A Configurable Middleware Framework for Small Embedded Systems that Supports Multiple Quality of Service Properties *

A. David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164-2752

Abstract

The majority of microprocessors manufactured in recent years have been deployed in embedded systems, often with real-time 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 paper 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 both device and application program constraints. This paper also describes the architectural taxonomy that was developed to specify and implement these constraints, the multiple Quality of Service domains that MicroQoSCORBA supports, and it presents an evaluation of our working framework. Our evaluation results illustrate the need to balance tradeoffs between application design, hardware resource constraints, and desired levels of multiple Quality of Service constraints.

1 Introduction

Real-time and embedded systems use over 95% of all microprocessors produced. Traditionally, each embedded system was a stand-alone device. But, as networking technology has become faster and also achieved commodity pricing, these stand-alone applications are becoming network aware. Furthermore these stand-alone applications are evolving into distributed real-time and embedded systems [1]. The sheer size of the embedded market is

*This research was supported in part by two Cisco University Research Program donations and grant NSF-CISE EHS-0209211 from the National Science Foundation’s Embedded and Hybrid Systems program.
staggering with more than 11 billion parts per year. This marketplace includes a wide range of devices and environments, from cell phones and small kitchen appliances to large jumbo-jets and ocean-going vessels. The range of application programs that run on these devices is even wider, since for example, a given embedded chipset can be used for many different applications for different customers.

Middleware, such as CORBA, has proven to be a valuable tool in dealing with the many facets of distributed systems [2]. Middleware’s advantages for standard enterprise systems are equally important for embedded systems development. In particular, CORBA provides an excellent object-oriented and platform-neutral middleware architecture for developing distributed applications [3]. It provides an elegant separation of a system’s interface from its implementation, along with its encapsulation of communication protocols and processing environments. Furthermore, CORBA provides a rich set of services and industry-specific APIs. This allows not only for language independence and a high-level programming building block, but can also mask heterogeneity and allow for many different implementation configurations and optimizations that can be employed transparently to the middleware client. Many developers are applying their existing middleware frameworks and tools to the emerging distributed, real-time and embedded systems market, but with varying degrees of success. Most high-end workstations have more onboard CPU cache than some embedded systems have available memory in both RAM and ROM. Thus, general-purpose frameworks developed for these memory-rich environments often fail to scale down to memory-starved environments. Other solutions have been developed for small, embedded devices, but typically they are point solutions, supporting one hard-coded set of constraints and tradeoffs. They thus do not have the flexibility to cover the wide space of the embedded systems market. In many cases they also fail to provide an embedded systems application designer with the ability to set constraints appropriate for a specific application’s functionality and target hardware.

We have designed and implemented MicroQoSCORBA, a middleware framework with a supporting tool set that can scale down to memory-starved environments. MicroQoSCORBA allows one to tailor middleware to both hardware properties and to an application’s configuration requirements. Limited memory is only one of the many facets of small, embedded devices. Thus, MicroQoSCORBA has been designed to address other systems facets such as fault tolerance, security, power usage, and system performance. Some of these facets are Quality of Service (QoS) properties and others are resource constraints.

As MicroQoSCORBA’s architecture was being defined to encompass each of these facets, it was determined that many traditional distributed systems architectural concepts (e.g., “client”, “server”, “push”, “pull”) were defined at too coarse a level and were not orthogonal enough for the fine-grained configuration control required for very small environments. To that end, we refined these concepts and broke them down into their respective orthogonal components.

The contributions of this paper are:

- A refined taxonomy of distributed system architectural concepts.
- A middleware framework that can be tailored for both application and device constraints, with a fine granularity of configuration constraints.
• A middleware framework for embedded systems that supports both functional and non-functional Quality of Service properties, which are fault tolerance, security, and timeliness.

• An initial evaluation of the trade-offs between multiple Quality of Service properties within an embedded system.

