CONFIGURABLE MIDDLEWARE-LEVEL INTRUSION DETECTION SUPPORT FOR EMBEDDED SYSTEMS

By

EIVIND NÆSS

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

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

May 2004

To the faculty of Washington State University:

The members of the Committee appointed to examine the thesis of EIVIND NÆSS find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

_____

Acknowledgment

I am greatly indebted to my advisor Dr. Dave Bakken, for his valuable support and assistance in mentoring me during the course of my education. This thesis would not have been possible without his help.

I wish to thank all of the committee members, Dr. Deborah A. Frincke, Dr. Carl Hauser, and Dr. John C. Shovic for their time, commitment and comments during this defense. I would particularly thank Dr. Deborah A. Frincke who has not only served as an external committee member, but also been a collaborator on my research from the very beginning and provided helpful comments and suggestions along the way.

In addition, I would like to thank the MicroQoSCORBA team and in particular Dr. David A. McKinnon (PhD'03), Thor Egil Skaug (MS'04), Kim Christian Swenson, and Ryan Johnston for their excellent support and expertise.

I am thankful for the support for this research provided 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.

Furthermore, I would especially like to thank Doreen Ramsuta for her patience and encouragement, and all my friends and people I have met while attending Washington State University for their help and support.

Finally, my sincerest thanks go to my family for their inspiration and support; I would not be where I am today without them.

Publications

- Næss, Eivind and Frincke, Deborah A. and Bakken, David E. “Configurable Middleware-Level Intrusion Detection for Embedded Systems”. Submitted to *Recent Advances in Intrusion Detection* (RAID), Sophia Antipolis, French Riviera, France - September 15-17, 2004.

CONFIGURABLE MIDDLEWARE-LEVEL INTRUSION DETECTION SUPPORT FOR EMBEDDED SYSTEMS

Abstract

by Eivind Næss, M.S.
Washington State University
May 2004

Chair: Dr. David E. Bakken

Embedded Systems account for more than 98% of all CPUs produced in recent years. They have become integral parts of a diverse range of systems from automobiles to critical infrastructures such as distribution and control of electricity and gas. Middleware is a layer of software above the operating system that provides higher-level building blocks for programmers to reduce the complexity of distributed systems. Embedded systems could benefit from using middleware because of the large heterogeneity of devices. However, intrusion detection research to date has not addressed embedded systems, and very little of it has explored the middleware layer. This thesis describes a model for application-based intrusion detection at the middleware layer by leveraging information that already exists at the middleware. It also describes the implementation of a configurable set of intrusion detection mechanisms for MicroQoSCORBA, a highly configurable CORBA middleware framework

designed for embedded systems. Finally, it presents an initial experimental evaluation of these mechanisms on two platforms.

Contents

List of Figures

List of Tables

Source Listings

To my parents

For their enduring love and support

Chapter 1

Introduction

Microprocessors are manufactured in a quantity of almost 10 billion parts per year, that is, more than 1 microprocessor for every human on earth, or 35 microprocessors per U.S. resident. We are surrounded by electronic gadgets such as cellular phones, PDAs, digital watches, and pagers. For example, there are about two dozens microprocessors integrated in an average car rolling off the assembly line [48].

Indeed, less than two percent of all CPUs produced in the year 2000 were intended for the high-end computing market [45]. Embedded systems have become ubiquitous, and in recent years increasingly networked. As embedded systems and their application programs become more distributed, their complexity and size increases. Embedded systems also form the foundation for monitoring and control of many of our critical infrastructures such as oil, gas, water and electricity supplies.

Middleware is a layer of software above the operating system which provides common programming abstractions across a distributed computing system. It helps shield programmers from the inherent complexities and heterogeneous interactions that are common in distributed systems. Middleware is especially suitable for critical infrastructures, particularly the electric power grid with its rapid proliferation of intelligent control and monitoring devices that has occurred in the last 5-10 years.

Encryption is supported in some form in most middleware frameworks. However, it provides limited protection for the range of security threats since the systems may be vulnerable to various other attacks. For example, intrusions can be performed by a mischievous insider with knowledge of the encryption and the required keys. Indeed, as quoted in New York Times from a statement made by Peter G. Neumann, a computer security pioneer, “If you think cryptography is the answer to your problem, then you don't know what your problem is”.

Intrusion detection is thus very necessary for embedded systems. However, to date there has been no published research on host-based intrusion detection tailored to the needs of embedded systems. Additionally, there has been little research into supporting intrusion detection at the middleware level.

1.1 Motivation

Intrusion detection systems are today widely available for both commercial and research purposes. Many of these come in a one-size-fits-all bundle and are often limited to monitoring of a specific computer resource such as the operating system or network. Embedded systems are often made for a specific purpose, and due to the large heterogeneity of devices it would make such general solutions impractical.

There are several advantages of implementing intrusion detection at the application layer instead of any of the lower levels i.e. the operating system or network level. For example, data can be captured by an application-based Intrusion Detection System (IDS) before being encrypted and sent to a remote host. Obtaining information about encrypted data and the internal state of an application are limited at the lower levels. Additionally, application-based IDSs have the ability to respond in real-time to behavior categorized as malicious, and thus possibly preventing the application from being compromised.

On the contrary, applications that take advantage of an application-based IDS risk a performance impact on its operations. In addition, it is more difficult to port the application specific IDS to other applications. Since an application specific

IDS limits its area of protection to the application it is made a part of, it can be necessary to deploy IDSs any at the lower levels to complement this property.

The model presented in this thesis focuses on implementing the mechanisms for intrusion detection as the distributed application is being developed, which has the benefit of allowing the system or application to be incrementally tested and tuned; and thus reducing the risk of incorrectly implemented sensors. Additionally, it has the potential to greatly reduce the time and cost to embed the sensors, especially if this has to be done by someone else at a later time that also had to make the effort to understand the code.

In this model, the middleware layer is used as a layer of abstraction that encapsulates the *intrusion detection system* (IDS). This encapsulation permits the IDS to be configured transparently to the application, which typically can be performed through a configuration tool bundled together with the middleware package. In this situation, configuration tools have the potential to reduce some of the configuration overhead associated with deploying an IDS.

Embedding sensors can be performed automatically in the middleware layer or the application. Being closer to the application enables a tighter integration between the IDS and the application. For example, in contrast to host-based IDSs integrated into the operating system, an application-level IDS is able to obtain

data before it is encrypted and sent between hosts in a distributed system. Additionally, this enables the possibility of allowing software developers to create customized application-level security policies and responses. The functionality provided by general responses with a *Commercial off the Shelf* (COTS) framework may be insufficient for a particular application. However, with a highly configurable and flexible architecture, it is relatively simple to add the necessary functionality or mechanisms to accommodate new requirements of the IDS.

Providing the IDS as a COTS software module pluggable into the middleware layer will still accommodate the viable time-to-market, usability, simplicity and effectiveness of the application being developed.

1.2 Summary of Contributions

This thesis investigates intrusion detection mechanisms for distributed embedded systems. The contributions of this thesis are the following:

- A model for middleware-based, application-level intrusion detection targeting small, resource constrained networked embedded devices.

- A highly configurable set of mechanisms for middleware-level intrusion detection. Among these is a new type of reusable sensors for intrusion detection that takes advantage of information available at the middleware layer.
- A set of reusable application-based security policies based on primitives provided by this intrusion detection framework.
- An experimental evaluation showing the cost overhead for each of the presented mechanisms that can be configured.

The remainder of this thesis is organized as follows: Chapter 2 gives background information on intrusion detection systems and a brief overview of MicroQoSCORBA; Chapter 3 discusses mistrust and vulnerabilities in embedded systems, especially those which are developed and deployed in critical infrastructures; Chapter 4 describes our proposed model for Embedded Middleware-Level Intrusion Detection. Chapter 5 presents our prototype based on this model. Chapter 6 presents the initial experimental evaluation of our framework. Chapter 7 describes related work in the area of intrusion detection, embedded intrusion detection and middleware. Chapter 8 provides a discussion

other related issues that pertain to the presented research. Finally, Chapter 9 concludes.

Chapter 2

Background

This chapter is divided into two sub-sections. The first part discusses general work in the field of intrusion detection and defines some of the most important terms that are frequently used in the latter part of this thesis. The last part of this chapter discusses MicroQoSCORBA, the middleware framework used for the experiments conducted in this thesis.

2.1 Intrusion Detection

Intrusion detection is still a large area of research and is increasingly becoming more important due to the number of networked computer systems and the posing threat of cyber-terrorism. In general, intrusion is defined as any set of actions that compromises the *Confidentiality, Integrity or Availability* (CIA) properties of a computer resource [19].

An *Intrusion Detection System* (IDS) is a computer system that makes an effort to perform detection of any attempt that compromises one or more of the CIA properties of a computer resource.

There are several different classifications that can be used to distinguish the particular IDSs. A *Host-based Intrusion Detection System* (HIDS) analyzes input data that pertains to the local host in which it is running. HIDS has successfully been implemented to use audit traces or log files from an application, system calls to the operating system, and checksums of system files to detect intrusions [15] [47]. A *Network-Based Intrusion Detection System* (NIDS) scrutinizes the streams of data that are transmitted onto the network to which the host is connected. *Application-Based Intrusion Detection Systems* (ABIDSs) are a subset of the HIDSs, which can monitor an application through e.g. log files or other output of an application. In some cases, it can be implemented as a personal firewall that inspects the packets at the application layer before they enter the network stack of the operating system [44]. Thus, the definition does not prohibit the ABIDS from being a part of an application instead of an external module. ABIDS has been implemented as a part of a web-server where it can detect ongoing attacks from clients [1][38].

The method of data collection can be divided into two categories that describe the way a component is being monitored. IDSs that typically acquire the data of the monitored component by a separate mechanism or tool, e.g. scrutinizing log files or network packets, are referred to as using an *indirect monitoring* strategy. On the other hand, if the data of the monitored component is obtained directly from it, e.g. via inline instrumentation of the source code, then the IDS is referred to as using a *direct monitoring* strategy. As a result of this, every IDS that is using a direct monitoring strategy are also host-based [50].

Direct monitoring can further be accomplished by using either an external or internal sensor. Whereas an *internal sensor* is implemented as a part of the component being monitored, an *external sensor* is implemented by source code separate from the monitored component [43]. For example, if the purpose of a sensor embedded in a middleware framework is to monitor the behavior of an application, then it is by definition an external sensor. However, if this sensor's purpose is to monitor the middleware framework, then it is by definition an internal sensor.

