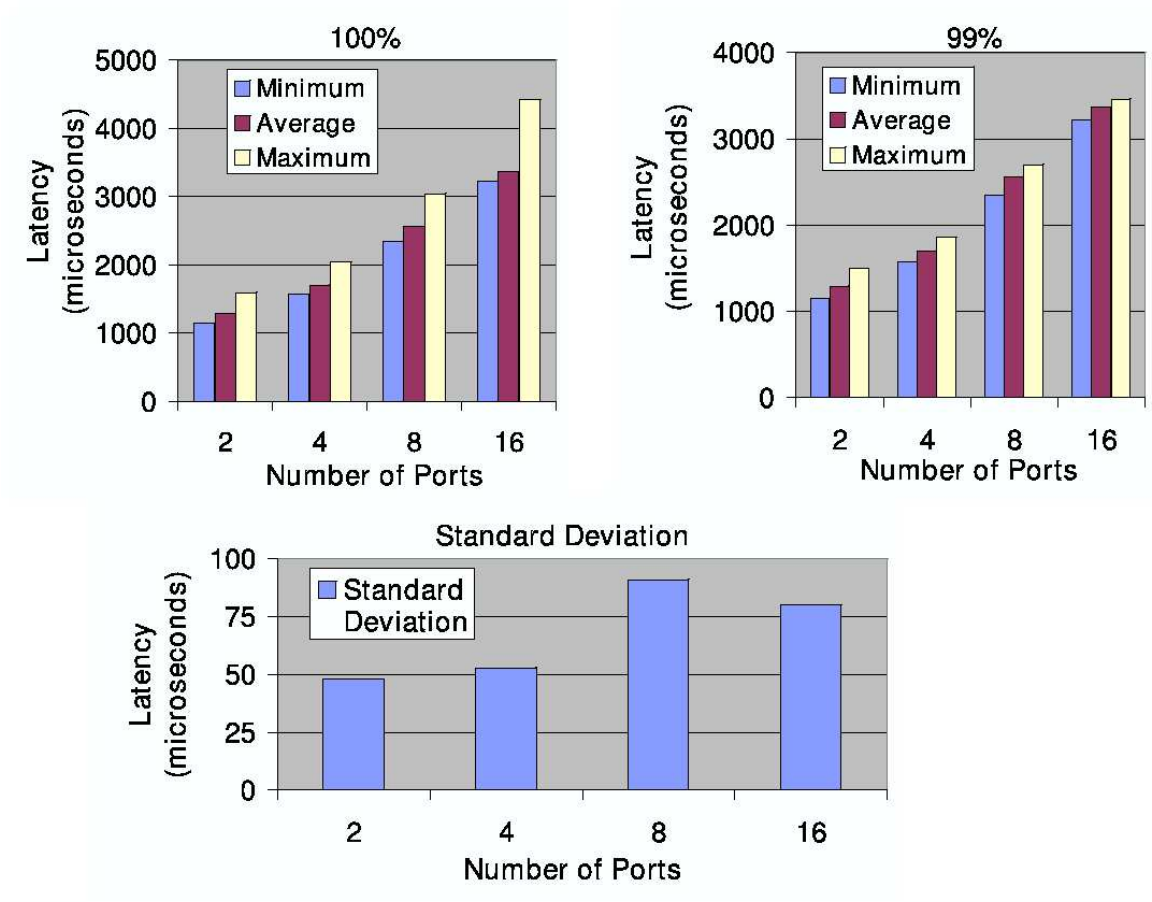

Figure 40: Latency Measurements for Reconnecting Component Connections

VI.4.3 Measuring the Update Latency Experienced by Clients

Rationale. Section VI.3.2 describes how SwapCIAO's component creation mechanisms are transparent to clients, efficient, and predictable in performing client-transparent dynamic component updates. Section VI.4.2 showed that SwapCIAO's standard POA mechanisms and the servant activator create new component implementations efficiently and predictably. We now determine whether SwapCIAO incurs any overhead – other than the work performed by the SwapCIAO's component creation mechanisms – that significantly affects client latency.

Methodology. The *incarnation delay* is defined as the period of time experienced by a client when (1) its operation request arrives at a server ORB after SwapCIAO has removed

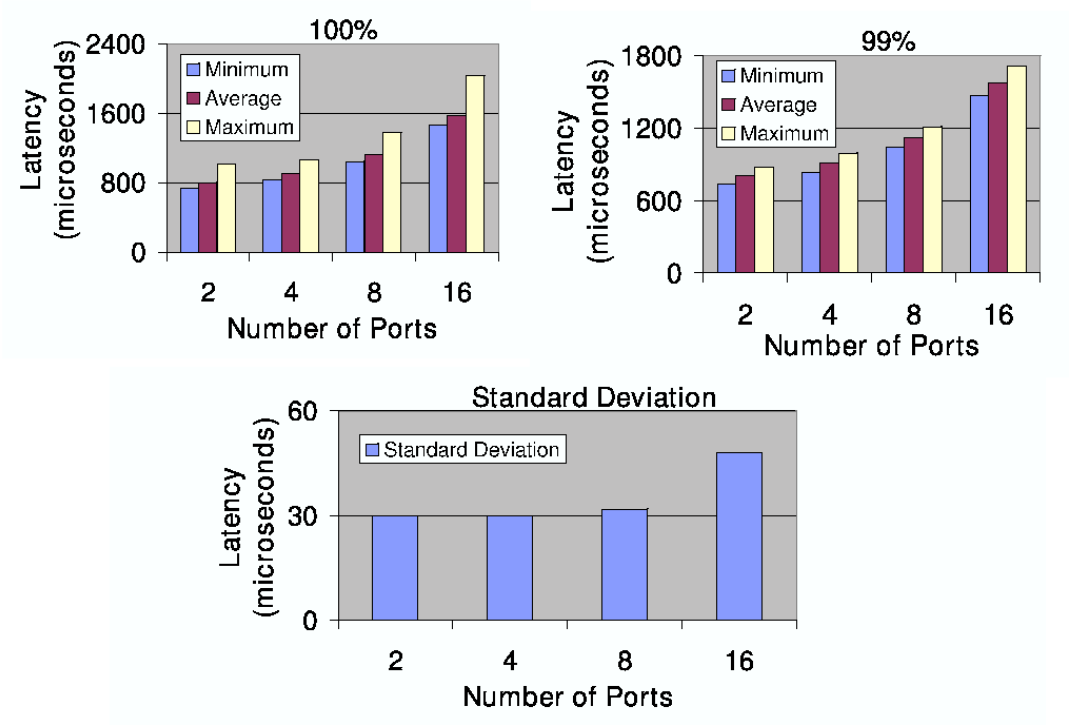

Figure 41: Latency Measurements for Component Removal

the component and (2) it receives the reply after SwapCIAO creates the component, restores the component's connections to peer components, and allows the updated component to process the client's request. The incarnation delay therefore includes the *creation time*, *reconnect time*, and *processing delay* (which is the time a new component needs to process the operation request and send a reply to the client). To measure incarnation delay, we (1) removed a component and (2) started a high-resolution timer when the client invokes a request on the component. We repeated the above experiment for 200,000 invocations and measured the latency experienced by the client for each invocation. We also varied the number of ports between 2, 4, 8, and 16 as described in Section VI.4.2 to measure the extent to which SwapCIAO's component creation process is affected by the number of ports connected to a component.

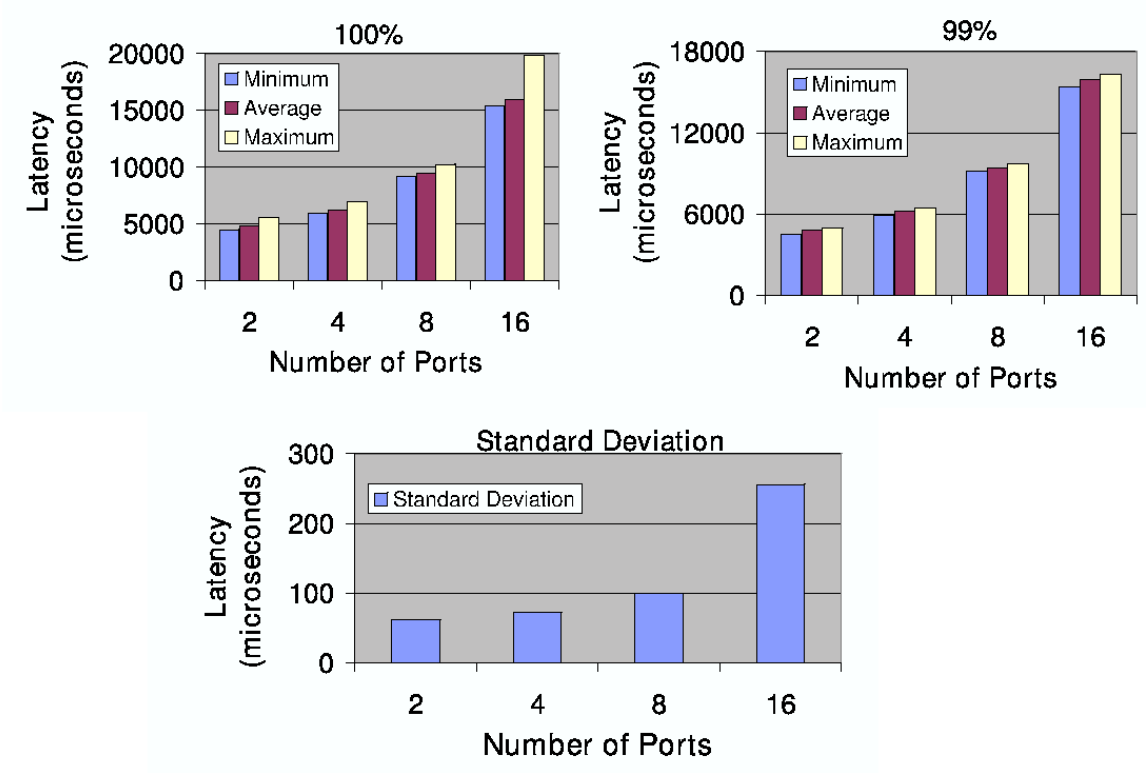

Figure 42: Client Experienced Incarnation Delays during Transparent Component Updates

Analysis of results. Figure 42 shows the delay experienced by a client as SwapCIAO creates a component with a varying number of connections to process client requests. By adding the delays in Figure 39, Figure 40, and Figure 38 and comparing them with the delays in Figure 42, we show how the incarnation delay is roughly equal to the sum of the creation time, reconnect time, and processing delay, regardless of whether the client invokes an operation on a updating component with ports ranging from 2, 4, 8, to 16.