The rest of this paper is organized as follows: first a motivating example is introduced; then in Section 3 MicroQoSCORBA’s architectural taxonomy is described, followed by discussions of application lifecycle epochs and multi-property Quality of Service support in Sections 4 and 5, respectively; after that, MicroQoSCORBA’s baseline architecture is presented in Section 6; our development environment is overviewed in Section 7; followed by our experimental evaluation in Section 8; related work is discussed in Section 9; and we conclude in Section 10.

2 Motivating Examples

The distributed and real-time embedded systems market is very broad and encompasses a wide range of applications and hardware platforms. In fact, the market is simply too broad to be represented by only a few examples, but in order to help focus the discussion of our middleware framework, we present the following three examples. The first two examples will be briefly presented, and the third will be covered in more depth because it will be referred to later in this paper.

2.1 Distributed Sensor Networks

Distributed sensor networks are not completely new, but the availability of low cost hardware that can communicate over wireless links is changing the landscape of sensor networks. Rather than just having static networks of high cost, stationary sensors, it is now feasible to consider disposable sensor networks. For example, sensors can be deployed from a remotely piloted vehicle to gather information about a dangerous chemical spill. With this kind of disposable sensor application in mind, one will naturally want to deploy devices that are as resource-constrained (i.e., inexpensive) as possible. MicroQoSCORBA was designed so that an application’s constraints can be used to help avoid selecting unnecessarily over provisioned hardware, thus saving overall costs.

2.2 Alarm and Monitoring Systems

A large number of embedded devices are deployed in monitoring and controlling the health of real-world systems. When faults are detected, alarms are raised so that an operator can take corrective action. Examples of these types of applications range from simple laboratory supervisory control and data acquisition (SCADA) systems, to command and control of large items such as ocean-liners and cargo ships. Many of these systems are developed in high unit cost, low volume markets. Thus, MicroQoSCORBA will not be desired so much for its ability to save on hardware costs, but for its ability to provide middleware solutions that
can enhance software development productivity. Additionally, since MicroQoSCORBA has been designed to be small, certifying its performance within safety critical systems should be simpler than certifying a “large” standard middleware object request broker (ORB).

### 2.3 Building Automation and Control

Consider a large office building. Its designers must consider the building’s purpose as well as the comfort and safety of the building’s occupants. The building will have hundreds, if not thousands, of individual rooms and offices, on dozens of floors. It will also have mechanical rooms for the building’s various utilities (e.g., electrical power, water, heating). The designers must consider the associated construction and maintenance costs. Another concern, applicable to a large construction company, will be to maximize the reuse of common technologies across the many similar, but yet one-of-a-kind buildings that the company constructs and maintains. Naturally, many trade-offs will exist and the company’s designers must balance concerns in all of the above areas.

While designing this large office building, numerous design trade-off questions must be asked. For example, should each office be equipped with “smart” embedded devices that can reason about the comfort and safety of the room’s occupant(s)? Or would it be more effective, if each floor had a “smarter”, more resource-rich device that gathered information from each office’s very resource-constrained devices? If the latter, then these dedicated floor level devices will be acting as a client to a centralized building server, and also as a server to the individual office systems. Furthermore, can these office sensors be classified as clients, if they were “optimized” to the point where they can only respond to queries (e.g., “What is the office temperature?”), but not generate any requests to the “smarter” floor-wide embedded system?

Another set of design questions that must be asked relate to various non-functional concerns, such as the building system’s tolerance to faults, security, and timeliness. It is very common for many critical systems to be designed with backups in mind (e.g., a backup power generator). Thus, a middleware framework should be designed so that it can also supply a level of fault tolerance for the critical systems within the building. Security of the control systems is also important because, for example, some managers will not be willing to have their room temperature set by a disgruntled employee.

Considering the many trade-offs that can occur within this example and others, resulted in the conclusion that a very fine-grained architectural taxonomy was needed. A fine-grained taxonomy allows one to more easily configure deeply embedded solutions. It is also needed because the use of “barely smart” sensors within a middleware framework requires a rethinking of what it means to be a node within a distributed system.

