A Configurable Security Subsystem in a Middleware Framework for Embedded Systems

A. David McKinnon *, David E. Bakken, John C. Shovic

School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164-2752

Abstract

Computer and network security is becoming increasingly important as both large systems and, increasingly small, embedded systems are networked. Middleware frameworks aid the system developer who must interconnect individual systems into larger interconnected, distributed systems. However, there exist very few middleware frameworks that have been designed for use within the embedded systems, which constitute the vast majority of CPUs produced each year, and none offer the range of security mechanisms required by the wide range of embedded system applications. This paper describes MicroQoSCORBA, a highly configurable middleware framework for embedded systems, and its security subsystem. It first presents an analysis of security requirements for embedded applications and what can and should be done in middleware. It then presents the design of MicroQoSCORBA's security subsystem and the wide range of mechanisms it supports. Experimental results for these mechanisms are presented for two different embedded systems and one desktop computer that collectively represent a wide range of computational capabilities.

Key words: Middleware, Security, CORBA, Embedded Systems

1 Introduction

Technological advances have been an enabling factor in the rapid increase in both the number and complexity of embedded systems that are being deployed. For some, this just means that the computing power that used to be

* Corresponding author.
Email address: mckinnon@acm.org (A. David McKinnon).

in our desktop workstations is now embedded in a small personal digital assistant (PDA). But the true advances in embedded systems are in those systems that we take for granted or just simply do not notice. For example, most individuals are not aware of the embedded processor in their microwave. Nor are they aware of the dozens (if not hundreds) of embedded processors contained within a new automobile.

As embedded processors have become smaller and more ubiquitous, new application domains are emerging, and wireless technologies have enabled a thriving cellular telephone marketplace. These and similar technologies are now being used to network together environmental sensors. These networks can be stationary and static–designed to measure a phenomena in a given area (e.g., air quality around a coal burning power plant). Alternately, they can be dynamically deployed in response to an environmental or man-made disaster (e.g., a hazardous chemical spill resulting from a train derailment).

Rapid advances in hardware technologies have exacerbated what is widely called the "software crisis." Software development skills and technologies have not kept pace with hardware advances and the emergence of new computing domains. For example cellular phones have been recalled because of software defects rather than hardware flaws [1]. Another concern is that many software developers and researchers are still focusing on traditional computing platforms and ignoring the embedded market, a market that includes 98% of all current computers and microprocessors [2]. Additional research must be focused on developing and deploying distributed systems because these systems are typically much harder to program than single-threaded applications that run on a workstation. Thus, even though the technology exists to build Internet-enabled refrigerators, few individuals have the desire or skills needed for such development efforts. Fewer still would be able to develop software that would allow these refrigerators to be safely and securely operated on the Internet.

Middleware frameworks are a key enabling technology that has great potential in alleviating the aforementioned "software crisis." Middleware is a software layer that exists below the application but above the operating system. Its purpose is to provide a common programming abstraction across all of the systems within a distributed system. This common middleware layer frees the software developer from low-level networking details and allows him or her to focus on an application's key details.

A software developer, aided by a middleware framework, is easily able to support the "functional" requirements of a distributed application, i.e., its application-specific business "logic." This is because the middleware development tools automatically generate the code that is needed to allow for the interoperability of heterogeneous, distributed systems. Thus, the software de-

veloper is able to focus on the objects and application program interfaces (API) within the system rather than on low-level networking details. Not only does this make the developer more productive, it also enables developers, who are not distributed systems experts, to deploy robust distributed systems.

In real-world applications, the "non-functional" properties, such as security and fault tolerance, within a distributed system are often just as important as the functional properties. For example, when withdrawing money from an automated teller machine (ATM) the user wants to receive her or his money in a timely manner. Likewise, access to the ATM should only be provided to authorized users in a secure manner. Thus we can see that within this ATM example domain, constraints exist for at least two non-functional properties (e.g., timeliness and security) and they both must be met. The level at which these non-functional requirements can be met varies from application to application (or perhaps even from day to day within a given, deployed application). Quality of Service (QoS) is a term that is used to describe the wide range of acceptable service levels for a given non-functional property.

Embedded systems exist in the real-world and so they often must achieve both functional and non-functional goals. However, few middleware frameworks have been developed that are suitable for the resource-constrained devices often deployed in embedded systems. Furthermore, few of the existing middleware frameworks support even one non-functional property let alone multiple non-functional properties. Additionally, while few programmers have expertise in distributed systems, many of those that do, are lacking expertise in programming with QoS mechanisms, especially low-level ones. Clearly, it is imperative that middleware frameworks support multiple QoS properties.

This paper describes the security subsystem of MicroQoSCORBA, a middleware framework developed from the ground up to support multiple non-functional properties within the embedded systems development environment.

The contributions of this paper are:

- an analysis of security requirements within the context of middleware frameworks targeting embedded systems.
- the design of a configurable security sub-system for a middleware framework.
- an experimental evaluation of performance and memory usage of security mechanisms across a wide range of hardware devices.

The rest of this paper is organized as follows: First, an overview of security as applied to embedded devices is given in Section 2; next a general overview of middleware as well as MicroQoSCORBA, our specific middleware framework, is presented in Section 3; Section 4 presents the security mechanisms supported in MicroQoSCORBA; evaluation results are presented in Section 5, related work is presented in Section 6, and we conclude in Section 7.

2 Security for Embedded Devices

This section presents an analysis of security requirements within the distributed embedded systems domain. First, a subsection discussing security requirements for embedded systems will be presented. The next subsection presents two motivating examples.

2.1 Embedded Systems Security Requirements

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

Embedded systems, by their very nature, have many application specific constraints imposed upon them that most desktop systems do not. In particular, embedded systems are often deeply coupled with their physical environment, whereas desktop computers operate in a virtual environment. In fact, this is one of Tennenhouse's three points in his call for more research on PROactive computing [2]. Namely, the 'P' in PROactive stands for *Getting physical*—exploring the coupling of devices with their environments; the 'R' is for *Getting real*—responding in real-time at faster-than-human speeds; and the 'O' is for *Getting out*—getting humans out of the decision making loop. Each of these three attributes are often desirable within embedded systems, but they are often not applicable within the standard desktop computing environment.

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

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

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

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

Denial of service attacks are even more acute within a sensor network. As Wood and Stankovic point out in [3], sensor networks may be attacked at the physical, link, network and routing, and transport layers. Furthermore, power must be taken into account when working with wireless, embedded devices, since one must assume that the adversary has a relatively unlimited power supply and thus could seek to exhaust the power budgets of the embedded devices [3].

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

Physical threats to embedded devices are not new because these devices have been deployed for many years as stand-alone devices—with the same physical security threats as they do now. What is new though are the virtual threats. As these traditionally stand-alone, isolated embedded devices are networked

they become subject to new threats. For example, a remote adversary can now access the devices without having to first acquire a physical presence. However, at times, little thought is put into analyzing this new threat vector. In part, some of this is due to the fact that few embedded systems developers have been trained with regard to computer and network security. Furthermore, these developers often lack tools that support security mechanisms, thus making it hard to design in security while still maintaining tight "time-to-market" deadlines and extremely const-conscious constraints.

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

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

2.2 Motivating Examples

The following two examples will be referred to later in this paper. The breadth of distributed systems is simply too broad to be represented by only two examples, however, these two examples do motivate the need or requirement that embedded systems middleware frameworks support multiple security QoS properties and levels. Both of these examples are discussed with respect to their respective requirements regarding the three classical security properties of confidentiality, integrity, and availability. These examples both illustrate that, even within a given application, multiple security QoS levels must be provided within real-world distributed embedded systems.

2.2.1 Building Automation and Control

"Smart buildings" is a broad term that encompasses a wide variety of building automation and control topics [4]. Consider a large office building. It may

have hundreds, if not thousands, of individual rooms and offices, on dozens of floors. The building may also have mechanical rooms for the building's various utilities (e.g., electrical power, water, heating). In order to keep this example short, only the heating and lighting within the building are discussed within this subsection.