These results validate our claim in Section VI.3.2 that SwapCIAO provides component updates that are transparent to clients. In particular, if SwapCIAO's servant activator did not transparently create the component and process the request, the client's delay incurred

obtaining a new object reference would be larger than the sum of the creation time, reconnect time, and the processing delay. We therefore conclude that SwapCIAO provides efficient and predictable client transparent updates.

VI.5 Summary

This chapter describes the design and implementation of SwapCIAO, which is a QoS-enabled component middleware framework based on Lightweight CCM that supports dynamic component updating. SwapCIAO is designed to handle dynamic operating conditions by updating component implementations that are optimized for particular run-time characteristics. The lessons learned while developing SwapCIAO and applying it to the ITS case study include:

- Standard Lightweight CCM interfaces can be extended slightly to develop a scalable and flexible middleware infrastructure that supports dynamic component updating. In particular, SwapCIAO's extensions require minimal changes to the standard Lightweight CCM server programming model. Moreover, its client programming model and client/server interoperability were unaffected by the server extensions. Developers of client applications in our ITS case study were therefore shielded entirely from SwapCIAO's component updating extensions.

- By exporting component implementations as DLLs, SwapCIAO simplifies the task of updating components by enabling their implementations to be linked into the address space of a server late in its lifecycle, *i.e.*, during the deployment and reconfiguration phases. These capabilities enabled developers in the ITS case study to create multiple component implementations rapidly and update dynamically in response to changing modes and operational contexts.

- SwapCIAO adds insignificant overhead to each dynamic component updating request. It can therefore be used even for normal operations in ITS applications that

do not require dynamic component updating. Moreover, due to the predictability and transparency provided by SwapCIAO, it can be used efficiently when operating conditions trigger mode changes.

CHAPTER VII

CONCLUDING REMARKS

Timeliness and high availability are two key quality of service (QoS) properties that must be assured for the correct operation of distributed real-time and embedded (DRE) systems. DRE systems are composed of multiple services and client applications that are deployed across local or metropolitan networks. Often these services and client applications are part of multiple end-to-end workflows that operate in environments that are constrained in the number of resources (*e.g.*, CPU, network bandwidth). Moreover, these systems operate in environments that are highly dynamic and where processor or process failures and system workload changes are common. System workloads in DRE systems could range from being statically known (closed DRE system) to being dynamic (open DRE system).

Middleware is a key software capability needed to support DRE systems. Designing middleware that satisfies the QoS requirements of DRE systems is hard because it needs to (1) integrate real-time and fault-tolerance by design, which is not straightforward due to the conflicting demands each QoS dimension imposes on the available resources, (2) be lightweight so that it is suitable for resource-constrained deployments, and (3) be adaptive so that availability and timeliness properties can be tuned dynamically at runtime to maintain soft real-time and fault-tolerant performance. Satisfying these three requirements needs a systematic and scientific approach to realizing such a middleware.

This dissertation describes the design, development and experimental evaluation of middleware-based mechanisms that provide both high availability and soft real-time performance simultaneously for both the open and closed types of DRE systems. Specifically, this research makes three contributions in the form of algorithms, architectures, and mechanisms that together address the above-described challenges as follows:

- First, it discusses a novel deployment and configuration framework for fault-tolerant DRE systems called DeCoRAM, which provides a novel replica allocation algorithm that is (1) *failure-aware, i.e.,* it handles multiple processor failures using passive replication and considers primary replicas, backup replicas, and state synchronization cost in the replica allocation problem, (2) *resource-aware, i.e.*, it minimizes number of processors used by opportunistically overbooking processors with multiple backup replicas after analyzing feasible failover patterns due to multiple processor failures, and (3) *real-time-aware, i.e.*, meet real-time performance requirements both in normal conditions and after multiple processor failures.

- Second, it presents the design and implementation of *Network QoS Provisioning Engine (NetQoPE)*, which is a model-driven, component middleware framework that deploys and configures applications in the nodes chosen by DeCoRAM's replica allocation algorithm and eliminates manual tasks developer heretofore used to implement replica allocation decisions. NetQoPE provides flexible and non-invasive QoS configuration and provisioning capabilities by leveraging CPU and network QoS mechanisms without modifying application source code.

- Third, it presents the Fault-tolerant, Load-aware and Adaptive middlewaRe (FLARe) for distributed soft real-time applications. FLARe features (1) the Load-aware and Adaptive Failover (LAAF) strategy that adapts failover targets based on system load; (2) the Resource Overload Management Redirector (ROME) strategy that dynamically enforces CPU utilization bounds to maintain desired server delays in face of concurrent failures and load changes; and (3) an efficient fault-tolerant middleware architecture that supports transparent failover to passive replicas. The dissertation

also describes SwapCIAO, which is a QoS-enabled component middleware framework based on Lightweight CCM that supports dynamic component updating. SwapCIAO is designed to handle dynamic operating conditions by updating component implementations that are optimized for particular run-time characteristics.

VII.1 Broader Impact and Future Research Directions

Although this research was conducted in the context of DRE systems, the principles are applicable to a wider range of distributed systems. Moreover, this research is by no means solving all the issues in the problem space. At the same time it opens up new opportunities for research. Below we present its broader applicability and some directions for future research based on our experience in designing and developing algorithms, architectures, and middleware mechanisms for fault-tolerant DRE systems.

1. **Tunable application performance versus consistency.** The requirement to provide both high availability as well as satisfactory response times for clients in a passively replicated environment is conflicting in many ways. For example, to provide better fault-tolerance, the *backup* replica's (there could be more than one *backup* replica) state must be made consistent every time the state of the *primary* replica changes. This approach reduces failure recovery time since any one of the available *backup* replicas can be promoted to be the new *primary* replica during failure recovery, and the clients could be quickly redirected to the new *primary* replica.

   However, this approach also increases response times perceived by client applications since the *primary* replica does not respond to the clients until the state of all the *backup* replicas is made consistent with the state of the *primary* replica. The response times perceived by the client applications depends on the time taken to synchronize the state of the *backup* replica operating in the slowest physical host. On the other hand, to provide satisfactory response times for clients and to minimize usage of available resources for fault-tolerance purposes, a *backup* replica's state can be made

consistent only during failure recovery. Although, this approach reduces network and CPU resource usage, it also incurs longer recovery times, which might not be acceptable for certain applications.

As described above, many different alternatives are available to synchronize the state of the *backup* replicas, and each of the alternatives can be characterized based on the response times provided to the clients, the recovery time after failures, and the resources consumed. It is important to provide policies for tradeoff between these three different aspects and mechanisms to tune these policies all at deployment time, when the applications and their replicas are deployed. This will help quantify deployment-time assurances on the consistency characteristics that can be provided to the applications via the middleware. Further, as the deployed applications and their replicas operate in dynamic environments (new applications are deployed; new hardware hosts are introduced; failures occur), such characteristics need to be tuned adaptively depending on the needs and importance of the applications.

2. **Resource-aware fault-tolerance through dynamic adaptation.** This dissertation demonstrated how to use our FLARe middleware to dynamically adjust failover targets at runtime in response to system load fluctuations and resource availability. We also demonstrated how FLARe adaptively maintains soft real-time performance for clients operating in the presence of failures and overloads with negligible runtime overhead. We now discuss some of the challenges in extending our work to perform more adaptations to maintain high availability and soft real-time performance simultaneously.

   Our adaptive and load-aware solution in FLARe is built upon a centralized monitoring architecture, where a centralized replication manager works with all the monitors in a distributed system to obtain and record performance characteristics at all the hardware nodes to make adaptive real-time fault-tolerance decisions. Although,

this architecture is reasonably applicable for a large number of DRE systems, this assumption is not valid for certain DRE systems, where the networks fail causing severe resource contention in the remaining links. To address this problem, one potential future approach to explore is to design and develop adaptive real-time fault-tolerance solutions that are based on a decentralized feedback architecture. Another potential future approach is also to integrate FLARe with network fault-tolerance techniques [28, 150], so that a centralized architecture could still be adopted, however, with better network high availability assurances.

3. **Enhancements to resource overload management.** This dissertation also demonstrated how our FLARe middleware employs the *Resource Overload Management and rEdirection (ROME)* strategy to dynamically enforce CPU utilization bounds to maintain desired server delays in face of concurrent failures and load changes. In the case of overloads, clients of the current primary replicas are redirected automatically to the chosen new backup replicas. This overload management strategy will succeed only if the CPU utilization at the least-loaded processor of the backup replicas does not exceed the *schedulable utilization bound* if the migration occurs. In scenarios where overloads cannot be mitigated using migrations, newer overload management solutions are required that work in conjunction with a real-time fault-tolerant middleware to maintain application required QoS.

   One potential future approach is to integrate FLARe with advanced overload management techniques such as *m,k* firm guarantees [127]. *m,k* firm guarantee techniques control the behavior of applications by modifying the real-time period [84] of their invocations. Since DRE systems are composed of services that participate in end-to-end flows, modifying the behavior of one of the services could impact the behavior (*e.g.*, real-time period of those services) of other services in the end-to-end application flows. If the real-time properties of end-to-end application flows are not managed

properly, application QoS assurances could be affected. Hence, another potential future approach is to integrate FLARe with end-to-end utilization control services [90], so that overload management techniques could be designed and modeled using rigorous control-theoretic techniques and can provide robust and analytically sound QoS assurance.