### 3 Architectural Taxonomy

The embedded applications space is very diverse in terms of interaction with other applications, the type and size of data transferred, whether or not the application is active or passive, etc. Moreover, the space of embedded devices also varies widely in terms of RAM, ROM, power consumption, etc. While the combined space of application and hardware
characteristics is quite wide, a given embedded system is typically designed with one task in mind, often a relatively simple one. Furthermore, a given system is often designed for a specific target hardware platform. Together, the application’s purpose and target platform dictate the tradeoffs and constraints that are presented to the middleware framework, which must then configure itself in order to meet the system’s (often constrained) requirements for RAM, ROM, power, and QoS.

MicroQoSCORBA was designed for distributed, embedded applications, therefore it was prerequisite to categorize some of the useful fundamental facets of embedded systems in order to support the wide range of tradeoffs outlined above. As shown in Table 1, four key categories of interest were identified: Embedded Hardware, Roles, Software I/O, and subsets of CORBA’s Interface Definition Language (IDL). We now discuss these categories in the following subsections.

3.1 Embedded Systems Hardware

The choice of what hardware to support is a critical factor for an embedded application. A priori knowledge about hardware design choices allows an MicroQoSCORBA 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 choose 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 our office building in Section 2 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) 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 1 to indicate that the size of individual systems and the networks to which they belong will vary widely.

3.2 Role Definitions

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 Micro-
Table 1: Refined Middleware Architectural Taxonomy

<table>
<thead>
<tr>
<th>Embedded Hardware</th>
<th>Roles (Client/Server/Peer)</th>
<th>Software Input/Output</th>
<th>IDL Subsets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Composition</strong></td>
<td>Control Flow</td>
<td>Data Flow</td>
<td>Interaction Style</td>
</tr>
<tr>
<td>Homogeneous</td>
<td></td>
<td></td>
<td>Sync</td>
</tr>
<tr>
<td>Asymmetric</td>
<td></td>
<td></td>
<td>Async</td>
</tr>
<tr>
<td><strong>Hardware I/O Support</strong></td>
<td></td>
<td></td>
<td>Data Direction</td>
</tr>
<tr>
<td>Serial, Parallel, 1-wire, Ethernet, IrDA, Bluetooth, GSM, GPRS</td>
<td></td>
<td></td>
<td>Parallelism</td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td></td>
<td></td>
<td>Service Location</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Processing Capabilities</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit, 16-bit, 32-bit, ...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QoSCORBA 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, then 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.

3.3 Communication Support

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 2-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 our example building, inexpensive serial cables can interconnect each room sensor with a centralized floor specific gateway computer that then bridges this information out on to the rest of the building network.

3.4 IDL Subsetting

CORBA has a rich and powerful Interface Definition Language (IDL) [4] 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.

To conclude this section, the following example is presented to help motive the benefits of MicroQoSCORBA’s support for IDL subsetting. Consider the building example presented in Section 2, 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.

4 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 paper, so only a few key constraints are shown in Table 2.

One of the first things to be noticed about Table 2 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 [5], 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 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 exam-
Table 2: Lifecycle Time Epochs

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

4.2 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?

4.3 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 of library code.

4.4 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.

4.5 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.

5 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 systems 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.

5.1 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. They are summarized in Table 3. 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 [6, 7]. Fault tolerance performance results are included in Section 8.