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

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

The lighting information has different security characteristics than the temperature information. The ability to control the lights essentially is a cost saving measure and this data is not needed as input into a control system. Thus, the lighting information does not need to be sent with high levels of availability nor integrity. However, the lighting information does provide an indirect measure of whether a room is occupied. Thus, some individuals will first want to check, via the network, to see if the lights are on in someone's office before leaving their office and walking down the hall. However, some office workers might feel that it is an invasion of their privacy if their coworkers can easily find out if the are in their office or not. If this is the case, then the lighting information should be transmitted on the network in a confidential manner. The next question to answer is at what level of protection should this data be sent. Encrypting this data such that it would be secure for dozens of

years is excessive since the coworker could, within a minute, walk down the hall and determine if the worker was in or not.

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

2.2.2 Status Information within the Electrical Power Grid

Considered as a whole the electrical power grid is a very large distributed embedded system composed of many smaller embedded systems. A simplification of the grid is to view it as being composed of entities that either produce or consume power and entities that control power production and distribution. The timely dissemination of status information within the power grid is critical to its reliable operation [5].

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

Some data flows within the grid are very confidential. With deregulation of the electrical power grid, many power producers are competing to sell power to consumers. Thus, a power generation facility is at a distinct economic disadvantage if it can not protect information about its future contract negotiations, planned power plant operations, or other run-time sensor data that allows an adversary to ascertain these details. For example, some power production fa-

cility will desire to keep confidential information about which of its many power generators are in operation so as to withhold information about their equipment maintenance schedules from their competitors.

3 MicroQoSCORBA Overview

This section will present the background material needed to understand MicroQoSCORBA before proceeding on to a discussion of MicroQoSCORBA's security mechanisms in Section 4. First a brief overview of CORBA will be given, followed by discussions of our middleware taxonomy and the architecture of MicroQoSCORBA. After that, the overall multi-property Quality of Service design of MicroQoSCORBA will be presented before discussing MicroQoSCORBA's security subsystems. For additional information on MicroQoSCORBA please refer to [6,7].

3.1 CORBA Overview

CORBA (the Common Object Request Broker Architecture) is a middleware standard supporting distributed objects that has been developed by the Object Management Group [8]. Remote objects are accessed transparently via the use of Object Request Brokers (ORBs) that reside on the local and remote systems. During the initial design stages of an application, the APIs for an application's object oriented services are described in CORBA's Interface Definition Language (IDL) [8,9]. Next the CORBA IDL specification is used by an IDL compiler to generate stub and skeleton routines. On the client side, the auto-generated code for the stub object implements the same interface as the remote object and thus allows the client program to make what appears to be a local object invocation on the remote object. On the server side, the auto-generated skeleton routines provide a network transparency layer. CORBA provides programming language independence and a high-level programming building block that can mask system heterogeneity. Furthermore, different implementation configurations and optimizations can be employed transparently (i.e., maintaining the same interface) to CORBA clients and servers by reconfiguring either the stub or skeleton code generation or the ORB functionality. In particular, this clean separation of interface and implementation allows our middleware framework which we call MicroQoSCORBA to support the functional and non-functional requirements of a given distributed embedded system.

The two examples presented in Section 2.2 could be implemented in a variety of ways. One possible implementation for the building example presented in

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

Fig. 1. Example Building Automation CORBA IDL

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

Fig. 2. Example Power Grid CORBA IDL

Section 2.2.1 is declared via the CORBA IDL specification shown in Figure 1. Likewise, the IDL code shown in Figure 2 refers to the power grid example discussed in Section 2.2.2. In the interest of brevity, both of these IDL examples are abbreviated, but they do contain sufficient details for the purpose of this paper. Object methods are nested within interface and module blocks. Thus, in Figure 1 the code

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

indicates that in order for a client to invoke a getLightStatus method on a remote object it must pass in a room identification number and the remote object will return a boolean value to indicate if the room's light is on or off. CORBA IDL does not allow for the specification of non-functional parameters. Thus, even though the two examples have specific security requirements (see Sections 2.2.1 and 2.2.2), these requirements can not be specified in CORBA IDL. Rather than being a shortcoming, this separation of interface and implementation is beneficial because it allows the designer to change a middleware implementation without having to modify the application's internal code. This means, for example, that if the encryption mechanism used to invoke the getLightStatus method is determined to be too insecure, this mechanism can be transparently replaced by simply modifying the CORBA ORB configurations instead of having to redesign and rewrite large portions of the building control application.

3.2 Middleware Functional Configurability

Embedded systems span a wide range of devices, each with their own level of functionality. In order to scale our middleware framework down to very small devices MicroQoSCORBA was built upon a foundation of many small, and fine-grained composable modules. These components were organized into four broad functional categories that provided support for hardware, middleware roles, networking, and IDL mappings. Each component was isolated so that it could be configured in to or out of a given MicroQoSCORBA-based application.

A few examples of MicroQoSCORBA's fine-grained configurable design will now be briefly described. MicroQoSCORBA allows for the addition/removal of hardware specific code (e.g., byte ordering). An analysis of middleware roles showed that fine-grained components that isolated control flows, data flows, and interaction styles should be developed. With regard to networking, data representation (i.e., wire protocols), transport, and protocol modules were designed. And finally, support for all of CORBA's IDL functionality is typically not needed, so a fine-grained IDL compiler was designed that adds in support for only those data and parameter types that are required for a given application (e.g., the IDL code shown in Figures 1 uses only three data types (`boolean`, `short`, `float`) and no CORBA `out` or `inout` parameter types). For more details on these and other fine-grained configuration choices please refer to [7] for a description of our middleware design taxonomy. This taxonomy was used to guide the architecture and development of MicroQoSCORBA.

3.3 MicroQoSCORBA Architecture

One of the key benefits of MicroQoSCORBA is its ability to target a wide range of embedded devices. This is accomplished by exploiting the various constraints that can be bound in the application's development cycle, as well as by using some novel adaptations in the standard CORBA architecture. A high-level architectural diagram that shows the key components of MicroQoSCORBA is shown in Figure 3. In brief, this diagram shows that a message from the client first passes through an IDL compiler-generated *Stub*, a customized *ORB*, and then through MicroQoSCORBA's *Protocol* and *Transport* layers before travelling across a network to a server. On the server-side, the process continues with the message being demarshalled by the server-side *Transport* and *Protocol* layers and then being passed through the *ORB*, *POA*, and *Skeleton* code before reaching the application-specific code contained in the servant implementation. The server's reply transverses this path in reverse.