4. **Application to other replication schemes.** ACTIVE and PASSIVE replication [61] are two common approaches for building fault-tolerant distributed applications that provide high availability and satisfactory response times for performance-sensitive distributed applications operating in dynamic environments. In addition to DRE systems, fault-tolerance has also been studied in the context of other systems [16, 17, 92, 121, 130, 134, 168] . In such systems, prior research efforts have focused on dynamically trading consistency for availability so that clients could be provided high availability with high throughput and shorter response times [79, 80, 159, 169].

   Chain replication [159] is one alternate replication scheme that is developed for such systems [80, 92]. In contrast to ACTIVE and PASSIVE replication schemes, that group replicas by their roles, chain replication groups replicas by the functionality provided to the clients. The replicas are divided into two groups: *read* and *write*. All the updates from the clients are forwarded to the *write* group, while all the read requests from the clients are forwarded to the *read* group. Weaker consistency is provided by employing a state update protocol that propagates updates from the *write* group to the *read* group using a chain, which connects all the replicas in the group. Client read requests are not blocked as the state update propagates, and this provides high throughput for the clients.

   One interesting future research agenda would be to experiment the effect of chain replication schemes on the timeliness assurances for DRE systems. Specifically, the

performance of the client read requests could be made predictable by taking advantage of a load balancer [4, 116, 117] (and by investigating new adaptive load balancing algorithms) that is built on top of middleware technologies that are most suited for developing DRE systems. Another interesting future research agenda would be to dynamically control the size of the *read* and *write* groups depending on the request patterns of the clients, there by providing predictable performance for client read as well as write requests.

5. **Fault-tolerance and scheduling in computational grids.** Large and complex scientific workflows rely on computational grids to have their large compute as well as data-intensive applications executed. With the large scale and extremely heterogeneous nature of the computational grids, executing those applications in a timely as well as a dependable manner becomes a huge challenge. Current fault-tolerance strategies [25, 52] for executing such complex workflows in a highly available manner rely on regularly obtaining checkpoints as the workflows execute, and restart the application from the last known checkpoint in case of a failure [42, 43].

   However, such reactive fault-tolerance schemes could incur significant performance loss because of computation repetition, and slow fault recovery time. Proactive fault-tolerance techniques [23, 82, 99] provide solutions for such problems by using failure prediction techniques to predict when a processor is more likely to fail, and subsequently migrating computations from that processor to other healthy processors. After computations are migrated, senders continue to send messages to those new processors to continue the overall workflow. The whole fault-tolerance and fault recovery process is orchestrated by the underlying middleware in an application transparent manner providing great flexibility in designing highly available computational grid applications.

   Many scientific applications and workflows are deadline-driven and hence need to

finish their computations within a certain time period. For such systems, even proactive fault-tolerance solutions might not work, as the time taken to complete a workflow depends on the processor that has been chosen for migration. After a failover or migration, the client perceived response times will depend on the loads of the processor hosting the new objects. Incorrect client redirections could overload a processor thereby affecting the response time(s) for the redirected client(s) and other clients that were already invoking remote operations on targets hosted on that processor. If a proactive fault-tolerance scheme has multiple processors available for migrating objects from a processor, *load-aware migration* decisions need to be made to determine the appropriate processor so that application performance is not affected after migration.

Hence, one potential future approach is to extend current proactive fault-tolerance schemes in the computational grids community with advanced *load-aware* overload as well as proactive fault-tolerance management schemes based on the adaptive resource management algorithms that have been designed and developed in the context of this dissertation.

APPENDIX A

UNDERLYING TECHNOLOGIES

This appendix summarizes the various technologies that are used to build the real-time fault-tolerant middleware solutions that are described in this thesis.

A.1 Overview of Lightweight CCM

The OMG Lightweight CCM (LwCCM) [108] specification standardizes the development, configuration, and deployment of component-based applications. LwCCM uses CORBA's distributed object computing (DOC) model as its underlying architecture, so applications are not tied to any particular language or platform for their implementations. *Components* in LwCCM are the implementation entities that export a set of interfaces usable by conventional middleware clients as well as other components. Components can also express their intent to collaborate with other components by defining *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components. *Homes* are factories that shield clients from the details of component creation strategies and subsequent queries to locate component instances.

Figure 43 illustrates the layered architecture of LwCCM, which includes the following entities:

- LwCCM sits atop an **object request broker** (ORB) and provides **containers** that encapsulate and enhance the CORBA portable object adapter (POA) demultiplexing mechanisms. Containers support various pre-defined hooks and strategies, such as

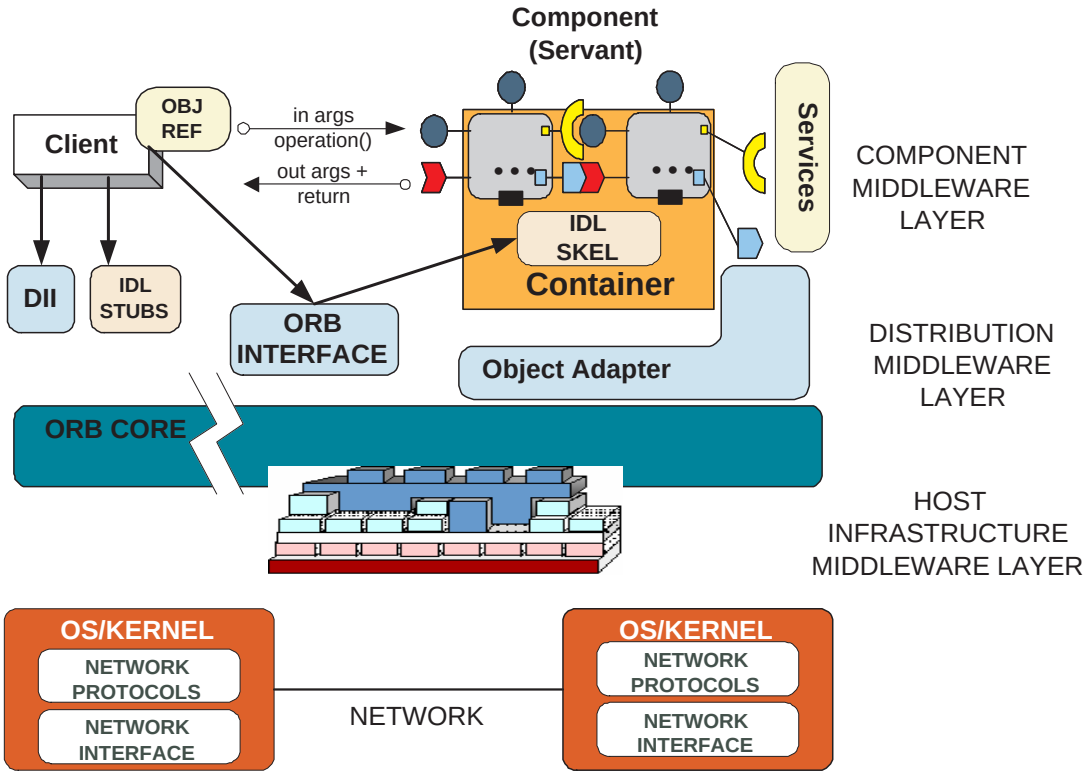

Figure 43: Layered LwCCM Architecture

persistence, event notification, transaction, and security, to the components it manages.

- A *component server* plays the role of a process that manages the homes, containers, and components.

- Each container manages one type of component and is responsible for initializing instances of this component type and connecting them to other components and common middleware services.

- The **component implementation framework** (CIF) consists of patterns, languages and tools that simplify and automate the development of component implementations which are called as **executors**. Executors actually provide the component's business logic.

- Component Implementation Definition Language (CIDL) is a text-based declarative language that defines the behavior of the components. In order to shield the component application developers from many complexities associated with programming POAs like servant activation and deactivation, a CIDL compiler generates infrastructure glue code called *servants*. Servants (1) activate components within the container's POA, (2) manage the interconnection of a component's ports to the ports of other components, (3) provide implementations for operations that allow navigation of component facets, and (4) intercept invocations on executors to transparently enact various policies, such as component activation, security, transactions, load balancing, and persistence.

- To initialize a instance of a component type, a container creates a component home. The component home creates instances of servants and executors and combines them to export component implementations to external world.

- Executors use servants to communicate with the underlying middleware and servants delegate business logic requests to executors. Client invocations made on the component are intercepted by the servants, which then delegate the invocations to the executors. Moreover, the containers can configure the underlying middleware to add more specialized services such as integrating event channel to allow components to communicate, add Portable Interceptors to intercept component requests, etc.

A.2 Overview of Component Middleware Deployment and Configuration

After components are developed and component assemblies are defined, they must be deployed and configured properly by deployment and configuration (D&C) services. The D&C process of component-based systems usually involves a number of service objects that must collaborate with each other. Figure 44 gives an overview of the OMG D&C model, which is standardized by OMG through the Deployment and Configuration

(D&C) [109] specification to promote component reuse and allow complex applications to be built by assembling existing components. As shown in the figure, since a component-based system often consists of many components that are distributed across multiple nodes, in order to automate the D&C process, these service objects must be distributed across the targeted infrastructure and collaborate remotely.