Embedded detectors are essentially internal sensors guarded by a conditional statement that looks for specific attacks and reports their occurrence [43]. An embedded detector is typically embedded into an application and used to detect

attempts utilizing existing exploits or previous vulnerabilities (e.g. buffer overflow). The use of embedded detectors is particularly interesting during an application's life cycle at a time when vulnerabilities have been reported, and before patches are released to the public.

Sensors as they are used in the area of intrusion detection should not be confused with transducers (i.e. sensors and actuators) that are frequently used as input or output of an embedded system. Similarly, it must be pointed out that real-time, as it is used in the area of intrusion detection, differs from when used with embedded systems. Real-time is often used to describe a computer system's characteristic of determinism and correctness by responding to externally generated input within a finite or specified delay. In intrusion detection, it describes the ability of the IDS to detect anomalies as they occur.

2.2 MicroQoSCORBA Overview

This section presents the MicroQoSCORBA, a highly configurable middleware framework for embedded systems, and some of the background material needed to understand MicroQoSCORBA [18][27][28][29][30]. This section is divided into two subsections; the first subsection gives a brief overview of CORBA

middleware followed by a more detailed section about the MicroQoSCORBA middleware framework.

2.2.1 CORBA Middleware

The *Common Object Request Broker Architecture* (CORBA) is a middleware architecture supporting distributed objects that has been developed by the *Object Management Group* (OMG) [35]. Objects that are distributed can be accessed transparently of their location through the *Object Request Broker* (ORB) that is located on both the local and remote hosts. As the initial step of designing a CORBA application, the objects' API is specified by use of the CORBA *Interface Description Language* (IDL). This specification is used by the CORBA IDL compiler that generates the stubs and skeletons. A stub is the client-side code generated by the IDL compiler that implements the same functions as the remote object thus allowing a remote invocation to appear as a local function call. A skeleton is the server-side equivalent of a client-side stub. CORBA middleware provide programmers with a higher level building block that can mask system heterogeneity and provide network transparency and language independence. Frameworks implemented according to the CORBA specification can additionally deploy configurations and optimizations transparently of the application while

maintaining the same interface as specified by the IDL specification. This makes it possible for MicroQoSCORBA to support additional requirements such as fault tolerance, encryption and intrusion detection.

2.2.2 MicroQoSCORBA Architecture

Embedded systems are very diverse, not only in terms of size, memory, computational power and power consumption, but also by their role of interaction with other applications and the amount of data transferred by the device; desktop computers are in many cases supercomputers in comparison. One of the key features of MicroQoSCORBA is the ability it has to target a wide range of embedded devices. MicroQoSCORBA is built upon a foundation of many small and highly configurable modules that allows it to scale down to the most resource constrained devices. These modules can be divided into different categories as a part of the middleware architecture taxonomy created especially for embedded systems; see Appendix A for further details on this middleware taxonomy.

MicroQoSCORBA is a highly configurable middleware framework designed with a “bottom-up” rethinking of the constraints and requirements that even pertains to the smallest resource constrained device. This framework has been created based on a set of composable and configurable modules that allows for

addition or removal of hardware specific code, e.g. support for different byte ordering. This is not only limited to hardware specific code, but also includes code for protocols and transport. For example, in extreme cases where the system does not need an Ethernet device, or it is simply too expensive for the target application, other transports such as serial or 1-wire [10] can be configured to accommodate the need of the application. Additionally, the IDL compiler provided with MicroQoSCORBA can be configured to add support for the data and parameter types that are absolutely necessary, and thus eliminate any unused code that can hold potential security vulnerabilities.

A high-level architectural diagram that shows the main components of MicroQoSCORBA is presented in Figure 1. This diagram illustrates the path of a message that is sent between two hosts in a distributed system. A message passes first through the IDL generated stub before it traverses through the customized ORB, protocol layer, transport layer, and finally reaches the physical network.

On the other side, the message propagates up through the transport and protocol layer before it is passed through the customized ORB and skeleton, and finally delivered to the remote application. The server's reply message traverses the same path in reverse.

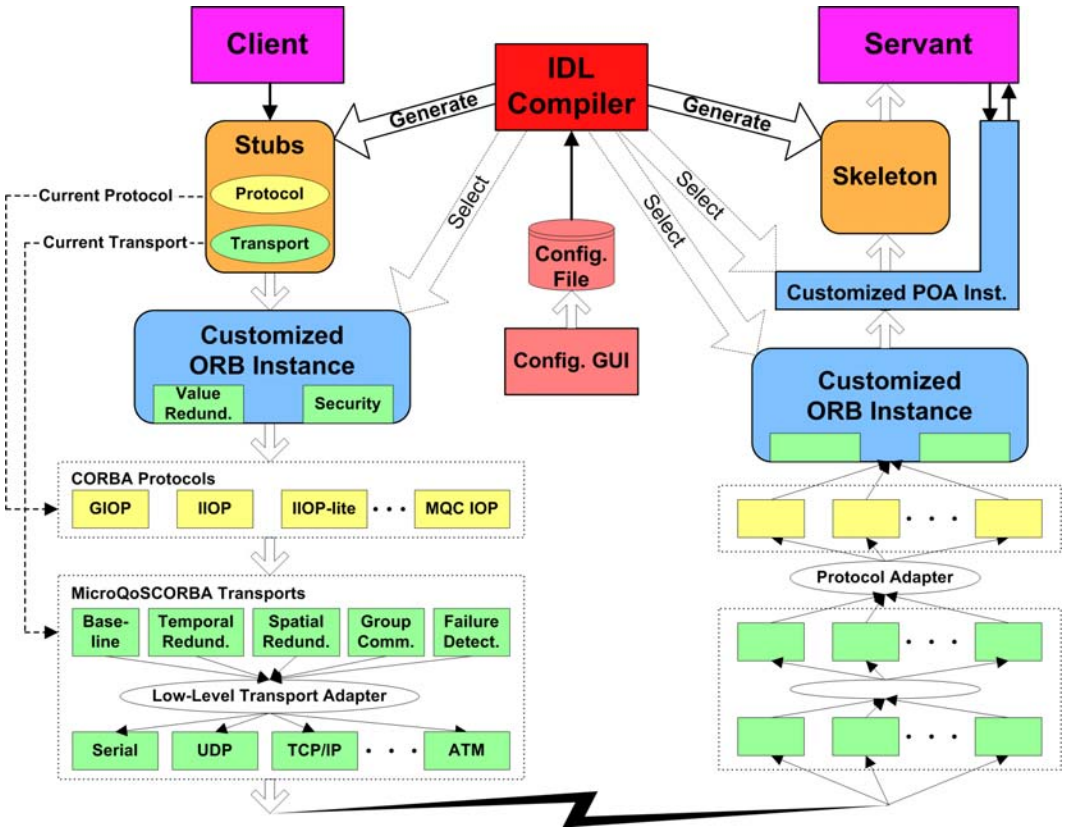

Figure 1. The MicroQoSCORBA architecture

As indicated by Figure 1, the IDL compiler has also a very central role in MicroQoSCORBA since it is used to generate or select many of the key components as specified by the developer. Every CORBA framework has their own IDL compiler which is often made very general and thus allows for little customizations or optimizations. The MicroQoSCORBA IDL compiler does not

only allow generation of customized stubs and skeletons for an optimized ORB, but also provides options for “hard-coding” a given protocol or transport into the client stubs. This subtlety removes the need for linking in some of the unneeded code from the transport and protocol layers.

Only a limited amount of optimization can be accomplished by the IDL compiler. However, the granularity of options available to MicroQoSCORBA allow for several optimizations to other parts of the framework (e.g. the ORB). For example, a developer may, at cost of the interoperability with other middleware frameworks that supports the CORBA standard, meet the application or hardware constraints by using a custom header format and non-standard data marshalling for the messages. The MicroQoSCORBA IDL compiler parses the IDL specification and a configuration file that can be generated by the accompanying configuration tool. These input files are used by the IDL compiler to generate the customized stub and skeleton code that is configured for the desired ORB or *Portable Object Adapter* (POA).

MicroQoSCORBA has also been designed and implemented with support for fault tolerance, security [12][27][28]. The modular and fine-grained architecture of MicroQoSCORBA is one of the key features which make this middleware framework unique. It allows for removal of unneeded code which accommodates

even the most resource-constrained embedded devices. This removal of unneeded system code also removes any unnecessary functionalities and any security vulnerabilities contained within this unused code [49].

Chapter 3

Vulnerability and Mistrust in Embedded Systems

Embedded systems are commonly used at the lower level in *Supervisory Control and Data Acquisition* (SCADA) systems. SCADA systems are a typical example of a *system of systems* (SoS) architecture where each computer system is a part of a hierarchical infrastructure.

As illustrated by Figure 2, electrical substations in the power grid are essentially smaller decentralized embedded systems spread out over a wide geographical area. The Distributed Control System (DCS) controller gathers data from the *Intelligent Electronic Devices* (IED) that further communicates with low level smart transducers such as sensors and actuators. Embedded systems in electrical substations are an important part of the data collection in a distributed control system for the electrical power grid.