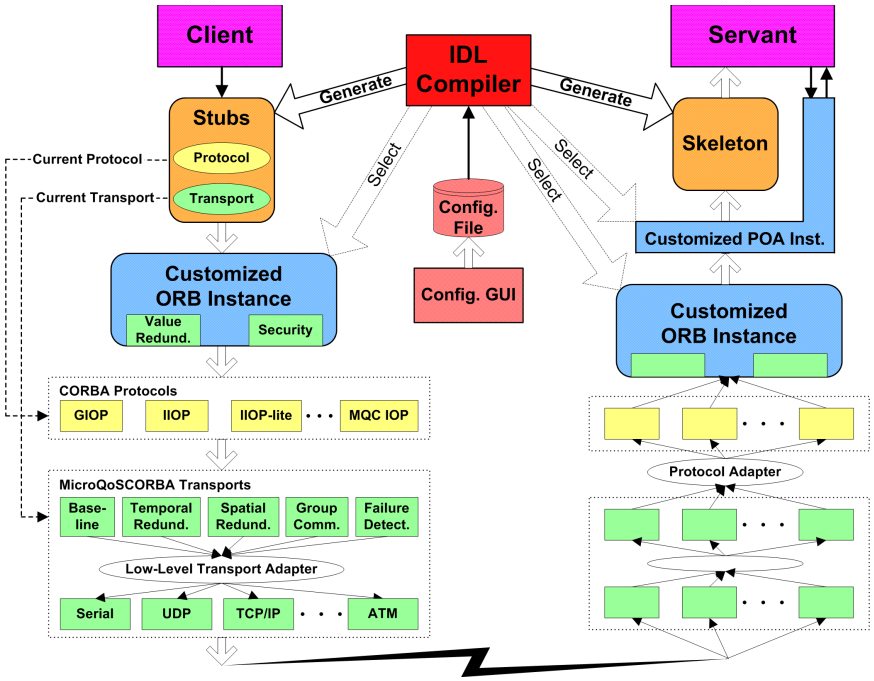

Fig. 3. MicroQoSCORBA Architecture

As can be seen in Figure 3, the MicroQoSCORBA's IDL compiler has a central architectural role because it is involved in generating or selecting many of the key components. Every CORBA development environment has an IDL compiler, but often these IDL compilers have overly-broad, all-encompassing designs. Rather than generate "one-size-fits-all" stub and skeleton code, the MicroQoSCORBA IDL compiler generates application and hardware specific stub and skeleton methods that are optimized for a customized ORB. Another subtlety of the MicroQoSCORBA architecture is that the IDL compiler selects and "hard codes" a given protocol and transport into the client-side stub routines. This removes ORB complexity and it eliminates linking unneeded protocol and transport code into the client-side application. The majority of MicroQoSCORBA's adapability is based upon constraining choices made during an application's initial design [6,7]. The results of these choices are leveraged by MicroQoSCORBA's IDL compiler.

A limited amount can be done by an IDL compiler to reduce (or improve) resource usage for a given application. Further improvements must be made by optimizing the ORB implementation. The fine granularity of MicroQoS-CORBA's design and implementation supports many ORB configurations and optimizations, each with their own associated profile of resource usage and performance. Most of MicroQoSCORBA's specialized ORB implementations maintain inter-operability with other standard ORB implementations. But, a developer may choose to forgo interoperability so that application or hardware specific constraints may be achieved (e.g., stripped down headers, non-standard data marshalling). The MicroQoSCORBA IDL compiler generates stub and skeleton code that is configured to use the desired ORB and

Portable Object Adapter (POA) implementations specified by the developer during the initial design of the embedded middleware application.

Many small, embedded devices have very limited communication abilities. For some applications, support for CORBA's Internet Inter-ORB Protocol (IIOP) may actually entail more code than is required for the application logic. Thus, support for one or more light-weight communication layers is needed. On the client side, the IDL generated stubs include a reference to the protocol and transport layer to be used. These references are given to the ORB so messages may be sent/received as needed. We note that the ORB could have used an abstract factory pattern [10], 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. MicroQoSCORBA supports very light-weight subsets of CORBA's communication's standards—subsets that will work because of design choices that allow for the removal of unneeded functionality (e.g., exceptions, data types such as structs and anys, and unneeded message types). Additional environment-specific transport layers will be developed as needed.

The fine-grained and composable architecture of MicroQoSCORBA is one of the key features of MicroQoSCORBA. This allows for the removal of unneeded system code resulting in cost savings because it allows MicroQoSCORBA to be deployed on devices with minimal memory footprints. The removal of unneeded system code, also removes unneeded functionality and any potential security vulnerabilities that may exist in the unused code.

3.4 Multi-Property Quality of Service

An embedded system's non-functional constraints are often as critical to the perceived success of the target application. For example, being able to withdraw money from an Automated Teller Machine (ATM) is a functional constraint. But, the responsiveness (i.e., timeliness) of the ATM and the desire to have only authorized users (i.e., security) withdraw money are also important. Furthermore, varying levels of service exist with which these non-functional constraints may be provided. Thus, one realizes that embedded applications must include and integrate multiple Quality of Service (QoS) constraints (e.g., security and timeliness). We have designed and implemented MicroQoSCORBA to support fault tolerance, security, and timeliness QoS constraints [6]. 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).

Most distributed applications require some level of fault tolerance in order to be successful. This is also true with embedded distributed systems, since in many cases they are mission critical (e.g., fly-by-wire systems for airplanes, anti-lock braking systems for cars). The following orthogonal mechanisms: temporal redundancy, spatial redundancy, value redundancy, failure detection, and group communication have been incorporated into MicroQoSCORBA so that MicroQoSCORBA can support a wide variety of fault tolerant requirements [7,11]. Although many fault tolerance mechanisms may be used to help ensure security, in particular system availability, these mechanisms are not the focus of this paper so we will instead refer you to [7,11] for further discussions of MicroQoSCORBA's fault tolerance mechanisms. A discussion of timeliness will also be omitted since the focus of this paper is on the security mechanisms of MicroQoSCORBA, which are discussed in the next section.

4 MicroQoSCORBA Security Subsystem

Security is one of the key non-functional properties supported by MicroQoSCORBA. In order to understand the current implementation of MicroQoSCORBA's security subsystem this paper will first present the overall design philosophy and goals with respect to MicroQoSCORBA's security subsystems. After that, the security design space and mechanisms are presented. This section then ends with a presentation of MicroQoSCORBA's current implementation.

4.1 Design Philosophy

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

First, MicroQoSCORBA is a middleware framework targeted at embedded systems. This conscious choice has impacted our research and development at several levels. At the lowest level, some security mechanisms are just too computationally expensive to run on low-end embedded systems. So a key part of this research has been focused on what mechanisms can we support as well as what mechanisms make sense to support. For example, does a microwave oven need to support encryption with long key lengths? Another consequence of targeting embedded systems is that, generally speaking, more a priori infor-

mation is known about an embedded application at design-time, thus making it possible to leverage design-time choices rather than incorporating in costly, run-time adaptation mechanisms as might be done within a workstation-based application. For example, an application with strong confidentiality security QoS requirements could be deployed on a private network–thus ensuring confidentiality while at the same time minimizing computational effort.

Second, we realized that a full range of security QoS is appropriate and, in fact, actually needed. For example, as will be shown in Section 5 a task that can run in less than a millisecond on a workstation can take seconds to run on an embedded device. Depending upon the application's timeliness constraints, this will be unacceptable. In which case, designers will have to chose a less secure mechanism for a low-end device. For example, Rot13 [12], a simple Caesar cipher [13], might be an adequate choice to ensure confidentiality if the primary threat was from honest insiders sniffing packets on a private network.

Third, we recognized the need to maintain baseline interoperability with existing CORBA implementations, but we needed the ability to avoid strict compliance with standards that were too heavy-weight. For example, both the Fault Tolerant CORBA [14] and the CORBA Security Service [15] specifications are extremely detailed and quite broad—thus full compliance to either of these standards would have left MicroQoSCORBA unsuitable for deployment on embedded systems. Part of our research was on composing multiple QoS properties—something which has been very sparsely addressed in active research nor is it addressed in working standards (which often lag research). Composing multiple QoS properties is challenging, even in the resource-rich workstation environment, so the fact that our research is targeting embedded systems is significant. Furthermore, we are unaware of any existing middleware frameworks, suitable for embedded systems, that allow for the composition of many QoS mechanisms.

4.2 Security Goals

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

Keep it small. As discussed previously, cost constraints generally dictate that embedded systems are deployed with minimal computational and memory resources. Thus, one of our prime goals is to ensure that both the code size and computational burden of MicroQoSCORBA's security subsystem are small.

This has been accomplished with two separate approaches. First, MicroQoS-CORBA was designed with a fine degree of granularity and composability, see [6]. We also carried this same approach forward when designing and implementing the security subsystems in order to ensure that only a minimal subset of the security mechanisms would need to be incorporated into a given application in order to meet its security constraints. Secondly, we have been judicious in our choice of security mechanism to be implemented—seeking to ensure that where possible "small" mechanisms are implemented first.