5.1.1 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
Table 3: Fault Tolerance Mechanisms

<table>
<thead>
<tr>
<th>Redundancy</th>
<th>Reliability</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporal</td>
<td>Group Communication</td>
<td>Sender FIFO</td>
</tr>
<tr>
<td></td>
<td>• Multiple Transmits</td>
<td></td>
</tr>
<tr>
<td>Spatial</td>
<td>• Best Effort</td>
<td>Causal</td>
</tr>
<tr>
<td></td>
<td>• Multiple Channels</td>
<td>• Logical Timestamps</td>
</tr>
<tr>
<td>Value</td>
<td>• Reliable</td>
<td>Total</td>
</tr>
<tr>
<td></td>
<td>• Uniform</td>
<td>• Sequencer based</td>
</tr>
<tr>
<td></td>
<td>• Atomic</td>
<td>• Token based</td>
</tr>
<tr>
<td></td>
<td>Failure Detection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Checksums</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CRC</td>
<td></td>
</tr>
</tbody>
</table>

5.1.2 Value Redundancy

Value Redundancy is implemented by including parity data, a checksum, or a cryptographically strong message digest or authentication code (see Section 5.2.2) 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.

5.1.3 Redundancy Examples

Given our building example (see Section 2.3) 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.
5.1.4 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 [8].

*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).

5.2 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. An overview of our MicroQoSCORBA security taxonomy is shown in Table 4. The confidentiality, integrity, availability and accountability constraints that MicroQoSCORBA supports will be discussed in the following subsections. Security evaluation results are presented in Section 8. Additional information on the design, implementation, and evaluation of MicroQoSCORBA’s security subsystems is included in [9, 10].

5.2.1 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
Table 4: Security Design Space

<table>
<thead>
<tr>
<th>Confidentiality</th>
<th>Integrity</th>
<th>Availability</th>
<th>Accountability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>Message Digests</td>
<td>Service Continuity</td>
<td>Authentication</td>
</tr>
<tr>
<td>• Dedicated Network</td>
<td>• MD4/5</td>
<td>• See Fault Tolerance</td>
<td>• Physical Tokens</td>
</tr>
<tr>
<td>• Secure Network</td>
<td>• SHA1/2</td>
<td></td>
<td>• Shared Secrets</td>
</tr>
<tr>
<td>Encryption</td>
<td>Message Authentication Codes</td>
<td></td>
<td>• Passwords</td>
</tr>
<tr>
<td>• Symmetric Key</td>
<td>• HMAC</td>
<td>Service Continuity</td>
<td>• Challenge / Response</td>
</tr>
<tr>
<td>AES, DES, Rot13, ...</td>
<td>Error Control / Correction Codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Public Key</td>
<td>• CRC32</td>
<td>Disaster Recovery</td>
<td></td>
</tr>
<tr>
<td>RSA, Elliptic Curves,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Digital Signatures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• DSA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• RSA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 8.5.3, 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 [11], were implemented because because they consume few system resources (e.g., run-time memory) and execute quickly, see Section 8.5.3. But the level of confidentiality they provide is minimal. Other more cryptographically strong ciphers, such as the Advanced Encryption Standard (AES) [12] 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.
5.2.2 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 [13] message authentication codes based upon the previously listed message digests.

5.2.3 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 5.1) 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 [6]. This support for replicated servers can aid greatly in designing high availability, embedded systems.

As illustrated in [14], 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.

5.2.4 Accountability

The accountability column shown in Table 4 lists several properties, a few of which will not be supported within our framework, but yet are listed for completeness. Namely, MicroQoSCORBA will support neither delegation 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) [15]. 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 designed in 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.
5.3 Timeliness

Initial timeliness efforts within MicroQoS CORBA are directed towards the computation aspects of a generic real-time model, which separates input and output, communication and computing [16]. Thus, both spatial and temporal profiling support is being developed for MicroQoS CORBA’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 the application and system developer valuable insight for the development of real-time applications using MicroQoS CORBA, by delineating the temporal and spatial costs for the various configurations.

Future efforts 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 frame-work within MicroQoS CORBA that will aid the developer in building real-time applications. We also plan on adding support for a deterministic Ethernet protocol such as ARINC 664 [17].

6 MicroQoS CORBA Architecture

One of the key benefits of the MicroQoS CORBA framework is its ability to target a range of embedded devices. This is accomplished by exploiting some novel adaptations in the standard CORBA architecture as well as binding constraints during the various epochs of an application’s lifecycle (see Section 4). The architecture of MicroQoS CORBA is shown in Figure 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.

