SUPPORTING FINE-GRAINED CONFIGURABILITY WITH MULTIPLE QUALITY OF SERVICE PROPERTIES IN MIDDLEWARE FOR EMBEDDED SYSTEMS

By

ARCHIBALD DAVID MCKINNON

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2003

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of ARCHIBALD DAVID MCKINNON find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

This accomplishment would not have been possible without the support and confidence provided by a large number of individuals. First and foremost I am thankful for the support of my family. My parents, Mikal and Janet, instilled in me a value of education and learning as well as a strong work ethic. Christine has constantly supported me (and proofread all of my papers). Samuel helped us move to Pullman, and Kimberly and Camilla joined us in the middle of this adventure and I am grateful for all three.

The faculty and staff at WSU have been of great help and assistance to me while here. In particular, I would like to thank my advisor, Dave Bakken, and my other committee members, Curtis Dyreson and John Shovic. Ruby Young also deserves a kind word of thanks. I am also grateful for the opportunity of being able to interact with many great individuals while serving as a member of WSU's UACCT and other committees.

I have had the experience of interacting with great students while at WSU. Naturally, I must mention all my fellow team members on the MicroQoSCORBA team: Olav Haugan, Tarana Damania, Kevin Dorow, Wesley Lawrence, Thor Skaug, Eivind Næss, Kim Swenson, and Ryan Johnson. Kjell "Harald" Gjermundrød, Ioanna Dionysiou, Richard Griswold, Brent House and many other graduate students have contributed to my enjoyable stay in Pullman. I also enjoyed my involvement in WSU's Graduate and Professional Student Association.

Over the years I have been influenced, guided, and mentored by a great number of individuals who have earned Ph.D.s. I would like to make a particular mention of the following: Mikal McKinnon, Doug Lemon, Chauncey Riddle, Melvin Luthy, Larry Christensen, Tony Martinez, Walt Hensley, Harry Miley, Deborah Frincke, Karen DePauw, Lane Rawlins, and Rollin Hotchkiss.

I am grateful to Battelle for its support via an education leave of absence from the Pacific Northwest National Laboratory as well as a graduate stipend during my first two years at Washington State University. The Carl M. Hansen foundation supported my first two years at WSU with a graduate fellowship. After that, the State of Washington provided financial support with both teaching and research assistantships. My research has also been supported in part by two Cisco University Research Program donations and by Grant NSF-CISE EHS-0209211 from the National Science Foundation.

And finally, this acknowledgment would be truly incomplete if I did not thank God for all of His divine assistance.

ATTRIBUTION

Portions of this dissertation were published or have been submitted for publication in the following conference and journal articles.

Conference Publication

1. A. David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic. A configurable middleware framework with multiple quality of service properties for small embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Network and Computing Applications (NCA2003)*, pages 197–204. IEEE Computer Society, April 2003.

Submitted for Journal Publication

1. A. David McKinnon, David E. Bakken, and John C. Shovic. A configurable security subsystem in a middleware framework for embedded systems. *Computer Networks*. Submitted for publication. 68 pages.

2. A. David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic. A configurable middleware framework for small embedded systems that supports multiple quality of service properties. *Software—Practice and Experience*. Submitted for publication. 68 pages.

A. David McKinnon was the first and primary author for each of the above publications. In particular, he was the chief architect and designer for the MicroQoSCORBA middleware framework that supports multiple Quality of Service properties and he designed and implemented MicroQoSCORBA's security subsystem. McKinnon also designed and implemented the automated build management and testing framework that was required because MicroQoSCORBA's high degree of configurability required support for a combinatorial large number

of finely-configured application builds. Haugan implemented the initial MicroQoSCORBA prototype, and its IDL compiler and configuration GUI [Hau01] under McKinnon's guidance. Damania implemented the UDP transports [Dam02], also under McKinnon's direction. Dorow assisted McKinnon in the design of the fault tolerance subsystem and he implemented MicroQoSCORBA's fault tolerance and group communications mechanisms [Dor02, DB03]. Lawrence conducted the temporal characterization of MicroQoSCORBA under McKinnon's direction [Law03]. Drs. Bakken and Shovic provided feedback and insight, as needed, on various topics.

SUPPORTING FINE-GRAINED CONFIGURABILITY WITH
MULTIPLE QUALITY OF SERVICE PROPERTIES IN
MIDDLEWARE FOR EMBEDDED SYSTEMS

Abstract

by Archibald David McKinnon, Ph.D.
Washington State University
December 2003

Chair: David E. Bakken

The majority of microprocessors manufactured in recent years have been deployed in embedded systems, often with real-time (timeliness) requirements, and increasingly they are being networked. Middleware frameworks offer many advantages to distributed systems designers and application programmers. However, there are very few middleware frameworks that are suitable for the low end of the embedded systems market, and they are only coarsely configurable. Furthermore, even fewer middleware frameworks of any size support multiple Quality of Service properties, such as fault tolerance, security, and timeliness. In this dissertation we describe the design and implementation of MicroQoSCORBA. It represents a fundamental, bottom-up rethinking of what middleware can and should support for resource-constrained devices. This framework can be tailored, with a fine degree of granularity, to support both device and application program

constraints. This dissertation describes the refined middleware architectural taxonomy that was developed to specify and implement these constraints and the multiple Quality of Service domains that MicroQoSCORBA supports. In particular, MicroQoSCORBA's security subsystem and its implemented security mechanisms are presented along with an analysis of security requirements from an embedded systems middleware perspective. This dissertation also presents an evaluation of our working middleware framework. The evaluation results illustrate the need to balance tradeoffs between application design, hardware resource constraints, and desired levels of multiple Quality of Service constraints.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iv

ATTRIBUTIONS vi

ABSTRACT viii

LIST OF TABLES xiv

LIST OF FIGURES xv

CHAPTER

1. INTRODUCTION 1

1.1 Distributed Systems 2

1.2 Middleware 4

1.3 Embedded Systems 7

1.4 Quality of Service 10

1.5 Thesis Statement 14

1.6 Dissertation Organization 15

2. RELATED WORK 17

3. MIDDLEWARE ARCHITECTURE TAXONOMY 22

3.1 Taxonomy Development Goals 22

3.2 Fine-Grained Middleware Design Taxonomy 24
3.2.1 Embedded Hardware 24
3.2.2 Roles 30
3.2.3 Roles—Control Flow 31
3.2.4 Roles—Data Flow 32
3.2.5 Roles—Interaction Style 32
3.2.6 Software Input/Output 36
3.2.7 IDL Subsets 37
3.2.8 Key Taxonomy Benefits 41

4. MICROQOSCORBA ARCHITECTURE 43
4.1 Lifecycle Epochs 43
4.1.1 Design 44
4.1.2 IDL Compilation 46
4.1.3 Application Compilation 46
4.1.4 System/Application Startup 47
4.1.5 Run Time 47
4.2 MicroQoSCORBA Architecture 47
4.2.1 IDL Compiler 49
4.2.2 Customized ORBs and POAs 49
4.2.3 Communications Layer 50
4.3 Multi-Property Quality of Service 51
4.3.1 Fault Tolerance 52

4.3.2 Security 53
4.3.3 Timeliness 57

5. SECURITY REQUIREMENTS FOR EMBEDDED SYSTEMS . . . 58
5.1 Embedded Systems Security Requirements 58
5.2 Motivating Examples 62
5.2.1 Building Automation and Control 63
5.2.2 Status Information within the Electrical Power Grid 65
5.2.3 CORBA IDL for the Motivational Examples 67
5.3 MicroQoSCORBA Security Design Philosophy 68
5.4 MicroQoSCORBA Security Goals 70
5.4.1 Keep it small 70
5.4.2 Implement what makes sense 71
5.4.3 Investigate multi-property QoS tradeoffs 71
5.5 Security Design Space 72
5.5.1 Confidentiality 74
5.5.2 Integrity 75
5.5.3 Availability 76
5.5.4 Accountability 77

6. DESIGN AND IMPLEMENTATION 78
6.1 Overview 78
6.2 MicroQoSCORBA Security Implementation 78
6.2.1 Extending MicroQoSCORBA 79

6.2.2 Implemented Mechanisms 80
6.3 Development Environment 82
6.3.1 MicroQoSCOBRA Configuration Tool 82
6.3.2 MicroQoSCOBRA IDL Code Generator 84
6.3.3 Building and Deploying the Application 85
6.3.4 Automated testing 87
6.4 Cross Platform Comparison Methodology 87
6.4.1 Steady State 88
6.4.2 Raw Timing Performance Results 89
6.4.3 Event Filtering 94
6.5 Implementation Status 96

7. EXPERIMENTAL EVALUATION 98
7.1 Testbed Hardware and Software Tools 98
7.2 Testbed Application 99
7.3 ORB Evaluation 102
7.4 MicroQoSCORBA Comparisons 104
7.4.1 Evaluated MicroQoSCORBA Configurations 105
7.4.2 Application Size and Memory Usage 106
7.4.3 Latency Results 108
7.5 Security Mechanism Evaluations 110
7.5.1 Application Size and Memory Usage 110
7.5.2 Performance 116

7.5.3 Analysis 124

8. CONCLUSION 127

8.1 Contributions 127

8.2 Future Work 129

APPENDIX

A. SOURCE CODE EXAMPLES 132

A.1 Config_macros.m4 132

A.2 Client.java.m4 135

A.3 Server.java.m4 147

A.4 Makefile.mqcc 151

A.5 Makefile 153

B. FAULT TOLERANCE MECHANISMS 158

B.1 Fault Tolerance 158

B.1.1 Temporal Redundancy 159

B.1.2 Value Redundancy 160

B.1.3 Redundancy Examples 160

B.1.4 Group Communication System / Multicast 161

BIBLIOGRAPHY 163

LIST OF TABLES

Page

3.1 Refined Middleware Architecture Taxonomy 25

4.1 Lifecycle Time Epochs 45

5.1 Security Design Space 73

6.1 Implemented Security Mechanisms 81

7.1 ORB Size Comparisons 103

7.2 ORB Latency Comparison (ms) 104

7.3 MicroQoSCORBA Java Class File Size (bytes) 107

7.4 MicroQoSCORBA End-to-End Latencies (ms) 109

7.5 Cipher Impacts on Java Class File Sizes (bytes) 112

7.6 Message Digest Impacts on Java Class File Sizes (bytes) 114

7.7 Cipher and Message Digest Impacts on Java Class File Sizes (bytes) 115

7.8 Cipher Timing Results (ms) 118

7.9 Message Digest Timing Results (ms) 120

7.10 Message Authentication Code Timing Results (ms) 122

7.11 Cipher and Message Digest Timing Results (ms) 124

B.1 Fault Tolerance Mechanisms 159

LIST OF FIGURES

Page

4.1 MicroQoSCORBA Architecture 48

5.1 Example Building Automation CORBA IDL 68

5.2 Example Power Grid CORBA IDL 68

6.1 Linux Timing Performance Histogram with Microsecond Resolution 90

6.2 Linux Timing Performance Histogram with Millisecond Resolution 91

6.3 SaJe Timing Performance Histogram with Microsecond Resolution 92

6.4 SaJe Timing Performance Histogram with Millisecond Resolution 92

6.5 TINI Timing Performance Histogram 93

7.1 Testbed Application IDL 101

A.1 Config_macros.m4 Listing 134

A.2 Client.java.m4 Listing 147

A.3 Server.java.m4 Listing 151

A.4 Makefile.mqcc Listing 153

A.5 Makefile Listing 157

Dedication

This dissertation is dedicated to my parents and my wife.

CHAPTER ONE

INTRODUCTION

Computing systems have evolved dramatically over the last several decades. Initially, computers were room sized devices dedicated to single-purpose applications. Now, computing systems are smaller, often general purpose, and much more commonplace. Significant advances have been made in the realm of embedded computing hardware in recent years. Microprocessors and small computers are now being embedded into a wide range of systems—from small room thermostats to large jumbo-jets. As a result, traditional desktop computers now use just a few percent of the microprocessors produced annually, while the rest are in embedded devices, often hidden from view.

Recent advances in networking technologies, especially wireless technologies, have made it more feasible to develop distributed systems composed of embedded devices. However, software development tools have not kept pace with the advances in hardware capabilities. The lack of adequate software development frameworks and tools for the development of distributed embedded systems with severe resource constraints that must be accommodated motivated the research that is presented in this dissertation.

This dissertation describes a novel middleware framework for resource-constrained embedded systems. In order to better understand its context and significance the following subsections are presented in this Chapter. First, distributed systems and then middleware are defined and discussed. Then an overview of

embedded systems is presented. Quality of Service is presented next. Following these four subsections, the thesis and contributions of this dissertation are given. This introductory chapter then concludes with the outline of the remaining chapters of this dissertation.

1.1 Distributed Systems

Distributed systems are computing systems that are composed of two or more computers that are interconnected and cooperate in order to achieve a common goal. A major advantage of distributed systems is that resources such as data or processing power can be accessed, and hence shared, by the individual computers that are within the distributed system. A well designed distributed system can leverage the strengths of each of its individual systems in order to achieve objectives beyond the capability of any of its individual systems. For example, in the the travel industry, a travel agent can access reservation information for dozens of airlines and thousands of hotels—information well beyond the abilities of a single personal computer.

There are many distributed system design architectures. Some distributed systems are designed and deployed in a very hierarchical manner. For example, consider Automated Teller Machines (ATM). Each ATM acts as a "client" of a larger banking system. When a bank customer desires to make a withdrawal the ATM must first query the bank's (centralized) database in order to determine whether the customer is authorized and able to withdraw the amount desired. Other distributed systems are very decentralized. For example, several peer-to-peer file

sharing networks operate without any centralized control. Collectively, thousands or millions of files can be accessible on a given peer-to-peer network even though each node within the network is capable of storing only a relatively small number of files. Distributed systems can also be designed and implemented so that the computational power of a super computer can be accessed and shared by a researcher's desktop workstation.

The many benefits of distributed systems do not come without challenges. Some of the key challenges derive from the heterogeneity of the individual systems within the distributed system. The three examples briefly mentioned in the preceding paragraph illustrate some of these challenges. Namely, each of these distributed systems can be deployed on varying hardware platforms (e.g., a bank's centralized database may be deployed on a mainframe computer, but it would be too costly to deploy ATMs with mainframe computers). Nor can one assume that a common operating system exists on all platforms (e.g., vendors ship different operating systems with workstations and super computers). The networking technology that interconnects each system within the distributed system will also be different (e.g., a cell-phone sharing files via a wireless link versus a home computer with a broadband connection). Yet another challenge is that, in many cases, different programming languages and tools will exist for the different systems within the distributed system (e.g., C for the ATM versus COBOL on the bank's mainframe or Java on the researcher's workstation versus FORTRAN on the super computer).

Each of the challenges just presented can be overcome. However, they each

add to the complexity of developing distributed applications. Individual developers simply do not have the time to become experts on multiple operating systems, networks, programming languages, etc. Furthermore, advances in software tools and development frameworks have simply not kept pace with advances in hardware technologies. Thus, we are confronted with what is commonly called the "software crisis" as developers with their limited time are forced to develop distributed application on ever increasingly complex hardware.

1.2 Middleware

One partial solution to the "software crisis" is middleware. Middleware is a layer of software that exists in between the operating system and application-specific software layers—hence the name middleware, because it exists in the middle. By occupying this strategic position within the software hierarchy, middleware is able to shield an application programmer from many of the low-level details inherent within a distributed system by providing a common programming interface across all of the systems within the larger distributed system. Thus, a portion of the "software crisis" is alleviated because the developer is freed from the low-level complexities that result from the use of multiple hardware platforms, operating systems, and networking technologies within large, distributed systems. This freedom, in turn, means that developers can focus more on application design and implementation.

Two key benefits of middleware are its ability to mask system heterogeneity and to provide transparency. Depending upon the middleware technology

used, dissimilar hardware, networks, operating systems, or programming languages may be used. Heterogeneous system components can be used, because each system's middleware development tools provides a common Application Programming Interface (API) to the distributed system developer. These common APIs also can provide transparency with regard to location, concurrency, replication, and other facets because the APIs do not typically present this information to the developer once a connection has been established between systems.

The middleware framework that is presented in this dissertation is based upon a distributed object paradigm. Distributed object middleware allows a developer to transparently access remote objects as if they were local objects. For example, if the method foo is invoked on a local object, lObj, with the syntax lObj.foo(), then an invocation on a remote object, rObj, is similarly given by rObj.foo(). Distributed object middleware generally uses some form of an interface definition language that allows the distributed system developer to specify the object and method calls that will be implemented within a distributed application. Given these specifications, the middleware tools then map the object and method calls to each system's specific hardware, network, and operating system APIs.

Consider, for example, a distributed ATM banking system. With a distributed object based middleware, an ATM system developer is able to specify account objects that have a withdraw(amount) method. This specification is then used by the distributed object middleware tools to auto-generate stub and skeleton routines that perform all of the low-level parameter marshalling, network transmission, and other functionality associated with account.withdraw(...) invocations.

Because the middleware tools handle all of the heterogeneity and transparency details, the developer can better focus on the core ATM issues of whether account.withdraw(...) invocations are performed according to the bank's business policies. Developers do not have to concern themselves with low-level socket programming, hand-coding layouts of parameter lists and data structures, etc., but rather all of this is generated for them, usually with strong type checking

As illustrated by the previous ATM example, middleware allows software developers to design at a very high-level. The developers are freed from focusing on the low-level implementation details (e.g., connecting to remote sockets, big endian vs. little endian byte ordering, etc.) commonly associated with distributed systems development, thereby allowing them to focus on correctly implementing the application's core logic and functionality.

Traditionally, middleware has been focused on the corporate computing environment, and, to a lesser extent but much earlier, on wide-area military environments [STB86]. Middleware tools were developed for these environments in order to increase developer efficiency by allowing the developer to focus on application constraints and logic rather than on the low-level implementation details of a distributed system. With the aid of middleware tools, developers can interconnect disparate systems. They can also connect newer desktop workstations to a corporation's legacy back-end computing systems. In fact, the ability to "wrap" a small middleware layer around an existing legacy application has extended the useful life of many of these systems because once again the legacy data and computing power can be easily accessed.

Thus, for a number of reasons, middleware has become a highly successful tool that reduces the complexity of developing distributed systems. This, in part, helps reduce the "software crisis" associated with distributed systems development. However, very few middleware frameworks exist that were designed for the low-end of the computing environment. Research on and development of middleware frameworks for distributed embedded systems is very limited compared to the large number and type of embedded systems deployed today [Ten00]. This is a significant concern, because as will be pointed out in the next section, distributed embedded systems are being deployed at a rapidly increasing rate.

1.3 Embedded Systems

Embedded systems are not easily defined by a fixed set of functional criteria because embedded systems span a broad range of the computing spectrum. However, embedded systems can be defined in terms of their purpose and design goals. In most cases, embedded systems are designed with a single, specific task to accomplish under the control of a computational entity. Furthermore, the "computer" in an embedded system is often both logically and functionally hidden from view.

Embedded systems, like traditional computational system, may be composed into systems of systems. For example, a smart thermostat is a very resource-constrained embedded system. The thermostat's sole purpose is to acquire and report the ambient temperature in a room. Within a large building, thermostats can be composed into a much larger system that controls the building heating,

ventilation, and air conditioning (HVAC) system. At a larger level, the building's HVAC system can also be classified as an embedded system. The HVAC system has a single purpose, controlling the building's environment, and this system does it without user intervention. In fact, most of the building occupants would not even be aware of all of the computing systems that are used to control the temperatures within building.

The size of embedded systems varies greatly—from miniaturized sensors to large jumbo-jet flight control systems. Resource constraints within embedded systems vary widely with regard to memory, computation, communications, and power requirements. Another classification trait of embedded systems is their production quantity. Some embedded systems are produced in very small quantities (e.g., satellites), while others are mass produced in very large quantities (e.g., microwave ovens, automotive electronic ignitions). The cross-product of embedded systems traits (e.g., size, resources, production quantities) covers a very large number of separate embedded niche markets, development techniques, and concerns.

A few common trends are appearing amongst all of the embedded systems niche markets. The first is that component miniaturization and decreased production costs have enabled a growing ubiquity of smart devices, or in other words, a ubiquity of embedded systems. Second, advances in communications technologies, especially wireless technologies, are enabling the development of networked embedded systems. Together, both of these trends are fuelling a rapid growth in the deployment of distributed embedded systems.

However, traditional computer science research and development programs are not aimed towards this new growth area. David Tennenhouse, Director of research for Intel and former Director of DARPA's Information Technology Office, has cited the need to focus more research on this growing embedded systems market and what he calls "PROactive" computing [Ten00]. The 'P' in "PROactive" stands for physical, meaning that research must be focused on coupling computing systems to the real world; the 'R' is for real-time, meaning that results much be achieved in real-time, and the 'O' stands for out, meaning getting the human out of the decision processing loop as far as possible. These three concepts of "PROactive" computing are not new ideas for embedded systems developers, but these are concepts that have been underemphasized by the research community that is focused on developing techniques applicable to traditional workstation-based systems that comprise only 2% of the processors currently being produced [Ten00].

Developing software for any system is a complex task. However, the mass-market nature of many embedded systems niche markets further complicates software development. This is because high production numbers means that saving even a few cents per unit can add up to significant savings. Thus, there is a strong desire to use resource constrained devices in order to save costs. Furthermore, time-to-market considerations mean that software developers must maintain high levels of productivity.

The need to tailor desired functionality to the available resources present within

an embedded system and the need to ensure high levels of programmer productivity strongly motivate the need for embedded systems middleware frameworks. We note that this is a non-trivial task. Embedded systems software development is difficult, distributed systems development is harder, and the composition of these tasks—distributed embedded systems development—is even harder!

1.4 Quality of Service

The term Quality of Service (QoS) was originally used in the networking community to refer to the management of network resources such as bandwidth and latency, in a local area network, and without adapting to changing conditions. Without the proper management of these network resources, properties such as performance and perceived usability may not be met even if, in steady-state situations an application is capable of delivering the desired levels of service. For example, proper bandwidth allocation within a network ensures that a video-on-demand application can provide uninterrupted service.

In the last 5–10 years, the term 'QoS' has become to be used in a broader sense which encompasses the management of an application's "non-functional" properties—properties that impact the perceived benefit of an application rather than its core logic and functionality. This broader definition of QoS includes properties such as security, dependability, and timeliness. In today's distributed computing environment, these broader QoS properties must be included in the overall design of an application because an application's performance can now depend

upon security, dependability, and timeliness as much as it does upon network performance.

Security is a key QoS property that must be designed into distributed systems. Security is a QoS property because of its non-functional nature. Or in other words, an application's security and its core functionality (i.e., application specific logic) are orthogonal design components. However, the success of an application might depend on its achieved security QoS. For example, no one would use an on-line banking service that allowed unauthorized users to withdraw money from their accounts, even if the bank could prove that every withdrawal was accounted for and it had a high degree of fault tolerance.

Confidentiality, integrity, and availability are three classical security properties, each of which can be provided at varying levels. For example, longer cipher key lengths generally provide increased confidentiality, thus increasing key lengths will increase the confidentiality QoS of an application. Integrity QoS also varies widely, and can be provided by mechanisms that range from weak, but simple, parity checks to strong cryptographic message digest mechanisms. An application's continuity of service can be ensured by employing appropriate availability mechanisms.

Fault tolerance is another key QoS property. A variety of mechanisms, each with varying strengths and weakness, can be employed to ensure that an application continues to function correctly in the presence of faults. Temporal, spatial, or value redundancy mechanisms can be used to overcome various communications faults. Redundancy must be tuned to a given application environment, in order to

ensure a desired level of dependability QoS.

Missing a timing deadline in a hard real-time system can, by definition, result in a fatal error. But in many other applications, achieving timely responses is a matter of perceived usefulness. Thus, depending upon application constraints, timeliness can either be a functional or a non-functional constraint and therefore a QoS system property. Some applications desire high response rates, while others require predictable patterns of behavior. As with other QoS properties, various mechanisms can be used to achieve the desired QoS timeliness constraints.

The difference between QoS properties and QoS mechanisms is in essence a difference between *what* versus *how*. When designing systems, QoS property constraints (i.e., the *what*) must be focused on. By doing this greater flexibility and insight may be gained. For example, a designer should focus on the level of confidentiality needed for a given distributed application before deciding which security cipher (and key length) to deploy.

Distributed applications often have multiple QoS property requirements. Two examples will be presented to illustrate this point. First, consider a payroll system with the requirements that only authorized managers can certify time cards and that payroll checks are deposited directly into the employees bank accounts. Security QoS is needed to ensure that only authorized managers can access the system. Fault tolerance QoS is needed to ensure that correct direct deposit amounts are sent to the bank and timeliness QoS is needed to ensure that the deposits are made on pay day (and not a day later). Second, consider an online bussiness—security QoS protects customer payment transactions, dependability QoS ensures

that orders are not lost, and timeliness QoS guarantees (perhaps probabilistically, not absolutely) that orders are processed and shipped in a timely manner. Full support for QoS properties within distributed systems will only be achieved when designers are presented with design frameworks and tools that treat Quality of Service properties as first-class system design requirements.

Support for multiple QoS properties is also needed for embedded systems development. Often, an embedded system's functional constraints are met in a manner that hard-codes a given QoS point solution (i.e., a solution that works for only one point in a multi-dimensional set of constraints) into the overall embedded system. Indeed, this is often the case precisely because embedded systems developers have no other choice, given the lack of rich support for QoS in middleware frameworks for embedded systems. Hard-coded, point solutions are costly to maintain and they also do not easily evolve to changing environments or requirments. Distributed systems environments often change, therefore tools need to be researched and developed which can present distributed embedded systems developers with a framework that is both highly configurable, in order to support the embedded systems environment, and also QoS aware.

Distributed systems, embedded systems, and distributed embedded systems all have both functional and multiple non-functional or QoS requirements. Research has been conducted on individual QoS properties, and in some cases this research has even been conducted within a distributed systems domain. However, little research has been conducted in the area of composing multiple QoS properties within distributed embedded systems, even within the simpler distributed

(non-embedded) systems domain. Without this research, the "software crisis" in distributed embedded systems computing will likely continue to grow as even more application developers, who lack both embedded systems and QoS property training and backgrounds, enter this growing market.

1.5 Thesis Statement

This Chapter so far has presented a broad overview of distributed systems, middleware, embedded systems, and Quality of Service. Rapid advances in the miniaturization of microprocessors and related components are leading to the deployment of embedded systems in ever increasing numbers. Furthermore, advances in communications technologies, especially wireless networking, are leading to the rapid deployment of distributed embedded systems. Not only are these distributed embedded systems being deployed in existing domains, but the availability of low cost components is the driving force behind the emergence of completely new computing domains such as wireless sensor networks.

The research behind and development of software frameworks and tools are unfortunately lagging the research behind their hardware counterparts thereby exacerbating the already existing "software crisis." New frameworks must be developed that will leverage existing distributed systems, embedded systems, middleware frameworks, and quality of service mechanisms because it is simply unreasonable to assume that any one individual can be an expert in each of these computing fields. These facts lead to the following thesis.

Middleware frameworks for embedded systems can be designed

with fine-grained composability and they can also support multiple Quality of Service properties.

The need for such capabilities has been presented above, but the feasibility of providing such capabilities was unknown when this research began.

In order to verify the above, this dissertation make the following research contributions:

- A fine-grained middleware architectural design taxonomy, to better capture the wide variation of distributed systems interactions and hence how they can be configured.
- The design and implementation of an architecture for a fine-grained and composable middleware framework.
- An analysis of security requirements for embedded systems.
- The design and implementation of a highly configurable security subsystem.
- An experimental evaluation of MicroQoSCORBA's performance on three hardware platforms.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents related work. Chapter 3 presents the middleware taxonomy that was developed in order to better architect a finely composable middleware framework. Chapter 4 presents the architecture of MicroQoSCORBA, the middleware framework that

was designed and implemented in support of this dissertation. Chapter 5 presents a discussion of security with regards to embedded systems. Chapter 6 presents the design and implementation of MicroQoSCORBA. Chapter 7 presents an evaluation of MicroQoSCORBA. And finally, Chapter 8 presents the conclusions of this dissertation.

CHAPTER TWO

RELATED WORK

We know of no CORBA or non-CORBA middleware framework that allows both the application and hardware constraints to be used to tailor the middleware, or of middleware for small, embedded devices that has been designed to support multiple QoS properties. MicroQoSCORBA is also allowing the constraints to be chosen at a much finer granularity than any other system of which we are aware, because we are designing it from the device level up rather than using the top-down approach of standard, reflective architectural reference models.

Emerging CORBA standards and products are only beginning to address the deeply embedded systems market. The minimum CORBA specification [Obj02e] removes dynamic interfaces and other features, but still only reduces the memory footprint by about half. The e*ORB^{TM}[Ver03] framework by Vertel gets much smaller and closer to what is needed with respect to memory footprint. However, this is a point solution that does not allow application developers to tailor their constraints in ways appropriate to their applications to meet resource constraints. Further, memory footprint is only one of a host of resource and quality of service (QoS) issues that must be addressed for small, embedded devices. The "Universal Interoperable Core" or UIC-CORBA from Ubicore (originally LegORB) [UIC03] is a component-based ORB targeted at Embedded devices and PDA's. The size goes from 16KB for a CORBA client running on a Palm OS device to 37KB for a CORBA client/server running on a Windows CE device. It does allow

some customization. However, while there are very few details available, there seems to be much less granularity in the choice of constraints than with MicroQoSCORBA, and there is no support for QoS. The nORB [SGS01, SGH+02] is another middleware framework that is targeted for embedded devices. But, unlike MicroQoSCORBA which was designed from the bottom up, it is an attempt at reifying ACE/TAO [TAO03, GLS01, SLM98] with respect to small, embedded devices. nORB will also seek to apply pattern languages within a middleware framework.

Another related work is the OMG Smart Transducers Interface Request For Proposal (RFP) [Obj00]. Smart transducers are small, single purpose devices (e.g., sensors and actuators) with some level of built in processing and communications support. This RFP is seeking to standardize a very lightweight communications API for these devices. Thus, this effort is focused on only a small part of the MicroQoSCORBA framework, namely the communication subsections.

Other CORBA-based frameworks have explicit support for Quality of Service or employ reflection, or both, as does MicroQoSCORBA, but are not intended to scale down to small devices. These systems also generally allow adaptivity much later in the design lifecycle than does MicroQoSCORBA, which is appropriate given they are not targeting small footprints. MULTE is a multimedia middleware platform that handles a range of latency and bandwidth requirements [Mul03, PEK+00]. A reflective architecture is implemented in the Open-ORB Python Prototype [ABE00]. The Open-ORB architecture uses reflection to

achieve a flexible and adaptable middleware solution. Open-ORB provides openness and a configurable component-based architecture. The dynamicTAO framework allows dynamic adaptation and allows for replacement of different strategy modules for concurrency, scheduling, and security; its footprint is never less than 1 MB [KRL+00, KCBC02].