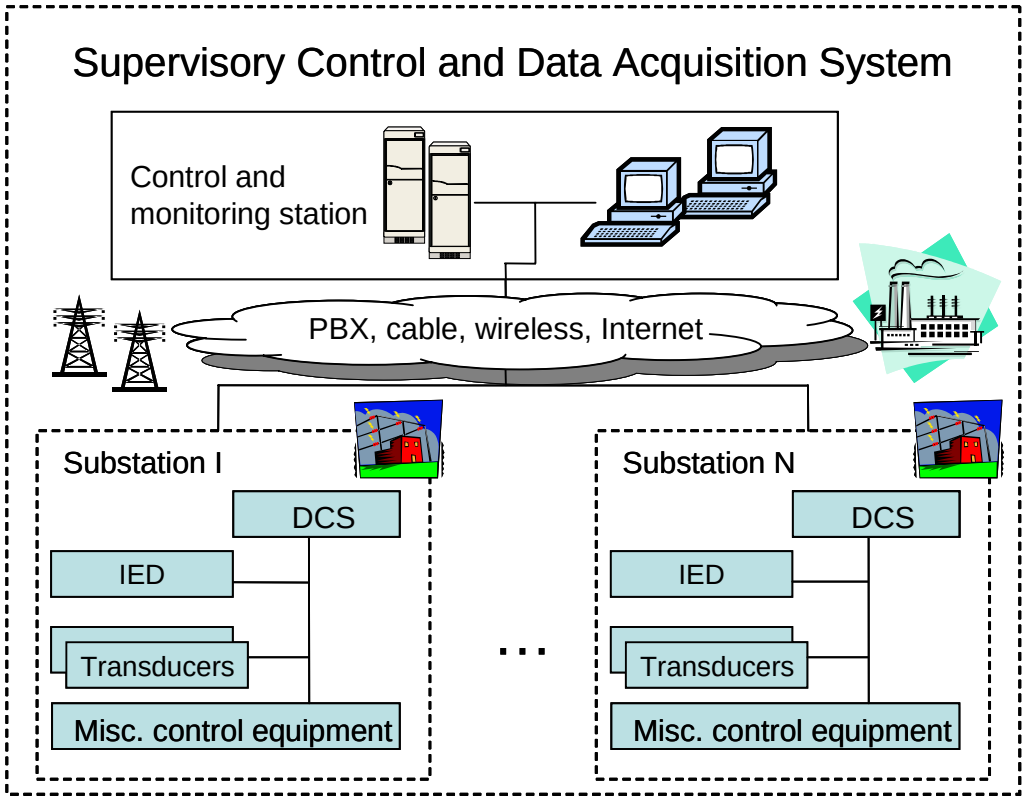

Figure 2. A SCADA system for the electrical power grid

The relatively low on-site security around these decentralized embedded systems makes them inherently more vulnerable to physical tampering as well as electronic attacks [9][32][33]. For example, an adversary can alter the system behavior by replacing components with different tolerance or accuracy, or even blindfold the sensors or actuators by feeding them a false input. Cyber attacks on SCADA devices are a big concern for many security experts [16]. Adversaries can ultimately gain electronic access to the system and reset switches to a higher

value, which could possibly damage the equipment, shut down transmission lines, and disable functionalities of the control system [9].

Securing the critical infrastructures is important, and security experts have good reasons to fear cyber attacks targeting these. Consider the effects of the blackout on August the 14, 2003, which was confirmed by authorities to have a natural cause. It affected 50 million people in 8 states and 2 provinces in USA and Canada respectively. This power outage was a direct cause of three deaths, shutdown of 22 US and Canadian power plants, closing of 10 major airports and cancellations of over 700 flights [17]. The total economic impact was estimated to be about 6.373 Billion dollars [2].

One can only imagine what mayhem it can create if terrorists can take out a few strategic parts of the power grid causing a major blackout in the middle of the winter time, and simultaneously prevent other critical infrastructures such as the 911 service from operating correctly; and this without having to set a single foot on that country's native soil. The following four paragraphs are examples of intrusions that have occurred in SCADA systems.

NSA demonstrated in 1998, how a cyber attack could be conducted against a remote substation for the electrical power grid, and possible shut down large parts of the grid [9]. A. S. Brown describes a method where a Pringle's can used as an

directional antenna, a laptop and some free software are enough to perform an intrusion into one of these remote sub-stations [6].

In August 2003, the U.S. nuclear regulatory commission determined that in January 2003, a computer worm was the cause of disabling a SCADA system for about 5 hours at the Davis-Besse nuclear power plant in Oak Harbor, Ohio [9].

A 12-year hacker managed in 1998 to break into the computer system that controls Arizona's Roosevelt Dam. Authorities stated that the boy had full access and control over the SCADA system that controls the massive dam's floodgates [16].

Furthermore, a disgruntled former employee was apprehended on April 23, 2000, in Queensland Australia, after his 46th successful intrusion into the SCADA system that controls the sewer treatment system. He ultimately released thousands of gallons of raw sewage into nearby rivers and parks [16].

Common for these intrusions presented here, and vulnerabilities described by Oman et al. [32][33], is that it appears to the embedded application as it is being accessed by legitimate personnel. The mechanisms for intrusion detection as provided in this thesis put focus on detecting malicious behavior by analyzing the input to an embedded application. For example, messages can be maliciously crafted to carry values that a host does not anticipate, hence, compromise the

integrity and possibly availability of the application. Furthermore, the behavior of an application can radically change dependent on the input. Application behavior modeling is frequently used in intrusion detection to create a fingerprint of the particular application, and further to detect deviations from this pattern [14][20][26][31]. Application behavior modeling has also been adopted by the model for middleware-level intrusion detection presented in this thesis.

Chapter 4

A Model for Embedded Middleware-Level Intrusion Detection

The model for an *embedded middleware-level intrusion detection system* (EMIDS) will now be presented. This chapter is further divided into five subsections. The first subsection describes the system model. Next, three discussions are provided in respect to the data collection mechanisms, application-level policies and responses as they influence the design choices of EMIDS. Finally, the last subsection gives a succinct summary of the presented model.

4.1 System Model

A traditional HIDS is placed at the operating system layer where it monitors the host's behavior. In contrast to the traditional HIDS, EMIDS is a model for an

application-based intrusion detection system based on anomaly and misuse detection.

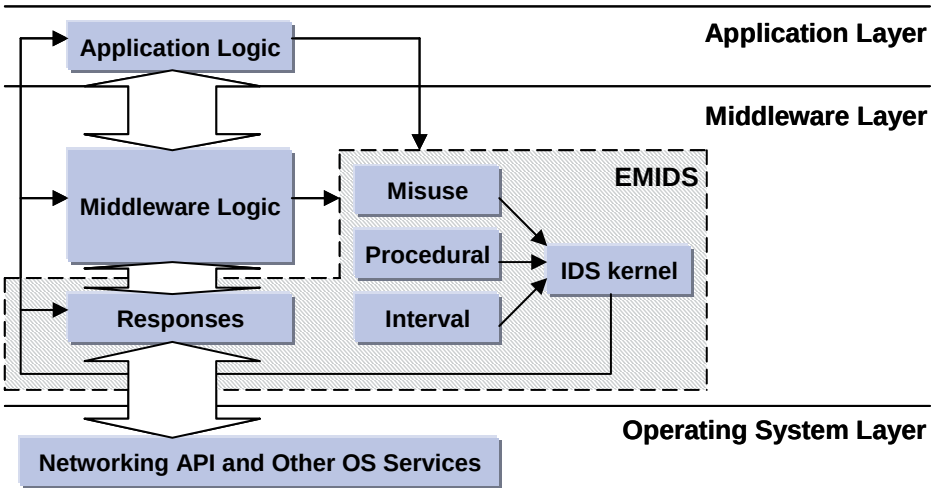

Figure 3. Architectural diagram of the EMIDS model

Implementing an application-based IDS can be accomplished both with and without the presence of a middleware layer. For distributed systems, the middleware layer serves as a higher-level of abstraction that provides an encapsulation where sensors can be embedded automatically and transparent to the application. In contrast to integrating the sensors into the operating system where they could normally observe the low level socket communication of an application, sensors can instead be embedded into the middleware layer observing an object's remote function calls.

Data collection is performed by sensors embedded into the application or middleware framework as illustrated in Figure 3. The output of the data collection phase is processed by an IDS-kernel, which is used for two things:

- Provide a high level of configurability, which can be extended to support dynamic reconfiguration of the security policies or responses.

- Support multiple types of sensors and responses through a generalized interface.

Depending on the internal sensor's functionality, the IDS-kernel triggers the configured responses or performs an extensive analysis of the collected data. The different sensor types are discussed in depth in the next section. Based on the outcome of the analysis or configuration of the IDS-kernel, the appropriate response will be triggered. Responses use information available in the middleware layer to take actions against a remote attack, e.g. delay or terminate a connection. It can additionally take actions against a particular part of an application, e.g. slow down the response time to the local user interface. See section 4.4 for more information in regards to responses.

4.2 Sensors and Detectors

Sensors and detectors incorporated into this model are divided into three different categories: interval-, procedural-, or misuse- based; each given a name that refers to their particular method of operation.

Interval-based sensors are used to monitor and detect anomalies in intervals of either the frequency of invocations to a given object, or the value of incoming data, i.e. an argument or return value of an object's method passed to the remote host. The interval-based sensors are external sensors that are inserted into the middleware framework automatically by an IDL compiler. This type of sensor is well suited for embedded systems as these systems frequently depend on their physical nature more than other systems, e.g. a distributed temperature sensor system made out of small embedded systems.

Procedural-based sensors are internal sensors that collect data based on the execution pattern of the application they are a part of, i.e. capturing the execution flow of an application, or more specifically, the order in which objects' methods are invoked. Sensors used for procedural detection can be automatically embedded in the entry and possibly exit of an object's methods as specified by the

developer. Procedural detection in EMIDS is based on the same model as described by Elbaum et al. [13].

Misuse-based detectors are a weaker form of embedded detectors that include misuse in addition to detecting specific attacks. They are typically inserted into locations in source code that contain (previously) known vulnerabilities; or locations that would naturally detect misuse [43]. Unlike the interval- and procedural- based sensors, misuse-based detectors cannot be embedded automatically into the application. However, this does not restrict them from being made configurable and inserted into the middleware without requiring any development effort from the application programmer. The surrounding logic of a misuse-based detector determines if an attack is present and triggers the configured responses. Similarly to the interval- and procedural- based sensors, the IDS-kernel provides a transparent mapping of detectors to the particular responses configured.

Both interval-based and procedural-based sensors could be replaced by an embedded detector that performs an equivalent detection. For example, an interval-based sensor can be replaced with a detector that compares the current value against an average of the previous values to determine if it is abnormal. This would require significantly more code embedded into the target's source file

compared to if it was handled by an IDS-kernel. In this situation, the IDS-kernel could considerably reduce the amount of code required for the instrumentation by providing a simple interface for data collection, which in the long run makes the source code more organized and maintainable.

4.3 Application-Based Security Policies

EMIDS is a model for intrusion detection that is capable of providing higher-level application-based security policies created especially for a given application. These policies are essentially formal policies, where the system or application can be described based on a mathematical model accompanied by a set of constraints that precisely define an abnormal behavior [5].

More specifically, an application can be statistically modeled by its execution pattern where an application-based security policy can specify a limit to any abnormal deviation from this model. This constraint is further referred to as a threshold, which determines when the IDS will respond. The model can be built over a period of time and stored into a profile that describes the application behavior. Policies as described here can also be used to model application constraints avoiding the use of a profile. The formulation of an application-based

security policy depends on the type of sensor or detector embedded, and may differ considerably in each case.