6.1 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.

MicroQoS CORBA’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 MicroQoS CORBA’s performance by removing support for unneeded IDL functionality (see Section 3.4).
Figure 1: MicroQoSCORBA Architecture
6.2 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 Sections 3 and 4, 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 8.4), 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.

6.3 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 [18], 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. 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 [19]. MicroQoSCORBA supports unreliable datagram support as well as two reliable datagram protocols, which are provided by wrapping standard datagrams within a reliability layer [20].

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.
7 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 MicroQoS-CORBA 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.

7.1 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. 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 [21].

7.1.1 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.

7.1.2 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.
7.1.3 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 (see Section 3.1). 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 [22] and heterogeneous hardware cannot be used together, since GIOPLite can only run on homogeneous hardware.

7.2 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 un-marshalling code to produce, and other choices that affect the configuration of the application code.

MicroQoSCORBA’s IDL compiler are not limited to just the 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 form a given application. This allows for a fine-grained control over the included library code or in other words is allows for the removal of unnecessary library methods. The IDL compiler also generates an application-specific set of macro definitions that are used to customize MicroQoSCORBA’s functionality. The purpose of these macros will be explained in the following subsection.
7.3 Building and Deploying the Application

Once the stubs and skeleton code has been 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 [23], 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 [24], the class files are processed by the JEMBuilder application and then downloaded to the SaJe board with the Charade application.

8 Experimental Evaluation

A working MicroQoSCORBA implementation has been developed and refined over several versions. MicroQoSCORBA was developed in Java because of its flexibility and cross-platform support. Several small devices have been deployed with JVMs (e.g., Dallas Semiconductor’s TINI board [23], aJile’s aJ-100 CPU [25]), which enabled MicroQoSCORBA to be tested and deployed on a wide variety of systems—from desktop workstations down to 8-bit CPUs. In the future support will be added for C/C++ so that MicroQoSCORBA can be deployed on devices too resource constrained to support a Java JVM.

This section is organized in to six subsections. The first presents the testbed hardware and software used in the experimental evaluation. After that the application used to evaluate MicroQoSCORBA’s performance is given in Section 8.2. Section 8.3 details how the performance results were measured and compared across the three testbed platforms used in this evaluation. Sections 8.4 and 8.5 present comparison of MicroQoSCORBA with other ORBs and between different MicroQoSCORBA configurations, respectively. This section then ends with the current status of MicroQoSCORBA’s implementation. For more evaluation details
and results, see [21, 20, 6, 26, 27, 9].

8.1 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 [28], and Dallas Semiconductors TINI board [29, 23]. 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 [30].

**SaJe.** Two SaJe boards from Systronix [24] 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. TINI-Convertor, 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.

8.2 Testbed Application

Our experimental evaluation was based upon executing the \text{foo.bar(...)} method given in the IDL specification shown in Figure 2. Client applications were implemented that made repeated \text{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 2.3) 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 7, 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 files. These two files had embedded in them macro definitions that could be 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 \text{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.

8.3 Cross-Platform Comparisons

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 weakness. 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 8.3.2 in order to help motivate the event filtering algorithm that is presented in Section 8.3.3.

8.3.1 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) was determined to be 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.

8.3.2 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 MicroQoSCORBA configuration are presented in the histograms of Figures 3, 4, 5, 6, and 7. 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 3 and 4, respectively. These histograms illustrate that although the vast majority of invocations are completed in less than 1-ms (see Figure 4). The overall, average elapsed time was 0.161 milliseconds (see Figure 3). 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, 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 5 and 6 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 ms (see Figure 5). But, the shape of the curve varies for the invocations with longer elapsed times. The data shown in Figure 6 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 3200 invocations, the results of
Figure 3: Linux Timing Performance Histogram with Microsecond Resolution

Figure 4: Linux Timing Performance Histogram with Millisecond Resolution
Figure 5: SaJe Timing Performance Histogram with Microsecond Resolution

Figure 6: SaJe Timing Performance Histogram with Millisecond Resolution
which are shown in Figure 7. 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.
8.3.3 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 3 (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.