MMLite [FHPR03] is a modular system architecture that allows a system to be built from object oriented components. Each application is built, either statically or dynamically, from a set of components that determine its behavior. Depending upon the choice of components, a system with a memory footprint as low as 10 KB has been achieved. The QoS properties of an MMLite application can be changed by selecting different subsets of components. The primary QoS property that MMLite targets is real-time behavior and this is achieved in part via the use of multiple implementations of the scheduling component. MicroQoSCORBA has a much broader QoS breadth (e.g., security, fault tolerance, and real-time behavior). Another key benefit of MicroQoSCORBA is its support for analysis tools that help quantify the impact of various component (both functional and non-function) costs and compositional properties. Another difference between MMLite and MicroQoSCORBA is that MMLite uses the COM [Bro95] object model and XML based SOAP [GHM+03] as its communication framework, whereas MicroQoSCORBA is CORBA based and uses GIOP to communicate.

GOPI is another middleware platform with a modular architecture [CBM02]. It was designed for multimedia systems and it supports the ability to both specify and manage QoS mechanisms.

Several OMG CORBA standards cover various security aspects and they have been implemented by a variety of companies [OMG03]. The standards are the CORBA Security Service [Obj02f], Common Secure Interoperability (CSIv2) [Obj02b], Authorization Token Layer Acquisition Service (ATLAS) [Obj02a], and Resource Access Decision (RAD) [Obj01]. Individually and as a group, each of these standards is large and therefore full compliance with any of these standards would preclude deployment on small, resource-constrained systems. Although a baseline MicroQoSCORBA configuration maintains interoperability with other CORBA implementations, MicroQoSCORBA's security subsystem bypasses conformance with any of these standards in order to reduce resource usage on small, embedded systems. Another standard, the Java Cryptography Extension (JCE) [JCE03] supports multiple security mechanisms via the dynamic class loading of a security provider's implementation. MicroQoSCORBA by design does not support dynamic class loading because of both hardware/platform specific requirements (i.e., TINI does not support dynamic class loading) as well as improved resource usage.

At least two recent projects have focused on evaluating the security properties of various security mechanisms. From 1997 to 2000, the US NIST conducted an evaluation of proposed symmetric-key encryption algorithms to be used as a new Advanced Encryption Standard (AES) [NBB+00, Bur03]. This process concluded with the selection of Rijndael as the new AES and the publication of a new Federal Information Processing Standard, FIPS-197 [Nat01]. The New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project [NES03] is

another project that evaluated the strength of multiple security mechanisms. Both the NIST and NESSIE evaluations focused on strong security primitives. MicroQoSCORBA supports both strong and weak mechanisms (e.g., XOR-base encryption, CRC32 checksums) because many resource-constrained systems simply can not execute stronger mechanisms in a timely manner. Additionally, these other efforts have not focused on end-to-end performance within a middleware framework.

The DARPA NEST program [DAR] is funding several related projects. The Berkeley Wireless Embedded Systems (WEBS) [WEB03] has developed SPINS [PST+02], a security protocol for sensor networks. SPINS, like MicroQoSCORBA is based upon the use of symmetric-key security primitives because it was designed for extremely resource-constrained devices running TinyOS [HSW+00]. TinyPK [Tin03b], was developed at BBN in order to provide public key encryption support for TinyOS. Wood and Stankovic present a security analysis of sensor networks in [WS02]. Each of these projects is focused on security at either the networking or OS implementations—levels well below MicroQoSCORBA's focus. In particular, none of these projects provides security within a middleware framework.

CHAPTER THREE

MIDDLEWARE ARCHITECTURE TAXONOMY

The development of a fine-grained middleware taxonomy is a key contribution of this dissertation. The first section of this chapter presents the motivation behind the development of this taxonomy. After that, the taxonomy will be presented and explained. This chapter then concludes with a discussion of the key benefits of developed taxonomy.

3.1 Taxonomy Development Goals

Two key goals were met by the development of the middleware taxonomy that is presented in the next section. The first goal was to focus MicroQoSCORBA on embedded systems middleware development. The second goal was to develop a fine-grained taxonomy that would enable the development of a finely composable middleware framework. Each of these goals will now be discussed in turn.

The space of embedded systems application development is both large and diverse. As mentioned in Section 1.3, embedded systems are typically developed to accomplish a single task or purpose and they are often deployed on resource-constrained devices due to cost constraints. Thus, a general purpose and therefore overprovisioned middleware system is not suitable for many distributed embedded systems development efforts. But, in order to design and implement a suitable middleware framework, a clear understanding of middleware requirements for embedded systems was needed.

Focusing on embedded systems requirements helped refine as well as point out all of the various middleware components within a distributed system. This is because, as one considers the very low-end of resource-constrained devices, one comes face-to-face with the question of "When does a stripped-down middleware environment lose so much functionality that it can no longer be called middleware?" Each component that could conceivably be "stripped out" was categorized for inclusion within the middleware taxonomy. These components were then organized into the orthogonal components presented in Table 3.1. Iterating through this process, while keeping a strong focus on middleware for embedded systems, accomplished this first goal.

The second goal behind the development of a middleware taxonomy was to enable the design and implementation of a finely composable middleware framework. Designing a composable framework requires a thorough understanding of what functionality and components are configurable within the framework at the finest possible granularity. The development of the taxonomy clearly identified the individual middleware components and roles within a distributed embedded system. Thus, the taxonomy did truly enable the fine-grained composable nature of MicroQoSCORBA because efforts behind the development of this taxonomy we later used to develop the overall fine-grained and composable architecture of MicroQoSCORBA.

3.2 Fine-Grained Middleware Design Taxonomy

The middleware taxonomy developed for this dissertation is presented in Table 3.1. The four broad categories within the taxonomy are: embedded hardware, roles, software input/output, and IDL subsetting. Each of these categories will be covered in depth in the following subsections.

3.2.1 Embedded Hardware

The choice of what hardware to support is a critical factor for an embedded application. A priori knowledge about hardware design choices allows a Micro-QoSCORBA designer to appropriately constrain code generation and other hardware specific optimizations. In a non-embedded environment, some designers will leave these choices unconstrained and simply assume that the underlying operating system will chose the appropriate constraints—but this is often a risky assumption in the embedded systems environment. One key hardware choice is the decision regarding the heterogeneity of the embedded system devices as well as the hardware to which these devices will be connecting. Typically, middleware is built to support a large degree of heterogeneity, but this does not have to be the case with embedded systems. Many embedded system designers have substantial control over their deployment environment, so they can (and often do) reasonably dictate a common platform for all devices within the system. For example, the design of an office building can call for identical room sensors/controls. It can also be less expensive, in a global sense, to deploy asymmetric hardware; especially if only a few nodes need to be resource-rich (e.g., floor-wide controllers)

Table 3.1: Refined Middleware Architecture Taxonomy

Embedded Hardware	**Roles (Client/Server/Peer)**			**Software Input/Output**	**IDL Subsets**
	Control Flow	**Data Flow**	**Interaction Style**		
System Composition • Homogeneous • Asymmetric ***Hardware I/O Support*** • Serial, Parallel, 1-wire, Ethernet, IrDA, Bluetooth, GSM, GPRS ***Resources*** • Memory • Power ***Processing Capabilities*** • 8-bit, 16-bit, 32-bit, . . .	***Connection Setup*** • Initiate setup • Receive setup requests ***Service Location*** • Hardwired-logic • Config. file • Name service • Other	***Data Direction*** • Bits in • Bits out • Bits in/out ***Parallelism*** • 1 message in transit • N messages in transit	***Sync*** • (Send/ Receive) ***Async*** • (One-way msgs) ***Msg. Push*** ***Msg. Pull*** ***Passive*** ***Pro-Active*** --- ***Event & Notification Services*** ***Publish / Subscribe***	***Data Representation*** • CORBA CDR • MQC CDR • . . . ***Protocols*** • TCP/IP • UDP • PPP • 1-wire ***Gateways*** • Data representation • Transports • Protocols	***Message Types*** • Request • Reply • Locate ***Parameter Types*** • CORBA in, out, inout ***Data Types*** • char, short, long, float, double, . . . ***Exceptions*** • System • User ***Message Payload*** • Fixed length • Variable length

and the rest (e.g., individual room controllers) can be resource-poor and thus less expensive.

To some degree, the choice of a system's hardware will also have an impact upon the role(s) and software I/O that a given device can support. Some extremely "small" devices will not need a full Ethernet solution or perhaps it may be too expensive for the target application. However, if hardware Ethernet support is removed, the device's software must be configured to use another networking technology. Another key hardware choice must be made regarding the capabilities of the processor. Processing capability is included in Table 3.1 to indicate that the size of individual systems and the networks to which they belong will vary widely.

Constraining hardware choices allows the MicroQoSCORBA designer to configure code generation and other optimizations performed by MicroQoSCORBA. Key choices that may be considered regard system composition, I/O support, available resources, and processing capabilities, each of which will now be discussed.

System Composition

One key choice is the decision regarding the heterogeneity of the embedded systems hardware as well as the hardware to which these devices will be connecting. Typically, middleware is built to support a large degree of heterogeneity, but this does not have to be the case with embedded systems. Many embedded systems designers have substantial control over their deployment environment, so they can (and often do) reasonably dictate a common platform for all devices within the system. It can also be less expensive, in a global sense, to deploy asymmetric

hardware, especially if only a few nodes need to be resource rich and the rest can be resource poor and thus less expensive.

Hardware Homogeneity. If a distributed embedded system's hardware is homogenous, many simplifying assumptions can be made. For example, if homogenous hardware is being used, then one may choose to eliminate support for CORBA's Common Data Representation (CDR). CDR is absolutely essential when different machines with different native data representation exists in the same application space, but when homogenous hardware is being used, there is no need to encode and decode data into a common representation because the native format could be used by all. Being able to eliminate CDR support will have an impact on both space and time within the MicroQoSCORBA applications. Space is saved because the code required to convert data in to and out of CDR can be eliminated. Time is saved since the calls that are required to encode and decode the data into CDR do not need to be made as data is sent and received from the network. Both of these savings will likely be small, but they might be enough to allow a less expensive processor to be used.

Asymmetric Hardware. In some cases it might make sense to deploy an asymmetric mix of devices in an embedded systems application. Generally the number of servers is lower than the number of clients. Thus, even small cost savings in the client hardware could be significant when multiplied across all of the client devices that will be deployed. Some existing applications already transfer some of the processing load from the clients to the server (e.g., send raw rather than

processed data). MicroQoSCORBA can shift some of the communications burden as well as processing load. This can be done, for example, by shifting from a "receiver makes right" to a "powerful makes right" paradigm. In "receiver makes right," each device is responsible for converting CDR data on the communications medium into its on native format. The switch to "powerful makes right" would require that the more powerful device would transmit data to the weaker device in the weaker device's native format. This adds an additional burden to the power device, but it saves both processing load and some application code size from the smaller device. One disadvantage is that instead of a linear number (i.e., $2N$) data conversion routines a quadratic number (N^2) routines will be required. But, the additional development costs might easily be recovered if millions of smaller, cost-effective devices are to be deployed.

Hardware I/O Support

Another key design choice will revolve around the communications hardware and software that will be deployed with the embedded system. Internet connectivity can almost be assumed for any desktop application, but in embedded environments this can not be readily assumed. Most embedded devices to date have been stand-alone devices, but that is changing. More and more devices are becoming interconnected, but the connection medium is not always based upon standard Internet protocols (e.g., Ethernet and TCP/IP). Many devices will have a simple UART based serial communication capability. Networking is becoming more ubiquitous and now some devices are now shipping with Ethernet support built-in. Wireless networking is also advancing and more devices are shipping with

wireless support built-in (e.g., infrared, Bluetooth, AirPort). As devices move up the communications food chain (e.g., digital I/O to Internet), more and more resources are consumed and required at the device. For this reason alone, some designers will opt for a less powerful or robust, but less expensive solution. If at the initial design stage, the choice is made to not support TCP/IP, then Micro-QoSCORBA will eliminate this support from the resulting application. But, the removed of standard Internet support, will require that an Environment Specific Inter-ORB Protocol (ESIOP) be used instead of standard IIOP. An ESIOP has the potential to more efficiently use the resources of the embedded device, but it will also mean that these devices will have to pass their data through a gateway device before their data can be sent to the Internet at large.

Resources

Memory and power are two resource issues that must be addressed by embedded systems designers. Memory is an issue because the RAM on some embedded processors is measures in bytes instead of kilobytes or megabytes as is often the case with desktop workstations. Many embedded systems are often battery powered which means that they have a limited lifespan. This lifespan can be increased or decreased depending upon the computational burdens placed upon the system. Both of these resource issues as well as others have an impact upon embedded systems and therefore they should be addressed by middleware frameworks.

Processing Capabilities

Another key choice must be made regarding the capabilities of the processor. The size of the system is also included in Table 3.1 to indicate that the size of the individual systems and the networks that they compose will vary widely. The capabilities of a given processor chosen for an embedded application will have an impact. For example, the processor that controls a washing machine will require less computing power than the autopilot for an airplane. Processors of varying speeds (e.g., MHz to GHz), design (e.g., RISC, CISC, 8-bit, 64-bit) and purpose will be used in various systems. MicroQoSCORBA will remove support for components that are not present in a given device. For example, the embedded system for a washing machine will not need the precision of 64-bit computation, so it would be pointless to bundle support for 64-bit numbers (integer for floating point values) into a washing machine. This is just one example of how a front-end design choice can trigger changes that will affect the outcome of a final product.

3.2.2 Roles

A device's role has an impact upon both the software and hardware deployed at the node. The initial design for MicroQoSCORBA was based upon a simple client/server/peer stratification of our devices. However, the current, more constrained, role taxonomy allows MicroQoSCORBA to more precisely configure an application with only its needed functionality and no more. For example, a parasitically powered device can sense a room's temperature, but it cannot initiate a

connection to a remote system. So in this extreme case, code that initiates connections with other devices need not be present. Some devices may only have the capability (or need) to either transmit or receive data. If this is the case, one can remove software from these devices that either receives or transmits data. Likewise other restrictions may be placed upon the device, depending upon its data flow or interaction style, allowing for reduced code size.

3.2.3 Roles—Control Flow

A key sub-role within the middleware role hierarchy is the the assignment of control flows. Namely, which systems will establish the control flows and how will they do it.

Connection Setup

Establishing the initial connection between two embedded systems can be accomplished by either system. If a node does not have to establish connections, then the logic that is required to lookup and connect to other systems can be removed from its functionality.

Service Location

Another key factor is establishing control flows is determining the location of the desired systems. Locating a node can be done in a variety of means, perhaps the least computationally expensive method is to simply ‘hard-code’ in a predetermined value. But, this method, while effective, is severely constrained over the life cycle of an application that must be deployed over several generations of hardware and capabilities. Object references can also be stored in configuration

files, or made available via naming services. By determining how systems can be identified and located, unnecessary code can be removed from an application.

3.2.4 Roles—Data Flow

The data flows, within a middleware system, are orthogonal to its control flows and interaction styles.

Data Direction

A middleware system can be configured to support receiving data (i.e., bits in), sending data (i.e., bits out), or both. If a system only sends data out, then the middleware support code on that system could have all of its data demarshalling code removed because there will be no incoming data that needs to be demarshalled.

Parallelism

The data flows within a distributed system can also be configured or designed to occur in parallel or in a synchronous manner. Disallowing the transmission of multiple messages at any given time simplies the bookkeeping logic associated with buffering and sorting messages into their proper order.

3.2.5 Roles—Interaction Style

The interaction style of an embedded device is a key design point. Not only will it effect the overall design of the distributed embedded system, but it will effect the design of individual nodes within the system. Segregating nodes into their individual roles is not enough because a given server could just as easily push its data to the clients as it could require the clients to pull the data. As in the case of role assignments, assigning interaction styles allows the MicroQoSCORBA

design process to accurately adapt the lifecycle stages to each embedded device.

Synchronous Messages

When data is either pushed or pulled from node to node, messages will be used. Traditionally, messages have been sent in pairs. First the message is sent, then the node waits for a reply from the destination node. This is intrinsically a synchronous model of communication.

Asynchronous Messages

One-way messages do not require a response from the message recipient. When used judiciously they can be used to implement an asynchronous communications paradigm. When one-way messages are chosen at the design stage, MicroQoSCORBA can optimize the resulting code because resources will not need to be consumed while waiting for a response.

Message Push

Knowing ahead of time that all data will be pushed out from a node allows MicroQoSCORBA to trim down (or even remove) the ability to respond to external commands and events. On the other hand, the node must keep track of all the other nodes that need the data pushed out to them. This will result in the potential for a large list of nodes that must have data pushed out to them. If the devices are too constrained to keep such a list, then this interaction style will probably need to be avoided.

Message Pull

If a device must pull its information from other nodes, then its own internal logic must keep track of elapsed time, external data sources, etc. so that it will know when it is appropriate to pull in new data. This additional logic will consume additional systems resources (e.g., memory, hardware timers), but it also ensures that only needed data is pulled into a node rather than pushing out data that may or may not be used by other nodes. Each node is responsible for locating the nodes from which it needs to pull data.

Message Push/Pull

In some cases nodes will need to both push some data out and pull some data in. This will likely happen when a node needs to consume some result, process it, and then deliver the processed result to yet another node. Using a push/pull interaction style will not provide a lot of potential on the part of MicroQoSCORBA to initially constrain the functionality of nodes.

Passive

Some embedded systems can be designed so that they operate in a passive manner—never initiating an interaction, but only responding to requests from other systems. In such cases, the potential exists to optimize the event loops within the middleware implementation because the designer knows that no self-initiated responses are required.

Pro-Active

Rather than being passive, some systems can be pro-active—sending data and commands before they are requested. For example, a temperature sensor could transmit its temperature value every minute whether it has received a temperature read request or not.

Exceptions

Exceptions are a useful error handling mechanism and interaction style. But, exceptions come with a considerable cost. Since exceptions occur at unpredictable times (i.e., when errors occur), mechanisms must be put into place that will catch the exceptions when they occur. Removing exception support could result in a significant reduction in both code footprint and the consumption of other resources within a node. Another argument for the removal of exceptions is that a thin client will simply not have the sophistication to deal with an anomalous result. If a server goes off-line, all that a truly thin client might be able to do is to retry and hope that a good result can be obtained on the second attempt.

Event and Notification Services

CORBA standard supports event and notification services. MicroQoSCORBA may support these services. Naturally, their support will have implications on the amount of required resources at individual nodes.

Publish / Subscribe

The publish / subscribe interaction style decouples in both space and time the production and consumption of events within a distributed system. Currently,

publish / subscribe is not implemented within MicroQoSCORBA, but it is listed in Table 3.1 for completeness.

3.2.6 Software Input/Output

One can not assume that all embedded devices will be able to support CORBA's Internet Inter-ORB Protocol (IIOP). For some applications the cost of implementing IIOP will be too costly because it requires TCP/IP. MicroQoSCORBA provides support for IIOP as well as Environment Specific Inter-ORB Protocols (ESIOP). When extremely resource-constrained devices need to communicate it may be appropriate for them to do so in a two-tiered approach. The resource-constrained devices can use a "resource poor" protocol to connect to a gateway machine that bridges data and commands to the Internet at large, with CORBA IIOP. For example, rather than running an Ethernet cable to an Ethernet enabled device in each room of a building, inexpensive serial cables can interconnect each room's sensor with a centralized floor specific gateway computer that serves as a bridge to the building's network.

Data Representation

Data within a middleware system must be represented in a uniform manner between systems so that each system can marshal and demarshal messages appropriately. One common representation is CORBA's Common Data Representation (CDR) [Obj02c]. If a limited subset of data items is to be marshalled and demarshalled the opportunity exists for the deployment of a more constrained data representation.

Protocols

Once the data representation has been selected a networking protocol must be used to transmit the middleware messages and data over the network. The protocols supported by the devices will have an impact upon the distributed system. For example, simple serial communications can be supported with a minimal amount of hardware and software support. On the other hand implementing a standards compliant TCP/IP stack will require a fair amount of hardware and software support. Thus, the choice of which protocols to support will play a key factor within this MicroQoSCORBA design stage.

Gateways

Cost considerations will, in some cases, dictate that non-standard data representations, protocols, or transports be used within a given distributed embedded system. In these cases, gateways can be established which bridge the non-standard conforming messages onto a standards conforming network.

3.2.7 IDL Subsets

CORBA has a rich and powerful Interface Definition Language (IDL) [McK03a] that is used to define an application's functional interfaces. Not all embedded systems require the full expressiveness and power of CORBA's IDL. For example, CORBA IDL supports 'Any's, a data structure that is not defined and typed at compile-time. Supporting Anys is costly in terms of both code size and application performance and Anys (and especially dynamic Anys) also introduce the

notion of run-time adaptability—something that many embedded systems designers prefer to avoid. Because of all of these considerations, MicroQoSCORBA does not support Anys, Dynamic Anys, nor other composite data structures (e.g., structs and arrays).

MicroQoSCORBA's IDL compiler scans an application's IDL file to determine which CORBA IDL message types (e.g., request, reply), parameter types (e.g., in, out, inout), and data types (e.g., boolean, char, long) are needed by the application. The IDL compiler then uses this information to custom generate stub and skeleton code. This information is also used to configure customized client ORB, server ORB, and portable object adaptor (POA) configurations that only support the required message, parameter, and data types. This means that when floating point numbers are not required (e.g., some applications will run on devices with no hardware for floating point numbers), support for marshalling and demarshalling floating point numbers will be removed from the client and server applications. Some applications on small devices are designed without support for CORBA exceptions, because the application does not have enough resources to support exception processing. In these cases, a developer can remove support for CORBA system and user exceptions from an MicroQoSCORBA application.

Message Types

CORBA supports several message types. Two of these, the request and reply message types are the most commonly used. Eliminating support for the other CORBA message types allows for reduced resource usage (e.g., code size). The IDL subset choice would then be used during the interface compilation stage to

produce an optimal set of stubs and proxies for the distributed application.

Parameter Types

Parameter type optimization can also be made when possible. CORBA supports ‘in’, ‘out’, and ‘inout’ parameter types. A system that is configured to only transmit data will not need to support either the ‘out’ or ‘inout’ parameter types. This also means that data demarshalling code can also be removed from the system as well.

Data Types

It is not at all unreasonable to remove support for some data types within MicroQoSCORBA’s IDL compiler. As mentioned earlier, some low-end, low-cost hardware will support only a limited number of data types (e.g., 8-bit and 16-bit ints). If the hardware being used can not support a given data types, say floating point values, then the removal of support for floating points values within the application could potentially save both space and time. Space would be saved since the encoding/decoding logic associated floating point numbers as well as the routines used to convert these values into other data types could be removed. Time would be saved if enough data types were removed so that the resulting CDR mappings could be streamlined and improved. Even if the embedded systems hardware supported a given data type, one might still want to remove MicroQoSCORBA support for it. For example, consider an embedded system that reports the current temperature. Perhaps the ADC temperature sensor has 14-bit accuracy, but the application only needs to know the temperature to within a few degrees. The

designer could choose to drop support for floating point values and instead report the temperature as an integer value. Some precision in the reported temperature value will be lost, but the ability to remove support for floating point values might result in other potential savings.

Exceptions

Support for exceptions is one easily removed IDL component. As discussed earlier this would have a significant impact on resource usage within the node. But, support for exceptions does not need to be an all or nothing proposition. A middle of the road approach could be to provide a limited set of predefined or generic exceptions. The applications developer could then map the application's exceptions onto these generic, pre-defined exceptions. This would be less than optimal from the viewpoint of debugging an application, but from the viewpoint of deploying a resource-constrained application it is necessary is some cases.

Message Payload

The size of a MicroQoSCORBA message may be determined at both design time and at interface compilation. A traditional CORBA implementation does not worry about constraining the size of messages. But in an embedded system, with limited resources, the message sizes must be analyzed. Large message sizes mean that large buffers must exist so that the message can be marshaled or demarshaled and transmitted (received). Constraining the payload size of messages at design time will give the MicroQoSCORBA developer more control over the memory requirements of the application.

IDL Subsetting Example

The following example is presented to help motive the benefits of MicroQoS-CORBA's support for IDL subsetting. Consider a building example, each room may have a temperature sensor that can measure 10 or 12 bits of precision. In some instances, a building designer will decide that 8 bits of precision are sufficient to control a room's temperature. Thus, the building designer will specify an IDL interface for the system that passes temperature information in only 8-bits of information. This will save both bandwidth (only 8 bits instead of 16- or 32-bits are sent), as well as code size since the ability to marshal and demarshal 16 and 32 bits values is eliminated. By itself this is not a large improvement, but combined with the fact that support for IDL exceptions can be removed and that message formats can be constrained based on these subsets of standard IDL, the potential exists for significant gains in both reduced resource usage of the hardware and simpler software.

3.2.8 Key Taxonomy Benefits

The refined fine-grained middleware architectural design taxonomy presented in this chapter is a key contribution of this dissertation. The analysis for this taxonomy provided a foundation upon which a robust and finely composable middleware framework was developed. Additional research could further refine and expand upon the design taxonomy presented in this Chapter, the results of which can then be used to identify additional fine-grained architectural components that

can be incorporated into MicroQoSCORBA's architecture. The current taxonomy highlights how embedded systems hardware constraints can be leveraged by a middleware framework in order to configure and deploy resource constrained middleware solutions on embedded systems.

CHAPTER FOUR

MICROQOSCORBA ARCHITECTURE

The design and implementation of an architecture for a fine-grained and composable middleware framework is a key contribution of this dissertation. The first section of this chapter presents a discussion of application lifecycle epochs. MicroQoSCORBA's architecture leverages the constraints that can be bound during each of these epochs in order to finely configure an application's resource usage and quality of service. Section 4.2 presents MicroQoSCORBA's architecture. The multiple quality of service properties that MicroQoSCORBA supports are presented in Section 4.3.

4.1 Lifecycle Epochs

During the successive stages in the lifetime of any distributed application program, designers must provide information on how the application may be configured, what tradeoffs will be supported, etc. To support this, MicroQoSCORBA has an underlying architecture and toolkit that span the complete development cycle from first concept in the design stages to application runtime. We divide the lifetime of a MicroQoSCORBA project into five epochs: Design, IDL Compilation, Application Compilation, System/Application Startup, and Run Time. During each of these epochs, various constraints are bound. During the application's lifecycle, as each constraint is bound, opportunities exist for reducing and/or refining many key facets of the application. A complete list of each constraint that may be bound

would be too large to include in this dissertation, so only a few key constraints are shown in Table 4.1.

These five lifecycle epochs apply to both embedded and non-embedded systems development. However, one of the first things to notice about Table 4.1 is that MicroQoSCORBA's base architecture focuses most of its effort on constraining choices early on in the lifecycle. This happens, in part, because the dedicated nature of many embedded systems allows for constraints to be determined early in the design process. Additionally, for many embedded applications, supporting too much adaptability during the latter stages (e.g., startup and especially run time) would result in costly, additional resource consumption (e.g., memory footprint, application context switches). Our MicroQoSCORBA approach contrasts with many other reflective middleware systems such as QuO [ZBS97], which leaves most constraints to be bound later in the cycle, in order to best facilitate runtime adaptivity. While MicroQoSCORBA cannot afford such late binding flexibility, in QuO it is necessary to support the runtime adaptivity necessary to deal with the dynamic characteristics inherent in the wide-area network environments it supports.

4.1.1 Design

The choices made in the design stage affect all future stages. It is during this stage that key decisions regarding the makeup of the embedded system's network will be made. For example, will all systems deployed share the same hardware configuration (homogeneity), what processor will be used (type, capability), will some nodes have more resources/processing power than others (symmetry), and/or what

Table 4.1: Lifecycle Time Epochs

Lifecycle Epoch	**Constraint Bound**	**Representative Example**
Design	HW Heterogeneity	Symmetric, Assymetric
	HW Choice	x86, TINI, ColdFire
	Communications HW	Symmetric, Assymetric
	Processing Capability	50 Mhz, 1 Ghz, 8bit, 32bit
	System size	Small, medium, large (e.g., transducers to jets)
	Power Usage	Line, battery, parasitic power
IDL Compilation	Communications Style	Passive, Proactive, Push, Pull
	Stub/Proxy Generation	Inline vs. Library usage
	Message Lengths	Fixed, variable length messages
	Parameter Marshalling	Fixed Formats
Application Compilation	Space/Time Optimizations	Loop unrolling, code migration, function and proxy inlining
	Library Usage	Static vs. dynamic library linkage
System / Application Startup	Device Initialization	Serial port baud-rate, handshaking
	Network Startup	Bootp, DHCP
	Major QoS adaptation	Select between QoS modules
Run Time	Minor QoS adaptation	Adjust QoS parameters

technology will be used for communication?

4.1.2 IDL Compilation

The IDL compilation stage begins to exploit many of the constraints bound during the previous epoch. For example if an 8-bit processor is being used, then support for larger data types may be dropped. The communication style and role of the devices will be set during this stage. Does the developer need this device to proactively push data to other nodes? Is another role more necessary? The IDL compiler is also able to change the functionality of the generated code depending upon hardware and role constraints. For example, is there enough memory present to inline the proxy/skeleton routines within the client/servant implementation? Can messages be constrained to be of a given size? Likewise, can optimization be made to the data marshalling routines?

4.1.3 Application Compilation

Additional configuration choices customizing the middleware can be made during the application compilation stage. Existing “off the shelf” tools and compilers perform these compilation steps, for the most part. Implementing a highly optimized compiler is beyond the scope of the MicroQoSCORBA project to date, but directing the performance of these compilers and tools is quite beneficial. Thus, if the developer knows that memory will be at a premium, the MicroQoSCORBA configuration tools can direct the compiler to optimize the compiled code so that space is conserved. Another constraint that is bound during this epoch is the choice of static versus dynamic linking library code.

4.1.4 System/Application Startup

When power is first applied to an embedded device, both the system and application will start running. The binding of a few run-time MicroQoSCORBA constraints may be delayed until this time. The embedded device may have some hardware configuration options that are set with buttons, switches, etc. and these settings could control the startup state of the embedded hardware. At startup, the device's networking parameters might be automatically configured (e.g., DHCP). Another key hardware factor is that ROM is often more plentiful than RAM. Thus, multiple implementations could be written and burned into the device's ROM, then at startup the appropriate implementation could be loaded into RAM. This mechanism would allow a device to adapt to its environment in a very coarse way.