Interval-based sensors are used to detect variations in value or time, which also include the frequency of a specific invocation on either the client or server-side. Application based policies often relate to the physical surroundings of an embedded system, e.g. it is physically impossible that a temperature measured by a temperature sensor is below absolute zero. If a temperature sensor reports temperatures below this point, they may be considered abnormal and hence possibly compromise the integrity and availability of the application.

Measuring the frequency of an invocation on either the client or server-side over a period of time can implicitly reveal an abnormally high rate of data transmissions on the underlying network, which may be a sign of a network based *denial of service* (DOS) attack that can ultimately compromise the availability of the device. Additionally, a profile can be created for the interval-based sensors to contain a collection of previous values needed to build a statistical model. Application-based security policies can be applied to this model to distinguish rare or abnormal values from values that occur frequently.

The procedural-based detection mechanism must be configured with a profile containing a model that describes the normal behavior of the application.

Associated with this profile is an algorithm that scans the execution trace for non-existing or rare sequences of function calls. The result of this analysis yields a value describing the likelihood for the combination of function calls to occur. An application-based security policy can be applied to the output of this algorithm to determine if the result exceeds a predefined maximum. Procedural-based sensors would typically detect attempts to compromise the integrity and availability of a running application.

Embedded detectors implement their own application-based security policies through the additional logic that is required by each detector. The policy simply describes the detectors functionality while the embedded detector implements it. Limiting the number of connections from a particular host is an example of an application-based security policy that can be embedded into a middleware framework. Verifying that the integrity of an application or its external dependencies remains the same between each time the application is started is another example of an application-based security policy. Obviously, the application-based security policy may differ vastly from the specific embedded detectors purpose. Furthermore, application-based security policies for an embedded detector can be created to detect any attempt to compromise the CIA properties of an application.

4.4 Responses

The definition of an Intrusion Detection System (IDS) given in section 2.1 does not include responses as a part of the system; although, it is an important aspect and a motivation factor for deploying an IDS. Auditing is perhaps the most natural response that can be provided by an EMIDS. Proper credentials such as smart cards or personal pin codes can be implemented to access the system. These can further be embedded into the audit log for the purpose of accountability.

Information available at the middleware layer can be leveraged by an EMIDS to provide responses that log events, delay invocations, ban the IP-address, or terminate the connection between the host and an adversary. However, responses in an EMIDS are not limited to the possibilities within a middleware framework. For example, the output of an analysis in the IDS-kernel can be used to trigger application specific code, or send a signal to a remote host or network device. In other words, if the remote invocations of a system are being significantly impeded by high network traffic, an EMIDS can notify other network infrastructure components that can better handle the possible DOS attack or ensure a higher bandwidth, (e.g. use of a bandwidth reservation scheme). Alternatively, it can

signal the application to adapt to a more hostile environment, which may include setting the IDS at a higher level of alertness.

Internal sensors or detectors should in general not alter the execution path as it may have an unpredictable outcome on the future execution of an application [50]. Policies for interval-based sensors could in some cases reject an invocation in real-time based on the values of an object's argument that naturally would be discarded at a later time by the application; and at the same time act proactively to prevent the application from possibly inducing an error.

Applications can be designed and implemented in a way that allows the IDS to respond to a particular part of an application. For example, if the attack was determined to be performed locally through a user interface, responses such as time-delay can take action against the particular part of the application that interacts with the user.

A SCADA system turns to the higher level in its hierarchy if it encounters problems it is not programmed to handle [9]. Embedded systems that are a part of a similar *System of Systems* (SoS) architecture could be built in this manner, or with a "human in the loop" type of response. The flexibility of implementing an IDS at the middleware layer gives the developers the opportunity to integrate

responses transparently from the application, or fine tune the already existing responses for a specific purpose.

4.5 Summary of the EMIDS Model

This chapter presented a model for embedded middleware-level intrusion detection. A model for an application-based intrusion detection system integrated into a middleware framework that can transparently of the application provide a range of data collection mechanisms, application-based security policies and responses.

EMIDS can, in contrast to other host- and network- based IDSs, detect anomalies in well-defined data instead of bit-level information that is obtained through reading a message sent to the network stack. Also, EMIDS is able to capture data before it is encrypted and sent to the remote host. The data collected can possibly provide a less noisy data set and thus provide a better foundation of creating application-based security policies.

Developers can either configure or develop their own application-based security policies that can be used to describe abnormal behavior within a given dataset. Furthermore, this model is also capable of providing responses that are

able to react to abnormal behavior in real-time. Section 4.4 outlines a range of plausible responses integrated into the middleware framework or an application, which can be configured transparently of the application.

Chapter 5

MIDES: A Configurable Framework Providing Middleware-Level Intrusion Detection

A prototype of a highly configurable middleware-based IDS framework based on the EMIDS model presented in chapter 4 was designed and implemented. This framework is called *Middleware-based Intrusion Detection for Embedded System* (MIDES). This chapter presents the implementation of a configurable set of intrusion detection mechanisms as a part of the MicroQoSCORBA middleware framework designed particularly for embedded systems [11][18][27][28][29][30].

This chapter is organized into three subsections. The first subsection gives a general overview of the MIDES framework followed by an extensive description of the application-based security policies that are implemented and can be configured. Finally, the last section gives a more thorough description of the responses that can be configured with the system.

5.1 Overview of MIDES

The first step in creating a distributed application in CORBA is to write an interface specification that defines each object's methods and their parameters. This interface specification is used as an input to the IDL compiler that creates the stubs and skeletons for the middleware framework. A stub is the client-side code generated by a CORBA framework's IDL compiler that provides distribution transparency, i.e. making the client's call to a remote object look like a local method call. A skeleton is the server equivalent of a client-side stub.

MicroQoSCORBA goes a step further by requiring an additional configuration file that describes the subset of features and configuration constraints that are to be enabled in the middleware sub-system. This information is used in the generation of the middleware framework, the environmental configuration, and configuration files for the IDS that are linked into their respective client or server application. Typical configuration options are provided through a graphical user interface in which a developer can visually select a particular type of transports or protocols in addition to security, fault tolerance, and intrusion detection mechanisms. See Appendix B for further details on the configuration tool used to configure MIDES. These functionalities can be

transparently configured or enabled without any necessary modification to the client or server implementation. However, the application needs to be recompiled with the new changes incorporated into the middleware sub-system and the stubs and skeletons.

The IDL compiler generates additionally two files that individually configure the IDS for the respective client or server implementation. These files are used to set up the relationship between the instances of data, policy, profile and responses.

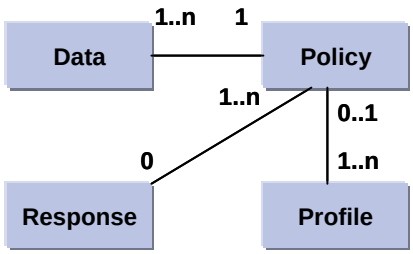

Figure 4. The relationship between data, policy, profile and responses

The data object is a general representation of an output from an internal sensor or detector. As illustrated in Figure 4, this data object is bound to the configured policy by the IDS kernel. This policy is further bound to the appropriate responses and profile as configured by the developer. Profiles are implemented so they can be reused if necessary by different policies referenced

by the same data point. Responses are configured globally where the functionality can be fitted for the specific purpose of the application. This allows responses to be instantiated only once which can reduce the configuration overhead and the amount of memory required.

A misuse based detector generates a data object that uniquely identifies the responses as they are configured with the IDS-kernel. This allows detectors the flexibility of being individually and transparently configured of the application. All of the sensors and detectors types as described in section 4.2 are supported by MIDES and will be thoroughly discussed in the next three subsections.

5.1.1 Interval-Based Sensors

The interval-based sensors are automatically inserted into the stubs and skeletons when generated by the IDL compiler. There are three different settings for the interval-based sensors that can be generated to accommodate different types of data collection. This includes settings for measuring the rate of invocations, response-times, and one or more of the function parameters (referred to as value-based).

The *Rate of invocation* measures the number of invocations per second on the server-side. This is particularly useful for applications that have a perpetual

behavior e.g. a distributed temperature sensor system at a nuclear power plant that sends the current temperature to a master node at explicit intervals.

The *response-time*, or the time it takes to perform an invocation, can be measured on the client-side. If there is a significant change in the response-time time, i.e. an invocation that takes normally a few microseconds takes seconds or minutes to perform, it can be classified as abnormal. A set of samples measured over a period of time can detect a significant increase in network traffic, which may be sign of a possible network based DOS attack. The response-time and rate of invocation mechanisms are later in this thesis also referred to as frequency based mechanisms.

The *value-based* mechanism is used to evaluate one or more of the function parameters including the return value for particular function defined in the interface specification given by the developer. This mechanism is particularly interesting for function parameters that follow a characteristic pattern that can be stochastically modeled, or classified into a specific interval. For example, consecutive temperatures measured over a period of time, or a specified range of temperatures, i.e. a temperature value of negative Kelvin is physically impossible, and thus out of range for most real-world applications.

Application-based security policies that are provided by MIDES can be applied to the absolute value, or to a profile containing previous values generated by the data collection of an interval-based sensor. In the latter case, it can be necessary to apply simple statistical computations to find the average and standard deviation of the values incorporated into this profile.

5.1.2 Procedural-Based Sensors

Procedural-based sensors are embedded into the application at the beginning and possibly the exit of each function. This type of internal sensor is used to build a profile, which holds a set of the most recent functions in the order they were invoked. Any deviation from this profile is detected in real-time by an algorithm that describes the level of anomalous application behavior. Applications are frequently improved and released with a broader set of functionality over their life cycles. Because of the increasing size and complexity of applications, it is crucial for this algorithm not only to work in linear time but also in bounded space (memory). MIDES can be configured to use either a simple sliding windows algorithm as described in [15] or a *Probabilistic Suffix Tree* (PST) as discussed in [26][36]; both algorithms run in a linear time. However the space bounds are significantly different. Whereas the PST is a parsimonious *n*-ary tree limited by

the number of nodes that holds any significant stochastic information; the sliding windows algorithm has an exponential worst case space bound. The following two subsections discuss these algorithms in detail.