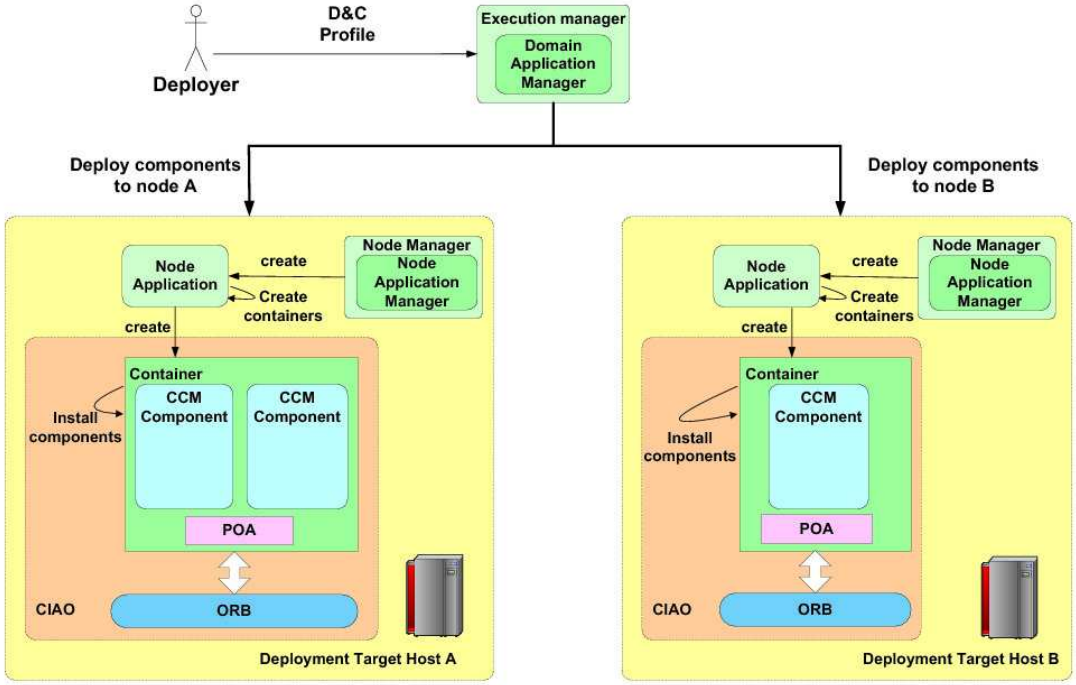

Figure 44: An Overview of OMG Deployment and Configuration Model

The run-time of the OMG D&C model standardizes the D&C process into a number of serialized phases. The OMG D&C Model defines the D&C process as a two-level architecture, one at the domain level and one at the node level. Since each deployment task involves a number of subtasks that have explicit dependencies with each other, these subtasks must be serialized and finished in different phases. Meanwhile, each deployment task involves a number of node-specific tasks, so each task is distributed.

A.3 Overview of Generic Modeling Environment (GME)

GME is a configurable toolkit for creating DSMLs and program synthesis environments. Third-generation programming languages, such as C++, Java, and C#, employ imperative techniques for development, deployment, and configuration of systems. For example, real-time QoS provisioning with object request brokers is conventionally done using imperative techniques that specify the QoS policies at the same level of abstraction as the mechanisms that implement those policies [123].

In contrast, GME-based DSMLs use a declarative approach that clearly separates the specification of policies from the mechanisms used to enforce the policies. Policy specification is done at a higher level of abstraction (and in less amount of detail), *e.g.*, using *models* and declarative configuration languages. Declarative techniques help relieve users from the intricacies of how the policies are mapped onto the underlying mechanisms implementing them, thereby simplifying policy modifications.

GME-based DSMLs are described using *metamodels*, which specify the modeling paradigm or language of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain, *e.g.*, which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the family of models that can be created using the resultant modeling environment.

For example, a DSML might represent the different hardware elements of a radar system and the relationships between them in a component middleware technology like LwCCM. Likewise, it might represent the different elements, such as *EJBComponent*, *EJBHome*, *EJBContainer* and *ApplicationServer*, that are present in a component middleware technology like EJB. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.

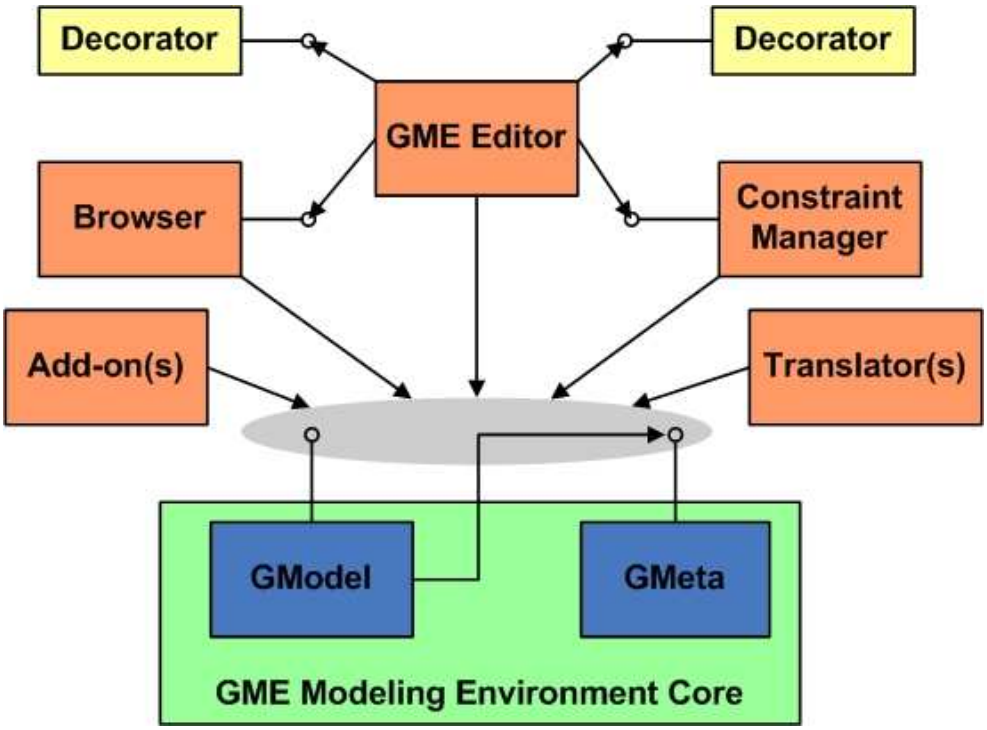

Figure 45: Overview of GME

To create metamodels and their associated DSMLs, GME uses a modular and component-based architecture as shown in Figure 45 (see [83] for a detailed overview of the GME architecture). Application developers create new DSMLs using the following core components of GME: (1) *GME Editor*, (2) *Browser*, (3) *Constraint Manager*, (4) *Translator*, and *Add-ons*. To support building large-scale and complex systems, GME's Editor and the Browser provide basic building blocks to model different entities of the system and express the relationships between those different entities. GME's Constraint Manager catches errors when models are constructed with incorrect relationships or associations. GME's Add-ons provide capabilities to extend the GME Editor, and its Translators support the analysis of models and synthesize various types of artifacts, such as source code, deployment descriptors, or simulator input.

A.4 Overview of Telcordia's Bandwidth Broker

Telcordia has developed a network management solution for QoS provisioning called the Bandwidth Broker [27], which leverages widely available mechanisms [18] that support Layer-3 DiffServ (Differentiated Services) and Layer-2 Class of Service (CoS) features in commercial routers and switches. DiffServ and CoS have two major QoS functionality/enforcement mechanisms:

- At the ingress of the network, traffic belonging to a flow is classified based on the 5-tuple (source IP address and port, destination IP address and port, and protocol) and DSCP (assigned by the Bandwidth Broker) or any subset of this information. The classified traffic is marked/re-marked with a DSCP as belonging to a particular class and may be policed or shaped to ensure that traffic does not exceed a certain rate or deviate from a certain profile.

- In the network core, traffic is placed into different classes based on the DSCP marking and provided differentiated, but consistent per-class treatment. Differentiated treatment is achieved by scheduling mechanisms that assign weights or priorities to different traffic classes (such as weighted fair queuing and/or priority queuing), and buffer management techniques that include assigning relative buffer sizes for different classes and packet discard algorithms, such as Random Early Detection (RED) and Weighted Random Early Detection (WRED).

These two features by themselves are insufficient to ensure end-to-end network QoS because the traffic presented to the network must be made to match the network capacity. What is also needed, therefore, is an adaptive admission control entity that ensures there are adequate network resources for a given traffic flow on any given link that the flow may traverse. The admission control entity should be aware of the path being traversed by each flow, track how much bandwidth is being committed on each link for each traffic class, and

estimate whether the traffic demands of new flows can be accommodated. In Layer-3 networks, there is more than one equal-cost between a source and destination; so we employ Dijkstra's all-pair shortest path algorithms. In Layer-2 network, we discover the VLAN tree to find the path between any two hosts.

Figure 46 illustrates the architecture (described in detail in [27, 28, 50]) of the Bandwidth Broker's network management solution for providing application QoS. The four

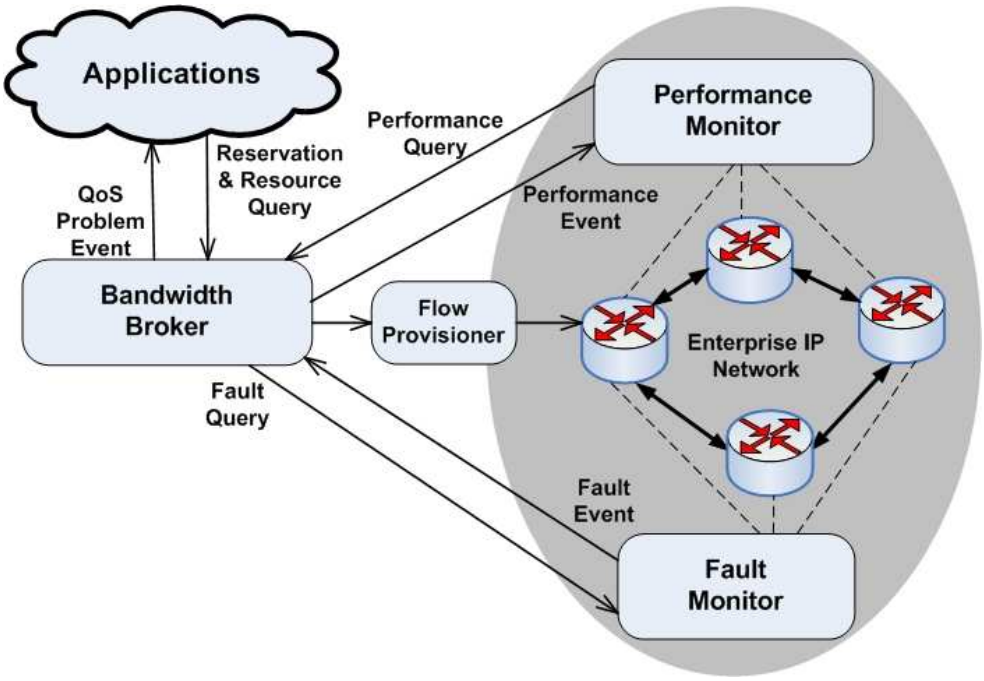

Figure 46: Overview of Telcordia's Bandwidth Broker

components of the QoS management architecture are (1) *Bandwidth Broker*, (2) *Flow Provisioner*, (3) *(Network) Performance Monitor*, and (4) *(Network) Fault Monitor*. These network QoS components provide adaptive admission control that ensures there are adequate network resources to match the needs of admitted flows.