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

Investigate multi-property QoS tradeoffs. Not only were we interested in the design and deployment of a security-enabled middleware framework, we were equally interested in deploying a middleware framework that could be used as a testbed within which multi-property QoS tradeoffs could be analyzed. Although an application might be developed and deployed with support for only one non-functional QoS property this does not mean that the application does not have multiple QoS property constraints—it simply means that the other constraints have been ignored. We believe that future embedded distributed systems will be required to explicitly support multiple non-functional QoS properties and thus our support for security within MicroQoSCORBA is a vital step in this direction. The focus of this paper is on the security subsystems of MicroQoSCORBA which limits our ability to present the tradeoffs that we have observed to date. For some initial evaluation results of MicroQoS-CORBA's security, fault tolerance, and timeliness results and their tradeoffs please refer to [6,7].

4.3 Security Design Space

One useful breakdown of security is to consider an application's constraints with regard to confidentiality, integrity, and availability properties as well as a broad range of other properties which we have grouped under the heading of accountability. We have taken a fairly broad view of these constraints within our middleware framework. Hardware and application constraints play a significant role in the overall system design, implementation, and deployment lifecycle of an embedded distributed application. MicroQoSCORBA was designed and implemented so that it fits within the overall embedded systems

Table 1
Security Design Space

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

design lifecycle. In particular, a designer can trade software mechanisms for hardware mechanisms that ensure the same constraints are met. The rest of this section will be devoted to these security constraints and the mechanisms that support them as shown in Table 1.

4.3.1 Confidentiality

The confidentiality column shown in Table 1 lists two headings: 'Physical' and 'Encryption'. This is because confidentiality of the data and control messages within a distributed embedded system may be achieved by both physical and logical means. For example, system designers could use separate networks for confidential information and commands. If physical mechanisms such as this are used, then MicroQoSCORBA's security mechanisms will not need to be embedded into an application.

Under the physical heading, we have listed dedicated networks as well as secure networks as two broad categories. As previously mentioned, physically isolating the embedded distributed application's data and commands provides a level of confidentiality. However, this is likely to incur a greater cost and thus the designer must strive to reach an appropriate balance between increased networking costs (e.g., isolated networks) versus increased costs due to sup-

porting higher computational loads on the embedded system's processors (e.g., missed timing deadlines or more expensive processors). Another option is to use an existing network, but yet deploy secure network interface cards (NIC) or protocols. Accessing the network would not incur any additional costs within MicroQoSCORBA's middleware layers, but yet data confidentiality would still be ensured because of the actions of the secure NICs or networking protocol layers. Deploying secure hardware or protocols stacks can also provide for increased integrity and availability even though it is not explicitly listed in the other columns of Table 1.

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

Under the private key heading only three ciphers are listed for brevity. AES [16], DES [17], and Rot13 [12] are shown to illustrate the embedded systems have confidentially requirements that range from very strong to very weak. In fact, Rot13 will likely only provide enough confidentiality to protect against only honest hackers. But, on the other hand Rot13 does have a very low computational overhead, thus making it potentially suitable for applications with low computational overhead margins.

4.3.2 Integrity

Data integrity is often critical within embedded systems because they interact with and react to their environments. For example, in the building automation and control example, given in Section 2.2.1, accurate room temperatures are needed to accurately control the building's HVAC systems.

Error control and error correction codes are listed in Table 1 because they are part of the quality of service continuum level that MicroQoSCORBA supports. For example, even though a standard CRC32 code is not cryptographically strong, it does provide a comparatively inexpensive means of detecting simple

bit errors.

Several cryptographically strong message digests are supported within MicroQoSCORBA in order to ensure data and command integrity. In some cases, not only must the data or command be unchanged, but the sender must be authenticated as well. In these cases, MicroQoSCORBA supports Message Authentication Codes (MAC). A MAC might be appropriate in the power grid example, given in Section 2.2.2, when a command to shutdown a power plant is given. Not only must the plant's operator know that the message has not been tampered with, but the operator must also be assured that it is coming from an entity that is authorized to shut down the plant's power production.

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

4.3.3 Availability

Availability is provided via the fault tolerance mechanisms briefly discussed in Section 3.4. For more information on MicroQoSCORBA's support for fault tolerance please see [7,11]. Disaster recovery is listed in Table 1 for completeness, but it is beyond the scope of a middleware framework and must be addressed by the designer of the distributed application.

4.3.4 Accountability

For the purpose of this paper, several other security properties have been gathered together and included them under the 'Accountability' heading in Table 1. Namely, these properties are 'Authentication,' 'Authorization,' 'Audit,' and 'Non-repudiation.' As with confidentiality, integrity, and availability, some accountability aspects are not within the capabilities of small embedded systems. Likewise, some aspects of accountability may also be provided with hardware mechanisms (e.g., hardware tokens for authenticating users and other entities). Non-repudiation is not supported because it relies upon a trust architecture that relies upon a PKI which is not supported by MicroQoSCORBA. Supporting local audit logs may be problematic on some systems because of their lack of both sufficient memory as well as non-volatile storage within which to keep the logs.

4.4 MicroQoSCORBA Security Implementation

MicroQoSCORBA was designed from the ground up to support a fine-grained composability of its various components. This greatly aided our efforts to design, architect, and implement the security subsystem within MicroQoSCORBA. This section will first describe the additions made to the baseline functionality of MicroQoSCORBA and then it will proceed to describe the mechanisms implemented.

4.4.1 Extending MicroQoSCORBA

One of the original subgoals within our development efforts has been to provide an easy to use GUI that a designer could use to specify the desired hardware and software constraints for a given embedded application [18]. This GUI was extended so that a system designer could specify their choice of confidentiality, integrity, and availability options to deploy in a given application.

MicroQoSCORBA has been developed in Java. However in order to support the various platforms in our testbed (see Section 5), Java 1, Java 2, and Java 2 Micro Edition all had to be supported. In order to provide this level of flexibility, we could have used factory and facade patterns [10], but we opted instead to implement the required platform specific details via the use of a macro processor. Using m4 allowed us to avoid the overhead associated with multiple levels of indirection that would be required to support these design patterns. Another key advantage, was that we were able to write macros that could allow us to insert or remove very small components—thus helping us achieve our goals of very fine-grained composability within MicroQoSCORBA.

We initially looked at supporting the Java Cryptography Extensions (JCE) [19]. However, JCE was architected with the assumption that at run-time a cryptographic provider's JCE implementation would be loaded and used by an application. MicroQoSCORBA does not support run-time adaptability because of the added complexity and resource usage that would be incurred. However, we were able to refactor the Open Source Cryptix Java JCE toolkit [20] to run in a non-dynamic manner on both Java 1.1.8 (required for TINI) and Java 2. Because we already had developed a macro capability, we were able to extend our configuration tool and IDL compiler so that they could generate security mechanism aware macro definitions. These macros were then used in the build process to bypass what used to be multiple levels of indirection in order to drill down to the actual algorithm implementations. Thus, with a few minor changes, we were able to reuse an existing and tested code-base without having to incur the penalties due to over-generalization associated with JCE.

Table 2
Implemented Security Mechanisms

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

4.4.2 Implemented Mechanisms

The Advanced Encryption Standard (AES) [16] is supported within our framework as well as other ciphers and mechanisms. AES was chosen for implementation within MicroQoSCORBA because it is a federal standard that was selected for its suitability for resource constrained devices as well as its cryptographic strength [21]. A complete list of implemented mechanisms is listed in Table 2. Null mechanisms were implemented in order to evaluate the overheads associated with encryption, message digests, and message authentication codes.

All of the encryption ciphers listed in Table 2 are secret key ciphers because, as explained in Section 4.3, MicroQoSCORBA does not currently support public key encryption mechanisms. The supported ciphers range from weak to strong. For example, the Caesar cipher as well as a simple XOR encryption scheme both provide very little confidentiality, but they are included within MicroQoSCORBA so that they may be used on hardware platforms that do not have enough computational resources (or long enough time deadlines) to support a stronger cipher. When MicroQoSCORBA's fixed-length message size option is selected, the complete message is encrypted (including the GIOP message headers), otherwise the GIOP message header are sent in plain-text because the message length must be read by the server's ORB.