5.1.2.1 Sliding Windows

The profile for sliding windows is stored in an *n*-ary tree that records all the possible number of combinations that occur in the training sample. This tree has an exponential space bound if every unique function and class were to be combined resulting in a full tree. Thus, object-oriented programs with low cohesion and coupling between objects makes the space bound significantly lower. However, this algorithm was chosen because of its simplicity and to illustrate the use of procedural detection.

Figure 5 shows an example of a tree that is inferred by sliding a window across the training sample (trace). This window is moved in increments of one over the trace. For example, *"3, 2, 4"*, *"2, 4, 3"* ... *"3, 2, 3"* are typical values that would be inserted into the tree under training.

The level of anomaly *P* is calculated by sliding a window of size *w* over the current trace with length *n* while recording the number of mismatches *M*. That is, the number of non-occurring windows in the tree inferred by the training sample (e.g. *"2, 2, 2"*). The level of anomaly *P* is given by Equation 1 [42].

$$P = M / (w * (n - (w + 1) / 2))$$

Equation 1. The level of anomaly using sliding windows

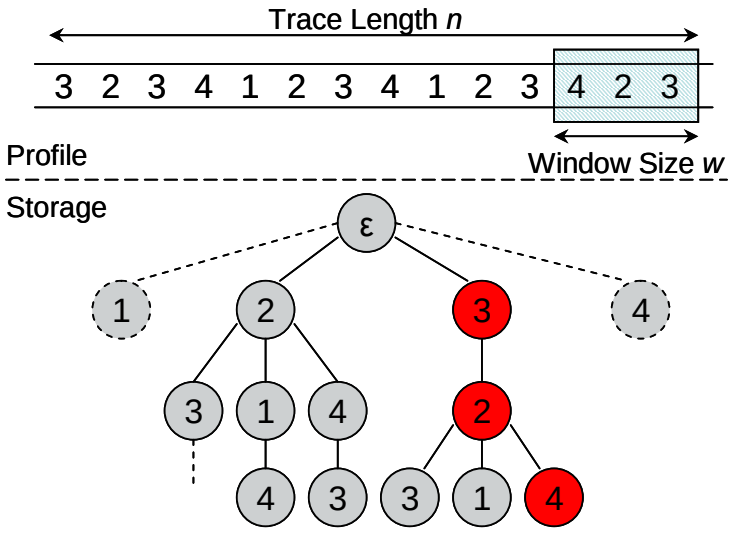

Figure 5. Example of a tree inferred by the Sliding Windows algorithm

Somayaji et al. [42], describes a sliding windows algorithm that uses a byte array of size $|S|$ x $|S|$ x (w - 1), where S is a finite number of symbols that represent all possible system calls in a UNIX system. This is a potential optimization for small applications, or application with a limited set of symbols. Thus, a tighter integration with the application eliminates the need to represent all the possible states, e.g. functions that are unconditionally invoked from within another function, and therefore allows the tree to grow according to the application's complexity.

5.1.2.2 Probabilistic Suffix Tree

It is very important for resource starved embedded systems that this algorithm detects anomalies efficiently in both time and space. Recent advances in the research area of intrusion detection have shown algorithms, such as a *Probabilistic Suffix Tree* (PST), that has a linear time and space bound [14][26]. The PST is an *n*-ary tree that models the stochastic behavior of an application. A suffix tree differs from other trees by its nodal relationship (parent / child). Parents in a suffix tree are described by a set of symbols that is a suffix to all its descendant nodes.

This is better illustrated in Figure 6 where node *'23'* is an example of a symbol *'2'* with the suffix *'3'*. The advantage of using a PST is that the number of nodes is always kept to a minimum level specified by its input parameters. It does so by excluding nodes that hold stochastic information already known to the tree. Compared to the tree inferred by the sliding windows algorithm, see figure 5, the PST is parsimonious by means of keeping the number of nodes to a minimum and by putting a restriction to its height.

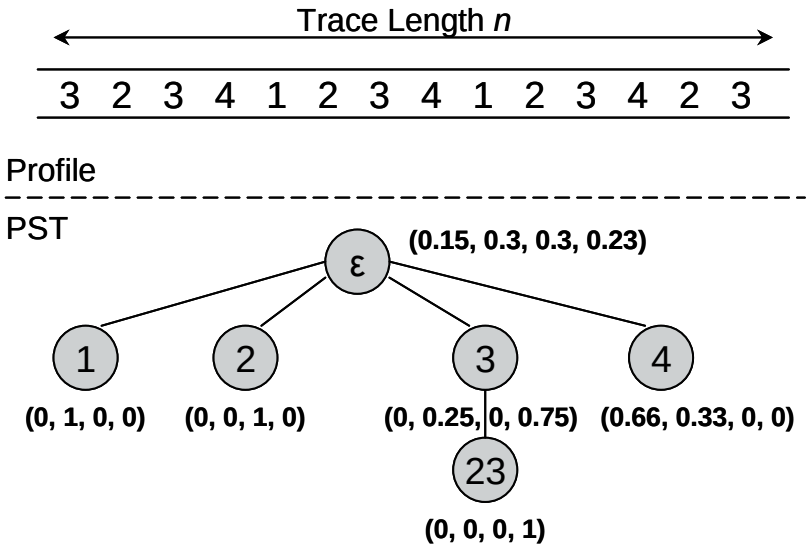

Figure 6. An example of a PST inferred from a training sample

In a PST, each node holds the probability of the next node's symbol as it can be inferred by the training sample, $P(\sigma|\text{suf}(\sigma))$. For example, as illustrated in Figure 6 the probability of symbol *'2'* to appear after a symbol *'3'* is given by the conditional empirical probability, P(2|3) = 1/4. This is calculated by dividing number of times that *'32'* occur in the training sample by the number of times that the suffix *'3'* appears. The last symbol in the training sample is not taken into account since it is unknown what symbol follows it.

The algorithm that is used to build a PST is depicted in Figure 7 [36]. This algorithm is comprised of three steps requiring five input parameters in addition to the training sample: The minimum probability $P_{\min}$ for which strings are

required to occur, a parameter α that defines the significance threshold for a conditional appearance of a symbol, the smoothing factor γ_{min}, a threshold factor r that describes the conditional appearance of descendant node from its ancestor, and the maximum length L of any permutation of the string s that can appear in the tree.

According to Figure 7, the first step in constructing a PST is to initialize the root node and its next symbol probabilities $P(\sigma)$. Now, every symbol σ that has a significant probability of appearing in the training sample becomes a child of the root node. Each of these symbols (or strings) is further used to grow permutations of strings that can appear in the final PST. This is done by appending another symbol from the alphabet to this string. Nodes are added to the appropriate parent in the tree if the string's conditional probability satisfies the conditions given in step 2c of Figure 7. The length of the strings grown is limited by L, which essentially restricts the height of the tree. Finally, the last step is required since no symbol is absolutely impossible right after any given subsequence i.e. this ensures that any node in the tree has a next symbol probability greater than zero.

Matching sequences in a PST is performed by calculating the cumulative probability of occurrences while traversing the tree. This is done by matching the symbols in the sequence one-by-one from the root down to the leaves in the PST.

If the subsequence for some reason does not exist in the tree, e.g. *'42'* in Figure 6, then the first symbol of the subsequence is removed until the sequence exists in the tree or the subsequence becomes empty. For example, matching the sequence *'423'* yields the cumulative probability of:

$$P(423) = \gamma_{root}(4) * \gamma_4(2) * \gamma_2(3) = 0.23 * 0.33 * 1 = 0.0759$$

Build-PST (P_{min}, α, γ_{min}, r, L)

1. Initialization: let T be a PST that consists of a single root node (with an empty label), and let $S \leftarrow \{\sigma \mid \sigma \in \Sigma \text{ and } P(\sigma) \geq P_{min}\}$.
2. Building the PST skeleton: While $S \neq \Phi$, pick any $s \in S$ and do:
   a) Remove s from S
   b) If there exists a symbol $\sigma \in \Sigma$ such that
      $P(\sigma|s) \geq (1 + \alpha)\gamma_{min}$
      and $P(\sigma|s) / P(\sigma|suf(s)) \geq r$ or $\leq 1/r$
      then add the node corresponding to s to T, and all the nodes on the path to s from the closest parent in T that is a suffix of s.
   c) If $|s| < L$ then add the strings $\{\sigma's \mid \sigma' \in \Sigma \text{ and } P(\sigma's) \geq P_{min}\}$ (if any) to S
3. Smoothing the prediction probabilities: For each s labeling a node in T, let $\gamma_s(\sigma) \equiv (1 - |\Sigma|\gamma_{min}) P(\sigma|s) + \gamma_{min}$

Figure 7. The Build-PST algorithm as described by Ron et al. [36]

Modeling application behavior using a PST is done by sliding a window over the most recent function calls. Figure 8 shows the output of smoothed PST, based

on the same training sample as in Figure 6, applied to a benign application behavior using a window size of three. Note that the output of a PST applied to any malicious behavior, or behavior not incorporated into the training sample, would have a significantly lower probability, which results in a large deviation from this model.

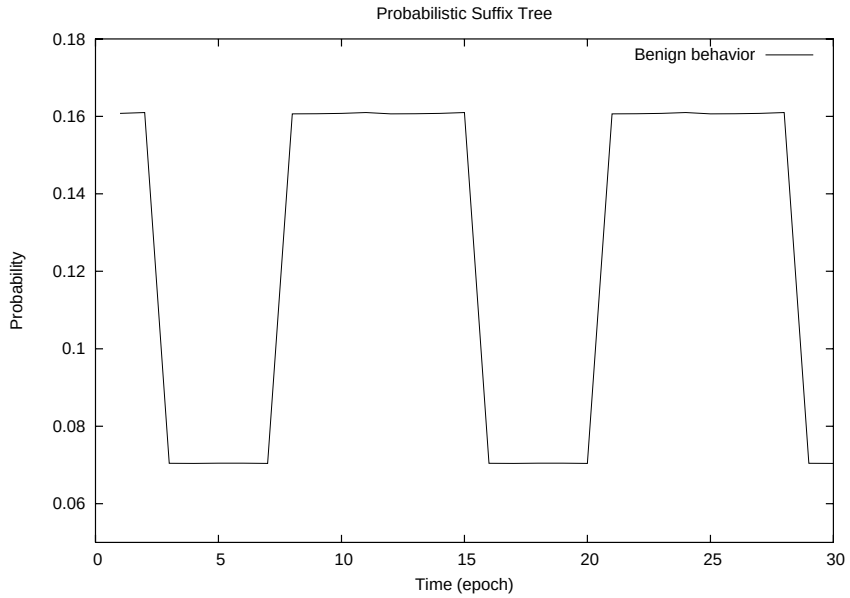

Figure 8. Benign application behavior measured by a PST

5.1.3 Misuse-Based Detectors

Support for configurable embedded detectors is provided by the IDS-kernel, and are referred to as misuse detectors in MIDES where the IDS-kernel maps the

outcome of a detector directly to a response. Misuse can be detected naturally in different places in source code including the middleware layer. There are implemented two embedded detectors in MicroQoSCORBA that exemplifies the use of configurable detectors. One of these detectors has been made to detect an excessive number of connections made by a client to a server as an indication of misuse and resource starvation. The second detector embedded in MicroQoSCORBA checks the integrity of the application at start up. This is particularly useful for detecting attempts to change the functionality of the application or the integrity of any of the application dependencies. The logic around these is further discussed in the next section as they are a part of the application-based security policies provided by MIDES.