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 paper. Thus, each of the results reported in this paper 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 typically 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.
Table 5: ORB Size Comparisons

<table>
<thead>
<tr>
<th>Size Metrics</th>
<th>MicroQoSCORBA</th>
<th>JacORB</th>
<th>TAO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client</td>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>Application</td>
<td>4,222 B</td>
<td>2,476 B</td>
<td>6,591 B</td>
</tr>
<tr>
<td>Java Memory</td>
<td>153.6 KB</td>
<td>160.6 KB</td>
<td>223.0 KB</td>
</tr>
<tr>
<td>Linux RSS</td>
<td>9.95 MB</td>
<td>9.62 MB</td>
<td>13.31 MB</td>
</tr>
</tbody>
</table>

8.4 ORB Comparisons

MicroQoSCORBA has successfully interoperated with other ORB implementations (e.g., JacORB [31] and TAO [32]). 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 [33]. First a memory usage comparison will be presented, followed by a run-time performance comparison.

The various ORB metrics shown in Table 5 were gathered on the Linux workstations for MicroQoSCORBA, JacORB, and TAO. The ORB comparison numbers are limited to Linux because JacORB is too 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 TAO. 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 6. 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 in Section 8.3. 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. 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.
Table 6: ORB Latency Comparison (ms)

<table>
<thead>
<tr>
<th>ORB</th>
<th>Linux</th>
<th></th>
<th>SaJe</th>
<th></th>
<th>TINI</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Filtered</td>
<td>Unfilrd</td>
<td>Filtered</td>
<td>Unfilrd</td>
<td>Filtered</td>
<td>Unfilrd</td>
</tr>
<tr>
<td>MicroQoSCORBA</td>
<td>0.170</td>
<td>0.171</td>
<td>4.162</td>
<td>4.425</td>
<td>248.6</td>
<td>256.8</td>
</tr>
<tr>
<td>JacORB</td>
<td>0.329</td>
<td>0.604</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>TAO</td>
<td>0.330</td>
<td>0.332</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

8.5 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 present, followed by their application sizes, after which their respective latencies will be presented.

8.5.1 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 (see Section 5.1.2) and temporal redundancy (see Section 5.1.1). The temporal redundancy settings tested were 2 and 4 temporally redundant retransmissions. A range of confidentiality (see Section 5.2.1) and integrity mechanisms (see Section 5.2.2) were tested. To evaluate MicroQoSCORBA’s value redundancy and integrity properties the following mechanisms were evaluated: a Parity-byte, a CRC32 code, and the message digests, MD5 and SHA1. For confidentiality, the Caesar [11], DES [34], TripleDES [34], and the Advanced Encryption Standard (AES) [12] ciphers were evaluated in electronic code book mode. Each of the aforementioned security and value redundancy configurations was also tested with temporal redundancy enabled, thereby providing a multi-property QoS evaluation. MicroQoSCORBA was configured to send messages via IIOP (CORBA GIOP version 1.2 over TCP/IP).

8.5.2 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
Table 7: MicroQoSCORBA Java Class File Size (bytes)