4.1.5 Run Time

A conscious choice has been made to limit the run time flexibility of a MicroQoSCORBA system. It is true that many embedded systems exist which have sufficient computing resources that can support flexibility at run time. But the growth area in embedded systems middleware is in the low end of the market where flexibility is neither required nor cost effective. In a few cases, increased flexibility might actually be detrimental in the sense that the predictability of a device's performance might suffer.

4.2 MicroQoSCORBA Architecture

One of the key benefits of the MicroQoSCORBA framework is its ability to target a range of embedded devices. This is accomplished by exploiting some novel

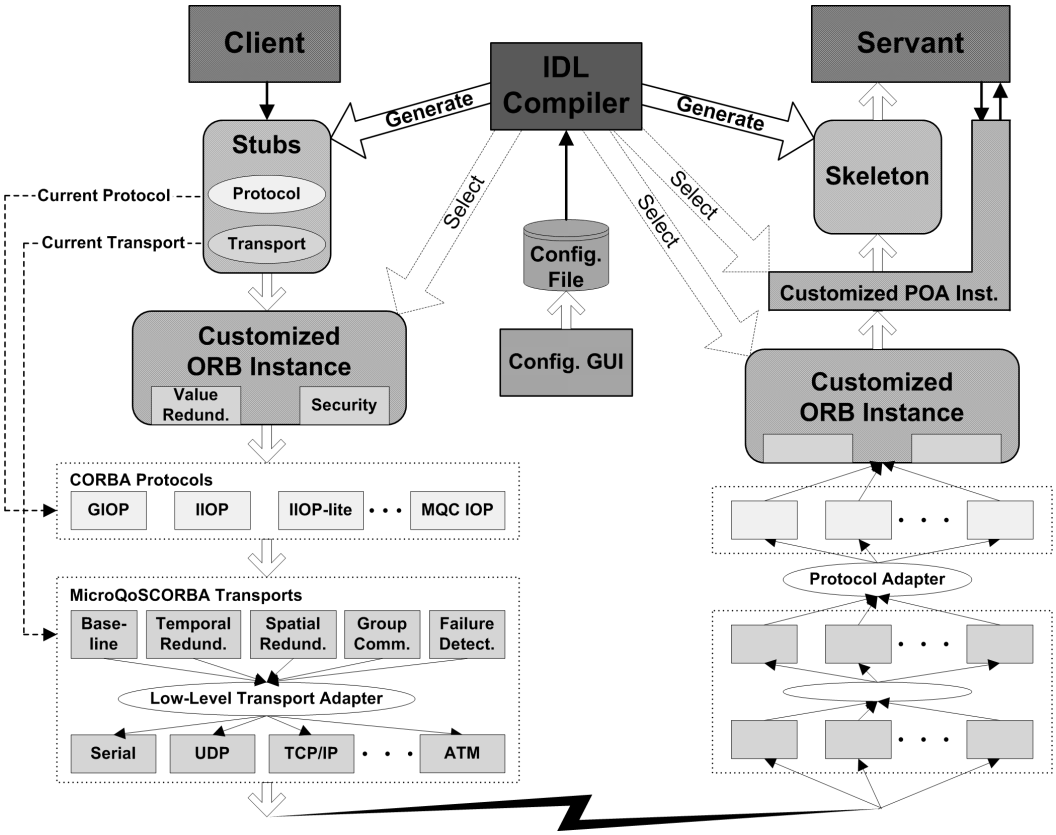

Figure 4.1: MicroQoSCORBA Architecture

adaptations in the standard CORBA architecture as well as binding constraints during the various epochs of an application's lifecycle (see Section 4.1). The architecture of MicroQoSCORBA is shown in Figure 4.1. Note that the IDL compiler has an increased role, customized Object Request Brokers (ORB) and Portable Object Adapters (POA) are used, and that the interaction between the ORB and the underlying communication layer has changed. We now discuss each of these in turn.

4.2.1 IDL Compiler

Every CORBA development environment has an Interface Definition Language (IDL) compiler. This compiler is responsible for parsing an application's IDL files and producing the appropriate middleware stub and skeleton code. Often, these IDL compilers assume that one canonical ORB implementation exists. For the standard desktop/workstation environment this is a reasonable assumption, since sufficient resources exist at the desktop to bundle in "everything" that is needed into one ORB implementation. But, a "one size fits all" solution does not scale down to small embedded devices.

MicroQoSCORBA's IDL compiler generates stubs and skeleton code that has been optimized for a customized ORB. The IDL compiler also selects and "hard codes" a given protocol and transport into the client-side stub routines. This removes ORB complexity and is also a work around that can eliminate the linking of unneeded protocol and transport code into the client-side application. The IDL compiler also optimizes MicroQoSCORBA's performance by removing support for unneeded IDL functionality (see Section 3.2.7).

4.2.2 Customized ORBs and POAs

Only so much can be done in the stub and skeleton code to reduce (or improve) resource usage for a given application. Thus, MicroQoSCORBA supports the ability to use customized ORBs and POAs. Depending upon the architectural and design choices discussed in Chapter 3 and Section 4.1, the IDL compiler selects a specialized ORB and POA for a given application. Then the IDL compiler

generates client-side stub and server-side skeleton code that is specialized for the previously selected ORB and POA. MicroQoSCORBA's ORB and POA provide enough functionality to interoperate with other ORBs (see Section 7.3), but neither the ORB nor POA provide the level of functionality required to support the latest CORBA standards.

MicroQoSCORBA is able to produce small client and server applications because it both uses customized ORBs and POAs and because its IDL compiler generates application and hardware specific stub and skeleton code. For many, MicroQoSCORBA's small footprint is one of its benefits. But, for others, especially those developing safety critical systems, the benefit is that MicroQoSCORBA does not deploy any unneeded software in its ORBs and POAs. This greatly reduces the effort of validating and maintaining a distributed embedded system.

4.2.3 Communications Layer

Many small, embedded devices have very limited communication abilities. For some applications, the support for CORBA IIOP interconnectivity may actually entail more code than is required for the application's logic. In these cases, support for a lighter-weight communication layer is needed. On the client side, the IDL generated stubs have a reference to the protocol and transport layer to be used. These references are given to the ORB so messages may be sent or received as needed. We note that the ORB could have used an abstract factory pattern [GHJV95], but that would have required linking in functionality for all of the MicroQoSCORBA's communication layers into a given application, something

that was neither needed nor desired.

UDP is one of the transport options supported in MicroQoSCORBA [Dam02]. Several TCP properties (e.g., being strictly sequential, fixed retransmission model geared for bulk data, poor wireless performance) were the primary reasons to look beyond the TCP transport protocol for MicroQoSCORBA. Additionally, not all devices have TCP support, for example most WAP enabled devices do provide datagram support at the wireless datagram protocol layer, but have no TCP support [All02]. MicroQoSCORBA supports unreliable datagram support as well as two reliable datagram protocols, which are provided by wrapping standard datagrams within a reliability layer [Dam02].

MicroQoSCORBA was architected and developed to support many protocols and transports. Environment and application specific protocols and transport layers can be designed and developed within the MicroQoSCORBA framework as needed. The advantage of using a specialized protocol or transport is that this specialized protocol or transport can be finely tuned to the exact needs of a given application and the abilities of the hardware upon which it is run.

4.3 Multi-Property Quality of Service

Meeting an application's non-functional constraints are often as important to the perceived success of an application as is meeting its functional constraints. For many embedded systems, non-functional constraints are important because of the system's deep coupling with its environment. For example, being able to control the temperature of a room in our office-building example is a functional constraint.

But, the responsiveness (i.e., timeliness) of heating/cooling systems and the desire to have only authorized users (i.e., security) change a room's temperature settings are important. So even though security and timeliness are QoS properties that apply to standard desktop and workstation class applications, these properties must be considered when designing embedded systems. Furthermore, many real-world applications must integrate multiple Quality of Service constraints (e.g., security, fault tolerance, and timeliness).

We have designed MicroQoSCORBA to support fault tolerance, security, and timeliness QoS constraints. In each of these subsystems there are multiple implementations of various QoS mechanisms, thus offering different tradeoffs of QoS versus resource consumption (e.g., encryption strength versus latency versus memory and power usage). The rest of this section is organized as follows: first, MicroQoSCORBA's fault tolerance support is overviewed; second, security is presented; and finally, a brief discussion of timeliness is presented.

4.3.1 Fault Tolerance

Most distributed applications require some level of fault tolerance in order to be successful. This is especially true with embedded distributed systems, since they are often mission critical components within larger systems (e.g., fly-by-wire systems for airplanes, anti-lock braking systems for cars). The following orthogonal fault-tolerant mechanisms have been incorporated into MicroQoSCORBA: temporal redundancy, spatial redundancy, value redundancy, failure detection, and

group communication. For more information on MicroQoSCORBA's fault tolerance subsystem, please refer to Appendix B and [Dor02, MHD+02]. Fault tolerance performance results are included in Section 7.4.

4.3.2 Security

Embedded systems designers must address computer and network security as their systems are being integrated into increasingly larger networks, even the Internet. MicroQoSCORBA has been designed with a wide variety of security mechanisms to support security QoS. The confidentiality, integrity, availability and accountability constraints that MicroQoSCORBA supports will be briefly discussed in the following subsections. Additional details about MicroQoSCORBA's security subsystem and these three security properties are presented in Chapter 5. Security evaluation results are presented in Section 7.5. Additional information on the design, implementation, and evaluation of MicroQoSCORBA's security subsystems is included in [McK03b, MBS].

Confidentiality

Confidentiality refers to the requirement that information can only accessed by entities that have been authorized to do so. Confidentiality of data and control messages is provided in MicroQoSCORBA via both physical and logical (e.g., encryption) mechanisms. For example, the designers of an office building can choose to put the mechanical room equipment on a separate network to ensure its privacy. If physical mechanisms such as this are used, then security mechanisms within MicroQoSCORBA will not be embedded into an application. On

the other hand, if the costs of physical protections (e.g., having multiple physical networks or special purpose hardware) are prohibitive, then confidentiality mechanisms within MicroQoSCORBA may be used. For example, the commands sent from the building engineer to the mechanical equipment room may be encrypted with an appropriately chosen cipher and key length. But, as will be shown in Section 7.4.3, encryption mechanisms are computationally expensive and must be planned for in an application's design as well as traded off against an application's real-time constraints.

MicroQoSCORBA supports several symmetric-key ciphers, covering a wide range of security strengths and performance impacts and tradeoffs. Two simple ciphers, an XOR cipher (where the plaintext is repeatedly XORed with a constant key value) and the Caesar cipher [Sta98], were implemented because because they consume few system resources (e.g., run-time memory) and execute quickly, see Section 7.4.3. But the level of confidentiality they provide is minimal. Other more cryptographically strong ciphers, such as the Advanced Encryption Standard (AES) [Nat01] are also supported because they provide stronger levels of confidentiality. AES was chosen because it is a federal standard that was selected for both its cryptographic strength as well as its suitability for resource-constrained devices. AES can be configured to use 128-, 192-, or 256-bit keys, thus providing varying levels of confidentiality QoS. Currently, MicroQoSCORBA supports the following symmetric-key ciphers: AES, Caesar, CAST5, DES, IDEA, MARS, RC2, RC4, SKIPJACK, Square, TripleDES, Twofish, and XOR.

Integrity

Integrity is a security property that is meet when mechanisms have been put into place to detect whether information is unaltered. Integrity constraints are not meet when information can be altered, added to, or partially deleted, either accidentally or otherwise, without detection. Integrity constraints of embedded applications vary widely. Once again, the overall system design will dictate some of the potential tradeoffs. In some cases, a parity-byte summation or CRC code will provide the level of integrity needed. If not, then the designer will have to decide if an application's real-time constraints allow for a comparatively slower message digest, authentication code, or digital signature mechanism to be used. MicroQoSCORBA currently supports the following message digests: Parity, CRC32, MD2, MD4, MD5, RIPEMD, RIPEMD-128, RIPEMD-160, SHA0, SHA1, SHA2-256, SHA2-384, SHA2-512, and Tiger. MicroQoSCORBA also supports HMAC [KBC97] message authentication codes based upon the previously listed message digests.

Availability

Availability is a property that concerns the need to ensure that a system can perform its given purpose as needed. Currently, MicroQoSCORBA relies heavily upon its fault tolerance subsystems (see Section 4.3.1) to help ensure service continuity. In addition to temporal and value redundancy, which were presented previously, MicroQoSCORBA also supports group communication with a variety of choices of message ordering and reliability [Dor02]. This support for replicated servers can aid greatly in designing high availability, embedded systems.

As illustrated in [WS02], many denial of service attacks against sensor networks operate below the middleware layer (e.g., link and transport layers). Embedded systems designers must therefore incorporate security and availability concepts into the initial designs of their applications. MicroQoSCORBA's fine-grained configurability allows one to easily configure and deploy an alternative lower level communication protocol in order to better protect against denial of service attacks.

Accountability

Accountability encompasses several properties such as authentication, authorization, audit, and non-repudiation. A few of these are not supported within our framework. Namely, MicroQoSCORBA will support neither delegation of authority nor non-repudiation. Delegation between peer embedded systems is generally not needed, thus our decision to drop delegation support. Effective non-repudiation requires user intervention (e.g., typing a pass phrase, providing a secure token) [Bla00]. Since many small embedded systems operate autonomously, this requirement cannot be met, therefore we have eliminated support for non-repudiation.

MicroQoSCORBA focuses on the trade-offs between hardware and software support and early constraining decisions made within an applications lifecycle. Thus, user and system accountability can be implemented via physical tokens (e.g., a hardware design choice) or via the use of software mechanisms (e.g., passwords). Likewise the choice to audit events must also be based upon hardware decision choices. For example, only one of our three testbed systems supports a

local storage device, thus the choice to audit events is also a choice to consume valuable system memory.

4.3.3 Timeliness

The initial timeliness efforts within MicroQoSCORBA are directed towards the computation aspects of a generic real-time model, which separates input and output, communication and computing [KV93]. Temporal and resource usage profiling support is being developed for MicroQoSCORBA's middleware framework. Valid configurations are being characterized with respect to computation time as well as static and dynamic memory requirements for a variety of environments. This information will provide a developer with insights for the development of real-time applications using MicroQoSCORBA, by delineating the temporal and resource usage costs for the various configurations. For more information on MicroQoSCORBA's support for timeliness and its temporal characterization please refer to [Law03].

Future research will address the communications and I/O considerations of the generic real-time system model. In addition, specializations of the generic model will be used to provide a framework within MicroQoSCORBA that will aid the developer who must both build and characterize real-time applications.

CHAPTER FIVE

SECURITY REQUIREMENTS FOR EMBEDDED SYSTEMS

This chapter presents an analysis of security requirements within the distributed embedded systems domain as well as MicroQoSCORBA's security subsystem. Security is one of the key non-functional properties supported by MicroQoSCORBA. In order to understand the current implementation of MicroQoSCORBA's security subsystem this chapter will first present the an overview of embedded systems security requirements in Section 5.1, followed two motivating examples in Section 5.2. MicroQoSCORBA's design philosophy and goals with respect to MicroQoSCORBA's security subsystems are presented in Sections 5.3 and 5.4, respectively. After that, a general overview of the security design space is presented in Section 5.5. The implementation of MicroQoSCORBA's security subsystem is presented in the next chapter in Section 6.2.

5.1 Embedded Systems Security Requirements

Good system security for standard desktop systems and small embedded devices share many common design features. Perhaps one of the most basic of these is that computer and network security is an overall system property–rather than just an individual component property. Not only must each individual component within the system be secure, but all of the components must also be integrated in a secure manner. Effective security must be pervasively designed in from the beginning as a coherent and cogent part of a system's architecture. It cannot be retrofitted after

the fact as evidenced by the constant stream of security vulnerabilities and patches released for a common desktop operating system family.

Embedded systems, by their very nature, have many application specific constraints imposed upon them that most desktop systems do not. In particular, embedded systems are often deeply coupled with their physical environment, whereas desktop computers operate in a virtual environment. In fact, this is one of Tennenhouse's three points in his call for more research on PROactive computing [Ten00]. Namely, the 'P' in PROactive stands for *Getting physical*—exploring the coupling of devices with their environments; the 'R' is for *Getting real*—responding in real-time at faster-than-human speeds; and the 'O' is for *Getting out*—getting humans out of the decision making loop. Each of these three attributes are often desirable within embedded systems, but they are often not applicable within the standard desktop computing environment.

Another key difference between embedded systems and desktop computing is their generality. Unlike desktop systems, embedded systems are often developed to accomplish a single task. Part of this is driven by simple design decomposition, but another part is driven by economics. In large volume applications (e.g., cell phones, automobiles, microwave ovens) saving just a few cents per unit can add up—literally to millions of dollars. This creates a strong market incentive to ship products with only *just enough* functionality and no more. Embedded systems designers must therefore consider the tradeoffs between performance, functionally, and cost during the initial design stages of an application. Conversely, they often must also try to reuse code as much as possible over the the life-cycle of a

product line, and sometime even across them. This suggests the need for a highly configurable development framework that supports fine-grained composability.

The combination of embedded systems being tightly coupled to their environment and the strong desire to save costs has a significant impact upon embedded system security. Namely, designers must determine just how much security is needed and at what cost. The ability to just add a "security patch" to the developed system will not work. This flawed logic of "we can just add security later" is even more flawed in embedded environments. This is because the embedded device will have been designed with just enough processing capability to meet its application goals and thus it will typically not have the extra computational power needed to support computational expensive security mechanisms that are added after the initial system design. Furthermore, the patch might close one security hole but yet open up another.

One, possibly unintended, consequence of deploying cost efficient hardware is that it tips the balance of power in the adversary's direction. When compared to a 4-bit or 8-bit microprocessor, a desktop system is, relatively speaking, a supercomputer—operating at speeds from one thousand to millions of times faster than a typical embedded device. Using this computational advantage, the foe will be able to effectively use brute force attacks against a slower, embedded processor. This means that foes with even modest means (e.g., access to a laptop computer) can be a serious threat to an embedded system.

Denial of service attacks are even more acute within a sensor network. As Wood and Stankovic point out in [WS02], sensor networks may be attacked at

the physical, link, network and routing, and transport layers. Furthermore, power must be taken into account when working with wireless, embedded devices, since one must assume that the adversary has a relatively unlimited power supply and thus could seek to exhaust the power budgets of the embedded devices [WS02].

Another concern with embedded devices is that they are much more susceptible to tampering because of their physical coupling with their environment. For example, it is difficult to ensure adequate physical security for an environmental temperature sensor that must be deployed in remote areas, where it is susceptible to the elements of nature as well as the examination or tampering of an adversary. At an extreme level, an adversary could even destroy the sensor itself.

Physical threats to embedded devices are not new because these devices have been deployed for many years as stand-alone devices—with the same physical security threats as they do now. What is new though are the virtual threats. As these traditionally stand-alone, isolated embedded devices are networked they become subject to new threats. For example, a remote adversary can now access the devices without having to first acquire a physical presence. However, at times, little thought is put into analyzing this new threat vector. In part, some of this is due to the fact that few embedded systems developers have been trained with regard to computer and network security. Furthermore, these developers often lack tools that support security mechanisms, thus making it hard to design in security while still maintaining tight "time-to-market" deadlines and extremely cost-conscious constraints.

Several adaptable security subsystems have been deployed in distributed systems composed of typical workstation-class computers. These systems are often deployed with excess reserve capacity and thus can afford the additional resource overheads (e.g., memory, processor) associated with adaptable run-time systems. However, embedded systems are often designed with little or no reserve capacity and thus are not able to adapt dynamically. Furthermore, many designers are not comfortable with single-purpose systems adapting at run-time because this makes it harder to analyze the systems overall performance and stability.

The requirements for close coupling with their environment and the desire to deploy cost-effective solutions dictate that building an embedded and "impregnable security fortress" just will not happen. Instead, the embedded system designer must make conscious choices as to what level of security is appropriate. Furthermore, each embedded system may have differing levels of security requirements in each of the three main security properties of confidentiality, integrity, and availability.

5.2 Motivating Examples

The following two examples will be referred to later in this dissertation. The breadth of distributed systems is simply too broad to be represented by only two examples, however, these two examples do motivate the need or requirement that

embedded systems middleware frameworks support multiple security QoS properties and levels. Both of these examples are discussed with respect to their respective requirements regarding the three classical security properties of confidentiality, integrity, and availability. These examples both illustrate that, even within a given application, multiple security QoS levels must be provided within real-world distributed embedded systems.

5.2.1 Building Automation and Control

"Smart buildings" is a broad term that encompasses a wide variety of building automation and control topics [Sno03]. Consider a large office building. It may have hundreds, if not thousands, of individual rooms and offices, on dozens of floors. The building may also have mechanical rooms for the building's various utilities (e.g., electrical power, water, heating). In order to keep this example short, only the heating and lighting within the building are discussed within this subsection.

Each office within the building is equipped with its own temperature sensor and lighting controls. The temperature sensor reports the room's current temperature as well as the desired set-point temperature. This information is used by the building's heating, ventilation and air-conditioning (HVAC) systems. The building was also designed to be energy efficient, so each room also has a motion detector that automatically turns off the room's lights when the room is determined to be unoccupied. This lighting (i.e., room occupancy) information is also distributed over the building's control network.

Although only a modest amount of information has been presented about this

building, enough information has been presented that will allow for the discussion of the confidentiality, integrity, and availability constraints within the building. It would be easy to simply decree that both the temperature and lighting data must be transmitted with high confidentiality, integrity, and availability–but this would likely incur excessive costs because rather than using low-end hardware, highly capable devices (i.e., expensive) would have to be deployed to each of the rooms within the building. A more reasonable approach is to recognize that the temperature information is likely non-confidential because, for most buildings, each room should have a common set-point. But since accurate temperature information is needed to control the HVAC system, this information should be sent reliably (e.g., low to high availability) and with a building specific level of data integrity to ensure that correct values are used within the HVAC control algorithms. If this is not done, then the building's HVAC control might be based upon spoofed values sent by a hacker or faulty values sent by a faulty (and possibly "jabbering") sensor.

The lighting information has different security characteristics than the temperature information. The ability to control a building's lighting is often desired as a cost saving measure (but in a few cases, safety or security considerations will require that lights be kept on). Thus, in most cases lighting information will generally not need to be sent with high levels of availability nor integrity. However, a room's lighting status does provide an indirect measure of whether a room is occupied. Thus, some individuals will first want to check, via the network, to see if the lights are on in someone's office before walking down the hall to visit the individual. However, some office workers might feel that it is an invasion of their

privacy if their coworkers can easily find out if the are in their office or not. If this is the case, then the lighting information should be transmitted on the network in a confidential manner. The next question to answer is at what level of protection should this data be sent. Encrypting this data such that it would be secure for dozens of years is excessive since the coworker could, within a minute, walk down the hall and determine if the worker was in or not.

Both the temperature and lighting systems send a relatively small amount of data (i.e., only one data point) in any given message. If timely response is not needed, data values could be sent in aggregated batches across the network thereby providing a tradeoff of latency and memory for bandwidth. There are, of course, embedded systems where large amounts of data need to be transmitted (e.g., a video surveillance system). But the amount of data being sent does not necessarily affect the decision on what security properties need to be meet when transmitting the data. However, the computational burden placed upon these systems will definitely increase as the size and frequency of transmitted messages increases.

5.2.2 Status Information within the Electrical Power Grid

Considered as a whole the electrical power grid is a very large distributed embedded system composed of many smaller embedded systems. A simplification of the grid is to view it as being composed of entities that either produce or consume power and entities that control power production and distribution. The timely dissemination of status information within the power grid is critical to its reliable operation [BBH+02].

Just as in our building example, status dissemination data flows within the

power grid have varying security property requirements. The frequency at which the power grid is operating is a critical input value needed to control power production. If more power is being consumed than generated or visa versa, the power grid's frequency will deviate from it's desired value (i.e., 50 or 60 Hz) and in an extreme case a blackout will occur. Thus this value must be reported with a high level of precision and integrity. However, protecting this value with high levels of confidentiality is pointless because this value can be easily measured by anyone with access to an electrical outlet. This is because, by the laws of physics, the frequency of a power grid varies uniformly throughout the complete grid, so for example the frequency measured in a home in Seattle, Washington will be identical to the frequency measured in a manufacturing plant in San Diego, California even though these two cities are more than 1200 miles apart.

Some data flows within the grid are very confidential. With deregulation of the electrical power grid, many power producers are competing to sell power to consumers. Thus, a power generation facility is at a distinct economic disadvantage if it can not protect information about its future contract negotiations, planned power plant operations, or other run-time sensor data that allows an adversary to ascertain these details. For example, some power production facility will desire to keep confidential information about which of its many power generators are in operation so as to withhold information about their equipment maintenance schedules from their competitors.

5.2.3 CORBA IDL for the Motivational Examples

The two examples presented in Section 5.2 could be implemented in a variety of ways. One possible implementation for the building example presented in Section 5.2.1 is declared via the CORBA IDL specification shown in Figure 5.1. Likewise, the IDL code shown in Figure 5.2 refers to the power grid example discussed in Section 5.2.2. In the interest of brevity, both of these IDL examples are abbreviated, but they do contain sufficient details for the purpose of this dissertation. Object methods are nested within interface and module blocks. Thus, in Figure 5.1 the code

```
boolean getLightStatus (in short roomID);
```

indicates that in order for a client to invoke a getLightStatus method on a remote object it must pass in a room identification number and the remote object will return a boolean value to indicate if the room's light is on or off. CORBA IDL does not allow for the specification of non-functional parameters. Thus, even though the two examples have specific security requirements (see Sections 5.2.1 and 5.2.2), these requirements can not be specified in CORBA IDL. Rather than being a shortcoming, this separation of interface and implementation is beneficial because it allows the designer to change a middleware implementation without having to modify the application's internal code. This means, for example, that if the encryption mechanism used to invoke the getLightStatus method is determined to be too insecure, this mechanism can be transparently replaced by simply modifying the CORBA ORB configurations instead of having to redesign and rewrite

```
module Building {
        interface Temperatures {
                boolean getTemp (in short roomID);
                void setTempSetPoint (in short roomID, in float tempSetPoint);
        };
        interface Lighting {
                boolean getLightStatus (in short roomID);
        };
};
```

Figure 5.1: Example Building Automation CORBA IDL

```
module PowerGrid {
        interface Status {
                float getFrequency ();
        };
        interface Production {
                float getMWProduction (in short generatorID);
                boolean setMWProduction (in short generatorID, in float mwProduction);
        };
};
```

Figure 5.2: Example Power Grid CORBA IDL

large portions of the building control application.

5.3 MicroQoSCORBA Security Design Philosophy

Embedded systems designers must address computer and network security as their systems are being integrated into larger and larger networks, even the Internet itself. MicroQoSCORBA has been designed with a wide variety of security mechanisms that supports a wide range of security service levels (i.e., security QoS). Our research and development on MicroQoSCORBA was shaped by the following three key design philosophies.

First, MicroQoSCORBA is a middleware framework targeted at embedded systems. This conscious choice has impacted our research and development at several levels. At the lowest level, some security mechanisms are just too computationally expensive to run on low-end embedded systems. So a key part of this research has been focused on what mechanisms we can support as well as what mechanisms make sense to support. For example, does a microwave oven need to support strong encryption and the costs associated with its additional resource usage and computational burden? Another consequence of targeting embedded systems is that, generally speaking, more a priori information is known about an embedded application at design-time, thus making it possible to leverage design-time choices rather than incorporating in costly, run-time adaptation mechanisms as might be done within a workstation-based application. For example, an application with strong confidentiality security QoS requirements could be deployed on a private network–thus ensuring confidentiality while at the same time minimizing computational effort.

Second, we realized that a full range of security QoS is appropriate and, in fact, actually needed. For example, as will be shown in Chapter 7 a task that can run in less than a millisecond on a workstation can take seconds to run on an embedded device. Depending upon the application's timeliness constraints, this will be unacceptable. In which case, designers will have to chose a less secure mechanism for a low-end device. For example, Rot13 [Rot02], a simple Caesar cipher [Sta98], might be an adequate choice to ensure confidentiality if the primary threat was from honest insiders sniffing packets on a private network.

Third, we recognized the need to maintain baseline interoperability with existing CORBA implementations, but we needed the ability to avoid strict compliance with standards that were too heavy-weight. For example, both the Fault Tolerant CORBA [Obj02d] and the CORBA Security Service [Obj02f] specifications are extremely detailed and quite broad—thus full compliance to either of these standards would have left MicroQoSCORBA unsuitable for deployment on embedded systems. Part of our research was on composing multiple QoS properties—something which has been very sparsely addressed in active research nor is it addressed in working standards (which often lag research). Composing multiple QoS properties is challenging, even in the resource-rich workstation environment, so the fact that our research is targeting embedded systems is significant. Furthermore, we are unaware of any existing middleware frameworks, suitable for embedded systems, that allow for the composition of many QoS mechanisms.

5.4 MicroQoSCORBA Security Goals

Some of our security goals are a direct consequence of our design philosophy. In particular, one of our key goals is to keep the security subsystem small. Another is to determine which mechanisms are appropriate for embedded systems, while a third goal seeks to investigate the tradeoffs between multiple QoS properties. Each of these goals will now be discussed.

5.4.1 Keep it small

As discussed previously, cost constraints generally dictate that embedded systems are deployed with minimal computational and memory resources. Thus, one of

our prime goals is to ensure that both the code size and computational burden of MicroQoSCORBA's security subsystem are small. This has been accomplished with two separate approaches. First, MicroQoSCORBA was designed with a fine degree of granularity and composability, see [MDD+03]. We also carried this same approach forward when designing and implementing the security subsystems in order to ensure that only a minimal subset of the security mechanisms would need to be incorporated into a given application in order to meet its security constraints. Secondly, we have been judicious in our choice of security mechanism to be implemented—seeking to ensure that where possible "small" mechanisms are implemented first.

5.4.2 Implement what makes sense

A vast number of security mechanisms have been developed and similarly there is no shortage of security related text books. However, many of these algorithms and metrics are simply not suitable for small embedded systems nor some embedded distributed applications. Thus part of our research has been focused on determining which mechanisms are appropriate for deployment on embedded devices within an overall embedded distributed application.

5.4.3 Investigate multi-property QoS tradeoffs

Not only were we interested in the design and deployment of a security-enabled middleware framework, we were equally interested in deploying a middleware framework that could be used as a testbed within which multi-property QoS tradeoffs could be analyzed. Although an application might be developed and deployed

with support for only one non-functional QoS property this does not mean that the application does not have multiple QoS property constraints—it simply means that the other constraints have been ignored. We believe that future embedded distributed systems will be required to explicitly support multiple non-functional QoS properties and thus our support for security within MicroQoSCORBA is a vital step in this direction. The focus of this dissertation is on the security subsystems of MicroQoSCORBA which limits our ability to present the tradeoffs that we have observed to date. For some initial evaluation results of MicroQoSCORBA's security, fault tolerance, and timeliness results and their tradeoffs please refer to [MDD+03, MHD+02].