5.2 Application-Based Security Policies in MIDES

Adversaries sometimes expend significant time and resources to find vulnerabilities in applications, and frequently use simple methods of testing to see if parts of the application have been ignored or poorly tested [49]. Messages can be carefully crafted to imitate a legitimate device and hence tweak values to compromise the integrity and availability of the application. In this situation, the

integrity is compromised when the system is processing a forged message. An attack on the availability is successful if this message results in a fault that terminates the application, e.g. an uncaught divide by zero fault.

Application-Based Security Policies in MIDES					
Policy	**Interval based**	**Procedural based**	**Profile required**	**Average Stddev**	**Description**
Maximum Value	Yes	Yes	No	None	Maximum allowed value
Minimum Value	Yes	Yes	No	None	Minimum allowed value
Delta Value	Yes	No	Yes	None	Difference in value over time - $\Delta(V/t)$
Maximum average	Yes	Yes	Yes	Average	Maximum distance from moving average
CDF	Yes	No	Yes	Both	Cumulative Distribution Function

Table 1. The application based policies currently provided by MIDES. Column two and three indicates whether or not the application-based security policy can be configured for the given internal sensor. Column four indicates the policy's need for profiling. The column second to the last, indicates the necessity of using a computed statistical average or standard deviation for the given profile.

Table 1 shows the various reusable application-based security policies that are implemented in MIDES. The Maximum and Minimum policies can typically reveal application semantic errors such as invalid temperature ranges. Furthermore, sudden increases in a value over a short period of time may also be determined to be abnormal. Similarly to the absolute minimum and maximum

temperature, it takes considerable time to physically heat a mass such as air or water. Delta value is a policy that is implemented with the intention to detect unexpected variations in values over a specified amount of time.

There are cases where comparing a value to its absolute maximum or minimum is insufficient. Measuring the temperature throughout a whole day would typically vary depending on the time of day. Maximum and minimum values will only create an artificially wide bound that is not able to detect smaller variations in a data set. Policies such as Maximum of Average can be used to check the maximum distance to the moving average for each sampled value, which overcomes this particular shortcoming of the maximum and minimum policies.

The *Cumulative Distribution Function* (CDF) implemented is an application-based policy that can be applied to distinguish rare values from values that occur frequently in a trace [22]; this requires that the data collected is normally distributed.

Also indicated by Table 1, the maximum, minimum and maximum of moving average policies can be applied to the output of a procedural based analysis, which essentially is a measurement of the likelihood that the trace of the execution pattern will occur.

The application-based security policies as presented in Table 1 do not address the application-based security policies for embedded detectors. This is due to the reason that policies for embedded detectors are not as general as for the procedural or interval based sensors. There are two different embedded detectors integrated into the MicroQoSCORBA middleware framework that tie directly to the implementation of MIDES. The application-based policies can be described as follows:

- There must not at any point in time be more than a configured number of connections from a single client to a given server.

- Any connection attempt by a single client to a given server must be longer apart in time than the configured time period.

- Any initializing instance of a client or server application must ensure the integrity of its own executable and any other external dependencies.

The first two policies as described here are embedded into the middleware framework at the point where connections are accepted or rejected by the server. A developer can specify the number of invocations per second, and the maximum number of simultaneous connections from a single client. See Appendix B for

how these application-based security policies for embedded detectors can be configured. These application-based security policies can detect attempts that compromise the availability of a device by creating an excessively amount of network connections consuming all of its network resources.

To ensure correct operation of the application, it is important to verify the integrity of the application or its external dependencies, e.g. dynamically linked libraries and configuration files. This application-based security policy is embedded into the middleware framework and checks the integrity of these dependencies at start up of the application. A developer can specify the files and checksum algorithm that are to be used in this operation. Similarly, if the device does not have a file system, but uses a flash memory instead to store its firmware, the same operation could be implemented to check the integrity over the memory area in which the application is stored.

These application-based security policies are implemented to exemplify the use of configurable embedded detectors targeting misuse within a middleware framework. Both responses and configurations for the application-based security policies can be configured transparently of the application.

5.3 Transparent Support for Application-Based Responses

Responses as described for the EMIDS model have been implemented in MIDES, and can be configured transparently of an application. Table 2 gives a brief overview of the implemented responses in MIDES, not counting individual responses as they relate to specific internal sensors.

MIDES is implemented as a part of a middleware framework that encapsulates application-level domain objects and interactions. This provides the opportunity to react to abnormal behavior or anomalies in real-time possibly before any harm could be done to the system. For example, the interval-based sensor using the value-based mechanism in MIDES can be configured to block an invocation as a response to an abnormality.

MIDES can similarly to *process Homeostasis* (pH), an IDS integrated into the kernel of a UNIX system [42], slow down parts of an application or network connection by embedding a time delay each time the code for a procedural-based sensor is executed. Audit and Time-delay are the only responses provided by MIDES, as indicated by the second column in Table 2, which are able to target the application.

Intrusions can be performed remotely in which responses can be targeted at a particular connection or all connections from a specific host. Responses that target the communication link between the two hosts include audit, time-delay, termination of the connection, and banning the IP-address of the remote host. Table 2 column three indicates the target for a corresponding response.

Application-Based Responses in MIDES			
Response	**Risk**	**Target**	**Description**
Audit	L	A / C	Generate an audit record
Time-delay	M	A / C	Delay the connection for a period of time
Terminate Connection	H	C	Terminate the connection to remote host
IP-Ban	H	C	Ban the IP-address of a remote host

Table 2. General responses that are supported by MIDES. The risk in case of a false positive are categorized as Low, Medium, or High (L/M/H). The target of the response can either be Application or Connection based, (A/C).

Time-delay can be configured to slow down a connection or a part of an application for a limited amount of time which depends on the severity threshold of the detected anomaly. IP-ban is used to ban an IP-address either permanently or over a limited period of time where the length of the period depends on a computed severity threshold. In the current implementation of IP-ban, the ban can be made permanent if a device is frequently being banned. Audit stores a pre-defined number of audit records in memory available to the application.

Additionally, it can be configured to dump all events to a file, or the standard output. The terminate connection response terminates the connection from a specific host with immediate effect.

All of the responses incorporate a common list of sources that can be trusted at any time, which stops a response from being activated. This list can be dynamically updated to accommodate a changing environment. Furthermore, responses can be configured to trigger at specific severity levels. This means that several responses can be configured for the same policy, but act at different severity levels. For example, an audit response can be triggered by small deviations from a policy, while more adverse deviations can result in triggering the audit and some other responses that are more severe e.g. Time-delay.

The second column in Table 2 illustrates the level of risk that is associated with each response. This risk is related to the case of a false positive where the response may possibly induce a self inflicted denial of service. The risk of a response can frequently be compared against external risks to the device. Performing a self-destruction by erasing all the memory areas is an example of a response with severe risk in case of a false positive. However, if the device is carried by soldiers and the risk of it being seized by the enemy is greater than if it would accidentally unavailable to one soldier, then self-destruction could be an

appropriate response. This is something that the application developer has to consider in each case before deploying a response.

Chapter 6

Performance Evaluation of MIDES

MIDES was implemented as a configurable and subsettable module of the Java version of the MicroQoSCORBA middleware framework [27][28][29][30]. It should be noted that all of the capabilities of MIDES could be implemented in the C++ version of MicroQoSCORBA; this would involve little more than a straightforward translation from Java to C++.

6.1 Experimental Setup Configuration

To illustrate the cross-platform capability, the test cases were run on two different platforms; one using a standard x86 configuration and the second using a Dallas Semiconductors TINI board [46]. For the first platform, two desktop PCs with a distribution of Linux Slackware 9.1 running the new 2.6 kernel were used. These identical PCs were powered by an Intel(R) Pentium(R) 2.4 GHz processor and

equipped with 1 GB of Memory. The test cases were compiled and executed by using Sun's Java 2 Software Development Kit, version 1.4.2-04, over a 100-Mbps network.

The TINI boards are powered by an 8-bit DS80C390 CPU running at 40 MHz and equipped with 512 Kbytes of memory. The test cases for this platform were run on a custom JVM that is compatible with a subset of Java version 1.1 [46], over a 10-Mbps network. Before the test cases could be run on the host board, they had to be converted into a compressed format suitable for the embedded devices by using the TINI-Convertor version 1.02e.

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

Source Listing 1. Interface description for the test application

The experiments were conducted by repeatedly executing `foo.bar(…)` as described by the Interface Description Language (IDL) specification in Source Listing 1. This example does not illustrate a real world example, but rather a simplified example made for the purpose of measuring timing and resource usage while changing the IDS parameters. Function `foo.bar(…)` simply returns the same value as given by `arg1`, see Appendix C for further implementation details.

There are several other factors that can influence the timing and resource results. Using different hardware architectures does not directly allow for a fair comparison e.g. the difference in memory makes the garbage collector run more frequently on the TINI boards than it does on the desktop PCs. Other sources that can make a difference includes CPU speed, native word length, jitter in network latencies and lost packets, frequent garbage collection, different JVM implementation, and noise generated by the operating system.