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

5 Experimental Evaluation

MicroQoSCORBA has been developed and refined over several versions. This section presents a brief evaluation of its security subsystems; for other evaluation results, see [18,22,11,6,7].

MicroQoSCORBA was developed in Java because of Java's flexibility and cross-platform support. A Java prototype will not allow one to target extremely small embedded devices, but several small devices have been deployed with JVMs (e.g., Dallas Semiconductor's TINI board [23], aJile's aJ-100 [24]), so one could quickly develop a cross-platform, working prototype. 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. MicroQoSCORBA has successfully interoperated with a number of other ORB implementations (e.g., JacORB [25] and TAO [26]). We have also compared our MicroQoSCORBA results with other standard Java ORBs [18,7,6]. In summary, MicroQoSCORBA has a significantly smaller memory footprint and executes faster. 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 an enterprize ORB's functionality. Therefore it has a smaller footprint and less message-oriented overhead.

Fairly comparing middleware frameworks is problematic, since each framework is designed and developed with different goals and priorities, and therefore each has a different set of strengths and weakness. Comparing embedded middleware frameworks is even more problematic. Not only do the standard middleware comparison issues apply, but simply comparing embedded hardware is often an "apples" to "oranges" comparison 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 and so forth. It is beyond the scope of this paper to present a full and fair comparison of MicroQoSCORBA to other ORBs running on both standard and embedded platforms. Furthermore, existing security-enabled ORBs would not be able to execute on (or even be ported to) all of our testbed platforms because of very limited capabilities. Thus, the following subsections will focus only on MicroQoSCORBA and its performance on three different hardware platforms.

5.1 Testbed Hardware

The hardware testbed used for our evaluation consisted of two Pentium 4, 1.5 GHz PCs running Red Hat Linux version 7.2, two Systronix SaJe boards [27],

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

Fig. 4. Testbed Application IDL

and two Dallas Semiconductor TINI boards [23]. The SaJe boards have a 100 MHz aJile Systems aJ-100 CPU that executes Java byte-code natively . The TINI boards are powered by a 40 MHz DS80C390 CPU. The Linux boxes communicated with each other at 100 Mb/sec, whereas the SaJe and TINI communicated at only 10 Mb/sec since these boards did not have 100 Mb/sec network interfaces.

The MicroQoSCORBA Java code and evaluation-specific code were compiled with Sun's Java 2 Software Development Kit version 1.4.1 (J2SDK1.4.1) for use on the Linux systems. Sun's J2SDK1.4.1 was also used to compile the Java code for the SaJe and TINI boards, but additional steps were required for these two embedded platforms. JemBuilder (version 3.1.5) was used to build a Java byte-code image for the SaJe boards and TINIConvertor (version 1.02e) was used to convert standard Java class files into a format suitable for the TINI board.

5.2 Testbed Application

Our testbed evaluation application was based upon the IDL code shown in Figure 4. We chose to not implement a more complex application (e.g., the building example of Section 2.2.1) so as to simplify the interpretation of our evaluation results.

Several versions of this example application were generated. First, baseline client and server program were generated for each of our three testbed platforms (Linux, SaJe, TINI). Second, various security enabled client and server programs were also built and run on our testbed environment. The following sections will provide the results of our evaluation of these various MicroQoS-CORBA configurations.

5.3 Application Size and Memory Usage

An application's size and memory usage depends upon the size of the application's Java class files that must be loaded into memory, the amount of data that must be allocated on the heap during run-time, and other system-specific

Table 3
Cipher Impacts on Java Class File Sizes (bytes)

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

and run-time library code that must be loaded and executed on behalf of the application. Measuring the size of the generated Java class files for the Micro-QoSCORBA ORB and timing example code is a straightforward task. These results are discussed in the following subsections.

5.3.1 Security Ciphers

The security cipher results (in bytes) are reported in Table 3. The client and server class file sizes are reported for each of the three testbed hardware platforms. If a cipher supports multiple key lengths, then the java class file sizes are for a configuration using a 128-bit key, if supported, in order to provide a better cross-cipher size comparison. It should be noted that for ciphers that support multiple key lengths, changing the key length caused the java class file sizes to vary by only a few bytes, if at all.

The reported SaJe values in Table 3 are larger than the Linux and TINI values because the SaJe development tools include some runtime class libraries in executable image that is downloaded into the SaJe's flash memory—whereas on Linux and TINI these run-time class libraries are not counted because they are part of their platform's respective software JVM. The TINI values are relatively small because a compressed archive of the required class files is downloaded to the TINI board.

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

5.3.2 Message Digests and Authentication Codes

The message digest results are reported in Table 4. The client and server class file sizes are reported for each of the three testbed hardware platforms. As explained in the previous subsection, the SaJe values are larger than the Linux values and the TINI values are smaller.

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

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

Table 4
Message Digest Impacts on Java Class File Sizes (bytes)

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

5.3.3 Cipher and Message Digests

The message digest results are reported in Table 5. The client and server class file sizes are reported for each of the three testbed hardware platforms. As explained in the previous subsection, the SaJe values are larger than the Linux values and the TINI values are smaller.

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

Table 5
Cipher and Message Digest Impacts on Java Class File Sizes (bytes)

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

5.4 Performance

Performance evaluation experiments were conducted on three hardware platforms (Linux, SaJe, and TINI). These three platforms vary widely in terms of overall performance. Furthermore, each platform's operating system and Java implementation impact the end-to-end performance of MicroQoSCORBA. Before presenting the MicroQoSCORBA security performance results, the methodology for gathering and computing these results is given. Then the performance of MicroQoSCORBA's ciphers, message digests, and message authentication codes are presented.

5.4.1 Cross Platform Comparison Methodology

The initial performance results gathered in end-to-end tests were not consistently reproducible. For example, increasing the number of iterations within the timing loop caused the performance numbers to drop. Additional analysis showed that this increase was due to two primary factors. First, enough iterations had to be run in order to load the CPU cache and get the network into a steady-state condition. Secondly, as more and more iterations are run, the Java garbage collector is called more often. On SaJe and TINI, the performance of their respective Java garbage collectors significantly impacted the timing results—thus skewing the overall cross-platform performance analysis. An algorithm was developed which separated the baseline performance from the "noise" associated with OS and Java implementation details of each platform. This algorithm will be presented later, but first a paragraph detailing how the startup impact were overcome, then raw, unprocessed timing histograms are presented for each platform. These results will be discussed, after which, the complete "noise" filtering algorithm will be presented.

Steady State. Tests were conducted on the three platforms in order to determine and optimal number of iterations to compute 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 optimizations and to overcome TCP's slow-start mechanisms. On the SaJe platform, three loops of 100 iterations was determined to be sufficient to enter a steady-state condition. On TINI, three loops of 10 iterations 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.