The Bandwidth Broker is responsible for admission control and assigning the appropriate traffic class to each flow. It tracks bandwidth allocations on all network links, rejecting

new flow requests when bandwidth is not available. The Flow Provisioner enforces Bandwidth Broker admission control decisions by configuring ingress network elements to ensure that no admitted flow exceeds its allocated bandwidth. The Flow Provisioner translates technology-independent configuration directives generated by the Bandwidth Broker into vendor-specific router and switch commands to classify, mark, and police packets belonging to a flow. The Fault Monitor is the main feedback mechanism for adapting to network faults and the Performance Monitor provides information on the current performance information of flows and traffic classes. The Bandwidth Broker uses this information to adapt its admission control decisions.

The Bandwidth Broker admission decision for a flow is not based solely on requested capacity or bandwidth on each link traversed by the flow, but is also based on delay bounds requested for the flow. The delay bounds for new flows must be assured without damaging the delay bounds for previously admitted flows and without redoing the expensive job of readmitting every previously admitted flow. Telcordia has developed computational techniques to provide both deterministic and statistical delay-bound assurance [28]. This assurance is based on relatively expensive computations of occupancy or utilization bounds for various classes of traffic, performed only at the time of network configuration/reconfiguration, and relatively inexpensive checking for a violation of these bounds at the time of admission of a new flow.

APPENDIX B

LIST OF PUBLICATIONS

Research on FLARe, DeCoRAM, NetQoPE, and SwapCIAO has led to the following journal, conference and workshop publications.

B.1 Refereed Journal Publications

1. Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons and Douglas C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *Elsevier Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA), Volume 73, Issue 1, September 2008, Pages 39-58*

2. Patrick Lardieri, Jaiganesh Balasubramanian, Douglas C. Schmidt, Gautam Thaker, Aniruddha Gokhale and Thomas Damiano, "A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems," *Elsevier Journal of Systems and Software: Special issue on Dynamic Resource Management in Distributed Real-Time Systems, editors C. Cavanaugh and F. Drews and L. Welch, Volume 80, Issue 7, July 2007, Pages 984-996*

3. Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha Gokhale and Nanbor Wang, "The Design and Performance of Component Middleware for QoS-enabled Deployment and Configuration of DRE Systems," *Elsevier Journal of Systems and*

Software, Special Issue on Component-Based Software Engineering of Trustworthy Embedded Systems, Volume 80, Issue 5, May 2007, Pages 668-677

4. Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale and Douglas C. Schmidt , "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," *Elsevier Journal of System and Software, Volume 73, Issue 2, March 2007, Pages 171-185*

5. Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Aniruddha Gokhale and Douglas C. Schmidt, "Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems," *Invited Paper to International Journal of Embedded Systems, Special Issue on Design and Verification of Real-time Embedded Software, Volume 2, Issue 3, 2006, Pages 142-155*

B.2 Refereed Conference Publications

1. Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha Gokhale, Christopher Gill and Douglas C. Schmidt, "Adaptive Failover for Real-time Middleware with Passive Replication," *Proceedings of the 15th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS 2009), San Francisco, USA, April 2009*

2. Friedhelm Wolf, Jaiganesh Balasubramanian, Aniruddha Gokhale, and Douglas C. Schmidt, "Component Replication Based on Failover Units," *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009), August 2009*

3. Brian Dougherty, Jules White, Jaiganesh Balasubramanian, Chris Thompson and Douglas C. Schmidt, "Deployment Automation with BLITZ'," *Proceedings of the Emerging Results Track of the 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada, May 2009*

4. Jaiganesh Balasubramanian, Aniruddha Gokhale, Douglas C. Schmidt and Nanbor Wang, "Towards Middleware for Fault-Tolerance in Distributed Real-time and Embedded Systems," *Proceedings of the 8th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2008), Oslo, Norway, June 2008*

5. Jaiganesh Balasubramanian, Sumant Tambe, Balakrishnan Dasarathy, Shrirang Gadgil, Frederick Porter, Aniruddha Gokhale and Douglas C. Schmidt, "NetQoPE: A Model-driven Network QoS Provisioning Engine for Distributed Real-time and Embedded Systems," *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS 2008), St. Louis, USA, April 2008*

6. Jaiganesh Balasubramanian, "FLARe: A Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems," *Proceedings of the 4th Middleware Conference Doctoral Symposium (MDS 2007), co-located with Middleware 2007, Newport Beach, California, USA, December 2007*

7. Sumant Tambe, Jaiganesh Balasubramanian, Aniruddha Gokhale and Tom Damiano, "MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-time and Embedded Systems," *Proceedings of the Annual International Service Availability Symposium (ISAS 2007), Durham, NH, USA, May 2007*

8. Nishanth Shankaran, Jaiganesh Balasubramanian, Douglas C. Schmidt and Gautam Biswas, "A Framework for (Re)Deploying Components in Distributed Real-time and Embedded Systems," *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006), Dijon, France, April 2006*

9. Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale and Douglas C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," *Proceedings of*

the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005), San Francisco, CA, March 2005

10. Jaiganesh Balasubramanian, Balachandran Natarajan, Jeff Parsons, Douglas C. Schmidt and Aniruddha Gokhale, "Middleware Support for Dynamic Component Updating," *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2005), Agia Napa, Cyprus, October 2005*

11. Gan Deng, Jaiganesh Balasubramanian, Douglas C. Schmidt and Aniruddha Gokhale, 'DAnCE: A QoS-enabled Component Deployment and Configuration Engine," *Proceedings of the Third Working Conference on Component Deployment (CD 2005), Grenoble, France, November 2005*

12. Jaiganesh Balasubramanian, Lawrence Dowdy, Douglas C. Schmidt and Ossama Othman, "Evaluating the Performance of Middleware Load Balancing Strategies," *Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2004), Monterey, CA, September 2004*

B.3 Refereed Workshop Publications

1. Jaiganesh Balasubramanian, Aniruddha Gokhale, Douglas C. Schmidt and Sherif Abdelwahed, "Investigating Survivability Strategies for Ultra Large Scale Systems," *Proceedings of the NSF TRUST Project Winter Workshop, Washington, D.C, January, 2006*

2. Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind Krishna and Gan Deng, "CoSMIC: Addressing the Deployment and Configuration Crosscutting Concerns of Distributed Real-time and Embedded Systems," *Proceedings of the Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages and Applications (OOPSLA 2004), Vancouver, BC, Canada, October 2004*

3. Arvind S. Krishna, Jaiganesh Balasubramanian, Aniruddha Gokhale, Douglas C. Schmidt, Diego Sevilla and Gautam Thaker, "Empirically Evaluating CORBA Component Model Implementations," *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages and Applications (OOPSLA 2003), Workshop on Middleware Benchmarking, Anaheim, CA, October 26, 2003*

4. Ossama Othman, Jaiganesh Balasubramanian and Douglas C. Schmidt, "The Design of an Adaptive Middleware Load Balancing and Monitoring Service," *Proceedings of the 3rd International Workshop on Self-Adaptive Software (IWSAS 2003), Arlington, VA, USA, June 9-11, 2003*

B.4 Submitted for Publication

1. Jaiganesh Balasubramanian, Aniruddha Gokhale, Friedhelm Wolf, Abhishek Dubey, Chenyang Lu, Christopher Gill and Douglas C. Schmidt, "Resource-Aware Deployment and Configuration of Fault-Tolerant Real-time Systems," *Submitted to the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), Stockholm, Sweden, April 2010*

2. Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale, Balakrishnan Dasarathy, Shrirang Gadgil and Douglas C. Schmidt, "A Model-driven QoS Provisioning Engine for Cyber Physical Systems'," *Submitted to the IEEE Transactions on Software Engineering, October 2009*

REFERENCES

[1] Tarek F. Abdelzaher, Scott Dawson, Wu chang Feng, Farnam Jahanian, Scott Johnson, Ashish Mehra, Todd Mitton, Anees Shaikh, Kang G. Shin, Zhiqun Wang, Hengming Zou, M. Bjorkland, and P. Marron. ARMADA Middleware and Communication Services. *Real-Time Systems*, 16(2-3):127–153, 1999.

[2] R. Al-Omari, A. K. Somani, and G. Manimaran. An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems. *J. Parallel Distrib. Comput.*, 65(5):595–608, 2005. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2004.09.021.

[3] Hakan Aydin. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Trans. Comput.*, 56(10):1372–1386, 2007. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.2007.70739.

[4] Jaiganesh Balasubramanian, Douglas C. Schmidt, Lawrence Dowdy, and Ossama Othman. Evaluating the Performance of Middleware Load Balancing Strategies. In *Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference*, Montery, CA, September 2004. IEEE.

[5] Jaiganesh Balasubramanian, Sumant Tambe, Balakrishnan Dasarathy, Shrirang Gadgil, Frederick Porter, Aniruddha Gokhale, and Douglas C. Schmidt. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS' 08*, pages 113–122, 2008.