In order to get a precise measurement by running the test cases, the system had to be brought up into a steady state where additional overhead costs related to initialization could be eliminated. On the Linux platform, this would allow the Java Hotspot JVM Just-In-Time compiler to boost the performance of the test application. The timing measurements presented in this paper are additionally filtered through a filtering algorithm that virtually eliminates the overhead of the Java garbage collection and abnormally long network latencies, i.e. latencies influenced by lost network packets or high network traffic. This overhead has been measured to be about 5% for the desktop system, and almost 25% for the TINI system [28][29].

The filtering algorithm assumes that the end-to-end latencies are normally distributed, and will filter out less than 0.02% of the desired latencies. This

filtering algorithm has shown strong effectiveness and correctness in calculating the end-to-end latencies, even with computationally expensive tasks running in the background. Nonetheless, each case was run three times with 3 million iterations on Linux and 1000 iterations on TINI. Only the best case (lowest) values were chosen in order to eliminate any possible jitter and to generate a highly accurate result. The memory usage results reported is an average of the collected values while executing the test cases.

All the tests were configured to use IIOP version 1.2 (CORBA's General Inter-Orb Protocol over TCP/IP) [35]. Also, the client and server was carefully designed and configured so that the responses of the IDS were never triggered. In other words, the results describe the cost of the IDS on normal application-level interactions, not the cost of the IDS issuing a response.

6.2 Test Results

The goal of the tests is to measure the general performance cost overhead of each different part of the IDS as it may be configured. Knowing the overhead cost of a using a particular IDS mechanism will enable a developer in an early stage of design to rule out any mechanisms that are too big or slow. This section provides

end-to-end latency and memory usage results of the different configuration of profiles, responses and policies. It will additionally provide results for application sizes arranged by the different sensors types. Finally, it provides a scalability analysis in terms of end-to-end latencies and memory usage by increasing the number of sensors inserted into the application or framework, or by increasing the numbers of unique data points that appear in a training sample.

6.2.1 Application Sizes

In Table 3, one can see a comparison of the application sizes while inserting different internal sensors into an application. This is accomplished by calculating the sizes of the Java byte-code class files. The application executables for TINI are compressed and converted into binary files ready for download and execution. This compression makes the executables for TINI unsurprisingly smaller than the Linux executables.

The interval-based sensor incorporates a maximum policy and an audit response that makes it naturally smaller than a procedural-based sensor, which incorporates a few extra data structures and an algorithm. Misuse-based detectors do not require any overhead of either policies or profiles and therefore require less

space. Note that these numbers do not include any extra overhead related to installing the Java virtual machine required to run the application.

Sensors/Detectors	Application sizes (bytes)							
	Linux				Tini			
	Client	%	Server	%	Client	%	Server	%
Baseline	63735	-	61250	-	23869	-	20599	-
Interval	83060	30.32%	80584	31.57%	29792	24.81%	26527	28.78%
Procedural (SW)	89286	40.09%	86854	41.80%	31711	32.85%	28393	37.84%
Misuse	87416	37.16%	78020	27.38%	33137	38.83%	29822	44.77%

Table 3. Application sizes listed by type of sensors integrated

6.2.2 End-to-End Latencies and Memory Usage

The end-to-end latencies as presented in Table 4 are measured by invoking the `foo.bar(…)` repeatedly using different configurations of sensors and application-based security policies. Maximum of Average and CDF compares the received value with a trace the 20 past values. There is a significant difference between polices that do not require a profile and those policies that do e.g. Maximum of Average and CDF. Increasing the length of the trace holding the previous values will increase the amount of computation required and thus increase the end-to-end latencies.

Because of the low resource availability on the TINI boards, it runs significantly slower than the PCs. This fact is especially reflected in the test cases

where the number of computations is significantly higher, i.e. when analyzing a profile by applying the Sliding Windows algorithm or the Cumulative Distribution Function. For the desktop PCs the end-to-end latencies for the CDF policy increases only by a few percent compared to the baseline. Thus, the TINI boards experience an increase of approximately 75% compared to its baseline.

Policies	End-to-End Latencies (ms)							
	Linux				Tini			
	Client	%	Server	%	Client	%	Server	%
Baseline (no ids)	0.117	-	0.117	-	254.44	-	254.44	-
Interval-based								
Maximum	0.122	4.27%	0.122	4.10%	290.48	14.16%	295.76	16.24%
Minimum	0.123	4.96%	0.122	3.93%	290.57	14.20%	295.41	16.10%
Δ(V/T)	0.123	5.38%	0.124	6.32%	307.43	20.82%	314.32	23.53%
Maximum of Average	0.124	5.73%	0.124	5.81%	310.54	22.05%	317.64	24.84%
CDF	0.124	6.32%	0.125	6.84%	426.88	67.77%	445.82	75.21%
Frequency-based	0.123	5.13%	0.122	4.27%	314.62	23.65%	317.60	24.82%
Procedural-based								
Sliding Windows	n/a	-	0.130	11.11%	n/a	-	555.83	118.45%
PST	n/a	-	0.125	6.84%	n/a	-	297.00	16.73%

Table 4. End-to-end latencies listed by the type of policies

Comparing the least and the most resource demanding policies on client-side to the baseline increases the latencies by 2.5% to 6.9% on the Linux based PCs. On the TINI boards, this overhead increases the end-to-end latencies by 14% to 75% from the best case to the worst case. Using the interval sensor to measure the rate of invocations and response-time is indicated by the frequency-based row in

Table 4. A few extra system-calls and object creations cause only a minor increase in the end-to-end latencies.

Policies	Memory Usage (bytes)							
	Linux				Tini			
	Client	%	Server	%	Client	%	Server	%
Baseline (no ids)	100968	-	109852	-	82832	-	79953	-
Interval-based								
Maximum	104122	3.12%	113221	3.07%	85536	3.26%	91090	13.93%
Minimum	104122	3.12%	113204	3.05%	85525	3.25%	90301	12.94%
Δ(V/T)	104309	3.31%	114084	3.85%	87072	5.12%	91279	14.17%
Maximum of Average	105285	4.28%	114767	4.47%	87893	6.11%	92314	15.46%
CDF	105269	4.26%	114758	4.47%	88160	6.43%	90759	13.52%
Frequency-based	104090	3.09%	113806	3.60%	85514	3.24%	91406	14.32%
Procedural-based								
Sliding Windows	n/a	-	116160	5.74%	n/a	-	93693	17.19%
PST	n/a	-	116912	6.43%	n/a	-	102874	28.67%

Table 5. Memory usage listed by type of policies

The procedural-based detection mechanism can either be configured with the Sliding Windows algorithm using trace length of 20 values and a window size of 3, or the PST algorithm using a window of same size. This overhead is measured by including one sensor inserted into server implementation. This increases the overhead of performing the invocations by about 6.8% to 11.1% on the Linux based PCs dependent on the algorithm applied to the profile. The increase of the end-to-end latencies is significantly larger on the TINI boards ranging from 17% to 118% for the respective Sliding Windows or PST algorithms.

According to Table 5, introducing the application-based security policies increases the memory overhead on the Linux based PCs by 3.1% to 4.28% and 3.1 to 4.5% for the client and server respectively. This overhead is significantly larger on the TINI boards, where the increase ranges from 0.7% to 3.8% on the client-side and approximately 13% to 14% on the server-side.

Only time-delay and IP-ban of the aforementioned responses as indicated in section 4.4 requires a direct implementation into the middleware layer. The code for IP-ban is executed upon any connection to the host as compared to the code for time-delay which is executed for each invocation. Hence, only the time-delay response introduces any significant delay to the end-to-end latencies. The overhead introduced by this response ranges from 1.9% to 3.8% on Linux and 15.9% to 38.2% on TINI compared to the baseline.

6.2.3 Scalability of the Data Collection Mechanisms

This section addresses the scalability issue of introducing data collection mechanisms for the procedural- or interval-based sensors inserted into the application or middleware framework respectively. The end-to-end latencies were measured in a similar way as the results presented in the previous section, but now by increasing the number of sensors embedded into the middleware

framework or application. Similarly, the memory usage for the procedural-based mechanism was measured by increasing the number of data points in the training sample. The procedural-based sensors were tested with the Sliding Windows algorithm using a total trace length of 20 and a window size of 3, and the PST using the parameters L = 3, P_{min}=0.01, α=0, $\gamma_{\min}$=0.001, r = 1.05 and a window size of 3. The interval-based sensor was configured as described in the previous section with a maximum policy and an audit response. Figure 10 and 11 show the memory usage by increasing the number of sensors inserted into the middleware framework, and not by introducing different types of policies analogous to the increasing the number of data points in the training sample for the procedural-based mechanism.

Figure 9 illustrates the increase in end-to-end latencies while changing the number of data points that are inserted into the server implementation. It is evident that the Sliding Windows algorithm is significantly slower than the PST. As it can be observed from Figure 9, the interval-based mechanism yields the lowest values. This is also expected since the number of required computations is fewer than any of the procedural-based mechanisms.

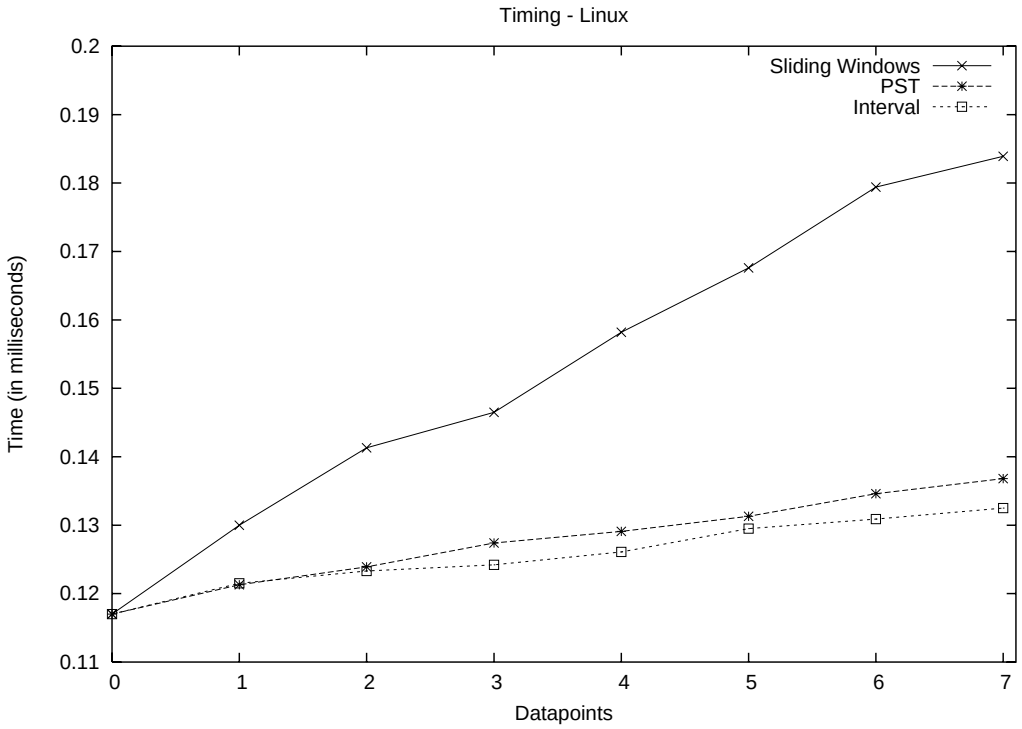

Figure 9. End-to-end latencies when increasing the number of data points on Linux

The same result is reflected in Figure 10, which describes the increase in end-to-end latencies using the TINI based platform. Alas, the number of data points that finished the test were fewer than for the Linux based platform. The eliminated data points were simply too slow or unreliable for the test application, and hence disregarded. It can be visually observed that the end-to-end latencies for both test platforms increases linearly with the number of data points. Thus, the

coefficients change with the type of data collection mechanism and the amount of computational power available to the system.