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 either Java Native Interface (JNI) or platform specific calls were issued in order to achieve a higher level of timing resolution. Raw performance results for the baseline MicroQoSCORBA configuration are presented in the histograms of Figures 5, 6, 7, 8, and 9. 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 5 and 6. These histograms illustrate that although the vast majority of invocations are completed quickly, with an average time of 0.161 milliseconds, approximately 5% of the baseline Linux invocations take longer 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 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 slower 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 7 and 8. 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

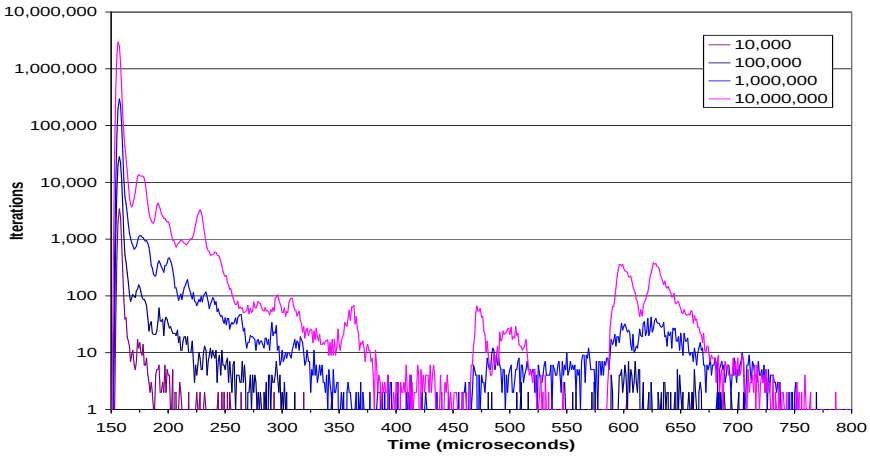

Fig. 5. Linux Timing Performance Histogram with Microsecond Resolution

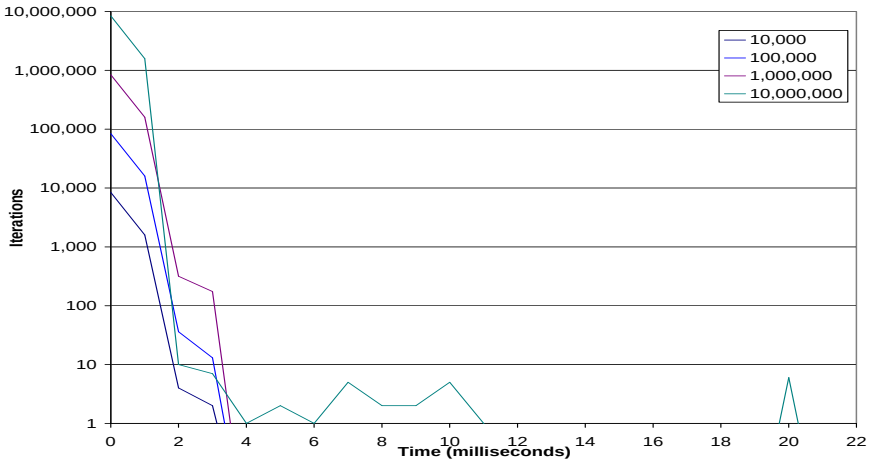

Fig. 6. Linux Timing Performance Histogram with Millisecond Resolution

peak located at 3.77 ms. But, the shape of the curve varies for the rest of the events. The data shown in Figure 8 illustrates the fact that Java garbage collection on the SaJe platform is very time consuming—adding over 270 ms to MicroQoSCORBA invocations that required 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 3.77 ms to 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 which are shown in Figure 9. The histogram shown in the aforementioned figure shows that the majority of the invocation calls complete with an average time of 134 ms. However, a significant number of events occur in the 150–225 ms range. Unlike the Linux and SaJe platforms which had 5% of their invocations impacted by garbage collection or other

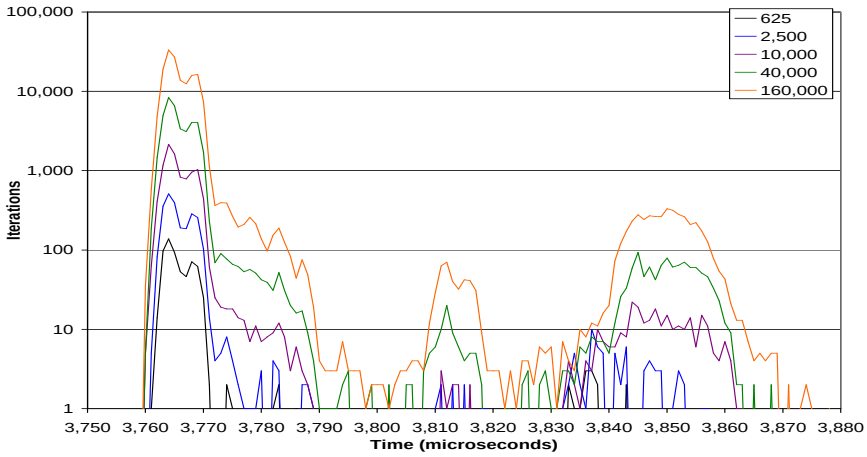

Fig. 7. SaJe Timing Performance Histogram with Microsecond Resolution

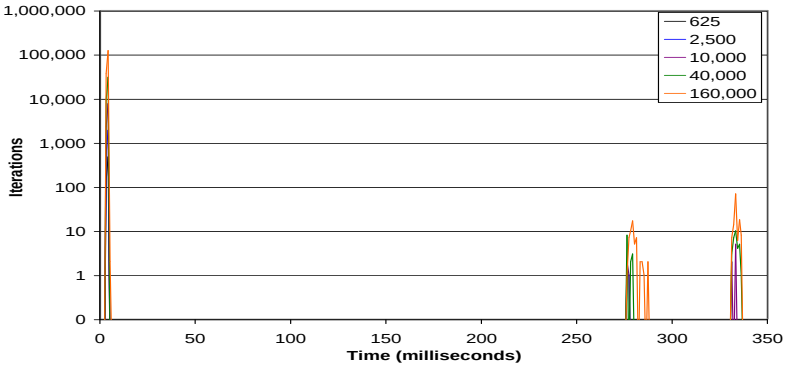

Fig. 8. SaJe Timing Performance Histogram with Millisecond Resolution

OS and network specific events, the TINI platform had over 25% of its invocations impacted by garbage collection and other system "noise." Thus, for small invocation counts an average time of 134 ms was recorded, but when a large number of invocations was issued the impact of TINI's java garbage collector caused this value to grow to 151 ms—an increase of 13%.

Thus, as can be seen from the experimental results gathered on these three testbed platforms memory management issues and other platform specific (e.g., OS and network implementations) can have a significant impact on overall system performance. Furthermore, this impact is platform specific. For example, on Linux and SaJe 5% of the invocations were impacted, whereas on TINI, for large invocation counts, over 25% were. Furthermore, on the TINI platform the garbage collector runs 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

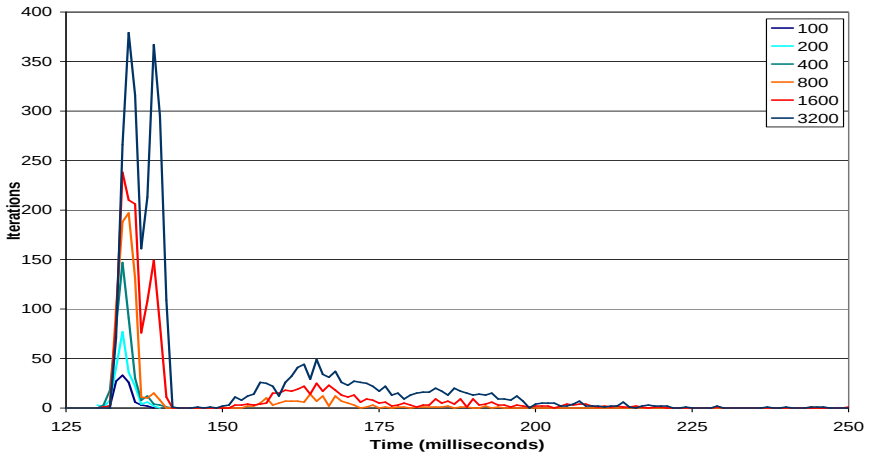

Fig. 9. TINI Timing Performance Histogram

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.

Event Filtering. The SaJe and TINI 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 be useful on all three testbed platforms. By appropriately filtering events, consistent timing results were obtained across a very wide range of invocation counts on each testbed platform. MicroQoS-CORBA's event filter also managed to filter out other, non-garbage collection impacts. 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 5 (many more "slow" events were recorded), but the filtered timing values remained unchanged from those computed when the MicroQoSCORBA client and server tasks were the only applications running the testbed Linux machines.