[6] Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale, Chenyang Lu, Christopher Gill, and Douglas C. Schmidt. FLARe: a Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems. Technical Report ISIS-08-812, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, October 2008.

[7] Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha Gokhale, Christopher Gill, and Douglas C. Schmidt. Adaptive Failover for Real-time Middleware with Passive Replication. In *Proceedings of the 15th Real-time and Embedded Applications Symposium (RTAS)*, pages 118–127, San Francisco, CA, April 2009.

[8] Krishnakumar Balasubramanian, Nanbor Wang, and Douglas C. Schmidt. Towards composable distributed real-time and embedded software. In *WORDS '03: Proceedings of the* 8^{th} *Workshop on Object-oriented Real-time Dependable Systems*, pages 226–233, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[9] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05:*

Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2302-1. doi: http://dx.doi.org/10.1109/RTAS.2005.4.

[10] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. *Journal of Computer Systems Science*, 73(2):171–185, 2007. ISSN 0022-0000. doi: dx.doi.org/10.1016/j.jcss.2006.04.008.

[11] Roberto Baldoni and Carlo Marchetti. Three-tier replication for ft-corba infrastructures. *Softw. Pract. Exper.*, 33(8):767–797, 2003. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.525.

[12] P. Barrett, A. Hilborne, P. Bond, D. Seaton, P. Verissimo, L. Rodrigues, and N. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *Proceedings of the 20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pages 481–488, June 1990.

[13] T. Bennani, L. Blain, L. Courtes, J. C. Fabre M. O. Killijian, E. Marsden, and F. Taiani. Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization. In *Proc. of DSN. (2004)*.

[14] Taha Bennani, Laurent Blain, Ludovic Courtes, Jean-Charles Fabre, Marc-Olivier Killijian, Eric Marsden, and Francois Taiani. Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization. In *DSN' 04*, pages 549–554, Florence, Italy, 2004.

[15] Alan A. Bertossi, Luigi V. Mancini, and Federico Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 10(9):934–945, 1999. ISSN 1045-9219. doi: dx.doi.org/10.1109/71.798317.

[16] Bettina Kemme and Gustavo Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 156–163, May 1998.

[17] Ken Birman, Robbert van Renesse, and Werner Vogels. Adding high availability and autonomic behavior to web services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 17–26, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services, 1998.

[19] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, Shanghai, China, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134292.

[20] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-backup Approach. In *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3.

[21] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-backup Approach. In *Distributed systems (2nd Ed.)*, pages 199–216. 1993.

[22] Zhongtang Cai, Vibhore Kumar, Brian F. Cooper, Greg Eisenhauer, Karsten Schwan, and Robert E. Strom. Utility-Driven Proactive Management of Availability in Enterprise-Scale Information Flows. In *Proceedings of ACM/Usenix/IFIP Middleware*, pages 382–403, 2006.

[23] Sayantan Chakravorty and L.V. Kale. A fault tolerance protocol with fast fault recovery. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007. doi: 10.1109/IPDPS.2007.370310.

[24] Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Shau-Yin Tseng. Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 249–258, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2800-7. doi: http://dx.doi.org/10.1109/RTAS.2007.30.

[25] Z. Chen, M. Yang, G. Francia, and J. Dongarra. Self adaptive application level fault tolerance for parallel and distributed computing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370604.

[26] EG Coffman Jr, MR Garey, and DS Johnson. Approximation algorithms for bin packing: a survey. 1996.

[27] B. Dasarathy, S. Gadgil, R. Vaidyanathan, K. Parmeswaran, B. Coan, M. Conarty, and V. Bhanot. Network qos assurance in a multi-layer adaptive resource management scheme for mission-critical applications using the corba middleware framework. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 246–255, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2302-1. doi: http://dx.doi.org/10.1109/RTAS.2005.34.

[28] Balakrishnan Dasarathy, Shrirang Gadgil, Ravi Vaidyanathan, Arnie Neidhardt,

Brian Coan, Kirthika Parmeswaran, Allen McIntosh, and Frederick Porter. Adaptive network qos in layer-3/layer-2 networks as a middleware service for mission-critical applications. *J. Syst. Softw.*, 80(7):972–983, 2007. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2006.09.030.

[29] G. de A Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *Computers, IEEE Transactions on*, 52(10):1332–1346, Oct. 2003. ISSN 0018-9340. doi: 10.1109/TC.2003.1234530.

[30] Miguel A. de Miguel. Integration of qos facilities into component container architectures. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 394, Washington, DC, USA, 2002. IEEE Computer Society.

[31] Dionisio de Niz and Raj Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems*, 2(3):196–208, 2006.

[32] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2516-4. doi: http://dx.doi.org/10.1109/RTAS.2006.30.

[33] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *Proc. of RTAS'06*, pages 231–242, Washington, DC, USA, August 2006. ISBN 0-7695-2516-4. doi: dx.doi.org/10.1109/RTAS.2006.30.

[34] S.K. Dhall and CL Liu. On a Real-time Scheduling Problem. *Operations Research*, pages 127–140, 1978.

[35] S.K. Dhall and CL Liu. On a Real-time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[36] T. Dumitras and P. Narasimhan. Fault-Tolerant Middleware and the Magical 1%. *In: Proc. of Middleware (2005)*, 2005.

[37] T. Dumitras and P. Narasimhan. Fault-Tolerant Middleware and the Magical 1%. *Middleware 2005: ACM/IFIP/USENIX 6th International Middleware Conference, Grenoble, France, November 28-December 2, 2005: Proceedings*, 2005.

[38] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building Adaptive Distributed Applications with Middleware and Aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software*

development, pages 66–73, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-842-3.

[39] E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic cpu management for real-time, middleware-based systems. In *RTAS '04: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 286, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2148-7.

[40] M. El-Gendy, A. Bose, S.-T. Park, and K.G. Shin. Paving the first mile for qos-dependent applications and appliances. In *IWQoS '04: Proceedings of the 12th International Workshop on Quality of Service*, pages 245–254, Washington, DC, USA, June 2004. IEEE Computer Society. doi: 10.1109/IWQOS.2004.1309390.

[41] M.A. El-Gendy, A. Bose, and K.G. Shin. Evolution of the internet qos and support for soft real-time applications. *Proceedings of the IEEE,* 91(7):1086–1104, July 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2003.814615.

[42] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Surveys*, 34(3):375–408, 2002.

[43] E.N. Elnozahy and J.S. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97–108, April-June 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.15.

[44] Paul Emberson and Iain Bate. Extending a Task Allocation Algorithm for Graceful Degradation of Real-Time Distributed Embedded Systems. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 270–279, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3477-0. doi: http://dx.doi.org/10.1109/RTSS.2008.24.

[45] Pascal Felber and Priya Narasimhan. Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems. *Computers, IEEE Transactions on*, 54(5):497–511, May 2004.

[46] Ian Foster, Markus Fidler, Alain Roy, Volker Sander, and Linda Winkler. End-to-end Quality of Service for High-end Applications. *Computer Communications*, 27(14):1375–1388, September 2004.

[47] Roy Friedman and Erez Hadad. Fts: A high-performance corba fault-tolerance service. In *Proc. of WORDS.(2002)*.

[48] Roy Friedman and Erez Hadad. Fts: A high-performance corba fault-tolerance service. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop*

on Object-Oriented Real-Time Dependable Systems (WORDS 2002), page 61, Washington, DC, USA, 2002. IEEE Computer Society.

[49] Lorenz Froihofer, Karl M. Goeschka, and Johannes Osrael. Middleware support for adaptive dependability. In *Middleware*, pages 308–327, 2007.

[50] Shrirang Gadgil, Balakrishnan Dasarathy, Frederick Porter, Kirthika Parmeswaran, and Ravi Vaidyanathan. Fast recovery and qos assurance in the presence of network faults for mission-critical applications in hostile environments. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 283–292, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2975-5. doi: http://dx.doi.org/10.1109/RTCSA.2007.39.

[51] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[52] Qi Gao, Weikuan Yu, Wei Huang, and D.K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 471–478, Aug. 2006. doi: 10.1109/ICPP.2006.26.

[53] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(3):272–284, 1997. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.584093.

[54] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An Algorithm for Automatically Obtaining Distributed and Fault-tolerant Static Schedules. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 159–168, June 2003.

[55] Aniruddha S. Gokhale, Balachandran Natarajan, Douglas C. Schmidt, and Joseph K. Cross. Towards real-time fault-tolerant corba middleware. *Cluster Computing*, 7(4):331–346, 2004. ISSN 1386-7857. doi: http://dx.doi.org/10.1023/B:CLUS.0000039493.73008.13.

[56] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *RTSS '97*, page 79, San Francisco, CA, USA, 1997. ISBN 0-8186-8268-X.

[57] Sathish Gopalakrishnan and Marco Caccamo. Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages

199–207, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2516-4. doi: http://dx.doi.org/10.1109/RTAS.2006.43.

[58] Zonghua Gu and Kang G. Shin. Synthesis of Real-Time Implementations from Component-Based Software Models. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 167–176, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7. doi: http://dx.doi.org/10.1109/RTSS.2005.38.

[59] Zonghua Gu, Sharath Kodase, Shige Wang, and Kang G. Shin. A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 78–85, Toronto, Canada, 2003. IEEE Computer Society. ISBN 0-7695-1956-3.

[60] Zonghua Gu, Sharath Kodase, Shige Wang, and Kang G. Shin. A Model-Based Approach to System-Level Dependency and Real-time Analysis of Embedded Software. In *RTAS'03*, pages 78–85, Washington, DC, May 2003. IEEE.

[61] Rachid Guerraoui and André Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[62] Jun He, Matti A. Hiltunen, Mohan Rajagopalan, and Richard D. Schlichting. Providing qos customization in distributed object systems. In *Middleware*, pages 351–372, 2001.