<table>
<thead>
<tr>
<th>Cipher &amp; Message Digest</th>
<th>Linux</th>
<th>SaJe</th>
<th>TINI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client</td>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>Baseline</td>
<td>63,607</td>
<td>61,062</td>
<td>259,077</td>
</tr>
<tr>
<td>Baseline w/Temp.Red.</td>
<td>59,478</td>
<td>58,617</td>
<td>258,437</td>
</tr>
<tr>
<td>Parity8</td>
<td>68,302</td>
<td>65,888</td>
<td>262,304</td>
</tr>
<tr>
<td>CRC32</td>
<td>68,687</td>
<td>66,275</td>
<td>262,506</td>
</tr>
<tr>
<td>MD5</td>
<td>73,547</td>
<td>71,137</td>
<td>265,871</td>
</tr>
<tr>
<td>SHA1</td>
<td>75,416</td>
<td>73,005</td>
<td>267,875</td>
</tr>
<tr>
<td>Caesar</td>
<td>74,635</td>
<td>72,088</td>
<td>263,436</td>
</tr>
<tr>
<td>DES</td>
<td>80,055</td>
<td>77,508</td>
<td>266,726</td>
</tr>
<tr>
<td>TripleDES</td>
<td>81,525</td>
<td>78,977</td>
<td>267,508</td>
</tr>
<tr>
<td>AES</td>
<td>85,182</td>
<td>82,634</td>
<td>270,606</td>
</tr>
</tbody>
</table>

in Table 7. For each platform in our testbed, both the client and server class file sizes are reported. The first column in Table 7, 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. 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, 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 appear to be high because the SaJe Java byte-code image contains all of the run-time support code for the SaJe board. In comparison, the TINI board requires an additional 448 Kb for its JVM and runtime environment [23] and the Linux JVM must link in multi-Mb 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.
Table 8: MicroQoSCORBA End-to-End Latencies (ms)

<table>
<thead>
<tr>
<th>Security Property</th>
<th>Linux</th>
<th>SaJe</th>
<th>TINI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No TR</td>
<td>TR2</td>
<td>TR4</td>
</tr>
<tr>
<td>Baseline</td>
<td>0.170</td>
<td>0.184</td>
<td>0.182</td>
</tr>
<tr>
<td>Parity</td>
<td>0.180</td>
<td>0.197</td>
<td>0.195</td>
</tr>
<tr>
<td>CRC32</td>
<td>0.178</td>
<td>0.194</td>
<td>0.195</td>
</tr>
<tr>
<td>MD5</td>
<td>0.199</td>
<td>0.218</td>
<td>0.218</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.204</td>
<td>0.232</td>
<td>0.231</td>
</tr>
<tr>
<td>Caesar</td>
<td>0.180</td>
<td>0.194</td>
<td>0.188</td>
</tr>
<tr>
<td>DES</td>
<td>0.201</td>
<td>0.226</td>
<td>0.227</td>
</tr>
<tr>
<td>TripleDES</td>
<td>0.238</td>
<td>0.294</td>
<td>0.297</td>
</tr>
<tr>
<td>AES-128</td>
<td>0.207</td>
<td>0.229</td>
<td>0.222</td>
</tr>
<tr>
<td>AES-192</td>
<td>0.202</td>
<td>0.222</td>
<td>0.222</td>
</tr>
<tr>
<td>AES-256</td>
<td>0.219</td>
<td>0.227</td>
<td>0.229</td>
</tr>
</tbody>
</table>

8.5.3 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 8. 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 8 also highlight the performance tradeoffs between weak and strong mechanisms. For example, a simple Parity byte can be 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 architexting and developing MicroQoSCORBA was to create
a testbed for multi-property QoS interactions. Simple as this evaluation example is, several results can already be observed. As already pointed out, 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.

8.6 Implementation Status

The initial analysis, design, and implementation of MicroQoSCORBA were completed in 2001 [21]. 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 [21]; 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 [6, 26] and a security subsystem [10, 9]. 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.

9 Related Work

We know of no CORBA or non-CORBA middleware framework that allows for 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 also allows constraints to be chosen at a much finer granularity than any other middleware framework of which we are aware, because it was designed 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 [35] removes dynamic interfaces and other features, but still only reduces the memory footprint by about half. The e*ORB$^\text{TM}$[36] framework by Vertel gets much smaller and closer to what is needed with
respect to memory footprint. However, it is a point solution that does not allow application
developers to tailor their constraints in ways appropriate for their applications to meet re-
source 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 “Uni-
versal Interoperable Core” or UIC-CORBA from Ubicore [37] (originally LegORB [38]) 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 run-
going 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 [39, 40] 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 [33, 32] with
respect to small, embedded devices. nORB also seeks to apply pattern languages within a
middleware framework.