MicroQoSCORBA's event filtering process proceeds in three steps. The first step, which has already been mentioned, consists of invoking a sufficiently large enough number of method calls in order to reach a steady state condition. Detailed timing data is not gathered during this initial stage of a timing run. During the second stage, detailed timing results for each invocation event is gathered and histogrammed. This data is then analyzed, as will be explained in the following paragraph. 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 was conducted via an iterative method. The high-level details are as follows. First, an average

and its standard deviation are computed from the complete timing histogram. Then the average and standard deviation values are used to compute an upper limit value. Next, only data points between the minimum value and the upper limit value are used to compute a new average and standard deviation, which in turn are used to compute a new upper limit value. This process continues until such time as the new upper limit value equals to the old upper limit value. During each iteration, the upper limit 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, non-garbage collected events. This approach was able to correctly identify the upper limit for the minimum time peak within just a few iterations. Event filtering was conducted after the completion of each timing loop so that performance overheads 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 event loops. For the most part, the values computed by filtering and analyzing the three separate timing histograms were consistent. But occasionally (especially on TINI), one run would be adversely affected by OS, network traffic, or Java garbage collection impacts, so the ability to filter out these abnormally high values added to the robustness of our overall event filtering methodology for MicroQoSCORBA.

5.4.2 Testbed Application Setup

A baseline MicroQoSCORBA application corresponding to the IDL shown in Figure 4 was designed and implemented. For each of the various MicroQoS-CORBA security configurations presented in the following sections, a client and a server Java application was built with the required functionality.

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

Three separate timing loops were conducted during the course of the testbed application. After each loop, the histograms were analyzed and their results saved for a comparison at the end of the programs' execution. On the Linux platform, 100,000 invocations were issued during each of the three loops, and on SaJe and TINI the values were 2500 and 200, respectively. As will be shown, some TINI configurations run very slowly (e.g., over 60 seconds per

invocation), so a timeout value set at 15 minutes per loop was used to ensure that that testing could complete in a timely manner. On Linux and SaJe the timeout values were set at 10 minutes. Even with the timeout values in place, the performance value reported for each configuration was based upon upon enough events so as to provide a high level of accuracy.

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

5.4.3 Cipher Performance

The security cipher performance values are shown in Table 6. Timing results were measured on each of the three testbed platform for three different packet lengths as explained previously. Each row in Table 6 refers to the execution of a given MicroQoSCORBA configuration. The values listed in the Baseline row correspond to a baseline MicroQoSCORBA configuration that does not support any security mechanisms. The Null configuration incorporates the methods needed to implement a security cipher but it does not actually encrypt or decrypt the data before sending it across the network. Several of MicroQoSCORBA's supported ciphers may be configured to use multiple key lengths. Unless otherwise noted by a value in parentheses, the results are reported for configuration using 128-bit keys.

Several of the rows in Table 6 are of particular interest—namely, the Baseline, Null, XOR, and Caesar cipher rows. Comparing the results listed in the Baseline and Null rows shows that a minimal overhead is added for incorporating the additional classes and method invocations needed to implement encryption and decryption of the MicroQoSCORBA messages. On TINI the Baseline and Null results for 512-byte message are slightly slower than than their respective 56-byte messages values. At first this seems contradictory, because it indicates that more data can be sent in less time. But this result can be explained by the fact that given an a priori knowledge of the packet length, MicroQoSCORBA is able to optimize its message marshalling and demarshalling routines. Furthermore, TINI's JVM was optimized for data I/O rather than computational power [23]. Taken together, this means that for this given MicroQoSCORBA

Table 6
Cipher Timing Results (ms)

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

configuration, more data to be sent in less time on the TINI platform.

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

The performance values for Linux are the fastest, followed by the SaJe and

TINI values. However, one might not have realized just how large of difference there is between these three platforms. For example, a 1024-byte message encrypted with Triple-DES can be sent and received in under two milliseconds on Linux, whereas on TINI the same task takes over 74 seconds. Relative performance on a given platform also varied widely. The performance ratio of the slowest configuration (i.e., Twofish on 1024-byte messages on Linux, Triple-DES on SaJe and TINI) divided by the quickest (8-bit XOR on 56-byte messages) is 11.5 on Linux, 22.0 on SaJe, and on TINI this ratio is 376.8.

5.4.4 Message Digest Performance

The message digest (MD) performance values are shown in Table 7. Timing results were measured on each of the three testbed platform for three different packet lengths as explained previously. Each row in Table 7 refers to the execution of a given MicroQoSCORBA message digest configuration. The values listed in the Baseline row correspond to a baseline MicroQoSCORBA configuration that does not support any security mechanisms. The Null configuration incorporates the methods needed to implement a message digest but it does not actually digest any data before sending it across the network. The length, in bits, of each message digest are as follows: Parity–8, CRC32–32, MD2–128, MD4–128, MD5–128, RIPEMD–128, RIPEMD-128–128, RIPEMD-160–160, SHA0–160, SHA1–160, SHA2-256–256, SHA2-384–384, SHA2-512–512, Tiger–192.

Several of the rows in Table 7 are of particular interest—namely, the Baseline, Null, Parity, and CRC32 rows. Comparing the results listed in the Baseline and Null rows shows that a minimal overhead is added for incorporating the additional classes and method invocations needed to implement message digests. Even though the Parity and CRC32 digests are not very secure, these digests execute quickly and are supported in MicroQoSCORBA so that they may be used in time critical applications. On Linux, the CRC32 MD outperforms the Parity MD, but on the other two platforms the Parity MD is faster. The slow implementation of MD2 is likely due to the fact that although this digest was designed for 8-bit hardware, it's implementation in a 32-bit Java virtual machine is less than optimal. With the exception of MD2, the SHA2 family of digests are the slowest digests due to their computational complexity.

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

Table 7
Message Digest Timing Results (ms)

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

Linux, 37.2 on SaJe, and on TINI this ratio is 522.4.

5.4.5 Message Authentication Code Performance

The message authentication code (MAC) performance values are shown in Table 8. Timing results were measured on each of the three testbed platform for three different packet lengths. Each row in Table 8 refers to the execution of a given MicroQoSCORBA message authentication code configuration. The values listed in the Baseline row correspond to a baseline MicroQoSCORBA configuration that does not support any security mechanisms. The Null configuration incorporates the methods needed to implement a message authentication code but it does not actually authenticate any data before sending it across the network. Currently, the supported MAC algorithm is HMAC and so each of the row labels shown in Table 8 refer to the underlying message digest used in the HMAC algorithm.

Table 8
Message Authentication Code Timing Results (ms)

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

Once again, the Null MAC illustrates that only a modest overhead is added with the addition of the classes and methods needed to implement MicroQoS-CORBA's MAC mechanisms. As expected, because HMAC builds a MAC based upon a given message digest, the HMAC values shown in Table 8 are all higher than their corresponding message digest performance values shown in Table 7.

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

Table 9
Cipher and Message Digest Timing Results (ms)

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

5.4.6 Cipher and Message Digest Combined Performance

All of MicroQoSCORBA's cipher, message digest, and message authentication code mechanisms can be composed together. Rather than present another large table, only a few results are presented in Table 9. For reference the baseline performance values are presented. Also presented for comparison are the combination of two light-weight mechanisms, XOR & Parity, as well as stronger mechanisms, Triple-DES & MD5, AES-128 & SHA1, and AES-256 & SHA2-512.

As shown by the results presented in Table 9, the performance differences between the three testbed platforms is significant. These results also show that the performance of the cipher and message digest composition can not be simply assumed to be a linear combination of the cipher and message digest values. For example, consider Triple-DES & MD5 on SaJe with variable-length packets (i.e., the 56-byte column). Adding the individual values and subtracting off one baseline value (so that the baseline is not double counted) results in 12.56 ms for an estimate, but the measured value is 15.31 ms.

5.5 Analysis

The previous results sections, Sections 5.4.3–5.4.6, have presented some limited analysis of the results. In particular, it has been shown that for each of the evaluated mechanisms, the performance between the testbed system vary dramatically and the performance between the "quickest" and "slowest" mechanisms within a category also varies dramatically. In this section, common analysis trends and results will be highlighted. First a discussion of hardware impacts will be given and then a discussion of these performance results in the broader multi-property QoS domain towards which MicroQoSCORBA is targeted.