5.5 Security Design Space

One useful breakdown of security is to consider an application's constraints with regard to confidentiality, integrity, and availability properties as well as a broad range of other properties which we have grouped under the heading of accountability. We have taken a fairly broad view of these constraints within our middleware framework. Hardware and application constraints play a significant role in the overall system design, implementation, and deployment lifecycle of an embedded distributed application. MicroQoSCORBA was designed and implemented so that it fits within the overall embedded systems design lifecycle. In particular, a designer can trade software mechanisms for hardware mechanisms that ensure the same constraints are met. The rest of this section will be devoted to these security constraints and the mechanisms that support them as shown in Table 5.1.

Table 5.1: Security Design Space

Confidentiality	**Integrity**	**Availability**	**Accountability**
Physical • Dedicated Network • Secure Network **Encryption** • Symmetric Key AES, DES, Rot13, … • Public Key RSA, Elliptic Curves, …	**Message Digests** • MD4/5 • SHA1/2 **Message Authentication Codes** • HMAC **Error Control / Correction Codes** • CRC32 **Digital Signatures** • DSA • RSA	**Service Continuity** • *See Fault Tolerance* **Disaster Recovery**	**Authentication** • Physical Tokens • Shared Secrets • Passwords • Challenge / Response **Authorization** • Access Controls • Data Protection **Audit** • Local Logs • Remote Logs **Non-Repudiation**

5.5.1 Confidentiality

The confidentiality column shown in Table 5.1 lists two headings: 'Physical' and 'Encryption'. This is because confidentiality of the data and control messages within a distributed embedded system may be achieved by both physical and logical means. For example, system designers could use separate networks for confidential information and commands. If physical mechanisms such as this are used, then MicroQoSCORBA's security mechanisms will not need to be embedded into an application.

Under the physical heading, we have listed dedicated networks as well as secure networks as two broad categories. As previously mentioned, physically isolating the embedded distributed application's data and commands provides a level of confidentiality. However, this is likely to incur a greater cost and thus the designer must strive to reach an appropriate balance between increased networking costs (e.g., isolated networks) versus increased costs due to supporting higher computational loads on the embedded system's processors (e.g., missed timing deadlines or more expensive processors). Another option is to use an existing network, but yet deploy secure network interface cards (NIC) or protocols. Accessing the network would not incur any additional costs within MicroQoSCORBA's middleware layers, but yet data confidentiality would still be ensured because of the actions of the secure NICs or networking protocol layers. Deploying secure hardware or protocols stacks can also provide for increased integrity and availability even though it is not explicitly listed in the other columns of Table 5.1.

MicroQoSCORBA uses encryption mechanisms to ensure the confidentiality

of data and commands that are sent within an embedded distributed application. Currently only symmetric key ciphers are implemented. This was a conscious choice based upon two factors. First, symmetric ciphers typically run faster than asymmetric or public key ciphers—thus requiring less computational power on each node within the distributed application. Secondly, in order to support public key ciphers a Public Key Infrastructure (PKI) would have to be implemented within the overall application. If only a few nodes are involved in the distributed application a simple PKI could be bootstrapped into the applications. However, when one considers a sensor network with thousands of identical nodes, the task of correctly implementing a PKI becomes unwieldy and beyond the scope of resource constrained embedded devices. Thus, we have chosen to postpone support for public key cryptography (both ciphers and digital signatures) until a later time.

Under the private key heading only three ciphers are listed for brevity. AES [Nat01], DES [Nat99], and Rot13 [Rot02] are shown to illustrate the embedded systems have confidentially requirements that range from very strong to very weak. In fact, Rot13 will likely only provide enough confidentiality to protect against only honest hackers. But, on the other hand Rot13 does have a very low computational overhead, thus making it potentially suitable for applications with low computational overhead margins.

5.5.2 Integrity

Data integrity is often critical within embedded systems because these systems interact with and react to their environments. For example, in the building automation and control example, given in Section 5.2.1, accurate room temperatures

are needed to accurately control the building's HVAC systems.

Error control and error correction codes are listed in Table 5.1 because they are part of the quality of service continuum level that MicroQoSCORBA supports. For example, even though a standard CRC32 code is not cryptographically strong, it does provide a comparatively inexpensive means of detecting simple bit errors.

Several cryptographically strong message digests are supported within MicroQoSCORBA in order to ensure data and command integrity. In some cases, not only must the data or command be unchanged, but the sender must be authenticated as well. In these cases, MicroQoSCORBA supports Message Authentication Codes (MAC). A MAC might be appropriate in the power grid example, given in Section 5.2.2, when a command to shutdown a power plant is given. Not only must the plant's operator know that the message has not been tampered with, but the operator must also be assured that it is coming from an entity that is authorized to shut down the plant's power production.

Digital signatures are not currently implemented within MicroQoSCORBA. In order to support digital signatures, public key encryption would also need to be supported, and as mentioned previously we have chosen to not implement public key mechanisms at this time.

5.5.3 Availability

Availability is provided via the fault tolerance mechanisms briefly discussed in Section 4.3.1. For more information on MicroQoSCORBA's support for fault tolerance please see Appendix B and [MHD+02, Dor02]. Disaster recovery is listed in Table 5.1 for completeness, but it is beyond the scope of a middleware

framework and must be addressed by the designer of the distributed application.

5.5.4 Accountability

For the purpose of this dissertation, several other security properties have been gathered together and included them under the ‘Accountability’ heading in Table 5.1. Namely, these properties are ‘Authentication,’ ‘Authorization,’ ‘Audit,’ and ‘Non-repudiation.’ As with confidentiality, integrity, and availability, some accountability aspects are not within the capabilities of small embedded systems. Likewise, some aspects of accountability may also be provided with hardware mechanisms (e.g., hardware tokens for authenticating users and other entities). Non-repudiation is not supported because it relies upon a trust architecture that relies upon a PKI which is not supported by MicroQoSCORBA. Supporting local audit logs may be problematic on some systems because of their lack of both sufficient memory as well as non-volatile storage within which to keep the logs.

CHAPTER SIX

DESIGN AND IMPLEMENTATION

This chapter presents the design and implementation of MicroQoSCORBA. First, a brief overview of MicroQoSCORBA's design and implementation is given in Section 6.1, followed by security implementation details in Section 6.2. Section 6.3 presents MicroQoSCORBA's development environment. The final section in this chapter presents MicroQoSCORBA's methodology for comparing end-to-end performance across the various testbed platforms that were used during the evaluation of MicroQoSCORBA (see Chapter 7).

6.1 Overview

The overall design and implementation of MicroQoSCORBA was guided by the refined middleware architectural taxonomy presented in Chapter 3 and the architecture given in Chapter 4. Haugan implemented the initial MicroQoSCORBA implementation along with its IDL compiler and configuration GUI in 2001 [Hau01]. Additional UDP transport layers were implemented by Damania [Dam02]. Support for fault-tolerance, MicroQoSCORBA's first supported QoS property, was added by Dorow [Dor02, DB03].

6.2 MicroQoSCORBA Security Implementation

MicroQoSCORBA was designed to support fine-grained composability of its various components. This greatly aided our efforts to design, architect, and implement the security subsystem within MicroQoSCORBA. This section will first describe

the additions made to the baseline functionality of MicroQoSCORBA and then it will proceed to describe the mechanisms implemented.

6.2.1 Extending MicroQoSCORBA

One of the original subgoals within our development efforts has been to provide an easy to use GUI that a designer could use to specify the desired hardware and software constraints for a given embedded application [Hau01]. This GUI was extended so that a system designer could specify their choice of confidentiality, integrity, and availability options to deploy in a given application.

MicroQoSCORBA has been developed in Java. However in order to support the various platforms in our testbed (see Section 7.1), Java 1, Java 2, and Java 2 Micro Edition all had to be supported. In order to provide this level of flexibility, we could have used factory and facade patterns [GHJV95], but we opted instead to implement the required platform specific details via the use of a macro processor. Using m4 allowed us to avoid the overhead associated with multiple levels of indirection that would be required to support these design patterns. Another key advantage, was that we were able to write macros that could allow us to insert or remove very small components—thus helping us achieve our goals of very fine-grained composability within MicroQoSCORBA. Further information on macro usage in MicroQoSCORBA is given in Appendix A.

We initially looked at supporting the Java Cryptography Extensions (JCE) [JCE03]. However, JCE was architected with the assumption that at run-time a cryptographic provider's JCE implementation would be loaded and used by an application. MicroQoSCORBA does not support run-time adaptability because of

the added complexity and resource usage that would be incurred. However, we were able to refactor the Open Source Cryptix Java JCE toolkit [Cry03] to run in a non-dynamic manner on both Java 1.1.8 (required for TINI) and Java 2. Because we already had developed a macro capability, we were able to extend our configuration tool and IDL compiler so that they could generate security mechanism aware macro definitions. These macros were then used in the build process to bypass what used to be multiple levels of indirection in order to drill down to the actual algorithm implementations. Thus, with a few minor changes, we were able to reuse an existing and tested code-base without having to incur the penalties due to over-generalization associated with JCE.

6.2.2 Implemented Mechanisms

The Advanced Encryption Standard (AES) [Nat01] is supported within our framework as well as other ciphers and mechanisms. AES was chosen for implementation within MicroQoSCORBA because it is a federal standard that was selected for its suitability for resource constrained devices as well as its cryptographic strength [Bur03]. A complete list of implemented mechanisms is listed in Table 6.1. Null mechanisms were implemented in order to evaluate the overheads associated with encryption, message digests, and message authentication codes.

All of the encryption ciphers listed in Table 6.1 are secret key ciphers because, as explained in Section 5.5, MicroQoSCORBA does not currently support public key encryption mechanisms. The supported ciphers range from weak to strong. For example, the Caesar cipher as well as a simple XOR encryption scheme both provide very little confidentiality, but they are included within MicroQoSCORBA

Table 6.1: Implemented Security Mechanisms

Secret Key Encryption Ciphers	Null, AES, Caesar, CAST5, DES, Triple DES, IDEA, MARS, RC2, RC4, Serpent, SKIPJACK, Square, Twofish, XOR
Message Digests	Null, CRC32, MD2, MD4, MD5, Parity, RIPEMD128, RIPEMD160, SHA0, SHA1, SHA256, SHA384, SHA512, Tiger
Message Authentication Codes	Null, HMAC (based upon supported message digests)

so that they may be used on hardware platforms that do not have enough computational resources (or long enough time deadlines) to support a stronger cipher. When MicroQoSCORBA's fixed-length message size option is selected, the complete message is encrypted (including the GIOP message headers), otherwise the GIOP message header are sent in plain-text because the message length must be read by the server's ORB.

A few non-cryptographically secure message digests were implemented for MicroQoSCORBA. They are a 32-bit CRC code as well as a parity-byte checksum. In addition to these message digests, many other message digests were refactored from the Cryptix JCE toolkit for use within MicroQoSCORBA. MicroQoSCORBA also supports message authentication codes (MAC) based upon the HMAC algorithm. Message digests and MACs are computed on the complete GIOP message (headers and data) and then they are appended to the message before being sent.

6.3 Development Environment

We have developed several tools, which aid the embedded systems developer in the design and configuration of a wide spectrum of architectural configuration choices. Our MicroQoSCORBA specific development tools currently include a GUI-based configuration tool and our IDL compiler. Additionally, our development environment also includes the use of a macro processor, m4, the make utility, several java development tools, and device specific tools that are required to target specific embedded hardware devices. We present our configuration tool, our IDL compiler, followed by the remaining steps required to build and deploy an application.

6.3.1 MicroQoSCOBRA Configuration Tool

The configuration tool's main purpose is to let the developer determine the base architecture on which to build the application by choosing from the constraints outlined in Section 4.1. Once selected, these constraints are stored in an application specific configuration file. The IDL compiler and other MicroQoSCORBA tools and its GUI configuration tool use these values to customize each application. We have divided our discussion of these constraints into different sections: Data types (IDL subsetting), Communication and Protocols, and Miscellaneous options. For more information on this tool, refer to [Hau01].

Data types

Many small, dedicated systems do not need support for all the different data types that a standard IDL mapping encompasses. We thus support only a subset of

the data types by selecting which specific types to use in our application. For example, the room temperature sensor in our office building example will likely have less than 16 bit of precision (2 bytes), so there will be no need to support double values (8 bytes of precision). Reducing the number of data types yields a smaller and simpler marshalling and un-marshalling library, and thus reduced code size and memory footprint. By removing support for user and/or system exceptions, we can generate smaller and simpler code for stubs and skeletons.

Communication and protocols

To allow for flexibility in the client/server application we need to be able to specify what communication layer and protocols to support. Embedded systems are developed on many different platforms, and so each application needs to be able to adapt to these different environments. We have chosen to consider the client and server separately. The relationship between a client and a server is usually many-to-one. A minimal client only needs to be able to use one specific transport and one specific protocol. This means that the client may be simple and small. The server however, often must have the ability to communicate with several different clients on different systems. The server side can, therefore, be configured to support more than one transport layer and protocol type.

Miscellaneous Other Configuration Constraints

An embedded systems developer generally knows which hardware a given system is going to be deployed to. MicroQoSCORBA can utilize this information

by using these hardware restrictions to bind and constrain various software implementation choices. This will reduce the code size and simplify the generated stub and skeleton code. The developer may set a number of constraints in the GUI in order to accurately reflect the hardware bindings. One of these choices is the use of homogeneous hardware. Once this GUI switch has been set, the GUI instructs the IDL compiler to generate simplified marshalling and unmarshalling code. Endian-ness can also be forced if needed. Other optimizations that are available are restriction of message payload, the number of interfaces, and the number of methods.

These restrictions can help to reduce the complexity of the client and server communication protocols and reduce the application's data flow. The application developer also has the option to compress interface names and method names into a fixed, n-bit integer. This will reduce the size and the overhead of a message. The final configuration option available is the choice of whether to use inline marshalling/unmarshalling or to use a library. Depending on the number of interfaces, methods, and parameters to the methods, choosing to use inline code instead of a library might increase the speed and reduce the code size of the application.

The developer can also check the configuration for errors or impossible combinations. For example, GIOPLite [SOO00] and heterogeneous hardware cannot be used together, since GIOPLite can only run on homogeneous hardware.

6.3.2 MicroQoSCOBRA IDL Code Generator

The IDL code generator reads the IDL definition file and generates files to ease implementation of the distributed system. The first task for the code generator is to

check the IDL definition for compliance with the IDL subsetting options selected by the designer via the GUI CASE tool. If any irregularities are detected, no code will be generated and an appropriate error message will be reported. Based upon the configuration selected by the developer, the code generation tool will generate application and device specific stubs and skeletons that reflect the choices already made by the developer. During code generation, the tool makes decisions regarding what transports and protocols to use, what type of marshalling and unmarshalling code to produce, and other choices that affect the configuration of the application code.

MicroQoSCORBA's IDL compiler is not limited to just stub and skeleton code generation. The compiler also autogenerates a 'Makefile' that influences the overall application build process. Depending upon the design-time choices configured in the configuration file, additional logic is added to each Makefile so that the appropriate MicroQoSCORBA library routines are either included or excluded from a given application. This allows for a fine-grained control over the included library code or in other words it allows for the removal of unnecessary library methods. The IDL compiler also generates an application-specific set of macro definitions (see Appendix A) that are used to customize MicroQoSCORBA's functionality. The purpose of these macros will be explained in the following subsection.

6.3.3 Building and Deploying the Application

Once the stubs and skeleton code are generated, the application must be compiled, built, and deployed. Although Java is intended to be truly cross platform

(i.e., “write once, run anywhere”) several differences exist between the JVMs supported by our testbed platforms, which require support for Java 2 Standard Edition (J2SE), Java 2 Micro Edition (J2ME), and Java 1 (JDK1.1.8). Most of these differences are relatively minor and easily solved with a macro-preprocessor approach. But, MicroQoSCORBA’s use of macros is not limited to just simply compensating for different Java versions—using macros has the added benefit of allowing for fine-grained configurability and optimizations. For example, the security mechanisms are wrapped inside of macro definitions. When the security macros are enabled, security functionality is included into the Java source files without the overhead of additional security-aware Java classes, nor the software maintenance burden of having two nearly identical modules, one security aware and other other unaware.

The autogenerated ‘Makefile’ contains targets for generating both the “client” and “server” applications for a given IDL based application. These targets include the necessary make logic to include only those Java class files that are needed for a given application, thus helping to limit the size of a MicroQoSCORBA application executable. The Makefile also has a “stats” target that computes various statistics about the compiled code such as the combined size of all of its classes.

Once the java code has been compiled it must be converted and downloaded into the specific hardware device. For a desktop machine this step is not needed. But this is a required step for our two test bed embedded systems platforms. For the TINI board [Loo01], first the java class files must be converted with TINIconvertor into a single file that then must be downloaded to the TINI board. For the

SaJe board [Sys03], the class files are processed by the JEMBuilder application and then downloaded to the SaJe board with the Charade application.

6.3.4 Automated testing

MicroQoSCORBA's fine-grained configurability allows for the creation of literally hundreds of individual middleware applications from the same baseline IDL definitions and client and server code. Expect [Lib94] scripts were created for each of the evaluation platforms that automated the running of multiple configurations.

6.4 Cross Platform Comparison Methodology

Fairly comparing middleware frameworks is problematic because each framework is designed and developed with different goals and priorities, and therefore each has a different set of strengths and weaknesses. In a similar manner, fairly comparing the same framework across a range of embedded hardware devices is also problematic because embedded systems have such a wide range of processing power, CPU instruction sets and native word lengths, operating systems (or lack thereof), network support, etc. A methodology was developed in order to help compare the best case performance of several ORBs running on the same hardware platform as well as well as comparing ORB performance across a range of embedded systems.

This methodology consisted of filtering out best-case performance results platform and network induced performance impacts. An algorithm was developed which separated the best-case, baseline performance from the "noise" associated

with OS, network, and Java implementation details of each platform. First, the an initial set of invocations are issued in order to get the testbed into a steady state condition. Then raw, unprocessed timing histograms are gathered for each performance run. Several of these detailed timing histograms are presented in Section 6.4.2 in order to help motivate the event filtering algorithm that is presented in Section 6.4.3.

6.4.1 Steady State

Tests were conducted on the three platforms in order to determine an approriate number of `foo.bar(...)` invocations to issue in order to enter a steady state condition. Three separate loops are run and the results of each are logged for further review, if needed. On Linux, three loops of 15,000 iterations are run for a total of 45,000 iterations. This large number of iterations is needed in order to trigger the Java HotSpot JVM just-in-time compiler optimizations and to overcome TCP's slow-start mechanisms. On the SaJe platform, three loops of 100 iterations (300 total) is sufficient to enter a steady-state condition. On TINI, three loops of 10 iterations (30 total) are run during the steady state portion of the performance tests. The low number of iterations on both the SaJe and TINI testbed platforms is partly because these two boards are more CPU-bound than network-bound while making an invocation, and thus do not need to overcome the TCP slow-start mechanisms to the same degree as on the Linux platform, which is network-bound.

6.4.2 Raw Timing Performance Results

For each of the three testbed platforms, a baseline client program was created that made multiple foo.bar(...) invocations. After each invocation, an elapsed time was computed and stored so that it could be later displayed in a histogram. Unfortunately, Java's standard timer has at best a 1-ms resolution, so on Linux Java's Native Interface (JNI) was used to access a microsecond resolution timer. On SaJe, a platform specific call was used to get a microsecond timing resolution and on TINI the timing resolution was increased from 10 ms to 1 ms via the use of a TINI specific timing call. Raw performance results for the baseline Micro-QoSCORBA configuration are presented in the histograms of Figures 6.1, 6.2, 6.3, 6.4, and 6.5. A family of lines is shown in each of these Figures, with each line corresponding to the results gathered for a specific number of foo.bar(...) invocations (e.g., 0.01, 0.1, 1, and 10 million invocations on Linux).

The Linux workstations had the most computational power of our three testbed platforms. This computational power benefited the overall performance results as well as the performance of the Java garbage collector. Four timing runs, consisting of 0.01, 0.1, 1, and 10 million invocations, were performed on the Linux testbed workstations. These results are displayed with both microsecond and millisecond resolution in Figures 6.1 and 6.2, respectively. These histograms illustrate that although the vast majority of invocations are completed in less than 1-ms (see Figure 6.2). The overall, average elapsed time was 0.161 milliseconds (see Figure 6.1). Approximately 5% of the baseline Linux invocations took a "long" time to complete (e.g., 0.6–0.7 ms, 6–11 ms, and even 20 ms). One should also note,

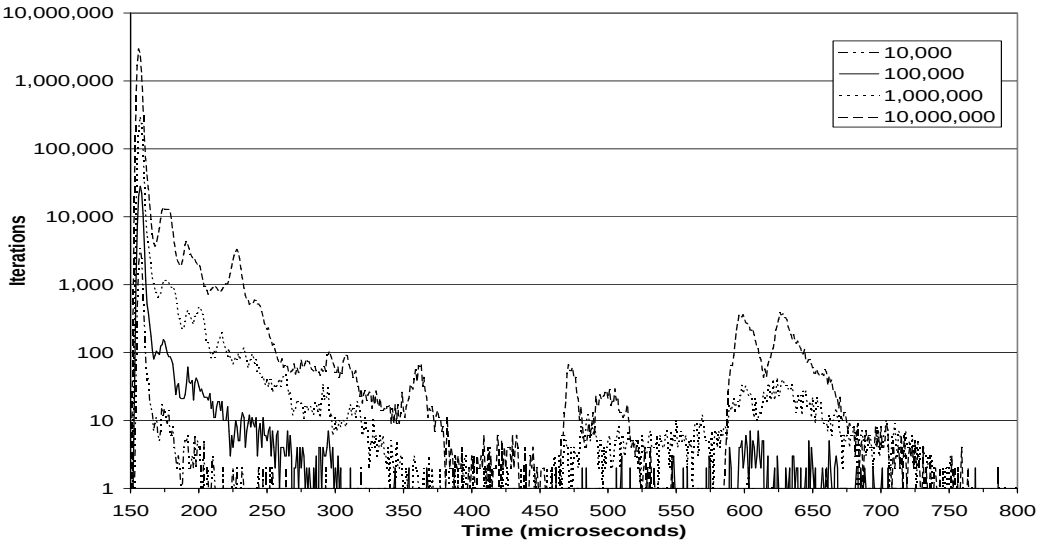

Figure 6.1: Linux Timing Performance Histogram with Microsecond Resolution

that the shape of the four sets of results, shown in the Linux histograms, scales uniformly with the total invocation count for the events of interest (i.e., those invocations that complete in 0.161 ms), but the shape is not uniform for those invocations with long elapsed times—showing that Java garbage collection performance and other OS and network specific slowdowns are adversely affecting MicroQoSCORBA's timing performance.

MicroQoSCORBA's performance for the baseline configuration running on the SaJe platform is shown in Figures 6.3 and 6.4 with microsecond and millisecond timing resolutions, respectively. Five curves, with event counts ranging from 650 to 160,000 invocations, are shown in the histograms of both of these figures. The shapes of all five of these curves match well for the initial peak located at 3.77

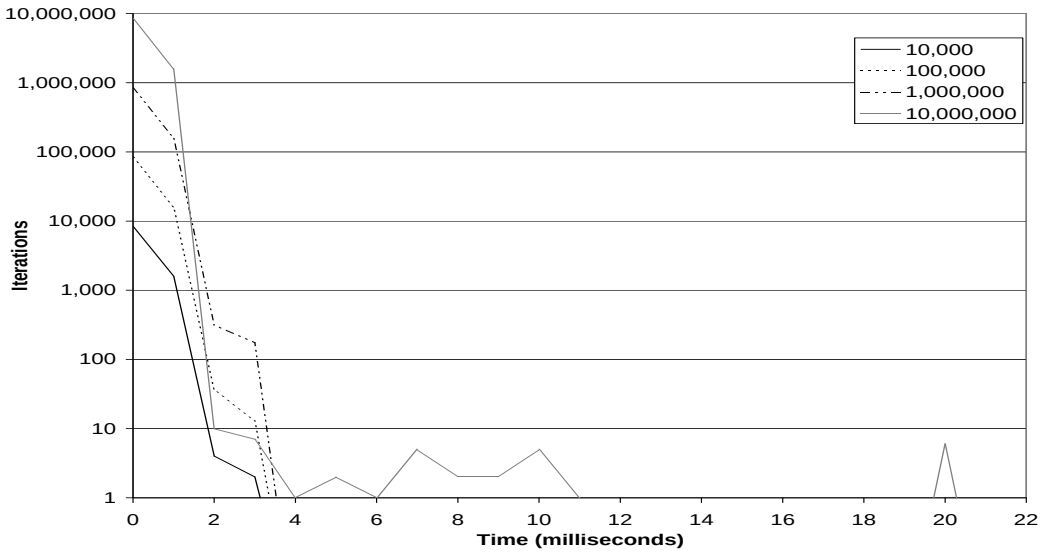

Figure 6.2: Linux Timing Performance Histogram with Millisecond Resolution

ms (see Figure 6.3). But, the shape of the curve varies for the invocations with longer elapsed times. The data shown in Figure 6.4 illustrates the fact that Java garbage collection on the SaJe platform is very time consuming—adding over 270 ms to the time of a MicroQoSCORBA invocation that requires the invocation of the SaJe Java garbage collector. When these "slow," garbage-collected invocations are included in the computation of an average invocation time, they cause the SaJe performance values to increase from a best-case average of 3.77 ms to an overall average of 4.11 ms.

The TINI platform, with only 512 Kbytes of memory and a 40 MHz 8-bit processor is the most computationally constrained platform of the three testbed platforms. Five TINI tests were conducted with 100, 200, 400, 800, 1600, and

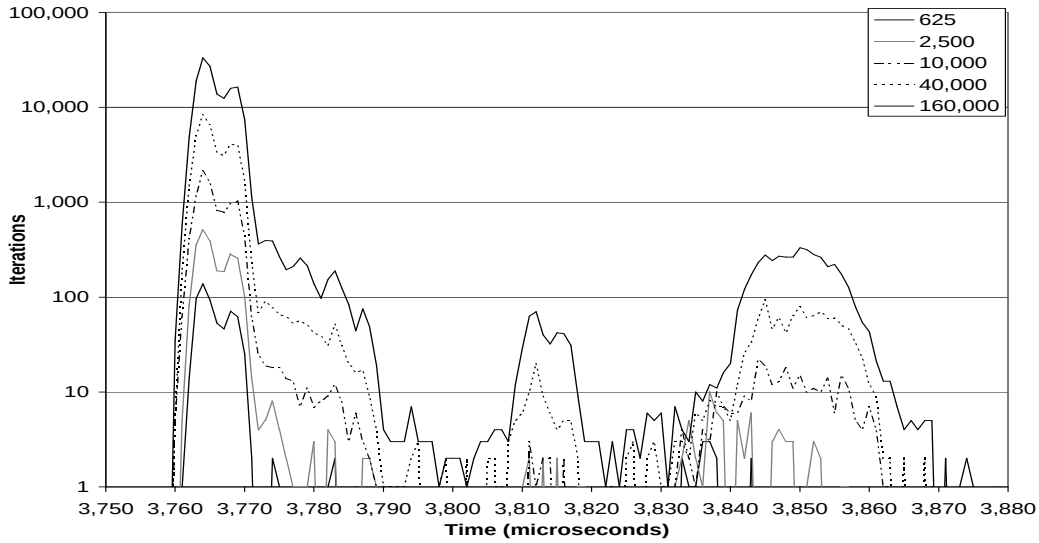

Figure 6.3: SaJe Timing Performance Histogram with Microsecond Resolution

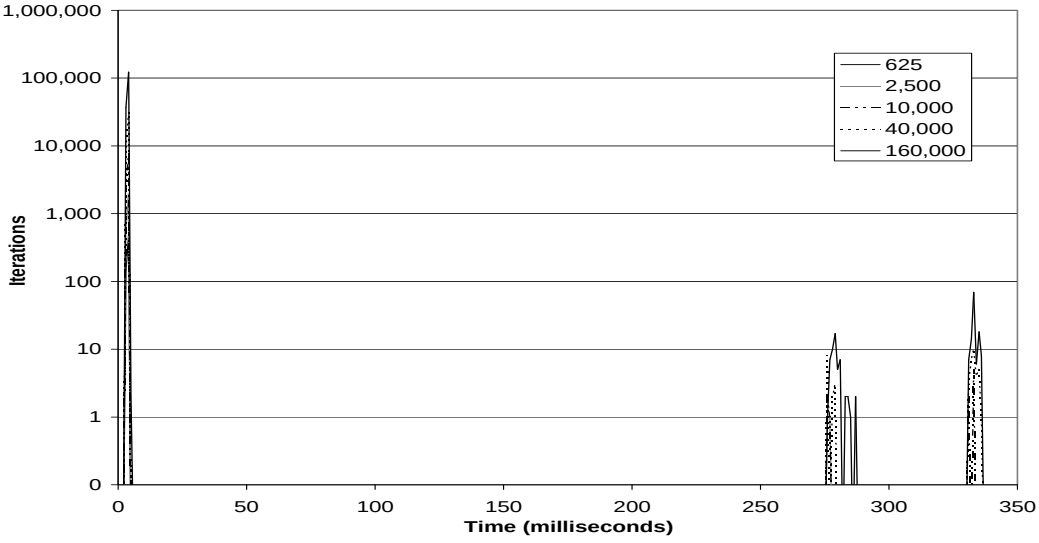

Figure 6.4: SaJe Timing Performance Histogram with Millisecond Resolution

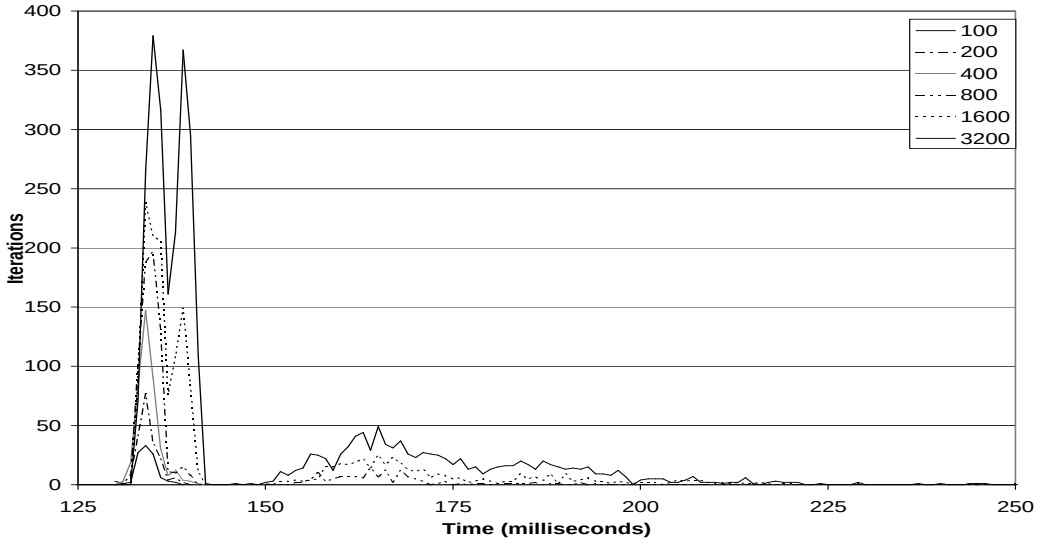

Figure 6.5: TINI Timing Performance Histogram

3200 invocations, the results of which are shown in Figure 6.5. The histogram shown in the aforementioned figure shows that the majority of the invocation calls complete with an average time of 134 ms. However, as the invocation counts increase (which causes the garbage collector to run more often) a significant number of events occur in the 150–225 ms range. This data shows that on the TINI platform over 25% of the invocations were impacted by garbage collection and other system "noise" whereas on the Linux and SaJe platforms only 5% of their invocations were similarly impacted. For small invocation counts an average time of 134 ms was recorded, but when a large number of invocations were issued the impact of TINI's java garbage collector caused this value to grow to an overall average of 151 ms—an increase of 13%.