[63] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[64] E. Douglas Jensen. Distributed Real-time Specification for Java. java.sun.com/aboutJava/communityprocess/jsr/jsr_050_drt.html, 2000.

[65] Scott Johnson, Farnam Jahanian, Akihiko Miyoshi, Dionisio de Niz, and Ragunathan Rajkumar. Constructing real-time group communication middleware using the resource kernel. *RTSS '2000: Proceedings of the 21st IEEE Real-Time Systems Symposium*, 00:3, 2000. ISSN 1052-8725. doi: doi.ieeecomputersociety.org/10.1109/REAL.2000.895990.

[66] Mick Jordan, Grzegorz Czajkowski, Kirill Kouklinski, and Glenn Skinner. Extending a j2ee™server with dynamic and flexible resource management. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 439–458, New York, NY, USA, 2004. Springer-Verlag New York, Inc. ISBN 3-540-23428-4.

[67] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Dynamic Scheduling of Distributed Method Invocations. In *21st IEEE Real-time Systems Symposium*, Orlando,

FL, November 2000. IEEE.

[68] Vana Kalogeraki, P. M. Melliar-Smith, L. E. Moser, and Y. Drougas. Resource Management Using Multiple Feedback Loops in Soft Real-time Distributed Systems. *Journal of Systems and Software*, 2007.

[69] Nagarajan Kandasamy, John P Hayes, and Brian T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Trans. on Comp.*, 52(2), 2003. ISSN 0018-9340. doi: doi.ieeecomputersociety.org/10.1109/TC.2003.1.

[70] Gabor Karsai, Sandeep Neema, Ben Abbott, and David Sharp. A Modeling Language and Its Supporting Tools for Avionics Systems. In *Proceedings of 21st Digital Avionics Systems Conference*, Los Alamitos, CA, August 2002. IEEE Computer Society.

[71] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[72] Panagiotis Katsaros and Constantine Lazos. Optimal object state transfer - recovery policies for fault tolerant distributed systems. In *Proc. of DSN. (2004)*.

[73] Khanna, S., *et al.* Realtime Scheduling in SunOS 5.0. In *Proceedings of the USENIX Winter Conference*, pages 375–390. USENIX Association, 1992.

[74] K. H. (Kane) Kim and Jeff J.Q. Liu. Techniques for Implementing Support Middleware for the PSTR Scheme for Real-Time Object Replication. *ISORC*, 00:163–172, 2004. doi: doi.ieeecomputersociety.org/10.1109/ISORC.2004.1300342.

[75] K. H. (Kane) Kim and Chittur Subbaraman. The pstr/sns scheme for real-time fault tolerance via active object replication and network surveillance. *IEEE Trans. on Know. and Data Engg.*, 12(2), 2000. ISSN 1041-4347. doi: dx.doi.org/10.1109/69.842258.

[76] Kane Kim. APIs Enabling High-Level Real-time Distributed Object Programming. *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, June 2000.

[77] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, 09(1):25–40, 1989. ISSN 0272-1732. doi: doi.ieeecomputersociety.org/10.1109/40.16792.

[78] Arvind S. Krishna, Douglas C. Schmidt, and Raymond Klefstad. Enhancing real-time corba via real-time java features. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 66–73, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2086-3.

[79] S. Krishnamurthy, W.H. Sanders, and M. Cukier. A Dynamic Replica Selection Algorithm for Tolerating Timing Faults. *DSN' 01*, pages 107–116, 2001.

[80] Sudha Krishnamurthy, William H. Sanders, and Michel Cukier. An Adaptive Quality of Service Aware Middleware for Replicated Services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, 2003. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2003.1247672.

[81] L. Zhang and S. Berson and S. Herzog and S. Jamin. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification, September 1997.

[82] Zhiling Lan and Yawei Li. Adaptive fault management of parallel applications for high-performance computing. *IEEE Transactions on Computers*, 57(12):1647–1660, 2008. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2008.90.

[83] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.963443.

[84] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *RTSS '89: Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, Washington, DC, USA, 1989. IEEE Computer Society. doi: 10.1109/REAL.1989.63567.

[85] F. Liberato, S. Lauzac, R. Melhem, and D. Mosse. Fault tolerant real-time global scheduling on multiprocessors. pages 252–259, 1999. doi: 10.1109/EMRTS.1999.777472.

[86] Frank Liberato, Rami Melhem, and Daniel Mossé. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Trans. Comput.*, 49(9):906–914, 2000. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.869322.

[87] Deni Llambiri, Alexander Totok, and Vijay Karamcheti. Efficiently Distributing Component-Based Applications Across Wide-Area Environments. In *Proc. of ICDCS'03*, 2003. ISBN 0-7695-1920-2.

[88] J. M. López, M. García, J. L. Díaz, and D. F. García. Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling. *Real-Time Syst.*, 24(1):5–28, 2003. ISSN 0922-6443. doi: http://dx.doi.org/10.1023/A:1021749005009.

[89] Chenyang Lu, Xiaorui Wang, and Christopher Gill. Feedback Control Real-time Scheduling in ORB Middleware. In *Proc. of RTAS. (2003)*.

[90] Chenyang Lu, Xiaorui Wang, and Xenofon Koutsoukos. Feedback Utilization Control in Distributed Real-time Systems with End-to-End Tasks. *IEEE Trans. on Par. and Dist. Sys.*, 16(6):550–561, 2005. ISSN 1045-9219. doi: dx.doi.org/10.1109/TPDS.2005.73.

[91] G. Manimaran and C. Siva Ram Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(3):312–319, 1998. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.674322.

[92] Tudor Marian, Ken Birman, and Robbert van Renesse. A scalable services architecture. In *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2677-2. doi: dx.doi.org/10.1109/SRDS.2006.7.

[93] Olivier Marin, Marin Bertier, and Pierre Sens. Darx: A framework for the fault-tolerant support of agent software. In *Proc. of ISSRE. (2003)*.

[94] Olivier Marin, Marin Bertier, and Pierre Sens. Darx: A framework for the fault-tolerant support of agent software. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 406, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3.

[95] Ashish Mehra, Dinesh C. Verma, and Renu Tewari. Policy-based diffserv on internet servers: The AIX approach (on the wire). *IEEE Internet Computing*, 4(5):75–80, 2000. URL citeseer.ist.psu.edu/mehra00policybased.html.

[96] S. Melro and P. Verssimo. Real-time and Dependability Comparison of Delta-4/XPA and MARS Systems, 1992.

[97] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[98] *Distributed Component Object Model Protocol (DCOM)*. Microsoft Corporation, 1.0 edition, January 1998.

[99] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-768-1. doi: http://doi.acm.org/10.1145/1274971.1274978.

[100] Klara Nahrstedt. To overprovision or to share via qos-aware resource management? In *HPDC '99: Proceedings of the 8th IEEE International Symposium on*

High Performance Distributed Computing, page 35, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0287-3.

[101] Klara Nahrstedt, Dongyan Xu, Duangdao Wichadakul, and Baochun Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine*, 39(11):140–148, November 2001.

[102] P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava. MEAD: Support for Real-time Fault-Tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.

[103] Priya Narasimhan. Trade-Offs Between Real-Time and Fault Tolerance for Middleware Applications. Workshop on Foundations of Middleware Technologies, November 2002.

[104] Priya Narasimhan, Tudor Dumitras, Aaron M. Paulos, Soila M. Pertet, Charlie F. Reverte, Joseph G. Slember, and Deepti Srivastava. MEAD: support for Real-Time Fault-Tolerant CORBA. *Concurrency - Practice and Experience*, 17(12):1527–1545, 2005.

[105] Priya Narasimhan, Raj Rajkumar, Gautam Thaker, and Patrick Lardieri. A versatile, proactive dependability approach to handling unanticipated events in distributed systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 2*, page 136.1, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9. doi: http://dx.doi.org/10.1109/IPDPS.2005.74.

[106] Andrey Nechypurenko, Tao Lu, Gan Deng, Emre Turkay, Douglas C. Schmidt, and Aniruddha S. Gokhale. Concern-based composition and reuse of distributed systems. In *ICSR*, volume 3107, pages 167–184, Madrid, Spain, 2004. Springer. ISBN 3-540-22335-5.

[107] Andrey Nechypurenko, Douglas C. Schmidt, Tao Lu, Gan Deng, and Aniruddha Gokhale. Applying MDA and Component Middleware to Large-scale Distributed Systems: a Case Study. In *Proceedings of the 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, Enschede, Netherlands, March 2004.

[108] *Light Weight CORBA Component Model Revised Submission*. Object Management Group, OMG Document realtime/03-05-05 edition, May 2003.

[109] *Deployment and Configuration Adopted Submission*. Object Management Group, OMG Document mars/03-05-08 edition, July 2003.

[110] Object Management Group. *Fault Tolerant CORBA, Chapter 23, CORBA v3.0.3*. Object Management Group, OMG Document formal/04-03-10 edition, March 2004.

[111] Object Management Group. *Lightweight CCM FTF Convenience Document*. Object Management Group, ptc/04-06-10 edition, June 2004.

[112] Object Management Group. *Real-time CORBA Specification*. Object Management Group, OMG Document formal/05-01-04 edition, August 2002.

[113] Object Management Group. *Real-time CORBA Specification v1.2 (static)*. Object Management Group, OMG Document formal/05-01-04 edition, November 2005.

[114] Y. Oh and S. H. Son. Scheduling Real-Time Tasks for Dependability. *The Journal of the Operational Research Society*, 48(6):629–639, 1997. ISSN 01605682. URL http://www.jstor.org/stable/3010227.

[115] OMG. *The Common Object Request Broker: Arch. and Specification*. OMG, 2002.

[116] Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt. Strategies for CORBA Middleware-Based Load Balancing. *IEEE Distributed Systems Online*, 2(3), March 2001.

[117] Ossama Othman, Jaiganesh Balasubramanian, and Douglas C. Schmidt. The Design of an Adaptive Middleware Load Balancing and Monitoring Service. In *LNCS/LNAI: Proceedings of the Third International Workshop on Self-Adaptive Software*, Heidelberg, June 2003. Springer-Verlag.