Another related work is the OMG Smart Transducers Interface Request For Proposal
(RFP) [41]. Smart transducers are small, single purpose devices (e.g., sensors and actuators)
with some level of built in processing and communications support. This RFP seeks 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
subsystem.

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. Quality Objects (QuO) [5] allows adaptivity much later in the design lifecycle
than MicroQoSCORBA does, which is appropriate given it is not targeting small footprints.
MULTE [42] is a multimedia middleware platform that handles a range of latency and
bandwidth requirements [42, 43]. A reflective architecture is implemented in the Open-
ORB Python Prototype [44, 45]. 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 [46, 47].

MMLite [48] is a modular system architecture that allows a system to be built from
object-oriented components. Targeted at embedded systems, 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. Selecting different subsets of components can change the QoS properties of an
MMLite application. The primary QoS property that MMLite targets is real-time behavior.
This 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).

Several OMG CORBA standards cover various security aspects [49, 50, 51, 52, 53]. Al-
though a baseline MicroQoSCORBA configuration maintains interoperability with other
CORBA implementations, MicroQoSCORBA’s security subsystem bypasses conformance
with these standards in order to reduce resource usage on small, embedded systems. The
Java Cryptography Extension (JCE) [54] supports multiple security mechanisms via the dy-
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.

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) [55, 56]. The New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project [57] is another project that evaluated the strength of multiple security mechanisms. The NIST and NESSIE evaluations focused on strong security primitives. MicroQoSCORBA supports both strong and weak mechanisms (e.g., XOR-base encryption, CRC32 checksums) for use on embedded systems with limited computational capabilities. Additionally, these other efforts have not focused on end-to-end performance within a middleware framework.

The DARPA NEST program [58] is funding several related projects. The Berkeley Wireless Embedded Systems (WEBS) [59] has developed SPINS [60], 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 [61]. Wood and Stankovic present a security analysis of sensor networks in [14]. These projects are focused on security, but their focus is at either the networking or OS implementation—levels well below MicroQoSCORBA’s focus. In particular, none of these projects is providing security within a middleware framework.

10 Conclusions

This paper describes MicroQoSCORBA, a new CORBA framework and development environment that has been designed for small embedded devices. It has been designed from the bottom up to allow the middleware to be tailored to both the hardware and the application’s requirements. We described a refined taxonomy of distributed system architecture concepts that are especially appropriate for small, embedded systems, and then overviewed a configurable middleware framework that enables a developer to constrain a variety of facets of the middleware to best achieve system footprint and QoS requirements. Finally, we presented an evaluation of MicroQoSCORBA that showed its performance and support for multiple QoS properties. The evaluation also illustrated the tradeoffs involved when multiple QoS properties are composed into a single distributed embedded application.

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 allow developers the ability to make informed decisions regarding resource usage vs. performance tradeoffs based on their application code. The existing implementation of MicroQoSCORBA has been designed with the goal of supporting many QoS issues (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.

11 Acknowledgements

This research was supported in part by two Cisco University Research Program Donations and grant NSF-CISE EHS-0209211 from the National Science Foundation’s Embedded and Hybrid Systems Program. We thank Chris Gill for his valuable comments and insights on a previous version of this paper and Ioanna Dionysiou and Kjell “Harald” Gjermundrød for their feedback and suggestions. We thank the Zen project [62] for providing their IDL compiler source code to use as a basis for our IDL compiler. We also thank the Cryptix team [63] for the use of many of the security mechanism implementations contained within their open source JCE implementation. Thor Skaug, Eivind Næss, Kim Swenson, and Ryan Johnson provided valuable assistance with testbed debugging.

References


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