Memory management and other platform specific (e.g., OS and network implementations) considerations have a significant impact on overall system performance. For example, on Linux and SaJe 5% of the invocations were impacted, whereas on TINI (on runs with high invocation counts) over 25% of the invocations were adversely impacted. On the TINI platform the garbage collector runs, on average, in less time than a single invocation, but on the Linux and SaJe platforms the Java garbage and other systems noise can add well over an order of magnitude of difference between a "fast" and "slow" invocation. A system designer must be aware of the best-case and worse-case performance results when designing an application. In order to compare the best-case system performance, the timing "noise" associated with each testbed platform's performance must be filtered out. The methodology for performing this filtering will now be discussed.

6.4.3 Event Filtering

The SaJe and TINI timing results that were previously presented largely motivated the need for filtering "noisy" or "slow" invocation events from the timing performance calculations. However, MicroQoSCORBA's event filtering methodology proved to also be useful on the Linux testbed platforms. For example, a few performance tests were run on the Linux workstations while other CPU-intensive tasks executed in the background. In these tests, the shape of the timing histograms varied significantly from those shown in Figure 6.1 (many more "slow" events were recorded), but the filtered, best-case timing values remained unchanged from those computed when the MicroQoSCORBA client and server tasks were the only applications running the testbed Linux machines. By appropriately filtering events,

consistent timing results were obtained on all three platforms across a very wide range of invocation counts.

MicroQoSCORBA's event filtering process proceeds in three steps. First, a sufficiently large number of method invocations are made in order to load the CPU caches and to reach a steady state network condition. Detailed timing data is not gathered during this initial stage of a timing run. During the second stage, the elapsed timer for each `foo.bar(...)` invocation is gathered and histogrammed. This data is then analyzed, as will be shortly explained. In the third stage, the results from multiple runs are compared and the lowest value is then reported.

The analysis and filtering of the performance histogram data is conducted via an iterative method. The high-level details are as follows. First, an average and its standard deviation are computed from the overall timing histogram. Then the average and standard deviation values are used to compute an upper limit threshold value. During the next iteration, only data points between the minimum value and the upper limit threshold value are used to compute a new average and standard deviation, which in turn are used to compute a new upper limit threshold value. This process continues until such time as the new upper limit value equals the old upper limit value. During each iteration, the upper limit threshold is computed to be the average plus 3.5 times the current standard deviation value. A value of 3.5 is used because, given the assumption that the events are normally distributed, this value will filter out less than 0.02% of the desired (e.g., non-garbage collected) events. This approach is able to correctly identify the upper limit threshold for the best-case timing peak within just a few iterations. Event filtering was conducted

after the completion of each timing loop run so that performance overheads associated with each invocation could be kept to a minimum during the execution of the main timing event loop. For further event filtering details please refer to Appendix A.2.

The number of separate histograms computed and compared is easily adjusted by the user. This value was set to three for the results reported in this dissertation. Thus, each of the results reported in this dissertation is the lowest value recorded during three separate, back-to-back timing runs. The values computed by filtering and analyzing the three separate timing histograms were typcially consistent. Occasionally (especially on TINI), the reported best-case timing value for one timing run would be significantly higher than the others due to OS, network traffic, or Java garbage collection impacts—thereby proving the usefulness of choosing the best of the three timing loop run values.

6.5 Implementation Status

The initial analysis, design, and implementation of MicroQoSCORBA were completed in 2001 [Hau01]. Since that time the taxonomy behind and architecture of MicroQoSCORBA have been continually refined and improved. MicroQoSCORBA has been developed in Java with support for both the Java 2 Standard and Micro Editions (J2SE and J2ME). This has allowed us to deploy MicroQoSCORBA to several hardware platforms (e.g., standard PCs, the TINI platform, and SaJe based systems). Furthermore, we have maintained interoperability between each of these diverse hardware platforms and other standard CORBA ORBs

(e.g., JacORB, TAO). MicroQoSCORBA currently supports: streamlined communication between homogeneous hardware; IOR and corbaloc object references; customized client and server ORB libraries; GIOP, IIOP, IIOP-lite, and a more streamlined IOP called MQCIOP, see [Hau01]; TCP and UDP networking layers; reliable UDP transmissions via custom Go-Back-N and Selective-Repeat wrappers over UDP; multiple IDL modules and interfaces; inclusion or exclusion (i.e., subsets) of simple CORBA data types (e.g., boolean, char, double); inclusion or exclusion of CORBA system and user exceptions; and optimizations based upon fixed length messages or IDL parameter types.

Furthermore, we have designed and implemented a fault tolerance subsystem [Dor02, DB03] and a security subsystem [MBS, McK03b]. The fault tolerance subsystem supports temporal, spatial, and value redundancies. This fault tolerance subsystem also supports failure detection and a wide variety of group communication and message ordering mechanisms. The security subsystem supports over a dozen symmetric key ciphers and over a dozen message digests and HMAC message authentication codes.

CHAPTER SEVEN

EXPERIMENTAL EVALUATION

An experimental evaluation was conducted in order to evaluate MicroQoSCORBA's resource usage and end-to-end latency performance on three separate hardware platforms that spanned a wide range of computational power. The results of these experiments will be presented, but first the testbed hardware and software tools will be presented as well as the testbed application in Sections 7.1 and 7.2, respectively. After that, a baseline MicroQoSCORBA configuration will be compared with other CORBA ORB implementations in Section 7.3. This chapter will then conclude with an evaluation of MicroQoSCORBA's security mechanisms in Section 7.5.

7.1 Testbed Hardware and Software Tools

Three hardware platforms were selected for MicroQoSCORBA's testbed setup. They are Linux on a desktop PC, Systronix's SaJe boards [SaJ03], and Dallas Semiconductors TINI board [TIN03a, Loo01]. Each of these platforms will now be described.

Linux. Two 1.5 GHz Pentium 4 desktop computers running Red Had Linux, version 7.2, were selected for the testbed. The PCs were connected via a 100-Mbps network. The software development environment used to compile and build the Java executables used for the evaluation was Sun's Java 2 Software Development Kit, version 1.4.1_03 [Jav03].

SaJe. Two SaJe boards from Systronix [Sys03] with 100 MHz aJile Systems aJ-100 CPUs were used in the testbed environment. The aJile processor executes Java byte-code natively in hardware, thus no separate JVM is required. These boards only support 10 Mbps network links. The SaJe boards support the Micro Edition of Java 2, rather than the Standard Edition. The most noticeable difference being that J2ME CLDC connection oriented classes were used for the networking rather than java.net.Socket classes. After the application class files are compiled, these classes files are converted into a format suitable for the SaJe board via the use of JEMBuilder, version 3.1.6, conversion tool. Then the final executable image is downloaded into the boards with the Charade tool.

TINI. The TINI boards are powered by a 40 MHz DS80C390 CPU. Like the SaJe boards, the TINI boards only support 10 Mbps networks. Unlike the Linux and SaJe platforms, TINI's limited JVM does not support Java 2, but only supports Java version 1.1.8. TINIConvertor, version 1.02e, was used to convert the standard Java class files into a compressed format suitable for download and execution on the TINI boards.

7.2 Testbed Application

Our experimental evaluation was based upon executing the foo.bar(...) method given in the IDL specification shown in Figure 7.1. Client applications were implemented that made repeated foo.bar(...) invocations to a server application.

The purpose of the evaluation was to quantify MicroQoSCORBA's resource usage and performance. Evaluating a complex application (e.g., the building example of Section 5.2.1) is feasible in MicroQoSCORBA, but doing so would require one to factor out the increased resource usage associated with the application's increased complexity when determining the performance and resource usage of MicroQoSCORBA's ORB. Thus, the decision was made to implement and test a simple testbed application even though it did not reflect the complexity typical of some real-world applications.

As explained in Section 6.3, MicroQoSCORBA's development environment supports the use of macro preprocessing Java files before they are compiled. This feature was beneficial during the evaluation because it allowed for the implementation of one macro-enabled client and one server application source code file (see Appendix A). These two files had embedded in them macro definitions that are enabled or disabled as needed in order to enable application specific QoS initialization features—thereby avoiding the need to create multiple QoS specific client and server source code files. For example, the syntax of making a `foo.bar(...)` call remains unchanged whether or not encryption is enabled, but when encryption is enabled the encryption key must be initialized by the client and server applications. This initialization is performed by code embedded inside of a macro, thus allowing the code to be included or excluded depending upon application specific macro definitions generated by MicroQoSCORBA's IDL compiler.

```
module timing {
      interface foo {
            long bar (in long arg1);
      };
};
```

Figure 7.1: Testbed Application IDL

Testbed Application Setup

The startup phase for test run consisted of 4500, 300, and 30 invocations for the Linux, SaJe, and TINI platforms, respectively. The high number of Linux invocation during the startup phase was required in order to ensure that the J2SE HotSpot compiler would have sufficient time to optimize the Java byte code corresponding to a given test configuration.

Three separate timing loops were conducted during the course of the testbed application. After each loop, the histograms were analyzed and their results saved for a comparison at the end of the programs' execution. On the Linux platform, 100,000 invocations were issued during each of the three loops, and on SaJe and TINI the values were 2500 and 200, respectively. As will be shown, some TINI configurations run very slowly (e.g., over 60 seconds per invocation), so a timeout value set at 15 minutes per loop was used to ensure that that testing could complete in a timely manner. On Linux and SaJe the timeout values were set at 10 minutes. Even with the timeout values in place, the performance value reported for each configuration was based upon enough events so as to provide a high level of accuracy.

Many of the security mechanisms implemented within MicroQoSCORBA are computationally complex. Thus, the packet lengths of the MicroQoSCORBA invocations were varied so as to illustrate the impact of the additional computation load associated with larger packet sizes. The packet lengths were set via a MicroQoSCORBA configuration option that allows the designer to specify whether packets are sent with a dynamically computed, minimum-length size, or whether the packets are sent with a hard-coded, fixed-length size. Because the testbed application only transmits one long value, the minimum-length packet is 56 bytes long. Packet lengths of 512 and 1024 bytes messages were also built by enabling fixed-length packet option and setting its value to 512 and 1024 bytes, respectively.

7.3 ORB Evaluation

MicroQoSCORBA has successfully interoperated with other ORB implementations (e.g., JacORB [Jac03] and TAO [TAO03]). This section will compare MicroQoSCORBA's resource usage and performance values with both JacORB and TAO. JacORB, like MicroQoSCORBA is implemented in Java. TAO is a C++ based ORB that has been designed and implemented to be suitable for usage in real-time applications [SLM98]. First a memory usage comparison will be presented, followed by a run-time performance comparison.

The various ORB metrics shown in Table 7.1 were gathered on the Linux workstations for MicroQoSCORBA JacORB (version 1.4.1), and TAO (version 1.3). The ORB comparison numbers are limited to Linux because JacORB is too

Table 7.1: ORB Size Comparisons

Size Metrics	**MicroQoSCORBA**		**JacORB**		**TAO**	
	Client	**Server**	**Client**	**Server**	**Client**	**Server**
Application	4,222 B	2,476 B	6,591 B	6,363 B	33,422 B	66,635 B
Java Memory	153.6 KB	160.6 KB	223.0 KB	243.1 KB	n/a	n/a
Linux RSS	9.95 MB	9.62 MB	13.31 MB	11.46 MB	4.68 MB	5.96 MB

large to run on either TINI or SaJe and TAO, being a C++ ORB, can not execute on either the TINI or SaJe platforms. The 'Application' size is based upon the size of the Java class files associated with the testbed application for MicroQoSCORBA and TINI and upon the unix 'size' command for TAO client and server executables. The 'Java Memory' row lists the amount of memory consumed on the Java heap for both MicroQoSCORBA and JacORB during the execution of the testbed application. This value is not applicable for TAO. Also during execution, the amount of physical memory (i.e., 'Linux RSS' value) consumed by each ORB's client and server applications is reported. It is significant to note that MicroQoSCORBA outperforms JacORB in all three categories. The TAO applications consume the least amount of physical memory because it does not use a Java VM.

Timing latency values were gathered for all three ORB as reported in Table 7.2. Latency values were also measured for MicroQoSCORBA on the SaJe and TINI platforms. For each testbed platform two values are reported. The 'Filtered' value is the best-case latency value reported by the event filtering algorithm described

Table 7.2: ORB Latency Comparison (ms)

ORB	Linux		SaJe		TINI	
	Filtered	**Unfiltrd**	**Filtered**	**Unfiltrd**	**Filtered**	**Unfiltrd**
MicroQoSCORBA	0.170	0.171	4.162	4.425	248.6	256.8
JacORB	0.329	0.604	n/a	n/a	n/a	n/a
TAO	0.330	0.332	n/a	n/a	n/a	n/a

in Section 6.4.3. The 'Unfiltered' value is the latency computed from all invocations. The best-case MicroQoSCORBA latency values are 0.170 ms, 4.162 ms, and 248.6 ms for the Linux, SaJe, and TINI platforms, respectively. On the Linux platform, the filtered JacORB and TAO numbers were virtually tied at 0.329 ms and 0.330 ms, respectively. But the overall, unfiltered latency value of 0.604 ms for JacORB almost doubled its filtered value while MicroQoSCORBA's and TAO's unfiltered values increased only slightly.

In summary, MicroQoSCORBA has a smaller memory footprint that JacORB and MicroQoSCORBA end-to-end latency values are almost twice as fast as either TAO's or JacORB's. This can be explained by the fact that MicroQoSCORBA is configurable with a fine degree of granularity and it supports, by design, a small fraction of either TAO's or JacORB's functionality. Therefore it has a smaller footprint and less message-oriented overhead.

7.4 MicroQoSCORBA Comparisons

The fine-grained configurability of MicroQoSCORBA allows for the creation of dozens of similar client and server applications that are all based upon the same

client and server application source code. The changes and optimization between each application are controlled by MicroQoSCORBA's IDL compiler which keeps track of each application's specified functional and non-functional constraints. The purpose of this section is to highlight the various resource constraints and performance tradeoff associated with various baseline and multi-property QoS configurations. First, the evaluated configuration will be presented, followed by their application sizes, after which their respective latencies will be presented.

7.4.1 Evaluated MicroQoSCORBA Configurations

Several versions of the testbed application were generated. First, baseline client and server programs were generated for each of our three testbed platforms (Linux, SaJe, TINI). After that, various QoS and multi-property QoS enabled client and server programs were also built and run on our testbed environment.

Both fault tolerance and security properties were evaluated. The fault tolerance mechanisms evaluated were value redundancy and temporal redundancy (see Appendix B). The temporal redundancy settings tested were 2 and 4 temporally redundant retransmissions. A range of confidentiality (see Section 4.3.2) and integrity mechanisms (see Section 4.3.2) were tested. For value redundancy and integrity a Parity-byte, a CRC32 code, and the message digests, MD5 and SHA1 were evaluated. For confidentiality, a Caesar cipher [Sta98], DES, TripleDES, and the Advanced Encryption Standard (AES) [Nat01] were evaluated in electronic code book mode. Each security option and value redundancy was also tested with temporal redundancy turned on, thereby providing a multi-property QoS evaluation. MicroQoSCORBA was configured to send messages via IIOP (CORBA

GIOP version 1.2 over TCP/IP).

7.4.2 Application Size and Memory Usage

An application's size and memory usage depends, in part, upon the size of the application's Java class files that must be loaded into memory, the amount of data that must be allocated on the heap during run-time, and other system-specific and run-time library code that must be loaded and executed on behalf of the application. Measuring the size of the Java class files for the testbed application is a straightforward task. These results (in bytes) are reported in Table 7.3. For each platform in our testbed, both the client and server class file sizes are reported. The first column in Table 7.3, lists the configuration options, namely the baseline option, baseline with temporal redundancy, and so forth. The class file sizes for the AES cipher did not vary depending upon whether a 128-, 192-, or 256-bit key length was used, so only one AES row is shown in Table 7.3. For all three platforms, adding temporal redundancy to a given application configuration (e.g., baseline, Caesar cipher) caused its class file size to decrease due to the fact that the temporal redundancy configurations use fixed length packets (instead of variable length packets), which require less code to marshal and demarshal. Only the temporal redundancy values for the baseline configuration are reported because the decreases for the other configurations decrease by a similar size.

One can see from the results shown in Table 7.3, that MicroQoSCORBA was able to produce small applications for both the Linux and TINI platforms. The reported TINI values are smaller than the reported Linux values, because the TINI JVM executes a compressed Java byte code file. The values for the SaJe board

Table 7.3: MicroQoSCORBA Java Class File Size (bytes)

Security Mechanism	Linux		SaJe		TINI	
	Client	Server	Client	Server	Client	Server
Baseline	63,607	61,062	259,077	254,246	23,867	20,764
Baseline w/Temp.Red.	59,478	58,617	258,437	252,819	22,506	19,928
Parity8	68,302	65,888	262,304	257,520	25,531	22,478
CRC32	68,687	66,275	262,506	257,726	25,726	22,675
MD5	73,547	71,137	265,871	261,084	28,845	25,796
SHA1	75,416	73,005	267,875	263,088	30,573	27,523
Caesar	74,635	72,088	263,436	258,585	27,481	24,388
DES	80,055	77,508	266,726	261,875	30,993	27,900
TripleDES	81,525	78,977	267,508	262,664	31,611	28,517
AES	85,182	82,634	270,606	265,762	35,156	32,062

appear to be high, but the SaJe Java byte-code image is completely self-contained. In contrast, the TINI board requires 448 Kb for its JVM and runtime environment [Loo01] and the Linux JVM must link in multi-Mb sized shared libraries in order to provide its runtime environment.

The class file size results also show that, with regard to class file size, the two fault tolerant mechanisms of value redundancy (i.e., Parity and CRC32 configurations) and temporal redundancy can be configured in with a relatively small extra cost. The MD5 and SHA1 message digests as well as the four cipher were larger in terms of Java class file sizes, with the AES cipher being the largest on each platform—adding over 20 Kb of code on the Linux platform.

7.4.3 Latency Results

A number of repeated foo.bar(...) invocations were made from the testbed client implementation to the servant implementation in order to determine an accurate round trip latency of a single foo.bar(...) invocation. On Linux, three runs of 100,000 invocations each were run and on SaJe and TINI the values were 2500 and 200, respectively. The SaJe and TINI invocation counts were significantly lower due to the fact that their CPUs are significantly slower than the Linux CPUs. The client and servant applications communicated over a 10/100 Mbit switched network. Our evaluation results are summarized in Table 7.4. Each row in the table represents a different QoS property, and the 'No TR', 'TR2', and 'TR4' columns report the temporal redundancy values.

The 1.5 GHz Linux machines had the lowest latency results, which was not surprising because they had the fastest CPUs. The TINI platform was the slowest of the three and several MicroQoSCORBA configurations requires seconds instead of milliseconds to send a message across the network. Perhaps the most interesting results that can be gathered from this data are the relative performance comparisons between the three testbed platforms. For example, the Linux non-fault tolerant TripleDES timing value was 1.40 times higher than its baseline performance, whereas the SaJe value was 2.70× higher and the TINI result was 12.75× higher. When the values from the 'TR4' column are compared to the same baseline values, the performance ratios are 1.75×, 7.20×, and 30.75×, respectively. The other values in Table 7.4 also highlight the performance tradeoffs between weak and strong mechanisms. For example, a simple Parity byte can be

Table 7.4: MicroQoSCORBA End-to-End Latencies (ms)

Security Mechanism	Linux			SaJe			TINI		
	No TR	TR2	TR4	No TR	TR2	TR4	No TR	TR2	TR4
Baseline	0.170	0.184	0.182	4.16	4.16	4.22	248	242	341
Parity	0.180	0.197	0.195	4.48	4.90	4.80	351	369	553
CRC32	0.178	0.194	0.195	4.64	5.18	5.31	415	466	688
MD5	0.199	0.218	0.218	6.59	8.87	10.90	1030	1,306	1,951
SHA1	0.204	0.232	0.231	7.08	9.52	11.85	1311	1696	2,537
Caesar	0.180	0.194	0.188	4.43	4.88	4.78	318	353	514
DES	0.201	0.226	0.227	6.80	9.80	12.26	1263	1,908	2,846
TripleDES	0.238	0.294	0.297	11.22	23.35	29.96	3168	5,105	7,640
AES-128	0.207	0.229	0.222	5.95	7.94	7.14	858	1,050	1,554
AES-192	0.202	0.222	0.222	6.17	8.25	9.86	951	1,181	1,751
AES-256	0.219	0.227	0.229	6.40	8.55	10.30	1047	1,312	1,945

added to a message in a small fraction of the time that SHA1, a secure message digest, can computed. Likewise, when time is of the essence a developer might have to choose to use a Caesar cipher on TINI so that messages can be sent in less than half a second.

One of the research goals of architecting and developing MicroQoSCORBA was to create a testbed for multi-property QoS interactions. Simple as this evaluation example is, several results can be observed. Security and fault tolerance can have a very significant impact depending upon the hardware platform used. In particular, the Linux machines can send thousands of messages a second with or without fault tolerance or security mechanisms enabled. The TINI board, on the other hand, requires over seven seconds to send a message encrypted with

TripleDES in a temporally redundant manner.

7.5 Security Mechanism Evaluations

The results presented in Section 7.4 present an overview of MicroQoSCORBA's multi-property QoS support, but they do not give a thorough evaluation of MicroQoSCORBA's security mechanisms. A key contribution of this dissertation was the design and implementation of MicroQoSCORBA's security subsystem, therefore a thorough evaluation of MicroQoSCORBA's security mechanisms will now be presented. First the application sizes will be presented for the ciphers and message digests, followed by the end-to-end latency values of MicroQoSCORBA's ciphers, message digests, and message authentication codes.

7.5.1 Application Size and Memory Usage

An application's size and memory usage depends upon the size of the application's Java class files that must be loaded into memory, the amount of data that must be allocated on the heap during run-time, and other system-specific and run-time library code that must be loaded and executed on behalf of the application. Measuring the size of the generated Java class files for the MicroQoSCORBA ORB and timing example code is a straightforward task. These results are discussed in the following subsections.

Security Ciphers

The security cipher results (in bytes) are reported in Table 7.5. The client and server class file sizes are reported for each of the three testbed hardware platforms. If a cipher supports multiple key lengths, then the java class file sizes are for a

configuration using a 128-bit key, if supported, in order to provide a better cross-cipher size comparison. It should be noted that for ciphers that support multiple key lengths, changing the key length caused the java class file sizes to vary by only a few bytes, if at all.

The reported SaJe values in Table 7.5 are larger than the Linux and TINI values because the SaJe development tools include some runtime class libraries in executable image that is downloaded into the SaJe's flash memory—whereas on Linux and TINI these run-time class libraries are not counted because they are part of their platform's respective software JVM. The TINI values are relatively small because a compressed archive of the required class files is downloaded to the TINI board.

In order of increasing java class file sizes, the various ciphers are sorted into the following order: Null, XOR (8-bit), Caesar, RC4, XOR (128-bit), IDEA, Square, RC2, DES, SKIPJACK, Triple-DES, AES, Twofish, Serpent, MARS, CAST5. Naturally, one would expect that a Null cipher which contains no computation logic would be the smallest. After the Null cipher, the XOR and Caesar ciphers are next—which also makes sense because they are both very simple. One can also observe that for a Linux client encryption overheads add from a minimum of 11.7 Kbytes of Java byte code for the XOR, and Caesar ciphers to an observed maximum of about 34.2 Kbytes for the complex CAST5 cipher implementation. DES adds 17.2 Kbytes and AES adds 22.3 Kbtyes of Java byte code to a client's baseline of 51.7 Kbytes of Java byte code.

Table 7.5: Cipher Impacts on Java Class File Sizes (bytes)

Security	**Linux**		**SaJe**		**TINI**	
Cipher	**Client**	**Server**	**Client**	**Server**	**Client**	**Server**
Baseline	51,704	51,995	248,519	249,319	19,168	19,050
Null	62,996	63,344	252,071	252,846	22,574	22,422
XOR-8	63,429	63,777	252,897	253,661	22,882	22,730
XORp-128	63,569	63,916	252,963	253,723	22,947	22,794
Caesar-8	63,452	63,800	252,920	253,684	22,896	22,744
AES-128	73,999	74,346	260,090	260,850	30,571	30,418
CAST5-128	85,936	86,283	275,448	276,208	41,851	41,698
DES-56	68,872	69,220	256,210	256,974	26,408	26,256
IDEA-128	67,408	67,755	254,176	254,936	24,959	24,806
MARS-128	75,660	76,007	261,874	262,634	32,786	32,633
RC2-128	68,867	69,214	255,907	256,678	26,677	26,524
RC4-128	63,463	63,810	252,655	253,415	22,893	22,740
Serpent-128	75,160	75,507	260,386	261,157	30,597	30,444
SKIPJACK-80	69,749	70,096	256,643	257,414	27,393	27,240
Square-128	68,672	69,019	255,334	256,094	25,896	25,743
Triple-DES-168	70,342	70,689	256,992	257,763	27,026	26,873
Twofish-128	74,443	74,790	260,414	261,185	30,411	30,258

Message Digests and Authentication Codes

The message digest results are reported in Table 7.6. The client and server class file sizes are reported for each of the three testbed hardware platforms. As explained in the previous subsection, the SaJe values are larger than the Linux values and the TINI values are smaller.

In order of increasing java class file sizes, the various message digests are sorted into the following order: Null, Parity, CRC32, MD2, MD4, SHA2-256, RIPEMD-128, MD5, SHA2-512, SHA2-384, RIPEMD-160, RIPEMD, SHA0, SHA1, and Tiger. After the Null message digest, the Parity and CRC32 message digests are the smallest, with size increases of 4.6 and 5.1 KBytes, respectively. Size-wise the Tiger message digest adds the most overhead with a size increase of 26.3 KBytes.

MicroQoSCORBA supports the HMAC message authentication algorithm. For both of the Linux and TINI platforms, the HMAC algorithm is implemented with additional classes and methods that add a constant amount to both of the client and server class file sizes. On Linux, the amount is 7,177 bytes for the Client and 7,165 for the Server. On TINI these values are 2,354 and 2,342 respectively. On the SaJe platform there is a small variation between the multiple configuration, but on average, the Client size is increased by 3,125 bytes and the Server size is increased by 3,102 bytes.

Table 7.6: Message Digest Impacts on Java Class File Sizes (bytes)

Message Digest	**Linux**		**SaJe**		**TINI**	
	Client	**Server**	**Client**	**Server**	**Client**	**Server**
Baseline	51,704	51,995	248,519	249,319	19,168	19,050
Null	56,289	56,627	251,644	252,503	20,749	20,678
Parity-8	56,464	56,799	251,763	252,614	20,852	20,778
CRC32	56,849	57,186	251,965	252,820	21,047	20,975
MD2	59,480	59,819	254,306	255,154	23,397	23,327
MD4	60,546	60,885	254,450	255,298	23,289	23,219
MD5	61,709	62,048	255,330	256,178	24,166	24,096
RIPEMD	62,389	62,725	255,781	256,636	24,717	24,644
RIPEMD128	61,405	61,738	255,444	256,280	24,257	24,181
RIPEMD160	62,262	62,595	256,164	257,011	24,982	24,906
SHA0	63,566	63,904	257,326	258,174	25,886	25,815
SHA1	63,578	63,916	257,334	258,182	25,894	25,823
SHA2-256	60,799	61,135	254,521	255,365	23,341	23,268
SHA2-384	62,158	62,494	255,653	256,508	24,353	24,280
SHA2-512	62,156	62,492	255,649	256,504	24,351	24,278
Tiger	78,029	78,366	270,676	271,520	41,456	41,384

Table 7.7: Cipher and Message Digest Impacts on Java Class File Sizes (bytes)

Cipher & Message Digest	**Linux**		**SaJe**		**TINI**	
	Client	**Server**	**Client**	**Server**	**Client**	**Server**
Baseline	51,704	51,995	248,519	249,319	19,168	19,050
XOR & Parity-8	67,824	68,200	255,815	256,611	24,427	24,307
Triple-DES & MD5	79,982	80,361	263,536	264,336	31,885	31,768
AES-128 & SHA1	85,508	85,886	268,627	269,427	37,158	37,040
AES-256 & SHA2-512	84,086	84,462	266,953	267,749	35,615	35,495

Cipher and Message Digests

The message digest results are reported in Table 7.7. The client and server class file sizes are reported for each of the three testbed hardware platforms. As explained in the previous subsection, the SaJe values are larger than the Linux values and the TINI values are smaller.

Only a few combinations are shown to illustrate that MicroQoSCORBA supports the composition of multiple security mechanisms. The given results illustrate that multiple security properties can be composed within MicroQoSCORBA. The composition of the two properties, a cipher and a message digest, results in a slightly larger class file size than just adding the respective file size deltas associated with the respective mechanisms.

7.5.2 Performance

Performance evaluation experiments were conducted on three hardware platforms (Linux, SaJe, and TINI). These three platforms vary widely in terms of overall performance. Furthermore, each platform's operating system and Java implementation impact the end-to-end performance of MicroQoSCORBA. The performance of MicroQoSCORBA's ciphers, message digests, and message authentication codes are now presented.