[118] Mihir Pandya and Miroslaw Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Trans. Comput.*, 47(10):1102–1112, 1998. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.729793.

[119] Soila Pertet and Priya Narasimhan. Proactive recovery in distributed corba applications. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 357, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2052-9.

[120] David Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, 14 (1):36–47, 1994. ISSN 0272-1732. doi: dx.doi.org/10.1109/40.259898.

[121] Francisco Prez-Sorrosal, Marta Patino-Martinez, Ricardo Jimenez-Peris, and Jaksa Vuckovic. Highly available long running transactions and activities for j2ee applications. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 2, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2540-7. doi: http://dx.doi.org/10.1109/ICDCS.2006.47.

[122] Sasikumar Punnekkat, Alan Burns, and Robert Davis. Analysis of checkpointing for real-time systems. *Real-Time Syst.*, 20(1):83–102, 2001. ISSN 0922-6443. doi: http://dx.doi.org/10.1023/A:1026589200419.

[123] I. Pyarali, D.C. Schmidt, and R.K. Cytron. Techniques for enhancing real-time corba

quality of service. *Proceedings of the IEEE*, 91(7):1070–1085, July 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2003.814616.

[124] Irfan Pyarali, Carlos O'Ryan, and Douglas C. Schmidt. A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2000. IEEE/IFIP.

[125] Xiao Qin, Hong Jiang, and David R. Swanson. An Efficient Fault-Tolerant Scheduling Algorithm for Real-Time Tasks with Precedence Constraints in Heterogeneous Systems. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, page 360. IEEE Computer Society, 2002. ISBN 0-7695-1677-7.

[126] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali. Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. In *Proc. of Middleware'03*, Rio de Janeiro, Brazil, June 2003. IFIP/ACM/USENIX.

[127] Parameswaran Ramanathan. Overload management in real-time control applications using m,k (m,k)-firm guarantee. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):549–559, 1999. ISSN 1045-9219. doi: dx.doi.org/10.1109/71.774906.

[128] Y. Ren, DE Bakken, T. Courtney, M. Cukier, DA Karr, P. Rubel, C. Sabnis, WH Sanders, RE Schantz, and M. Seri. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. *Computers, IEEE Transactions on*, 52(1): 31–50, 2003.

[129] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In *Proc. of Middleware 2006*.

[130] Louis Rilling, Swaminathan Sivasubramanian, and Guillaume Pierre. High availability and scalability support for web applications. In *SAINT '07: Proceedings of the 2007 International Symposium on Applications and the Internet*, page 5, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2756-6. doi: http://dx.doi.org/10.1109/SAINT.2007.14.

[131] Tom Ritter, Marc Born, Thomas Unterschütz, and Torben Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences (HICSS'03)*, page 318, Honolulu, HI, January 2003.

[132] Wendy Roll. Towards Model-Based and CCM-Based Applications for Real-time Systems. In *Proceedings of the IEEE International Symposium on Object-Oriented*

Real-time Distributed Computing (ISORC), May 2003.

[133] S. M. Sadjadi and P. K. McKinley. Act: An adaptive corba template to support unanticipated adaptation. In *Proc. of ICDCS. (2004)*, 2004.

[134] Jorge Salas, Francisco Perez-Sorrosal, no-Martínez Marta Pati and Ricardo Jiménez-Peris. Ws-replication: a framework for highly available web services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 357–366, New York, NY, USA, 2006. ACM. ISBN 1-59593-323-9. doi: http://doi.acm.org/10.1145/1135777.1135831.

[135] Richard Schantz, John Zinky, David Karr, David Bakken, James Megquier, and Joseph Loyall. An Object-level Gateway Supporting Integrated-Property Quality of Service. *ISORC*, 00:223, 1999. doi: doi.ieeecomputersociety.org/10.1109/ISORC.1999.776381.

[136] Richard E. Schantz, Joseph P. Loyall, Craig Rodrigues, and Douglas C. Schmidt. Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1189–1208, 2006. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.v36:11/12.

[137] Richard D. Schlichting and Fred B. Schneider. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983. ISSN 0734-2071. doi: doi.acm.org/10.1145/357369.357371.

[138] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[139] Douglas C. Schmidt and Steve Vinoski. The CORBA Component Model Part 3: The CCM Container Architecture and Component Implementation Framework. *The C/C++ Users Journal*, September 2004.

[140] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[141] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. In *CrossTalk - The Journal of Defense Software Engineering*, pages 10–16, Hill AFB, Utah, USA, nov 2001. Software Technology Support Center.

[142] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[143] Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, 2004. ISSN 0922-6443. doi: http://dx.doi.org/10.1023/B:TIME.0000045315.61234.1e.

[144] Praveen Kaushik Sharma, Joseph P. Loyall, George T. Heineman, Richard E. Schantz, Richard Shapiro, and Gary Duzan. Component-based dynamic qos adaptations in distributed real-time and embedded systems. In *CoopIS/DOA/ODBASE (2)*, pages 1208–1224, Agia Napa, Cyprus, 2004. Springer.

[145] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003, May 2003.

[146] Deborah Snoonian. Smart Buildings. *IEEE Spectrum*, 40(8):18–23, 2003.

[147] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. VEST: An Aspect-Based Composition Tool for Real-Time Systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 58–69, Toronto, Canada, 2003. IEEE Computer Society. ISBN 0-7695-1956-3.

[148] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. Vest: An aspect-based composition tool for real-time systems. In *Proc. of RTAS'03*, page 58, Washington, DC, USA, 2003. ISBN 0-7695-1956-3.

[149] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 71–84, Berkeley, CA, USA, 2005. USENIX Association.

[150] Randall Stewart and Qiaobing Xie. *Stream Control Transmission Protocol (SCTP) A Reference Guide*. Addison-Wesley, Boston, 2001.

[151] D.A. Stuart, M. Brockmeyer, A.K. Mok, and F. Jahanian. Simulation-Verification: Biting at the State Explosion Problem. *IEEE Transactions on Software Engineering*, 27(7):599–617, July 2001.

[152] SUN. Java Remote Method Invocation (RMI) Specification. java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html, 2002.

[153] Wei Sun, Yuanyuan Zhang, Chen Yu, X. Defago, and Y. Inoguchi. Hybrid Overloading and Stochastic Analysis for Redundant Real-time Multiprocessor Systems.

In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 265–274, Oct. 2007. doi: 10.1109/SRDS.2007.11.

[154] Sun Microsystems. *Java Specification Request, JSR 117, J2EE APIs for Continuous Availability*. Sun Microsystems, JSR 117 edition, April 2001.

[155] Dipa Suri, Adam Howell, Nishanth Shankaran, John Kinnebrew, Will Otte, Douglas C. Schmidt, and Gautam Biswas. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006.

[156] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, Santa Fe, NM, 1998.

[157] Bhuvan Urgaonkar and Prashant Shenoy. Sharc: Managing cpu and network bandwidth in shared clusters. *IEEE Trans. Parallel Distrib. Syst.*, 15(1):2–17, 2004. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2004.1264781.

[158] Bhuvan Urgaonkar, Arnold Rosenberg, and Prashant Shenoy. Application Placement on a Cluster of Servers. *International Journal of Foundations of Computer Science*, 18(5):1023–1042, 2007.

[159] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[160] Fuxing Wang, Krithi Ramamritham, and John A. Stankovic. Determining redundancy levels for fault tolerant real-time systems. *IEEE Transactions on Computers*, 44(2):292–301, 1995. ISSN 0018-9340. doi: dx.doi.org/10.1109/12.364540.

[161] Nanbor Wang and Christopher Gill. Improving real-time system configuration via a qos-aware corba component model. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90273.2, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2056-1.

[162] Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher. Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *24th Computer Software and Applications Conference (COMPSAC)*, pages 492–499, Taipei, Taiwan, October 2000. IEEE.

[163] Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran. Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. *IEEE Distributed Systems Online*, 2(5), July 2001.

[164] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran. Evaluating Meta-Programming Mechanisms for ORB Middleware. *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, 39(10):102–113, October 2001.

[165] Nanbor Wang, Christopher Gill, Douglas C. Schmidt, and Venkita Subramonian. Configuring Real-time Aspects in Component Middleware. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, October 2004. Springer-Verlag.

[166] P. Wang, Y. Yemini, D. Florissi, and J. Zinky. A distributed resource controller for qos applications. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*, pages 143–156, Los Alamitos, CA, USA, 2000. IEEE Computer Society. doi: 10.1109/NOMS.2000.830381.

[167] Duangdao Wichadakul, Klara Nahrstedt, Xiaohui Gu, and Dongyan Xu. 2k: An integrated approach of qos compilation and reconfigurable, component-based run-time middleware for the unified qos management framework. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 373–394, London, UK, 2001. Springer-Verlag. ISBN 3-540-42800-3.

[168] Huaigu Wu, Bettina Kemme, and Vance Maverick. Eager replication for stateful j2ee servers. In *CoopIS/DOA/ODBASE (2)*, pages 1376–1394, 2004.

[169] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 29–42, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: http://doi.acm.org/10.1145/502034.502038.

[170] Ying Zhang and K. Chakrabarty. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(1):111–125, Jan. 2006. ISSN 0278-0070. doi: 10.1109/TCAD.2005.852657.

[171] Qin Zheng, B. Veeravalli, and Chen-Khong Tham. On the Design of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computational Grids with Low Replication Costs. *Computers, IEEE Transactions on*, 58(3):380–393, March 2009. ISSN 0018-9340. doi: 10.1109/TC.2008.172.

[172] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3 (1):1–20, 1997.

[173] H. Zou and F. Jahanian. Optimization of a real-time primary-backup replication service. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable*

Distributed Systems, page 177, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-9218-9.

[174] H. Zou and F. Jahanian. A Real-time Primary-backup Replication Service. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):533–548, 1999.