5.5.1 Hardware Impacts

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

5.5.2 Multi-property QoS Impacts

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

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

6 Related Work

MicroQoSCORBA is a middleware framework that has been designed and implemented for small, embedded systems. A general overview of MicroQoS-CORBA and it's support for multiple Quality of Service properties is given in [6]. The focus of this paper has been to motivate the design as well as to provide implementation details and an evaluation of the security QoS provided within MicroQoSCORBA. Thus, we will limit our presentation of related work to security for resource constrained, embedded systems and middleware frameworks.

Several OMG CORBA standards cover various security aspects and they have been implemented by a variety of companies [28]. The standards are the CORBA Security Service [15], Common Secure Interoperability (CSIv2) [29], Authorization Token Layer Acquisition Service (ATLAS) [30], and Resource Access Decision (RAD) [31]. Individually and as a group, each of these standards is large and therefore full compliance with any of these standards would preclude deployment on small, resource-constrained systems. Although a baseline MicroQoSCORBA configuration maintains interoperability with other CORBA implementations, MicroQoSCORBA's security subsystem bypasses conformance with any of these standards in order to reduce resource usage on small, embedded systems. Another standard, the Java Cryptography Extension (JCE) [19] supports multiple security mechanisms via the dynamic class loading of a security provider's implementation. MicroQoSCORBA by design does not support dynamic class loading because of both hardware/platform specific requirements (i.e., TINI does not support dynamic class loading) as well as improved resource usage.

At least two recent projects have focused on evaluating the security properties of various security mechanisms. From 1997 to 2000, the US NIST conducted an evaluation of proposed symmetric-key encryption algorithms to be used as a new Advanced Encryption Standard (AES) [32,21]. This process concluded with the selection of Rijndael as the new AES and the publication of a new Federal Information Processing Standard, FIPS-197 [16]. The New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project [33] is another project that evaluated the strength of multiple security mechanisms. Both the NIST and NESSIE evaluations focused on strong security primitives. MicroQoSCORBA supports both strong and weak mechanisms (e.g., XOR-base encryption, CRC32 checksums) because many resource-constrained systems simply can not execute stronger mechanisms in a timely manner. Additionally, these other efforts have not focused on end-to-end performance within a middleware framework.

The DARPA NEST program [34] is funding several related projects. The

Berkeley Wireless Embedded Systems (WEBS) [35] has developed SPINS [36], 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 [37]. TinyPK [38], was developed at BBN in order to provide public key encryption support for TinyOS. Wood and Stankovic present a security analysis of sensor networks in [3]. Each of these projects is focused on security, but their focus is at either the networking or OS implementations—levels well below MicroQoSCORBA's focus. In particular, none of these projects is providing security within a middleware framework.

7 Conclusions

This paper has presented an extensive set of results regarding the design, implementation, and performance of MicroQoSCORBA's security subsystem. Future work will entail further analysis of the cross-property impacts of composing security properties with QoS properties other than just timeliness. In particular, performance studies of how MicroQoSCORBA's security and fault tolerance properties compose is needed. Additionally, we will continue to port MicroQoSCORBA to other hardware platforms in order to better ascertain and map out how hardware design choices impact the performance of MicroQoSCORBA.

This paper has described the security subsystem of MicroQoSCORBA, a middleware framework developed from the ground up to support multiple non-functional properties within the embedded systems development environment. In particular an analysis of security requirements within the context of middleware frameworks targeting embedded systems has been presented, along with the design of MicroQoSCORBA's configurable security sub-system. In conclusion this paper presented and an evaluation of security mechanisms across a wide-range of hardware devices.

8 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.

The authors would like to thanks other members of the MicroQoSCORBA team, present and past, who helped in the development of MicroQoSCORBA. They are Thor Skaug, Eivind Naess, Olav Haugan, Tarana R. Damania, Kevin

E. Dorrow, and Wesley E. Lawrence. They would also like to thank the Cryptix team [20] for the use of many of the security mechanism implementations contained within their open source JCE implementation.

References

[1] B. Charny, Docomo issues recall of new videophone, Online (November 2001). URL http://news.com.com/2100-1033-276186.html

[2] D. Tennenhouse, Proactive computing, Communications of the ACM 43 (5) (2000) 43–50.

[3] A. D. Wood, J. A. Stankovic, Denial of service in sensor networks, Computer 35 (10) (2002) 54–62.

[4] D. Snoonian, Smart buildings, IEEE Spectrum (2003) 18–23.

[5] D. Bakken, A. Bose, C. Hauser, I. Dionysiou, H. Gjermundrød, L. Xu, S. Bhowmik, Towards more extensible and resilient real-time information dissemination for the electrical power grid, in: Proceedings of Power Systems and Communications Infrastructure for the Future, International Institute for Critical Infrastructure, Beijing, China, 2002.

[6] A. D. McKinnon, K. E. Dorow, T. R. Damania, O. Haugan, W. E. Lawrence, D. E. Bakken, J. C. Shovic, A configurable middleware framework with multiple quality of service properties for small embedded systems, in: Proceedings of the 2nd IEEE International Symposium on Network and Computing Applications (NCA2003), IEEE Computer Society, 2003, pp. 197–204.

[7] A. D. McKinnon, O. Haugan, T. R. Damania, K. E. Dorow, W. E. Lawrence, D. E. Bakken, A configurable middleware framework with multiple QoS properties for small embedded systems, TR 2002-37, School of EECS, Washington State University (October 2002).

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

[9] A. D. McKinnon, Encylopedia of Distributed Computing, Kluwer Academic Publishers, 2003, Ch. Interface Definition Language, to appear.

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

[11] K. E. Dorow, Configurable fault-tolerance for MicroQoSCORBA, Master's thesis, Washington State University (December 2002).

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

[13] W. Stallings, Cryptography and Network Security: Principles and Practice, 2nd Edition, Prentice Hall, Upper Saddle River, New Jersey 07458, 1998.

[14] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 3.0.2, Object Management Group, Framingham, MA, 2002, Ch. 23–Fault Tolerant CORBA, pp. 23–1–23–106.
URL http://www.omg.org/docs/formal/02-06-59.pdf

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

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

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

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

[19] Java cryptography extension, Online (2003).
URL http://java.sun.com/products/jce/

[20] Cryptix, Online (2003).
URL http://cryptix.org/

[21] W. E. Burr, Selecting the advance encryption standard, Security and Privacy 1 (2) (2003) 43–52.

[22] T. R. Damania, Unreliable datagram support for configurable CORBA middleware, Master's thesis, Washington State University (July 2002).
URL http://microqoscorba.eecs.wsu.edu/Damania-Thesis.pdf

[23] D. Loomis, The TINI™Specification and Developer's Guide, Addison-Wesley, 2001.

[24] aJile, Online (2003).
URL http://www.aJile.org/

[25] JacORB, Online (2003).
URL http://www.jacorb.org/

[26] TAO, Online (2003).
URL http://www.cs.wustl.edu/~schmidt/TAO.html

[27] SaJe, Online (2003).
URL http://www.systronix.org/

[28] Omg secsig security links, Online (2003).
URL http://secsig.omg.org/old_secsig/links.html

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

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

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

[32] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, E. Roback, Report on the development of the advanced encryption standard (AES), NIST Report SAND2003-1215, NIST, Computer Security Division (October 2000).
URL http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf

[33] New european schemes for signatures, integrity, and encryption, Online (2003).
URL http://www.cryptonessie.org/

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

[35] Wireless embedded systems, Online (2003).
URL http://webs.cs.berkeley.edu

[36] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, D. E. Culler, Spins: security protocols for sensor networks, Wireless Networks 8 (5) (2002) 521–534.

[37] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, in: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, ACM Press, 2000, pp. 93–104.

[38] Tinypk, Online (2003).
URL http://www.is.bbn.com/projects/lws-nest/