Cipher Performance

The security cipher performance values are shown in Table 7.8. Timing results were measured on each of the three testbed platform for three different packet lengths as explained previously. Each row in Table 7.8 refers to the execution of a given MicroQoSCORBA configuration. The values listed in the Baseline row correspond to a baseline MicroQoSCORBA configuration that does not support any security mechanisms. The Null configuration incorporates the methods needed to implement a security cipher but it does not actually encrypt or decrypt the data before sending it across the network. Several of MicroQoSCORBA's supported ciphers may be configured to use multiple key lengths. Unless otherwise noted by a value in parentheses, the results are reported for configuration using 128-bit keys.

Several of the rows in Table 7.8 are of particular interest—namely, the Baseline, Null, XOR, and Caesar cipher rows. Comparing the results listed in the Baseline and Null rows shows that a minimal overhead is added for incorporating the

additional classes and method invocations needed to implement encryption and decryption of the MicroQoSCORBA messages. On TINI the Baseline and Null results for 512-byte message are slightly slower than than their respective 56-byte messages values. At first this seems contradictory, because it indicates that more data can be sent in less time. But this result can be explained by the fact that given an a priori knowledge of the packet length, MicroQoSCORBA is able to optimize its message marshalling and demarshalling routines. Furthermore, TINI's JVM was optimized for data I/O rather than computational power [Loo01]. Taken together, this means that for this given MicroQoSCORBA configuration, more data can be sent in less time on the TINI platform.

The XOR cipher, with either an 8-bit or 128-bit key, and the Caesar ciphers are not very secure. However, all three of these ciphers execute relatively quickly and because of this they are supported in MicroQoSCORBA so that they could be used in time critical applications where only a very modest amount of confidentiality is required. The AES cipher was the quickest of the strong, cryptographically secure ciphers on all three platforms. Triple-DES was the slowest cipher on SaJe and TINI, but on Linux Twofish ran slower.

The performance values for Linux are the fastest, followed by the SaJe and TINI values. However, one might not have realized just how large of difference there is between these three platforms. For example, a 1024-byte message encrypted with Triple-DES can be sent and received in under two milliseconds on

Table 7.8: Cipher Timing Results (ms)

Security	Linux			SaJe			TINI		
Cipher	**56**	**512**	**1024**	**56**	**512**	**1024**	**56**	**512**	**1024**
Baseline	0.161	0.247	0.351	3.77	5.88	8.29	134	129	141
Null	0.165	0.249	0.358	3.87	5.98	8.44	152	143	159
XOR (8)	0.166	0.264	0.378	4.02	8.42	13.31	198	918	1,706
XOR	0.167	0.279	0.409	4.18	11.18	18.83	263	1,984	3,837
Caesar (8)	0.163	0.267	0.385	4.02	8.56	13.59	203	953	1,776
AES	0.194	0.406	0.647	5.26	23.04	41.93	647	6,460	12,600
AES (192)	0.198	0.420	0.686	5.45	25.57	46.85	726	7,504	14,641
AES (256)	0.199	0.449	0.717	5.64	28.06	51.75	805	8,545	16,690
CAST5 (40)	0.189	0.502	0.848	5.20	24.19	44.51	591	6,267	12,299
CAST5	0.189	0.530	0.908	5.48	28.20	52.41	702	7,843	15,433
DES (56)	0.190	0.538	0.914	6.15	37.84	71.53	1,052	12,916	25,500
IDEA	0.206	0.828	1.490	6.47	42.40	80.57	980	11,814	23,298
MARS	0.203	0.616	1.102	6.52	39.73	74.73	1,308	15,053	29,739
RC2	0.197	0.608	1.061	6.24	39.19	74.22	1,115	13,843	27,415
RC4	0.168	0.364	0.576	4.65	19.92	36.37	448	5,209	10,290
Serpent	0.196	0.636	1.109	7.10	47.30	89.67	1,481	17,454	34,237
SKIPJACK (80)	0.203	0.800	1.478	7.29	54.32	104.28	1,447	18,617	36,782
Square	0.177	0.408	0.665	6.14	34.77	64.96	902	9,820	19,192
Triple-DES (168)	0.229	0.997	1.796	10.13	95.28	185.64	2,764	37,720	74,602
Twofish	0.238	1.050	1.916	9.68	81.35	156.77	1,723	20,620	40,478

Linux, whereas on TINI the same task takes over 74 seconds. Relative performance on a given platform also varied widely. The performance ratio of the slowest configuration (i.e., Twofish on 1024-byte messages on Linux, Triple-DES on SaJe and TINI) divided by the quickest (8-bit XOR on 56-byte messages) is 11.5 on Linux, 22.0 on SaJe, and on TINI this ratio is 376.8.

Message Digest Performance

The message digest (MD) performance values are shown in Table 7.9. Timing results were measured on each of the three testbed platform for three different packet lengths as explained previously. Each row in Table 7.9 refers to the execution of a given MicroQoSCORBA message digest configuration. The values listed in the Baseline row correspond to a baseline MicroQoSCORBA configuration that does not support any security mechanisms. The Null configuration incorporates the methods needed to implement a message digest but it does not actually digest any data before sending it across the network. The length, in bits, of each message digest are as follows: Parity–8, CRC32–32, MD2–128, MD4–128, MD5–128, RIPEMD–128, RIPEMD-128–128, RIPEMD-160–160, SHA0–160, SHA1–160, SHA2-256–256, SHA2-384–384, SHA2-512–512, Tiger–192.

Several of the rows in Table 7.9 are of particular interest—namely, the Baseline, Null, Parity, and CRC32 rows. Comparing the results listed in the Baseline and Null rows shows that a minimal overhead is added for incorporating the additional classes and method invocations needed to implement message digests. Even though the Parity and CRC32 digests are not very secure, these digests execute quickly and are supported in MicroQoSCORBA so that they may be used in

Table 7.9: Message Digest Timing Results (ms)

Security	**Linux**			**SaJe**			**TINI**		
Mechanism	**56**	**512**	**1024**	**56**	**512**	**1024**	**56**	**512**	**1024**
Baseline	0.161	0.247	0.351	3.77	5.88	8.29	134	129	141
Null	0.166	0.245	0.352	3.87	5.98	8.40	157	151	163
Parity	0.165	0.307	0.467	4.08	8.38	13.11	233	995	1,843
CRC32	0.167	0.291	0.438	4.22	9.91	16.23	295	1,684	3,220
MD2	0.311	1.356	2.512	22.67	162.27	311.44	7,673	62,869	121,772
MD4	0.184	0.323	0.485	5.50	13.14	21.40	720	2,672	4,751
MD5	0.184	0.318	0.471	6.20	17.35	29.36	919	3,870	7,010
RIPEMD	0.190	0.362	0.564	6.28	17.86	30.32	1,013	4,434	8,079
RIPEMD-128	0.193	0.358	0.556	6.57	19.55	33.52	1,206	5,571	10,224
RIPEMD-160	0.200	0.393	0.623	7.46	24.54	42.89	1,603	7,884	14,581
SHA0	0.200	0.356	0.549	6.61	19.41	33.19	1,148	5,109	9,334
SHA1	0.202	0.360	0.558	6.70	19.95	34.22	1,202	5,440	9,953
SHA2-256	0.213	0.470	0.757	14.00	62.79	114.93	3,421	18,600	34,795
SHA2-384	0.311	0.876	1.455	15.05	57.08	99.24	4,124	18,676	33,132
SHA2-512	0.315	0.878	1.466	15.30	57.32	99.48	4,168	18,713	33,186
Tiger	0.227	0.472	0.762	6.76	20.71	35.71	1,187	5,495	10,078

time critical applications. On Linux, the CRC32 MD outperforms the Parity MD, but on the other two platforms the Parity MD is faster. The slow implementation of MD2 is likely due to the fact that although this digest was designed for 8-bit hardware, it's implementation in a 32-bit Java virtual machine is less than optimal. With the exception of MD2, the SHA2 family of digests are the slowest digests due to their computational complexity.

The performance values for Linux are the fastest, followed by the SaJe and TINI values. However, there is very large difference between these three platforms. For example, sending a 1024-byte MicroQoSCORBA message with an MD2 digest takes 2.5 ms on Linux, 311 ms on SaJe and over 2 minutes (121.7 seconds) on TINI. Relative performance on a given platform also varied widely. The performance ratio of the slowest configuration (i.e., MD2 on 1024-byte messages) divided by the quickest (i.e., Parity on 56-byte messages) is 15.2 on Linux, 37.2 on SaJe, and on TINI this ratio is 522.4.

Message Authentication Code Performance

The message authentication code (MAC) performance values are shown in Table 7.10. Timing results were measured on each of the three testbed platform for three different packet lengths. Each row in Table 7.10 refers to the execution of a given MicroQoSCORBA message authentication code configuration. The values listed in the Baseline row correspond to a baseline MicroQoSCORBA configuration that does not support any security mechanisms. The Null configuration incorporates the methods needed to implement a message authentication code but it does not actually authenticate any data before sending it across the network.

Table 7.10: Message Authentication Code Timing Results (ms)

Security Mechanism	Linux			SaJe			TINI		
	56	**512**	**1024**	**56**	**512**	**1024**	**56**	**512**	**1024**
Baseline	0.161	0.247	0.351	3.77	5.88	8.29	134	129	141
Null	0.167	0.251	0.355	4.06	6.17	8.58	208	201	213
Parity	0.175	0.311	0.465	4.26	8.56	13.29	296	1,063	1,907
CRC32	0.173	0.297	0.437	4.54	10.23	16.54	410	1,802	3,336
MD2	0.583	1.637	2.804	55.72	195.34	344.51	20,909	76,103	135,021
MD4	0.236	0.367	0.533	9.96	17.60	25.88	2,253	4,206	6,283
MD5	0.225	0.355	0.512	12.99	24.14	36.15	3,117	6,067	9,206
RIPEMD	0.258	0.437	0.643	13.36	24.94	37.40	3,524	6,945	10,591
RIPEMD-128	0.260	0.427	0.629	14.59	27.59	41.54	4,349	8,721	13,380
RIPEMD-160	0.281	0.467	0.684	18.23	35.34	53.64	6,025	12,311	19,009
SHA0	0.258	0.415	0.600	14.53	27.36	41.12	4,038	8,006	12,248
SHA1	0.256	0.424	0.615	14.93	28.21	42.44	4,276	8,519	13,034
SHA2-256	0.330	0.589	0.879	46.01	94.78	146.92	13,791	28,980	45,170
SHA2-384	0.921	1.483	2.070	66.00	108.02	150.14	22,599	37,170	51,612
SHA2-512	0.926	1.488	2.080	66.30	108.32	150.45	22,651	37,197	51,682
Tiger	0.337	0.611	0.874	15.31	29.29	44.27	4,263	8,572	13,162

Currently, the supported MAC algorithm is HMAC and so each of the row labels shown in Table 7.10 refer to the underlying message digest used in the HMAC algorithm.

Once again, the Null MAC illustrates that only a modest overhead is added with the addition of the classes and methods needed to implement MicroQoS-CORBA's MAC mechanisms. As expected, because HMAC builds a MAC based upon a given message digest, the HMAC values shown in Table 7.10 are all higher

than their corresponding message digest performance values shown in Table 7.9.

As with the cipher and message digest results, the MAC performance values for Linux are the fastest, followed by the SaJe and TINI values. However, there is very large difference between these three platforms. For example, sending a 1024-byte MicroQoSCORBA message with an MD2 digest takes 2.8 ms on Linux, 344 ms on SaJe and over 2 minutes (135 seconds) on TINI. Relative performance on a given platform also varied widely. The performance ratio of the slowest configuration (i.e., MD2 on 1024-byte messages) divided by the quickest (i.e., Parity on 56-byte messages for SaJe and TINI and CRC32 for Linux) is 16.2 on Linux, 80.9 on SaJe, and on TINI this ratio is 456.2.

Cipher and Message Digest Combined Performance

All of MicroQoSCORBA's cipher, message digest, and message authentication code mechanisms can be composed together. Rather than present another large table, only a few results are presented in Table 7.11. For reference the baseline performance values are presented. Also presented for comparison are the combination of two light-weight mechanisms, XOR & Parity, as well as stronger mechanisms, Triple-DES & MD5, AES-128 & SHA1, and AES-256 & SHA2-512.

As shown by the results presented in Table 7.11, the performance differences between the three testbed platforms is significant. These results also show that the performance of the cipher and message digest composition can not be simply assumed to be a linear combination of the cipher and message digest values. For example, consider Triple-DES & MD5 on SaJe with variable-length packets (i.e.,

Table 7.11: Cipher and Message Digest Timing Results (ms)

Security Mechanism	Linux			SaJe			TINI		
	56	**512**	**1024**	**56**	**512**	**1024**	**56**	**512**	**1024**
Baseline	0.161	0.247	0.351	3.77	5.88	8.29	134	129	141
XOR & Parity	0.170	0.322	0.498	4.33	10.92	18.13	299	1,787	3,414
Triple-DES & MD5	0.269	1.025	1.882	15.31	109.50	209.50	4,704	42,550	82,629
AES-128 & SHA1	0.235	0.538	0.867	8.99	37.63	68.29	2,002	11,959	22,592
AES-256 & SHA2-512	0.375	1.095	1.859	19.83	82.14	145.56	5,892	28,350	51,131

the 56-byte column). Adding the individual values and subtracting off one baseline value (so that the baseline is not double counted) results in 12.56 ms for an estimate, but the measured value is 15.31 ms.

7.5.3 Analysis

The previous results sections, Sections 7.5.2–7.5.2, have presented some limited analysis of the results. In particular, it has been shown that for each of the evaluated mechanisms, the performance between the testbed systems varied dramatically and the performance between the "quickest" and "slowest" mechanisms within a category also varied dramatically. In this section, common analysis trends and results will be highlighted. First a discussion of hardware impacts will be given and then a discussion of these performance results in the broader multi-property QoS domain towards which MicroQoSCORBA is targeted.

Hardware Impacts

The performance results presented in this dissertation were gathered on three separate hardware platforms. In part this was done to show the versatility of MicroQoSCORBA, but in part it was also done to show the impact that hardware has upon the design of distributed embedded systems. Generally speaking, all of the Linux performance results occurred in sub-millisecond time intervals, whereas the TINI results occurred spanned tens of seconds, and even minutes at times. Also the relative performance of "slow" vs. "fast" mechanisms varied greatly. On Linux these values were in the range of a factor of 10–20x, on SaJe these ratios ranged from 20–80x, and on TINI the ratios were well over a factor of 350x. Given these wide variances, results such as those presented in this dissertation are essential so that a system designer can begin to understand the impact that the hardware design choices will have upon a given distributed embedded system.

Multi-property QoS Impacts

Security properties, such as confidentiality and integrity, comprise just a few of the overall system properties that must be considered when designing distributed embedded systems. MicroQoSCORBA was designed and implemented to support the composition of multiple QoS properties within a given middleware framework. Although the primary focus of this dissertation is on MicroQoSCORBA's security subsystem, the presented performance results do highlight another QoS property, namely timeliness.

Generally speaking, increased security comes with an increased time cost. However, the choice of an appropriate security mechanism is important because in

a few cases, increased security can be achieved without an increased performance penalty. For example, except for the Linux 56-byte messages, all of the AES configurations on all three platforms outperformed DES, a weaker cipher. On TINI the computational burden associated with strong security mechanisms (e.g., AES, Triple-DES, SHA2) is especially noticeable. Rather than just spanning a few milliseconds at best (as on Linux or TINI), the slower mechanisms on TINI could take over half a minute to complete. Even a simple XOR encryption scheme takes over a second for a long message. Thus, the results presented clearly show that a designer must choose to balance security strength and timeliness when designing a distributed embedded system.

CHAPTER EIGHT

CONCLUSION

This dissertation concludes by presenting the contributions of this research, followed by a discussion of future work.

8.1 Contributions

This dissertation began with a general introduction to distributed systems, middleware, embedded systems, and quality of service in Chapter 1. After that, related work was presented in Chapter 2. Together these two chapters motivate the need for and the previously existing lack of support for fine-grained configurability with multiple quality of service properties in middleware for embedded systems.

Chapter 3 presented a refined, fine-grained distributed systems architectural taxonomy. This architectural taxonomy is a key contribution of this dissertation because it identifies and explains the relationships between fine-grained architectural design components.

As explained in Chapter 4, MicroQoSCORBA's architecture was designed to support the fine-grained components that were identified in Chapter 3. Furthermore, MicroQoSCORBA's architecture supports fine-grained configurability with multiple Quality of Service properties.

An analysis of security requirements for embedded systems from an embedded systems middleware perspective was presented in Chapter 5. This analysis showed that security must be designed into a distributed embedded application.

The analysis also showed that the level of security provided must be tailored to both the application and hardware constraints of a given distributed embedded application.

The design and implementation of MicroQoSCORBA, a new middleware framework, was presented in Chapter 6. This chapter also presented details about the development environment and tools that support the creation of MicroQoSCORBA applications. A methodology for filtering out Java and OS specific impacts in order to determine the best-case performance of a given hardware platform was presented at the end of this chapter.

In Chapter 7, an experimental evaluation of MicroQoSCORBA's performance results were presented. These results compared MicroQoSCORBA's performance with other ORBs as well as the various resource usage and performance tradeoffs required to support multiple QoS properties. MicroQoSCORBA's end-to-end latency was significantly better than either JacORB or TAO's end-to-end latencies. This is because MicroQoSCORBA's fine-grained configurability removes unneeded functionality, thereby removing unneeded overheads associated with end-to-end middleware method invocations. MicroQoSCORBA's fine-grained composability enabled it to operate with a smaller memory footprint than JacORB, another (coarse-grained) Java-based ORB. The QoS evaluation results illustrate that both the hardware platform and security mechanism choices impact the end-to-end performance of MicroQoSCORBA.

In summary, the key contributions of this dissertation are the following.

- A fine-grained middleware architectural design taxonomy, to better capture

the wide variation of distributed systems interactions and hence how they can be configured.

- The design and implementation of an architecture for a fine-grained and composable middleware framework.
- An analysis of security requirements for embedded systems.
- The design and implementation of a highly configurable security subsystem.
- An experimental evaluation of MicroQoSCORBA's performance on three hardware platforms.

Furthermore, the thesis of this dissertation:

> **Middleware frameworks for embedded systems can be designed with fine-grained composability and they can also support multiple Quality of Service properties.**

has been demonstrated by the research that supported the design, implementation, and evaluation of MicroQoSCORBA.

8.2 Future Work

Future work for MicroQoSCORBA will include profiling tools and additional QoS support. Profiling tools will be developed and integrated with the MicroQoSCORBA toolkit in order to give developers the ability to make informed decisions regarding resource usage versus performance tradeoffs based on their application code. The existing implementation of MicroQoSCORBA has been designed with

the goal of supporting several QoS properties (e.g., security, fault tolerance, timeliness, network performance). In the future, additional concrete implementation of these QoS subsystems and their mechanisms will be developed, integrated, and profiled with respect to each other and multi-property QoS support. We also plan to investigate how MicroQoSCORBA's very fine granularity and composability can lend itself to easier validation within mission critical applications.

APPENDIX

APPENDIX A

SOURCE CODE EXAMPLES

Five files are listed in this Appendix. The 'Config_macros.m4' file defines the *m4* macros required for the macro-preprocessing of MicroQoSCORBA's macro-enabled source code files. The 'Client.java.m4' and 'Server.java.m4' files are two macro-enabled source code files that contain the code for the MicroQoSCORBA client and server applications, respectively. These two files were used to gather the experimental performance evaluation results presented in Chapter 7. The 'Makefile.mqcc' and 'Makefile' files are used in the automated build process.

A.1 Config_macros.m4

The source code listed in Figure A.1 defines the *m4* macros used by MicroQoSCORBA. This file, 'Config_macros.m4', is autogenerated by MicroQoSCORBA's IDL compiler and fine-tuned to a given distributed system's hardware and application constraints.

Config_macros.m4

```
1  divert(-1)
2  define(‘__J2SE__’, ‘$*’)dnl
3  define(‘__CLDC__’)dnl
4  define(‘__TCP_TP__’, ‘$*’)dnl
5  define(‘__UDP_TP__’)dnl
6  define(‘__UNRELIABLE_UDP__’)dnl
7  define(‘__GO_BACK_N_UDP__’)dnl
8  define(‘__STOP_N_WAIT_UDP__’)dnl
9  define(‘__SERIAL_TP__’)dnl
10 define(‘__ONEWIRE_TP__’)dnl
11 define(‘__GIOP_PROTOCOL__’, ‘$*’)dnl
12 define(‘__GIOPLITE_PROTOCOL__’)dnl
13 define(‘__MQCIOP_PROTOCOL__’)dnl
14 define(‘__GIOP10__’)dnl
15 define(‘__GIOP11__’)dnl
16 define(‘__GIOP12__’, ‘$*’)dnl
17 define(‘__GIOP13__’)dnl
18 define(‘__FIX_SIZE__’)dnl
```

```
define(‘__VAR_SIZE__’, ‘$*’)dnl
define(‘__TEMPORAL__’)dnl
define(‘__NOTEMPORAL__’, ‘$*’)dnl
define(‘__SPATIAL__’)dnl
define(‘__NOSPATIAL__’, ‘$*’)dnl
define(‘__TEMPSPACE__’)dnl
define(‘__NOTEMPSPACE__’, ‘$*’)dnl
define(‘__GROUPCOM__’)dnl
define(‘__NOGROUPCOM__’, ‘$*’)dnl
define(‘__GCUNIFORM__’)dnl
define(‘__GCNONUNIFORM__’)dnl
define(‘__VALUERED__’)dnl
define(‘__NOVALUERED__’, ‘$*’)dnl
define(‘__SECURITY__’)dnl
define(‘__SEC_CIPHER_ALG__’)dnl
define(‘__SEC_CIPHER_MODE__’)dnl
define(‘__SEC_CIPHER_MODE_ECB__’)dnl
define(‘__SEC_CIPHER_MODE_CBC__’)dnl
define(‘__SEC_CIPHER_PADDING__’)dnl
define(‘__SEC_CIPHER_PADDING_NONE__’)dnl
define(‘__SEC_CIPHER_PADDING_PKCS5__’)dnl
define(‘__SEC_CIPHER_KEY_LEN__’)dnl
define(‘__SECURITY_MD__’)dnl
define(‘__SEC_MD_ALG__’)dnl
define(‘__SECURITY_MD_NOT_HMAC__’, ‘$*’)dnl
define(‘__SECURITY_MAC__’)dnl
define(‘__SEC_MAC_ALG__’)dnl
define(‘__SEC_MAC_ALG_PARAMS__’)dnl
define(‘__SEC_MAC_KEY_LEN__’)dnl
define(‘__SECURITY_ORBKEY__’)dnl
define(‘__SECURITY_AUDIT__’)dnl
define(‘__SEC_AUDIT_ALG__’)dnl
define(‘__SEC_AUDIT_LOG_LEN__’)dnl
define(‘__DEBUG__’)dnl
define(‘__DEBUG1__’)dnl
define(‘__DEBUG2__’)dnl
define(‘__DEBUG3__’)dnl
define(‘__DEBUG4__’)dnl
define(‘__DEBUG5__’)dnl
define(‘__DEBUG_MEMORY__’, ‘$*’)dnl
define(‘__DEBUG_TIMING__’)dnl
define(‘__OS__’, ‘LINUX’)dnl
ifelse(__OS__, ‘LINUX’, ‘
  define(‘__LINUX__’, ‘$*’)dnl
  define(‘__TIMER_INIT__’, ‘JNITimer.init(1000000)’)dnl
  define(‘__GET_TIME__’, ‘JNITimer.currentTime()’)dnl
  define(‘__TICKS_PER_MS__’, ‘1000’);dnl
  define(‘__SS_ITERATIONS__’, ‘15000’)dnl
  define(‘__SS_LOOP_CNT__’, ‘3’)dnl
  define(‘__MAIN_LOOP_CNT__’, ‘3’)dnl
  define(‘__TIMEOUT_SEC__’, ‘600’)dnl
’, ‘
  define(‘__LINUX__’)dnl
’)dnl
ifelse(__OS__, ‘DARWIN’, ‘
  define(‘__DARWIN__’, ‘$*’)dnl
  define(‘__TIMER_INIT__’, ‘;’)dnl
```

```
  define(‘__GET_TIME__’, ‘JNITimer.currentTime()’)dnl
  define(‘__TICKS_PER_MS__’, ‘1000’);dnl
  define(‘__SS_ITERATIONS__’, ‘15000’)dnl
  define(‘__SS_LOOP_CNT__’, ‘3’)dnl
  define(‘__MAIN_LOOP_CNT__’, ‘3’)dnl
  define(‘__TIMEOUT_SEC__’, ‘600’)dnl
’, ‘
  define(‘__DARWIN__’)dnl
’)dnl
ifelse(__OS__, ‘GENERIC’, ‘
  define(‘__GENERIC__’, ‘$*’)dnl
  define(‘__TIMER_INIT__’, ‘;’)dnl
  define(‘__GET_TIME__’, ‘System.currentTimeMillis()’)dnl
  define(‘__TICKS_PER_MS__’, ‘1’);dnl
  define(‘__SS_ITERATIONS__’, ‘15000’)dnl
  define(‘__SS_LOOP_CNT__’, ‘3’)dnl
  define(‘__MAIN_LOOP_CNT__’, ‘3’)dnl
  define(‘__TIMEOUT_SEC__’, ‘600’)dnl
’, ‘
  define(‘__GENERIC__’)dnl
’)dnl
ifelse(__OS__, ‘SAJE’, ‘
  define(‘__SAJE__’, ‘$*’)dnl
  define(‘__TIMER_INIT__’, ‘;’)dnl
  define(‘__GET_TIME__’, ‘com.ajile.jem.rawJEM.getTime()’)dnl
  define(‘__TICKS_PER_MS__’, ‘1000’);dnl
  define(‘__SS_ITERATIONS__’, ‘100’)dnl
  define(‘__SS_LOOP_CNT__’, ‘3’)dnl
  define(‘__MAIN_LOOP_CNT__’, ‘3’)dnl
  define(‘__TIMEOUT_SEC__’, ‘600’)dnl
’, ‘
  define(‘__SAJE__’)dnl
’)dnl
ifelse(__OS__, ‘TINI’, ‘
  define(‘__TINI__’, ‘$*’)dnl
  define(‘__TIMER_INIT__’, ‘;’)dnl
  define(‘__GET_TIME__’, ‘com.dalsemi.system.TINIOS.uptimeMillis()’)dnl
  define(‘__TICKS_PER_MS__’, ‘1’);dnl
  define(‘__SS_ITERATIONS__’, ‘10’)dnl
  define(‘__SS_LOOP_CNT__’, ‘3’)dnl
  define(‘__MAIN_LOOP_CNT__’, ‘3’)dnl
  define(‘__TIMEOUT_SEC__’, ‘900’)dnl
’, ‘
  define(‘__TINI__’)dnl
’)dnl
undefine(‘index’)dnl
undefine(‘len’)dnl
divert
/* Do NOT edit this file--It was autogenerated by m4 from a *.java.m4 file */
```

Config_macros.m4

Figure A.1: Config_macros.m4 Listing

A.2 Client.java.m4

The source code for the ‘Client.java.m4’ file is listed in Figure A.2. This file contains the baseline code needed to support invoking methods as well as additional QoS related code that is compiled in to or out of the client application depending upon the macro definitions in the ‘Config_macros.m4’ file. Furthermore, this file also contains the actual implementation of the event filtering method discussed in Section 6.4.3.

Client.java.m4

```
1  //include('Config_macros.m4')
2
3  import mqc.*;
4  import mqc.holders.*;
5  import mqc.Config;
6
7  __SECURITY_ORBKEY__(
8  import mqc.security.InvalidKeyException;
9  )
10 __SAJE__(
11 import com.ajile.lang.Math;
12 )
13
14 public class Client
15 {
16     static final int REPEAT_CNT = __MAIN_LOOP_CNT__; // number of times to repeat
17                                                      //    the main timing loop
18     __DEBUG_TIMING__(
19     static       long  dt_cnt    = 0;    // count of dt[] elements
20     static       long  dt_sum    = 0;    // E[x]  of dt histogram
21                                          //  (unnormalized by cnt)
22     static       long  dt_sum2   = 0;    // E[x2] of dt histogram
23                                          //  (unnormalized by cnt)
24     static       float dt_avg    = 0;    // avg time based upon dt
25     static      double dt_stdev  = 0.0;  // standard deviation
26     static      double dt_stdev2 = 0.0;  // standard deviation squared
27     static  double STDEV_ERR     = 99.9; // value for stdev if an error occurs
28     static      long[] r_cnt     = new long[REPEAT_CNT];   // i_th dt_cnt
29     static     float[] r_avg     = new float[REPEAT_CNT];  // i_th dt_avg
30     static    double[] r_stdev   = new double[REPEAT_CNT]; // i_th stdev
31     static final int   DT_SIZE   = 500;  // 500 for linux w/TR, 5000 for SaJe
32                                          //   w/TR, 1000 for TINI w/TR
33     static final int   DTW_SIZE  = 300;  // 300 for linux w/TR, 1000 for SaJe
34                                          //   w/TR, 100 for TINI w/TR
35     static final int[] dt   = new int[DT_SIZE];  // store either individual event
36     static final int[] dtw  = new int[DTW_SIZE]; // times or bins of event counts
37     static       int[] dt_b;
38     static       int   dt_offset = 0;    // dt[] offset, bin dt[i] correspond to
39                                          //   time i+dt_offset
```

```
    static     boolean dt_binEvents;     // false -> ind. event times,
                                         //    true -> event cnt bins in dt[]
    static       float nSigma    = 3.5f; // default width of the peak (in stdev)
    static       int   timeoutSec = __TIMEOUT_SEC__;  // timeout (in sec) for the
                                                      //  the main timing loop
    static    int MAX_PEAK_WIDTH = 75;   // timing peak should be found within
                                         //    the first MAX_PEAK_WIDTH bins
    static double MAX_OK_STDEV   = 4.0;  // max value of an ok/good peak stdev
    )
    static      String info;

  public static void main(String[] args)
  {
    int    i;
    int    maxIterations;
    String propertyStr;
    int    ticksPerMilliSecond = 1;
    __DEBUG_TIMING__(
    long   dt_time,  dt_time0;
    int    dt_delta;
    long   dtw_time, dtw_time0;
    int    dtw_delta;
    )
    __DEBUG_MEMORY__(
    //showMemory();
    System.gc();
    )

    __DEBUG_TIMING__(
    // check to see if the user wants a custom peak width
    propertyStr = System.getProperty("nSigma");
    if (propertyStr != null) {
      nSigma = 0.01f * Integer.parseInt(propertyStr);
      System.out.println("nSigma: " + formatFloat(nSigma));
    }
    )

    __DEBUG_TIMING__(
    // check to see if the user wants a custom timeout
    propertyStr = System.getProperty("TO");
    if (propertyStr != null) {
      timeoutSec = Integer.parseInt(propertyStr);
      System.out.println("timeoutSec: " + timeoutSec);
    }
    )

    // initialize the timer (calls and values supplied by the following macros)
    __TIMER_INIT__;
    ticksPerMilliSecond = __TICKS_PER_MS__;

    Object object;
    C_ORB orb = new C_ORB();
    __DEBUG__( System.out.println("ORB created!");)

    __SECURITY__(
    System.out.println( "\n*** " + orb.cipherEncrypt.toString() +
                        "  keyLen: __SEC_CIPHER_KEY_LEN__");
```

```
    // load the 'user' supplied key into the ORB
    byte[] orbKey = {10,11,12,13,14,15,16,17};
    try {
      orb.initCipherKey(orbKey);
    }
    catch (InvalidKeyException e) {
      System.out.println(e);
    }
    )
    __SECURITY_MD__(
    System.out.println("*** Using Message Digest: __SEC_MD_ALG__");
    )
    __SECURITY_MAC__(
    System.out.println("*** Using MAC: __SEC_MAC_ALG__ (__SEC_MAC_ALG_PARAMS__)");
    // load the 'user' supplied MAC key into the ORB
    byte[] macKey = {20,21,22,23,24,25,26,27};
    try {
      orb.initMacKey(macKey);
    }
    catch (InvalidKeyException e) {
      System.out.println(e);
    }
    )

    // Find the server to connect to (via a corbaloc)
    //
    propertyStr = System.getProperty("corbaloc");
    if (propertyStr != null) {
      __DEBUG__( System.out.println("Using 'corbaloc' property: "+propertyStr);)
      object = orb.corbaloc_to_object(propertyStr);
    }
    else {
      __DEBUG__( System.out.println("No 'corbaloc' property, using args[0]: " +
                                    args[0]);)
      object = orb.corbaloc_to_object(args[0]);
      //object = orb.string_to_object(args[0]);
    }
    __DEBUG__( System.out.println("corbaloc to object!");)

    // Get the number of iterations
    //
    maxIterations = 1000;  // The default number of iterations
    propertyStr = System.getProperty("cnt");
    if (propertyStr != null) {
      maxIterations = Integer.parseInt(propertyStr);
    }
    System.out.println("Iteration 'cnt' set to " + maxIterations);
    __DEBUG_TIMING__(
    if (maxIterations > DT_SIZE) {
        dt_binEvents = true;  // dt[i] == # of events that took i ms to complete
    } else {
        dt_binEvents = false; // dt[i] == delta time of event i
    }
    )

    timing.foo fooObj = timing.fooHelper.narrow(object);
    __DEBUG__( System.out.println("Narrow! finished");)
```

```
154
155     // configID contains info about the current MQC settings/options
156     String configID = "cfg:p" + Config.intPacketSize + ":";
157     __SECURITY__(
158         configID += orb.cipherEncrypt.toString() + "-__SEC_CIPHER_KEY_LEN__";
159     )
160     configID += ":__SEC_MAC_ALG__:__SEC_MD_ALG__";
161
162     long ssStartTime = 0;  // steady state start time
163     long ssTotalTime = 0;  // steady state total time
164     long startTime   = 0;  // start of "real" timing loop time
165     long totalTime   = 0;  // total time for the "real" timing loop
166
167     //
168     // STEADY STATE TIMING LOOP
169     //--------------------------
170     //
171     __DEBUG_MEMORY__(
172     //showMemory();
173     System.gc();
174     )
175
176     System.out.println("---Steady State Begin---");
177
178     __DEBUG_TIMING__(
179     // init variable needed to compute the stdev of time deltas of each call
180     for (i = 0;  i < DT_SIZE;  i++) {
181       dt[i] = 0;
182     }
183     for (i = 0;  i < DTW_SIZE;  i++) {
184       dtw[i] = 0;
185     }
186     )
187     ssStartTime = __GET_TIME__;
188     __DEBUG_TIMING__(
189     dt_time0 = __GET_TIME__;
190     )
191
192     __DEBUG_TIMING__(
193     int    dt_min = 999999999;
194     int    ss_min;
195     int    ss_max;
196     int    ss_upperLimit = 999999999;
197     )
198     int    ss_nLoop = __SS_LOOP_CNT__;
199     int    ss_jcnt  = __SS_ITERATIONS__;
200     for (int j = 0;  j < ss_nLoop;  j++) {
201         __DEBUG_TIMING__(
202         dt_sum  = 0;
203         dt_sum2 = 0;
204         dt_cnt  = 0;
205         ss_min  = 999999999;
206         ss_max  = 0;
207         dt_time0 = __GET_TIME__;
208         )
209         for (i = 0;  i < ss_jcnt;  i++) {
210             fooObj.bar(i);
```

```
            __DEBUG_TIMING__(
            dt_time  = __GET_TIME__;
            dt_delta = (int) (dt_time - dt_time0);
            if (dt_delta < ss_min) ss_min = dt_delta;
            if (dt_delta > ss_max) ss_max = dt_delta;
            if (dt_delta < ss_upperLimit) {
                dt_sum  += dt_delta;
                dt_sum2 += (long) dt_delta * (long) dt_delta;
                dt_cnt++;
            }
            dt_time0 = dt_time;
            )
        }
        __DEBUG_TIMING__(
        if (ss_min < dt_min) {
            dt_min = ss_min;
        }
        // semi-fragile code, probably should check that dt_cnt > 1.  -ADM
        dt_avg    = (float) dt_sum / dt_cnt;
        dt_stdev2 = (double)(dt_sum2 - (dt_sum * dt_sum)/dt_cnt) / (dt_cnt-1);
        if (dt_stdev2 >= 0.0 ) {
            dt_stdev = Math.sqrt(dt_stdev2);
        } else {
            System.err.println("Oops! Likely integer overflow -- dt_stdev2= " +
                       formatDouble(dt_stdev2) + " < 0.0 -- Setting stdev to " +
                       formatFloat((float)STDEV_ERR) + "!");
            dt_stdev = STDEV_ERR;
        }
        ss_upperLimit = (int) (dt_avg + nSigma * dt_stdev);
        System.out.println(" - " + (j+1)*ss_jcnt + ":   avg: " +
                           formatFloat(dt_avg) + "  stdev: " +
                           formatDouble(dt_stdev) +
                           "  min/max: " + ss_min + "-" + ss_max +
                           "  upL: " + ss_upperLimit);
        )
    }
    __DEBUG_TIMING__(
    float  dt_ssAvg   = (float) (dt_avg / ticksPerMilliSecond);
    double dt_ssStdev = dt_stdev / ticksPerMilliSecond;
    long   dt_ssCnt   = dt_cnt;
    )

    ssTotalTime = (__GET_TIME__ - ssStartTime);
    float ssAvgTime = (float) (ssTotalTime / ticksPerMilliSecond) /
                              (ss_nLoop * ss_jcnt);

    __DEBUG_TIMING__(
    System.out.println("---Steady State Info---");
    )
    System.out.println("Overall:  " + formatFloat(ssAvgTime) + "    n/a     " +
             (ss_nLoop * ss_jcnt) + "  " +
             formatFloat((float)ssTotalTime/(1000*ticksPerMilliSecond)) + "s");
    __DEBUG_TIMING__(
    System.out.println("  ->end:  " + formatFloat(dt_ssAvg) + "  " +
                       formatDouble(dt_ssStdev) + "  " + dt_ssCnt);
    )
    System.out.println("---Steady State End---" );
```

```
268
269     __DEBUG_TIMING__(
270     int dd = (int) (5.0 * (dt_ssStdev * ticksPerMilliSecond)) + 10;
271     if (dd > (int) (.2 * DT_SIZE)) {
272       dd = (int) (.2 * DT_SIZE);
273     }
274     dt_offset = dt_min - dd;
275     if (dt_offset < 0) {
276       dt_offset = 0;
277     }
278     )
279
280
281     //
282     // MAIN TIMING LOOP
283     //------------------
284     //
285     int  iterations = 0;
286     long stop_time;
287     for (int r = 0;  r < REPEAT_CNT;  r++) {
288         __DEBUG_TIMING__(
289         i = -1;
290         dt_delta  = -1;
291         dtw_delta = -1;
292         try {
293         )
294             __DEBUG_TIMING__(
295             // init variable needed to compute stdev of time deltas of each call
296             for (i = 0;  i < DT_SIZE;  i++) {
297                 dt[i] = 0;
298             }
299             for (i = 0;  i < DTW_SIZE;  i++) {
300                 dtw[i] = 0;
301             }
302             )
303
304             startTime = __GET_TIME__;
305             __DEBUG_TIMING__(
306             dt_time0  = startTime;
307             dtw_time0 = dt_time0 / 1000;
308             stop_time = dt_time0 + timeoutSec * 1000 * ticksPerMilliSecond;
309             )
310
311             int d;
312             iterations = maxIterations;
313             for (i = 0; i < iterations; i++) {
314                 fooObj.bar(i);
315                 __DEBUG_TIMING__(
316                 dt_time   = __GET_TIME__;
317                 dt_delta  = (int) (dt_time - dt_time0);
318                 dt_time0  = dt_time;
319                 dtw_time  = dt_time / 1000;
320                 dtw_delta = (int) (dtw_time - dtw_time0);
321                 dtw_time0 = dtw_time;
322                 if (dt_binEvents) {
323                     d = dt_delta - dt_offset;
324                     if (d < DT_SIZE) {
```

```
                    dt[d]++;
                } else {
                    dt[DT_SIZE-1]++;
                }
            } else {
                // store raw (unbinned) times into dt[]
                dt[i]  = dt_delta;
            }
            // wide events are always binned
            if (dtw_delta < DTW_SIZE) {
                dtw[dtw_delta]++;
            } else {
                dtw[DTW_SIZE-1]++;
            }
            if (dt_time > stop_time) {
                iterations = i;
                break;
            }
            )
        }
        totalTime = (__GET_TIME__ - startTime);
        float avgTime = (float) (totalTime/ticksPerMilliSecond)/iterations;
        float ssDelta = ssAvgTime / avgTime;

        __DEBUG_TIMING__(
        dt_stdev = findStdev(false, iterations);  // warning computes dt_cnt,
                                                  // dt_avg, dt_sum, dt_sum2
        dt_avg   /= ticksPerMilliSecond;
        dt_stdev /= ticksPerMilliSecond;
        )

        info = "@@@   Loop" + r + ":  " + (int)(totalTime/
               (1000*ticksPerMilliSecond)) +  "  " + formatFloat(avgTime);
        __DEBUG_TIMING__(
        info += "  " + formatDouble(dt_stdev) + "  " + iterations +
                "  @@  r" + formatFloat(ssDelta);
        )
        info += "  @@  " + configID;

        __DEBUG_TIMING__(
        dt_stdev  = findStdev(true, iterations);  // warning computes dt_cnt,
                                                  // dt_avg, dt_sum, dt_sum2
        dt_avg   /= ticksPerMilliSecond;
        dt_stdev /= ticksPerMilliSecond;
        float dt_ssDelta = dt_ssAvg / dt_avg;
        )

        System.out.println(info);   // print non-GC info

        __DEBUG_TIMING__(
        System.out.print("@@@ gcLoop" + r + ":  " +
                     (int)(totalTime/(1000*ticksPerMilliSecond)) + "  " +
                      formatFloat(dt_avg)+ "  " + formatDouble(dt_stdev) +
                      "  " + dt_cnt + "  @@  r" + formatFloat(dt_ssDelta));
        System.out.println("  @@  " + configID);
        )
```

```
382         __DEBUG_TIMING__(
383         }
384         catch (java.lang.ArrayIndexOutOfBoundsException e) {
385             System.err.println("i: " + i + " dt_offset/dt/dtw: " + dt_offset +
386                                "  " + dt_delta + "  " + dtw_delta);
387             dt_cnt   = -1;
388             dt_avg   = 999999;
389             dt_stdev = STDEV_ERR;
390         }
391         )
392 
393         __DEBUG_TIMING__(
394         r_cnt[r]   = dt_cnt;
395         r_avg[r]   = dt_avg;
396         r_stdev[r] = dt_stdev;
397         )
398     }
399 
400     __DEBUG_TIMING__(
401     System.out.println("---Statistical Info---");
402     System.out.print("Loop s:  " + formatFloat(ssAvgTime));
403     System.out.print("  " + formatDouble(dt_ssStdev) + "  " + dt_ssCnt);
404     System.out.println();
405 
406     dt_cnt   = r_cnt[0];
407     dt_avg   = r_avg[0];
408     dt_stdev = r_stdev[0];
409     for (int r = 0;  r < REPEAT_CNT;  r++) {
410         System.out.println("Loop " + r + ":  " + formatFloat(r_avg[r]) +
411                            "  " + formatDouble(r_stdev[r]) + "  " + r_cnt[r]);
412         if (r_avg[r] < dt_avg) {
413             dt_cnt   = r_cnt[r];
414             dt_avg   = r_avg[r];
415             dt_stdev = r_stdev[r];
416         }
417     }
418     System.out.println("@@@ Summary: " + formatFloat(dt_avg) + "  " +
419                        formatDouble(dt_stdev) + "  " +
420                        dt_cnt + " @@- " + configID);
421     )
422 
423     __DEBUG_MEMORY__(
424     showMemory();
425     )
426 
427     __SAJE__(
428     // hacks needed to cause the SaJe (Charade) environment to shutdown
429     // First, cause the server to shutdown
430     fooObj.bar(-1);
431     // then it is our turn to shutdown
432     shutDown();
433     )
434   }
435     __SAJE__(
436     /**
437      * shutDown -- Used on the SaJe platform to cause a S/W break
438      * point that will timeout a Charade 'run X' command.
```

```
     */
    static void shutDown() {
        boolean b = false;
    }
    )

    /**
     * formatFloat -- used to print a float with four decimal digits
     * (This routine is needed because CLDC does not support printing floats).
     *
     * @param f  floating point number to print
     */
    static String formatFloat(float f) {
        f += 0.00005;
        long fint = (long) f;
        long ffra = (long) (10000 * ((f + 1) - fint));
        StringBuffer ffraStr = new StringBuffer( String.valueOf(ffra));
        ffraStr.setCharAt(0, '.');
        return String.valueOf(fint) + ffraStr;
    }

    __DEBUG_TIMING__(
    /**
     * formatDouble -- used to print a double with six decimal digits
     * (This routine is needed because CLDC does not support printing doubles).
     *
     * @param d  double to print
     */
    static String formatDouble(double d) {
        d += 0.0000005;
        long dint = (long) d;
        long dfra = (long) (1000000 * ((d + 1) - dint));
        StringBuffer dfraStr = new StringBuffer( String.valueOf(dfra));
        dfraStr.setCharAt(0, '.');
        return String.valueOf(dint) + dfraStr;
    }
    )


    __DEBUG_TIMING__(
    /**
     * findStdev -- find the standard deviation of a set of point.
     *
     * @param nonGC   if true, computate the stdev of only the
     *                non-Garbage Collected, if false, then use all events
     * @param iCnt    number of non-binned events in dt[] (see dt_binEvents)
     * @return double the raw standard deviation (ie, non-scaled value).
     */
    static double findStdev(boolean nonGC, int iCnt) {
        //WARNING: global variables dt_cnt, dt_sum, dt_sum2 are all modified
        //within this routine (yes--an ugly hack...)

        int    i, iStart, iPeakStart, iStop;
        int    dt_max;
        int    dt_b[];  // bins
        double stdev2, stdev;
        String dt_info;
```

```
496
497         if (dt_binEvents) {
498             // bins already computed, just alias dt[]
499             dt_b   = dt;
500         } else {
501             // need to compute number of bins needed
502             dt_offset = 3600000;     // an hour (in ms)
503             dt_max    =       0;
504             for (i = 0;  i < iCnt;  i++) {
505                 if (dt[i] < dt_offset) dt_offset = dt[i];
506                 if (dt[i] > dt_max)    dt_max    = dt[i];
507             }
508             dt_offset--;
509             dt_max++;
510             if (dt_max > dt_offset + 2*DT_SIZE ) {
511                 dt_max = dt_offset + 2*DT_SIZE;
512             }
513             // create the bins
514             dt_b = new int[dt_max - dt_offset + 1];
515             // stuff the bins
516             for (i = 0;  i < iCnt;  i++) {
517                 __DEBUG__( if (nonGC) System.err.print(" " + dt[i]); )
518                 if (dt[i] - dt_offset < 2*DT_SIZE) {
519                     dt_b[dt[i] - dt_offset]++;
520                 } else {
521                     dt_b[2*DT_SIZE]++;
522                 }
523             }
524             __DEBUG__( if (nonGC) System.err.println(); )
525         }
526
527         // Initially, use the first MAX_PEAK_WIDTH non-zero channels to compute
528         // the average and stdev of the timing peak
529         iStart = 0;
530         while (dt_b[iStart] == 0  &&  iStart < dt_b.length-1) {
531           iStart++;
532         }
533         iPeakStart = iStart+1;
534         while (dt_b[iPeakStart] < 2  &&  iPeakStart < dt_b.length-1) {
535           iPeakStart++;
536         }
537
538         iStop = dt_b.length;
539         if (nonGC) {
540             // the last bin contains overflow values--discard it
541             iStop--;
542         }
543         if (iStop > iPeakStart + MAX_PEAK_WIDTH) {
544             iStop = iPeakStart + MAX_PEAK_WIDTH;
545         }
546         if (nonGC) {
547             System.err.println("iStart/iPeakStart/iStop: " + (iStart+dt_offset)+
548                        "  " + (iPeakStart+dt_offset) + "  " + (iStop+dt_offset));
549         }
550         int iStop0;
551         long dtb, dti;
552         double x, x2;
```

```
553
554         do {
555             iStop0 = iStop;
556             // sum up the dt[] bins
557             dt_cnt  = 0;
558             dt_sum  = 0;
559             dt_sum2 = 0;
560             for (i = iStart;  i < iStop;  i++) {
561                 dt_cnt  += dt_b[i];
562                 dtb = dt_b[i];
563                 dti = dt_offset + i;
564                 dt_sum  += dtb * dti;
565                 dt_sum2 += dtb * dti * dti;
566                 // check for overflowed values (they make dt_sum2 negative)
567                 if (dt_sum2 < 0) {
568                     System.err.println("Oops! Likely integer overflow problem.");
569                     System.err.println("dt_sum2 < 0: " + dt_sum2 +
570                                        " at i=" + (dt_offset + i));
571                 }
572             }
573 
574             // compute the average and its standard dev.
575             dt_avg  = (float)   dt_sum  / dt_cnt;
576             x  = (double) dt_sum;
577             x2 = (double) dt_sum2;
578             stdev2  = (x2 - (x * x)/dt_cnt) / (dt_cnt-1);
579             if (stdev2 >= 0) {
580                 stdev = Math.sqrt(stdev2);
581             } else {
582                 System.err.println("Oops! Likely integer overflow -- stdev2= " +
583                           formatDouble(stdev2) + " < 0.0 -- Setting stdev to " +
584                           formatFloat((float)STDEV_ERR) + "!");
585                 stdev = STDEV_ERR;
586             }
587             if (nonGC) {
588                 iStop = (int) (dt_avg + (nSigma * stdev) + 0.5) - dt_offset;
589                 if (iStop >= dt_b.length) {
590                     iStop = dt_b.length - 1;
591                 }
592                 if ((iStop >= iStop0) && (stdev > MAX_OK_STDEV)) {
593                     // Compute an alternate upper limit for the first (nonGC)
594                     // peak. The value (dt_avg - iStart) is *assumed* to be the
595                     // bottom (lower-time) tail of the nonGC peak. This means
596                     // that (dt_avg - iStart) should be another approximation
597                     // to the value of 3.5 * nSigma (of the nonGC peak). If this
598                     // value gives a smaller iStop use it.
599                     int iStop2 = (int) ((dt_avg - dt_offset) +
600                                         ((dt_avg - dt_offset) - iPeakStart));
601                     System.err.println("avg,iStop,iStop2: " +
602                                        formatFloat((float)dt_avg) + "  " +
603                                        (dt_offset+iStop) + "  " + (dt_offset+iStop2));
604                     if (iStop2 < iStop) {
605                         iStop = iStop2;
606                     }
607                 }
608                 if (iStop < iStart + 1) {
609                     iStop = iStart + 1;
```

```
                }
                System.err.println("x-y,avg,stdev,iStop: " +
                    (dt_offset+iStart) + "-" + (dt_offset+iStop0) + "  " +
                    formatDouble(dt_avg) + "  " + formatDouble(stdev) +
                    "  " + (dt_offset+iStop));
            }
        } while (iStop < iStop0);

        // Pretty-Print the data
        if (nonGC) {
            System.out.println("--- " + (dt_offset+iStart) + "  " +
                               (dt_offset+(iStop0-1)) + " ---non-GC---");
            printArray(dt_b, dt_offset);
            System.out.println("---Wide bins---");
            printArray(dtw, 0);
        }

        if (! dt_binEvents) {
            dt_b = null;    // release the dt_b array--it is no longer needed
        }

        return stdev;
    }

    /**
    * printArray -- a pretty printer for the dt/dtw arrays
    *
    * @param a     an array to print
    * @param offset  index offset of the array
    */
    static void printArray(int a[], int offset)
    {
        int     lastZero = 0;
        boolean inZeros  = true;
        System.out.println( offset + "  " + a[0]);
        for (int i = 1;  i < a.length;  i++) {
            if (a[i] > 0) {
                if (inZeros) {
                    inZeros = false;
                    if (i-1 > lastZero) {
                        System.out.println((offset+i-1) + "  " + a[i-1]);
                    }
                }
                System.out.println((offset+i) + "  " + a[i]);
            } else {
                if ( inZeros ) {
                    // do nothing
                } else {
                    inZeros  = true;
                    lastZero = i;
                    System.out.println((offset+i)+ "  " + a[i]);
                }
            }
        }
    }
    )
```

```
667     __DEBUG_MEMORY__(
668     static Runtime runtime = Runtime.getRuntime();
669
670     /**
671      * showMemory -- print our the free/used memory
672      */
673     static void showMemory() {
674         long tMem, fMem, uMem, fMem1, uMemMax, fMemMax;
675
676         tMem    = runtime.totalMemory();
677         fMem    = runtime.freeMemory();
678         fMemMax = fMem;
679         if (tMem == fMem) {
680             // TINI hack, tMem and fMem are reported as equal
681             tMem = 334240;  // MAGIC NUMBER appears to be the proper value
682         }
683         uMem    = tMem - fMem;
684         __DEBUG__( System.out.println("c.Memory (t/f/u): \t" +
685                                 tMem + " \t" + fMem + " \t" + uMem);)
686
687         System.gc();
688
689         for (int i = 0;  i < 100;  i++) {
690             fMem1 = runtime.freeMemory();
691             __DEBUG__( System.out.println("c.fMem: " + fMem1););
692
693             if (fMem1 > fMemMax ) {
694                 fMemMax = fMem1;
695             } else {
696                 if (i > 1) break;
697             }
698             try {
699                 Thread.sleep(500);
700             } catch (InterruptedException ie) {}
701         }
702         uMemMax = tMem - fMemMax;
703
704         System.out.println("@@@ c.Memory (t/f/u/du): \t" + tMem + " \t" +
705                         fMemMax + " \t" + uMemMax + " \t" + (uMemMax - uMem));
706     }
707     )
708 }
```

Client.java.m4

Figure A.2: Client.java.m4 Listing

A.3 Server.java.m4

The source code for the ‘Server.java.m4’ file is listed in Figure A.3. This file contains the baseline code needed to implement a CORBA servant object as well

as additional QoS related code that is compiled in to or out of the client application depending upon the macro definitions in the ‘Config_macros.m4’ file.

Server.java.m4

```
//include(‘Config_macros.m4’)

import mqc.*;
import mqc.holders.*;

import java.io.*;

__SECURITY_ORBKEY__(
//import java.security.InvalidKeyException;
import mqc.security.InvalidKeyException;
)

public class Server
{
  public static void main(String[] args)
  {
    __DEBUG_MEMORY__(
    //ShowMemory();
    System.gc();
    )

    S_ORB orb = new S_ORB();
    __DEBUG__( System.out.println("ORB created!");)

    __SECURITY__(
    System.out.println( "\n*** " + orb.cipherEncrypt.toString() +
                        "  keyLen: __SEC_CIPHER_KEY_LEN__");
    // load the ’user’ supplied key into the ORB
    byte[] orbKey = {10,11,12,13,14,15,16,17};
    try {
      orb.initCipherKey(orbKey);
    }
    catch (InvalidKeyException e) {
      System.out.println(e);
    }
    )
    __SECURITY_MD__(
    System.out.println("*** Using Message Digest: __SEC_MD_ALG__");
    )
    __SECURITY_MAC__(
    System.out.println("*** Using MAC: __SEC_MAC_ALG__ (__SEC_MAC_ALG_PARAMS__)");
    // load the ’user’ supplied MAC key into the ORB
    byte[] macKey = {20,21,22,23,24,25,26,27};
    try {
      orb.initMacKey(macKey);
    }
    catch (InvalidKeyException e) {
      System.out.println(e);
    }
    )

    __GROUPCOM__(
```

```
    // create a new groupcom object
    System.out.println("Creating GroupCom object");
    String strListenPort = args[0];
    String strHost = System.getProperty("host");
    if(strHost == null) {
      strHost = "localhost.localdomain";
    }
    int intListenPort = Integer.parseInt(strListenPort);
    orb.instanciateGroupCom(strHost, intListenPort);

    if (intListenPort != 10000) {
      String grpStr = System.getProperty("grp");
      if (grpStr == null) {
        if (args.length > 1) {
          grpStr = args[1];
        } else {
          grpStr = "localhost.localdomain:10000";
        }
      }
      System.out.println("Using 'grp' = " + grpStr);

      orb.joinGroup(grpStr);

      try {
        Thread.sleep(500);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    )


    POA rootPOA = new POA(orb, "RootPOA");
    fooImpl test = new fooImpl();

    Object object = rootPOA.servant_to_reference( test );

//    //-- Generate the IOR --
//    String ior      = orb.object_to_string(object);
//    __DEBUG__( System.out.println("object to string");)
//    System.out.println(ior);

    //--Generate the corbaloc
    String corbaloc = orb.object_to_corbaloc(object);
    __DEBUG__( System.out.println("object to corbaloc!");)
    System.out.println(corbaloc);

    __J2SE__(
    try {
//      FileWriter out = new FileWriter(new File("timing.ior"));
//      out.write(ior);
//      out.close();

      FileWriter out = new FileWriter(new File("timing.corbaloc"));
      out.write(corbaloc);
      out.close();
    }
```

```
110     catch(IOException e)
111     {
112       System.out.println("Error writing IOR/corbloc");
113       return;
114     }
115     )
116
117     orb.run();
118
119     __DEBUG_MEMORY__(
120     // Give the ORB time to start running/waiting for client connections
121     try {
122       Thread.sleep(500);
123     } catch (InterruptedException ie) {}
124     ShowMemory();
125
126     __DEBUG__(
127     for (;;) {  // continuous loop of ShowMemory values
128       try {
129         Thread.sleep(5000);
130       } catch (InterruptedException ie) {}
131       ShowMemory();
132     }
133     )
134     )
135   }
136
137
138   __CLDC__(
139   static void AllDone() {
140     // This function is needed so that a S/W break point can be set
141     // that will cause Charade to timeout of its 'run X' command
142   }
143   )
144
145
146   __DEBUG_MEMORY__(
147   static Runtime runtime = Runtime.getRuntime();
148
149   static void ShowMemory() {
150     long tMem, fMem, uMem, fMem1, uMemMax, fMemMax;
151
152     tMem    = runtime.totalMemory();
153     fMem    = runtime.freeMemory();
154     fMemMax = fMem;
155     if (tMem == fMem) {
156       // TINI hack, tMem and fMem are reported as equal
157       tMem = 334240;  // MAGIC NUMBER appears to be the proper value
158     }
159     uMem    = tMem - fMem;
160     __DEBUG__(
161         System.out.println("s.Memory (t/f/u): \t"+tMem+" \t"+fMem+" \t"+uMem);
162     )
163
164     System.gc();
165
166     for (int i = 0;  i < 100;  i++) {
```

```
167       fMem1 = runtime.freeMemory();
168       __DEBUG__( System.out.println("s.fMem: " + fMem1););
169
170       if (fMem1 > fMemMax ) {
171         fMemMax = fMem1;
172       } else {
173         if (i > 1) break;
174       }
175       try {
176         Thread.sleep(500);
177       } catch (InterruptedException ie) {}
178     }
179     uMemMax = tMem - fMemMax;
180
181     System.out.println("@@@ s.Memory (t/f/u/du): \t" +
182                        tMem + " \t" + fMemMax + " \t" + uMemMax +
183                        " \t" + (uMemMax - uMem));
184   }
185   )
186 }
```
Server.java.m4

Figure A.3: Server.java.m4 Listing

A.4 Makefile.mqcc

The ‘Makefile.mqcc’ file, shown in Figure A.4, defines several targets used during MicroQoSCORBA’s automated build process. The ‘idl’ target (see Line 10) runs MicroQoSCORBA’s IDL compiler. The ‘linux’, ‘tini’, and ‘saje’ targets build MicroQoSCORBA client and server applications for these three hardware platforms in our evaluation testbed.

Server.java.m4
```
1  JAVA=$(JAVA_BIN)/java
2  JAVAC=$(JAVA_BIN)/javac
3
4  CASEDIR=$(MQC_HOME)/CASE
5
6  PACKAGE=timing
7  CONFIG_FILE=config.mqcc
8  CONFIG_DIR=../config
9
10 idl:
11     $(JAVA) -jar $(CASEDIR)/MqcIdlCompiler.jar $(PACKAGE).idl $(CONFIG_FILE)
12
13 # The following rule assumes that several test/evaluation configuration files
14 # exist in the CONFIG_DIR directory. Typing 'make -f Makefile.mqcc XYZ' file
15 # will copy the 'XYZ' config file into this directory and rename it config.mqcc.
```

```
# Next, executable files for Linux, TINI, and SaJe will be built and archived
# according to the ark/jark target rules that are in the base Makefile.

%:
    $(MAKE) cfgFile=$@ -f Makefile.mqcc saje
    $(MAKE) cfgFile=$@ -f Makefile.mqcc tini
    $(MAKE) cfgFile=$@ -f Makefile.mqcc linux

linux:
    echo ""
    echo "making Linux..."
ifdef cfgFile
    cp $(CONFIG_DIR)/$(cfgFile) $(CONFIG_FILE)
endif
    $(JAVA) -jar $(CASEDIR)/MqcIdlCompiler.jar $(PACKAGE).idl $(CONFIG_FILE)
    $(MAKE) clean
ifdef cfgFile
    $(MAKE) ARK=$(PACKAGE)/$(cfgFile) ark
else
    $(MAKE) jar
endif

tini:
    echo ""
    echo "making TINI..."
ifdef cfgFile
    cp $(CONFIG_DIR)/$(cfgFile) $(CONFIG_FILE)
endif
    $(JAVA) -jar $(CASEDIR)/MqcIdlCompiler.jar $(PACKAGE).idl $(CONFIG_FILE)
    sed -e "s/LINUX')/TINI')/" < Config_macros.m4 > Config_macros.m4.tmp
    mv Config_macros.m4.tmp Config_macros.m4
    $(MAKE) clean
ifdef cfgFile
    $(MAKE) ARK=$(PACKAGE)/$(cfgFile) tark
else
    $(MAKE) tini
endif

saje:
    echo ""
    echo "making SaJe..."
ifdef cfgFile
    sed -e "s/CLDC=false/CLDC=true/" < $(CONFIG_DIR)/$(cfgFile) > $(CONFIG_FILE)
else
    sed -e "s/CLDC=false/CLDC=true/" < $(CONFIG_FILE) > $(CONFIG_FILE).tmp
    mv $(CONFIG_FILE).tmp $(CONFIG_FILE)
endif
    $(JAVA) -jar $(CASEDIR)/MqcIdlCompiler.jar $(PACKAGE).idl $(CONFIG_FILE)
    sed -e "s/LINUX')/SAJE')/" < Config_macros.m4 > Config_macros.m4.tmp
    mv Config_macros.m4.tmp Config_macros.m4
    $(MAKE) clean
ifdef cfgFile
    $(MAKE) ARK=$(PACKAGE)/$(cfgFile) sark
else
    $(MAKE) saje
endif

```

```
73 gui:
74     $(JAVA) -jar $(CASEDIR)/MqcGui.jar
```
Server.java.m4

Figure A.4: Makefile.mqcc Listing

A.5 Makefile

MicroQoSCORBA's IDL compiler autogenerates an application specific 'Makefile' based upon the configuration options specified by the distributed system developer. The example 'Makefile' shown in Figure A.5 was generated for a baseline (i.e., no QoS support) MicroQoSCORBA application. The 'JAVA_SRC' variable (see Line 58) contains a list of all of the macro-enabled Java files that will be used. The 'vpath' directives (see Lines 74 and 76) direct *make* to the directories containing the needed MicroQoSCORBA library components. The '%.java' target (see Line 97) macro processes the '*.java.m4' files into application specific '*.java' files which are then used by the Java compiler. This 'Makefile' contains 'linux', 'tini', and 'saje' targets which are used to build the MicroQoSCORBA applications for each of these three hardware platforms. The 'ark', 'tark', and 'sark' targets build and archive applications for the linux, tini, and saje platforms, respectively.

Server.java.m4
```
1  #Makefile generated by MicroQoSCORBA code generator tool
2  #Make the necessary changes to the variables below
3
4  #Java compiler path
5  JAVA  := $(JAVA_BIN)/java
6  JAVAC := $(JAVA_BIN)/javac -target 1.1
7  JAR   := $(JAVA_BIN)/jar
8  #PC JAVAC := c:/jdk1.4.1/bin/javac -target 1.1
9
10 #Directory where you want the client and server classes to go
11 CLIENT_DIR := ./client
12 SERVER_DIR := ./server
13
```

```
#The path to were the MQC libraries are stored
PATH  := $(MQC_HOME)
MPATH := $(PATH)/mqc
JPATH := ./mqc

#The Home for the ARKived binaries, etc.  (Note: ARK is passed in from
#Makefile.mqcc and it is prepended with $(HOME). HARK_DOS is needed for SaJe)
HARK     := $(MQC_ARK)/$(ARK)
HARK_DOS := $(MQC_ARK_DOS)/$(ARK)

#Path to Third Party Tools, etc. (eg, aJile, TINI
TOOLS_DIR := $(MQC_TOOLS)
TINI_BIN  := $(TOOLS_DIR)/tini1.02e/bin
AJILE_DIR := $(TOOLS_DIR)/aJile

CLIENT_SRC = \
    Client.java \
#   CLIENT_SRC end

#Server sources
SERVER_SRC = \
    Server.java \
#   SERVER_SRC end

#Not neccessary to make any changes below this line

RM    := /bin/rm -f
FIND  := /usr/bin/find
M4    := /usr/bin/m4
WC    := /usr/bin/wc -c
XARGS := /usr/bin/xargs
MKDIR := /bin/mkdir -p
ECHO  := echo
COPY  := /bin/cp
TOUCH := /bin/touch
SED   := /bin/sed

CP = -classpath .:$(PATH):$(PATH)/jni
CLI_CP = $(CP):$(CLIENT_DIR)
SRV_CP = $(CP):$(SERVER_DIR)

BCP=-bootclasspath $(TOOLS_DIR)/J2mewtk/lib/midpapi.zip:\
        $(TOOLS_DIR)/aJile/Runtime_cldc/Rts

JAVA_SRC = \
    IORParser.java \
    Profile.java \
    Transport.java \
    ORB.java \
    S_ORB.java \
    POA.java \
    C_ORB.java \
    Delegate.java \
    GIOPDelegate.java \
    TCPTransport.java \
    S_TCPHandler.java \
    S_TCPTransport.java \
```

```
#   JAVA_SRC end

##-- where to look for the *.java.m4 files --
vpath %.java.m4 $(MPATH):$(MPATH)/protocols:$(MPATH)/transports:\
        $(MPATH)/ior:$(MPATH)/util##-- where to look for the *.java files --
vpath %.java    $(JPATH):$(JPATH)/protocols:$(JPATH)/transports:\
        $(JPATH)/ior:$(JPATH)/util

.PHONY: all jar linux tini saje ark tark sark clean realclean stats

all: $(CLIENT_DIR)/Client.class $(SERVER_DIR)/Server.class

jar: Client.jar Server.jar

linux: jar

##-- append tini-specific classpath info when building tini targets --
tini: CLI_CP = $(CLI_CP):$(TINI_BIN)/tiniclasses.jar

tini: Client.tini Server.tini

##-- append SaJe-specific classpath info when building SaJe targets --
saje: CLI_CP = $(CLI_CP):$(AJILE_DIR)/Runtime_cldc/Rts

saje: .SaJeClient .SaJeServer

%.java : %.java.m4
    $(MKDIR) $(patsubst $(PATH)%, .%, $(dir $<))
    $(RM) $(patsubst $(PATH)%.java.m4, .%.class, $<)
    $(M4) $< > $(patsubst $(PATH)%.m4, .%, $<)

%.tini : %.jar
    $(JAVA) -classpath $(TINI_BIN)/tini.jar TINIConvertor -f $^ \
            -d $(TINI_BIN)/tini.db -o $@

$(CLIENT_DIR)/mqc/Config.class: Config.java
    $(JAVAC) $(BCP) -classpath . -d $(CLIENT_DIR) $^

$(SERVER_DIR)/mqc/Config.class: Config.java
    $(JAVAC) $(BCP) -classpath . -d $(SERVER_DIR) $^

Client.java: Client.java.m4
    $(M4) $< > $@

$(CLIENT_DIR)/Client.class: $(CLIENT_SRC) $(JAVA_SRC) \
                            $(CLIENT_DIR)/mqc/Config.class
    $(JAVAC) $(BCP) $(CLI_CP) -d $(CLIENT_DIR) Client.java

Client.jar: $(CLIENT_DIR)/Client.class Client.MANIFEST.MF
    cd $(CLIENT_DIR);   $(JAR) -cvmf ../Client.MANIFEST.MF ../Client.jar \
                               -C .. config.mqcc *.class mqc timing

Client.MANIFEST.MF:
    $(ECHO) "Main-Class: Client" > $@

Server.java: Server.java.m4
    $(M4) $< > $@
```

```
128
129 $(SERVER_DIR)/Server.class: $(SERVER_SRC) $(JAVA_SRC) $(SERVER_DIR)/mqc/Config.class
130     $(JAVAC) $(BCP) $(SRV_CP) -d $(SERVER_DIR) Server.java
131
132 Server.jar: $(SERVER_DIR)/Server.class Server.MANIFEST.MF
133     cd $(SERVER_DIR);   $(JAR) -cvmf ../Server.MANIFEST.MF ../Server.jar \
134                                -C .. config.mqcc *.class mqc timing
135
136 Server.MANIFEST.MF:
137     $(ECHO) "Main-Class: Server" > $@
138
139 SaJeClient.ajp: SAJE_CLIENT.AJP
140     $(SED) -e "s#OUTPUT_DIR#$(HARK)_Client.saje#" < $< > $@
141
142 .SaJeClient: SaJeClient.ajp Client.jar
143     -$(JAVA) -cp $(AJILE_DIR)/JemBuilder/jembuilder.jar \
144              -Dlax.root.install.dir=$(AJILE_DIR) com.ajile.jembuilder.JemBuilder \
145              $(AJILE_DIR)/JemBuilder/library.properties SaJeClient.ajp
146     $(ECHO) "Converting Unix path names to DOS path names..."
147     $(COPY) $(HARK)_Client.saje/SaJeClient.sod $(HARK)_Client.saje/SaJeClient.sod.unix
148     $(SED) -e "s#$(HARK)#$(HARK_DOS)#" < $(HARK)_Client.saje/SaJeClient.sod.unix \
149                                        > $(HARK)_Client.saje/SaJeClient.sod
150     $(COPY) $(HARK)_Client.saje/build.jcf $(HARK)_Client.saje/build.jcf.unix
151     $(SED) -e "s#$(HARK)#$(HARK_DOS)#" < $(HARK)_Client.saje/build.jcf.unix \
152                                        > $(HARK)_Client.saje/build.jcf
153     $(TOUCH) $@
154
155 SaJeServer.ajp: SAJE_SERVER.AJP
156     $(SED) -e "s#OUTPUT_DIR#$(HARK)_Server.saje#" < $< > $@
157
158 .SaJeServer: SaJeServer.ajp Server.jar
159     -$(JAVA) -cp $(AJILE_DIR)/JemBuilder/jembuilder.jar \
160              -Dlax.root.install.dir=$(AJILE_DIR) com.ajile.jembuilder.JemBuilder \
161              $(AJILE_DIR)/JemBuilder/library.properties SaJeServer.ajp
162     $(ECHO) "Converting Unix path names to DOS path names..."
163     $(COPY) $(HARK)_Server.saje/SaJeServer.sod $(HARK)_Server.saje/SaJeServer.sod.unix
164     $(SED) -e "s#$(HARK)#$(HARK_DOS)#" < $(HARK)_Server.saje/SaJeServer.sod.unix \
165                                        > $(HARK)_Server.saje/SaJeServer.sod
166     $(COPY) $(HARK)_Server.saje/build.jcf $(HARK)_Server.saje/build.jcf.unix
167     $(SED) -e "s#$(HARK)#$(HARK_DOS)#" < $(HARK)_Server.saje/build.jcf.unix \
168                                        > $(HARK)_Server.saje/build.jcf
169     $(TOUCH) $@
170
171 ark: linux
172     $(COPY) Client.jar  $(HARK)_Client.jar
173     $(COPY) Server.jar  $(HARK)_Server.jar
174     echo "Computing Client side stats..."                >  $(HARK)_stats
175     $(FIND) $(CLIENT_DIR) -name "*class" | $(XARGS) $(WC) >> $(HARK)_stats
176     echo "Computing Server side stats..."                >> $(HARK)_stats
177     $(FIND) $(SERVER_DIR) -name "*class" | $(XARGS) $(WC) >> $(HARK)_stats
178
179 tark: tini
180     $(COPY) Client.tini $(HARK)_Client.tini
181     $(COPY) Server.tini $(HARK)_Server.tini
182     echo "Computing TINI Client side stats..."           >  $(HARK)_stats_tini
183     $(FIND) $(CLIENT_DIR) -name "*class" | $(XARGS) $(WC) >> $(HARK)_stats_tini
184     echo "Computing TINI Server side stats..."           >> $(HARK)_stats_tini
```

```
185     $(FIND) $(SERVER_DIR) -name "*class" | $(XARGS) $(WC) >> $(HARK)_stats_tini
186     echo "Computing TINI stats..."                        >> $(HARK)_stats_tini
187     $(WC) Client.tini Server.tini                         >> $(HARK)_stats_tini
188
189 sark: saje
190     $(MKDIR) $(HARK)_Client.saje
191     $(MKDIR) $(HARK)_Server.saje
192     $(COPY) Client.jar  $(HARK)_Client.saje/Client.jar
193     $(COPY) Server.jar  $(HARK)_Server.saje/Server.jar
194     echo "Computing SaJe Client side stats..."                >  $(HARK)_stats_saje
195     $(FIND) $(CLIENT_DIR) -name "*class" | $(XARGS) $(WC)     >> $(HARK)_stats_saje
196     echo "Computing SaJe Server side stats..."                >> $(HARK)_stats_saje
197     $(FIND) $(SERVER_DIR) -name "*class" | $(XARGS) $(WC)     >> $(HARK)_stats_saje
198     echo "Computing SaJe BIN stats..."                        >> $(HARK)_stats_saje
199     $(FIND) $(HARK)_Client.saje -name "*bin" | $(XARGS) $(WC) >> $(HARK)_stats_saje
200     $(FIND) $(HARK)_Server.saje -name "*bin" | $(XARGS) $(WC) >> $(HARK)_stats_saje
201
202 clean:
203     $(RM) Client.jar Server.jar
204     $(RM) Client.tini Server.tini
205     $(RM) SaJeClient.ajp .SaJeClient
206     $(RM) SaJeServer.ajp .SaJeServer
207     $(RM) Client.java Server.java
208     $(FIND) . -name "*.class"       | $(XARGS) $(RM)
209     $(FIND) $(JPATH) -name "*.java" | $(XARGS) $(RM)
210
211 realclean: clean
212     $(FIND) . -name "*.jar"     | $(XARGS) $(RM)
213     $(FIND) . -name "*.tini"    | $(XARGS) $(RM)
214     $(FIND) . -name "*.sod"     | $(XARGS) $(RM)
215     $(RM) Client.MANIFEST.MF Server.MANIFEST.MF
216     $(RM) *.corbaloc *.ior
217     $(RM) -R $(CLIENT_DIR)
218     $(RM) -R $(SERVER_DIR)
219     $(RM) -R $(JPATH)
220
221 stats:
222     echo "Computing Client side stats..."
223     $(FIND) $(CLIENT_DIR) -name "*class" | $(XARGS) $(WC)
224     echo "Computing Server side stats..."
225     $(FIND) $(SERVER_DIR) -name "*class" | $(XARGS) $(WC)
226     echo "Computing TINI stats..."
227     $(WC) Client.tini Server.tini
228
229 #End of file
```
Server.java.m4

Figure A.5: Makefile Listing

APPENDIX B

FAULT TOLERANCE MECHANISMS

The overall architecture, design and implementation of MicroQoSCORBA were key contributions of this dissertation. A key part of this research was the architectural design of how fault tolerance mechanisms were to be designed and implemented into MicroQoSCORBA's overall architecture. However, the design and implementation of MicroQoSCORBA's fault tolerance subsystem was completed as part of Kevin E. Dorow's Masters Thesis [Dor02]. Because fault tolerance is a key QoS property, this appendix has been included in order to provide a better, overall understanding of MicroQoSCORBA's multi-property QoS support. For additional information on MicroQoSCORBA's fault tolerance mechanisms please refer to [Dor02, DB03]. The text for the rest of this appendix appears in a paper that has been submitted for publication (see [MDD+]).

B.1 Fault Tolerance

Most distributed applications require some level of fault tolerance in order to be successful. This is especially true with embedded distributed systems, since they are often mission critical components within larger systems (e.g., fly-by-wire systems for airplanes, anti-lock braking systems for cars). The following orthogonal fault-tolerant mechanisms have been incorporated into MicroQoSCORBA: temporal redundancy, spatial redundancy, value redundancy, failure detection, and

Table B.1: Fault Tolerance Mechanisms

Redundancy	**Reliability**	**Ordering**
Temporal • Multiple Transmits **Spatial** • Multiple Channels **Value** • Checksums • CRC	**Group Communication** • Best Effort • Reliable • Uniform • Atomic **Failure Detection**	**Sender FIFO** **Causal** • Logical Timestamps **Total** • Sequencer based • Token based

group communication. They are summarized in Table B.1. Brevity will only allow for a brief overview of two of these mechanisms, namely temporal and value redundancy. For more information on MicroQoSCORBA's fault tolerance subsystem, please refer to [Dor02, MHD+02]. Fault tolerance performance results were presented in Section 7.4.

B.1.1 Temporal Redundancy

Temporal Redundancy is implemented in the communications channel. It tolerates up to (k) omission failures by allowing the application to specify a fixed number of automatic retransmissions (k+1). The number of omission failures that need to be tolerated is provided by the application at system startup, and the generated code performs the retransmissions automatically, without any handshaking. When a message is received, it is checked to see if it is a duplicate so that the message can be either disregarded or used in some comparison/voting mechanism

(depending on the user's selected configuration options). In order to avoid having to implement a large cache of messages within the ORB, a sender based sequence number is inserted into each message header so that the receiving ORB only needs to maintain a list of senders and the sequence numbers of the last few messages received.

B.1.2 Value Redundancy

Value Redundancy is implemented by including parity data, a checksum, or a cryptographically strong message digest or authentication code (see Section 5.5.2) in messages to verify that the content of the data, upon receipt, is correct. When selected, this property configures the ORB to send redundant information (e.g., checksums, error correcting codes) with each transmitted message. This information is processed as messages are received. If an error is detected (e.g., a mismatched checksum value) then an error message can be delivered to the client application. It is also possible to configure the ORB so that if a value error is detected an automatic retransmission is requested.

B.1.3 Redundancy Examples

Given our building example (see Section 5.2.1) each of the three previously mentioned redundancy mechanisms may potentially be used to provide increased tolerance to faults. Electrical interference in a large equipment room might cause the sporadic loss or corruption of messages. One solution is to thoroughly ground and shield the equipment and communications infrastructure. But, a less costly

approach in many environments to simply retransmit the messages (temporal redundancy). Another approach is to use error-correcting codes (i.e., value redundancy) to recover the original values from corrupted messages that are sent within the equipment room. The equipment room might also house a hot water heater that could explode if it over-heated. Thus, multiple data links (e.g., spatial redundancy) might be used to ensure that in the case of an accidental cable cut the temperature of the water heater would still be reported to the office building's main control room.

B.1.4 Group Communication System / Multicast

Group Communication supports sending a message to multiple recipients, with varying ordering and reliability requirements. The MicroQoSCORBA application program gives the user the ability to specify the level of reliability for the group communication (unreliable, best effort, reliable, uniform). The user may also specify the ordering requirements of the group communication (sender FIFO, causal, total). Based on these selections, the required support is built into the generated client and server code. Below is a listing and brief description of the specific types of group communication mechanisms that are supported. The details of the algorithms used in the implementations of these mechanisms can be found in [Bir97].

Nonuniform Failure-Atomic Multicast. The implementation is a three-phase ACK-based protocol in which messages are delivered immediately upon receipt.

Dynamically Uniform Failure-Atomic Multicast. The implementation is a four-phase ACK-based protocol in which message delivery does not occur until

all members have received the message.

FIFO Ordered Multicast-FIFO. Ordering of messages is implemented on top of the two Failure-Atomic multicast protocols described previously by adding an ordered message id for each sender to the message header and controlling the delivery of messages by the ids.

Causal Ordered Multicast. A Vector Timestamp protocol is employed on top of the two Failure-Atomic protocols to create this ordering (assuming that every message is multicast to all group members).

Totally Ordered Multicast. A moving sequencer algorithm is used on top of the two Failure Atomic protocols to supply total ordering (ignoring the causality requirement).

BIBLIOGRAPHY

[All02] Open Mobile Alliance. Wireless applicatoin protocol architecture specification. Online, 2002. See http://www.wapforum.org/what/technical.htm.

[BBH+02] David Bakken, Anjan Bose, Carl Hauser, Ioanna Dionysiou, Harald Gjermundrød, Lin Xu, and Sudipto Bhowmik. Towards more extensible and resilient real-time information dissemination for the electrical power grid. In *Proceedings of Power Systems and Communications Infrastructure for the Future*. International Institute for Critical Infrastructure, Beijing, China, September 2002.

[Bir97] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, May 1997.

[Bla00] Bob Blakley. *CORBA Security: An Introduction to Safe Computing with Objects*. Addison Wesley Longman, Inc., One Jacob Way, Reading, MA 01867, 2000.

[Bro95] Kraig Brockshmidt. *Inside OLE*. Microsoft Press, Redmond, WA, 2nd edition, May 1995.

[Bur03] William E. Burr. Selecting the advance encryption standard. *Security and Privacy*, 1(2):43–52, March/April 2003.

[CBM02] Geoff Coulson, Shakun Baichoo, and Oveeyen Moonian. A retrospective on the design of the GOPI middleware platform. *Multimedia Systems*, 8:340–352, 2002.

[Cry03] Cryptix. Online, 2003. See http://cryptix.org/.

[Dam02] Tarana R. Damania. Unreliable datagram support for configurable CORBA middleware. Master's thesis, Washington State University, July 2002. See http://microqoscorba.eecs.wsu.edu/Damania-Thesis.pdf.

[DAR] DARPA. BAA #02-12: Network of embedded software technology (NEST). DARPA BAA #02-12. See http://dtsn.darpa.mil/ixo/solicitations/nest/CBD_02-12.htm.

[DB03] Kevin E. Dorow and David E. Bakken. Flexible fault tolerance in configurable middleware for embedded systems. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, November 2003.

[Dor02] Kevin E. Dorow. Configurable fault-tolerance for MicroQoSCORBA. Master's thesis, Washington State University, December 2002.

[FHPR03] Alessandro Forin, Johannes Helander, Paul Pham, and Jagadeeswaran Rajendiran. Component based invisible computing. In *IEEE Real-Time Embedded Systems Workshop, part of the 22nd IEEE Real-Time Systems Symposium*, December 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework. Online, May 2003. See http://www.w3.org/TR/soap12-part1/.

[GLS01] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time corba scheduling service. *Real-Time Systems: the International Journal of Time-Critical Computing Systems*, 20(2), March 2001. Special issue on Real-Time Middleware, guest editor Wei Zhao.

[Hau01] Olav Haugan. Configuration and code generation tools for middleware targeting small, embedded devices. Master's thesis, Washington State University, December 2001. See http://microqoscorba.eecs.wsu.edu/Haugan-Thesis.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104. ACM Press, 2000.

[Jac03] JacORB. Online, 2003. See http://www.jacorb.org/.

[Jav03] Java. Online, 2003. See http://java.sun.com/.

[JCE03] Java cryptography extension. Online, 2003. See http://java.sun.com/products/jce/.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Request for Comments (RFC) 2104, February 1997. See ftp://ftp.rfc-editor.org/in-notes/rfc2104.txt.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33, June 2002. See http://www.ime.usp.br/~kon/papers/cacm02.pdf.

[KRL+00] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Magalhaes, and Roy Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Proceedings of the FIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, April 2000. See http://devius.cs.uiuc.edu/2k/dynamicTAO/.

[KV93] H. Kopetz and P. Verissimo. Real-time and dependability concepts. In S. Mullender, editor, *Distributed Systems*. ACM-Press, Addison-Wesley, 2nd edition, 1993.

[Law03] Wesley E. Lawrence. Temporal characterization of configurable middleware for embedded systems. Master's thesis, Washington State University, December 2003. Expected.

[Lib94] Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reily, December 1994.

[Loo01] Don Loomis. *The TINI™ Specification and Developer's Guide*. Addison-Wesley, June 2001.

[MBS] A. David McKinnon, David E. Bakken, and John C. Shovic. A configurable security subsystem in a middleware framework for embedded systems. *Computer Networks*. Submitted for publication.

[McK03a] A. David McKinnon. Interface definition language. In J. Urban and P. Dasgupta, editors, *Encylopedia of Distributed Computing*. Kluwer Academic Publishers, 2003. To appear.

[McK03b] A. David McKinnon. *Supporting Multiple Quality of Service Properties and Fine-grained Configurability in Middleware for Embedded Systems*. PhD thesis, Washington State University, December 2003.

[MDD+] A. David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic. A configurable middleware framework for small embedded systems that supports multiple quality of service properties. *Software—Practice and Experience*. Submitted for publication.

[MDD+03] A. David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic. A configurable middleware framework with multiple quality of service properties for small embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Network and Computing Applications (NCA2003)*, pages 197–204. IEEE Computer Society, April 2003.

[MHD+02] A. David McKinnon, Olav Haugan, Tarana R. Damania, Kevin E. Dorow, Wesley E. Lawrence, and David E. Bakken. A configurable middleware framework with multiple QoS properties for small embedded systems. TR 2002-37, School of EECS, Washington State University, October 2002. See http://microqoscorba.eecs.wsu.edu/TR-2002-37/.

[Mul03] Multe. Online, 2003. See http://http://www.unik.no/~multe.

[Nat99] National Institute of Standards and Technology (NIST). Data Encryption Standard (DES). Federal Information Processing Standards Publication 46-2, October 1999. See http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[Nat01] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, November 2001. See http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NBB+00] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr,

Morris Dworkin, James Foti, and Edward Roback. Report on the development of the advanced encryption standard (AES). NIST Report SAND2003-1215, NIST, Computer Security Division, October 2000. See http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf.

[NES03] New european schemes for signatures, integrity, and encryption. Online, 2003. See http://www.cryptonessie.org/.

[Obj00] Object Management Group. *Smart Transducers Interface Request For Proposals*. Object Management Group, Framingham, MA, December 2000. See http://www.omg.org/formal/2000-12-13.pdf.

[Obj01] Object Management Group. *Resource Access Decision (RAD) Specification, Version 1.0*. Object Management Group, Framingham, MA, April 2001. See http://www.omg.org/formal/01-04-01.pdf.

[Obj02a] Object Management Group. *Authorization Token Layer Acquisition Service (ATLAS), Version 1.0*. Object Management Group, Framingham, MA, October 2002. See http://www.omg.org/formal/02-10-01.pdf.

[Obj02b] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 3.0.2*, chapter 24–Secure Interoperability (CSIv2), pages 24–1–24–66. Object Management Group, Framingham, MA, July 2002. See http://www.omg.org/docs/formal/02-06-28.pdf.

[Obj02c] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 3.0.2*. Object Management Group, Framingham, MA, December 2002. See http://www.omg.org/formal/02-12-06.pdf.

[Obj02d] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 3.0.2*, chapter 23–Fault Tolerant CORBA, pages 23–1–23–106. Object Management Group, Framingham, MA, July 2002. See http://www.omg.org/docs/formal/02-06-59.pdf.

[Obj02e] Object Management Group. *Minumum CORBA, Version 1.0*. Object Management Group, Framingham, MA, August 2002. See http://www.omg.org/formal/02-08-01.pdf.

[Obj02f] Object Management Group. *Security Service Specification, Version 1.8*. Object Management Group, Framingham, MA, March 2002. See http://www.omg.org/formal/02-03-11.pdf.

[OMG03] Omg secsig security links. Online, 2003. See http://secsig.omg.org/old_secsig/links.html.

[PEK+00] Thomas Plagemann, Frank Eliassen, Tom Kristensen, Robert Macdonald, and Hans Olav Rafaelsen. Flexible and extensible qos management for adaptable middleware. In *Proceedings of International Workshop on Protocols for Multimedia Systems (PROMS 2000)*, October 2000. Cracow, Poland.

[PST+02] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. Spins: security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002.

[Rot02] rot13. The on-line hacker Jargon File, Version 4.3.3, September 2002. See http://www.catb.org/~esr/jargon/html/entry/rot13.html.

[SaJ03] SaJe. Online, 2003. See http://www.systronix.org/saje.

[SGH+02] Venkita Subramonian, Chris Gill, Huang-Ming Huang, Stephen Torri, Jeanna Gossett, Tom Corcoran, and Douglas Stuart. Fine-grained middleware composition for the boeing NEST OEP. In *OMG Real-Time and Embedded Distributed Object Computing Workshop*, July 2002. See http://www.omg.org/news/meetings/workshops/RT_2002_Workshop_Presentations.

[SGS01] Venkita Subramonian, Christopher Gill, and David Sharp. Towards a pattern language for networked embedded software technology middleware. In *OOPSLA 2001 Workshop Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems*, October 2001. See http://www.cs.wustl.edu/~cdgill/PDF/OOPSLA01_NEST.pdf.

[SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design and performance of real-time object request brokers. *Computer Communications*, 21:294–324, April 1998.

[Sno03] Deborah Snoonian. Smart buildings. *IEEE Spectrum*, pages 18–23, August 2003.

[SOO00] Douglas C. Schmidt, Carlos O'Ryan, and Ossama Othman. Applying patterns to develop a pluggable protocols framework for orb middleware. In Linda Rising, editor, *Design Patterns in Communications*. Cambridge University Press, second edition, 2000.

[Sta98] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, Upper Saddle River, New Jersey 07458, second edition, July 1998.

[STB86] Richard E. Schantz, Robert H. Thomas, and Girome Bono. The architecture of the CRONUS distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259, May 1986.

[Sys03] Systronix. Online, 2003. See http://www.systronix.org/.

[TAO03] TAO. Online, 2003. See http://www.cs.wustl.edu/~schmidt/TAO.html.

[Ten00] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, 2000.

[TIN03a] TINI. Online, 2003. See http://www.ibutton.com/TINI.

[Tin03b] Tinypk. Online, 2003. See http://www.is.bbn.com/projects/lws-nest/.

[UIC03] Universal interoperable core. Online, 2003. See http://www.ubi-core.com/index.html.

[Ver03] Vertel. Online, 2003. See http://www.vertel.com/.

[WEB03] Wireless embedded systems. Online, 2003. See http://webs.cs.berkeley.edu.

[WS02] Anthony D. Wood and John A. Stankovic. Denial of service in sensor networks. *Computer*, 35(10):54–62, October 2002.

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theorey and Practive of Object Systems (Special Issue on CORBA and the OMG*, 3(1), April 1997. See http://www.dist-systems.bbn.com/papers/1997/TAPOS.