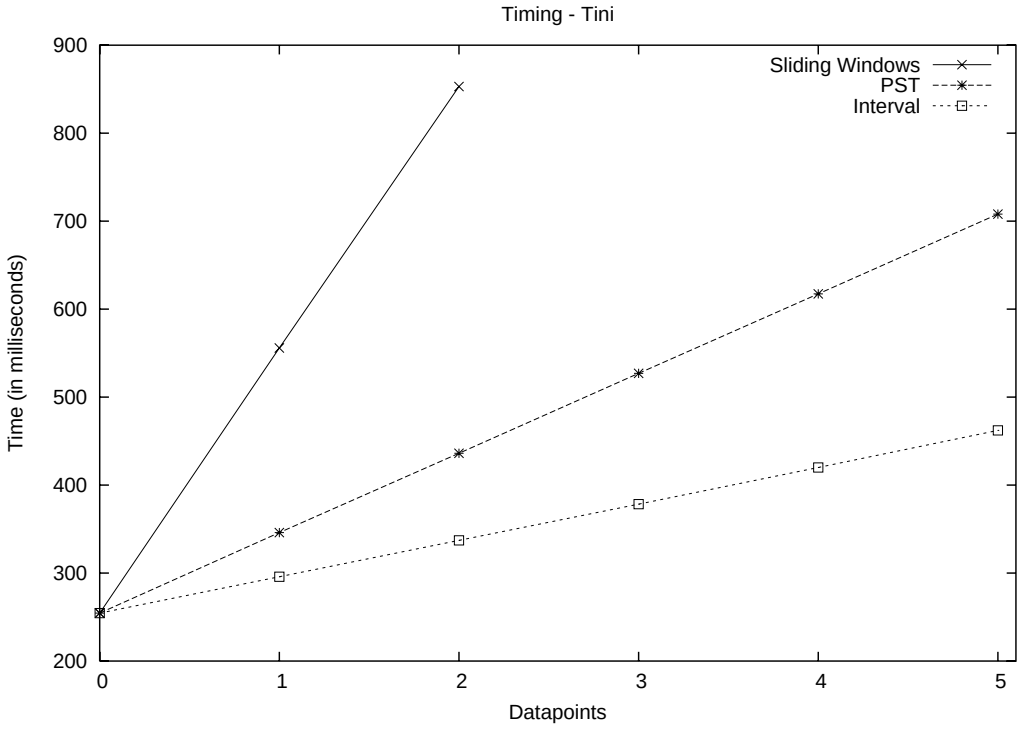

Figure 10. End-to-end latencies when increasing the number of data points on TINI

Figure 11 and 12 compare the resource usage in terms of memory on the different platforms. These values are an average of memory usage sampled during the execution of the test application. The result for the Sliding Windows and the PST algorithms differs significantly between the two test platforms. The PST algorithm uses more memory due to its inherent complexity. However, this could

be optimized by using fewer abstractions and reducing the size of large data structures in the implementation. The peak that can be observed in Figure 11 can also observed in Figure 12. This peak which cannot be observed for the Sliding Windows algorithm is caused by the varying size of the PST. The PST is optimal in both speed and memory for use with relatively big applications that has a large quantity of unique data points [26][36]. However, this is not so for the memory aspect of the test applications used in this research.

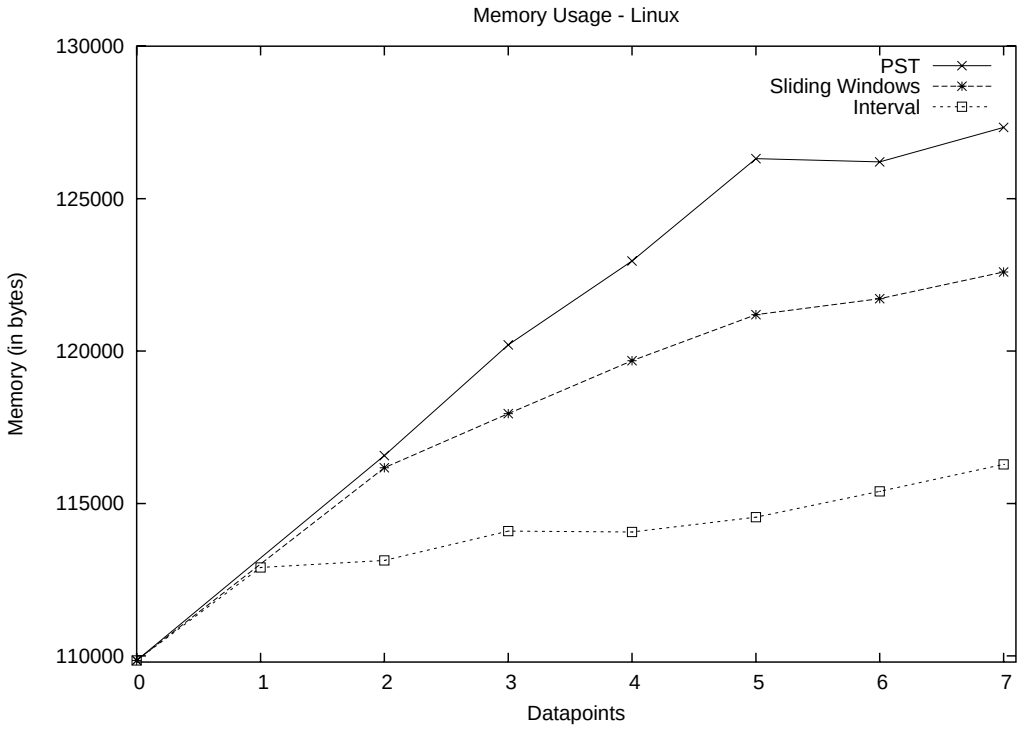

Figure 11. Memory usage when increasing the number of data points on Linux

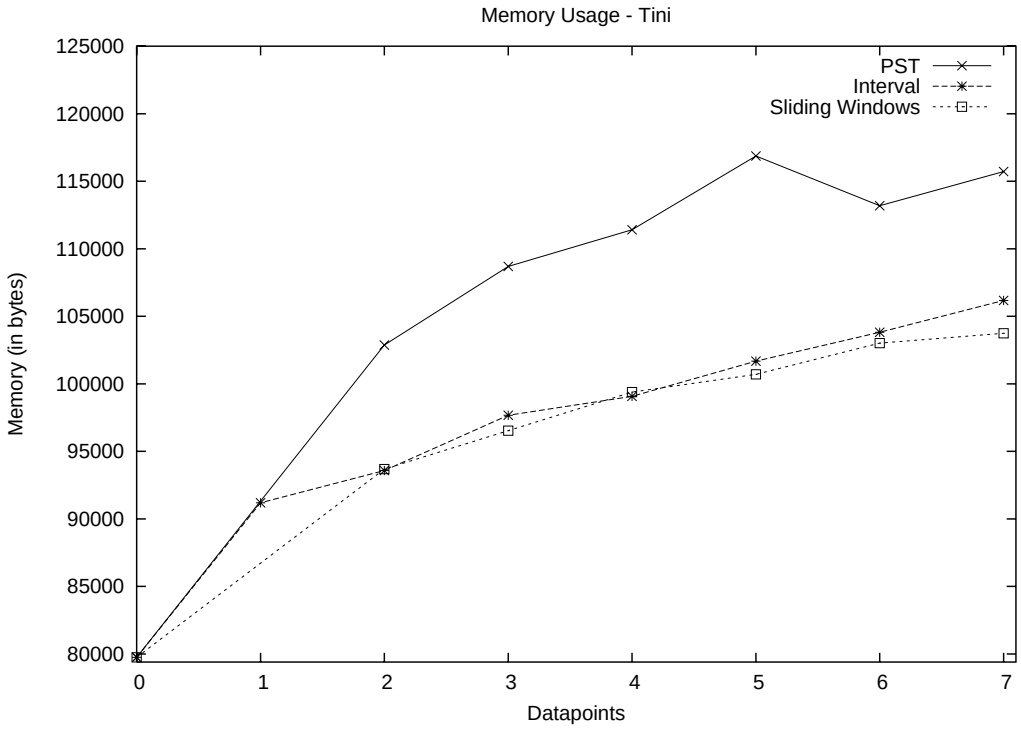

Figure 12. Memory usage when increasing the number of data points on TINI

6.2.4 Experiment Summary

Chapter 6 reported an estimate of cost overhead related to performance and resource usage by introducing an EMIDS integrated into a middleware framework. Subsection 6.2.1 described the general cost overhead in terms of an increasing memory footprint required for an embedded application using any of the provided sensors or detectors in MIDES.

Subsection 6.2.2 presented the end-to-end timing results by invoking `foo.bar(…)` repeatedly using different configurations of sensors and application-based security-policies. It additionally gave a brief overview of the memory usage that is associated with each configuration. Furthermore, it presented a brief discussion on the effect that directly implemented responses have on the end-to-end latencies.

The following section presented measurements that address scalability issues of the data collection mechanisms. This is subsequently performed by inserting an increasing number of sensors embedded into the application or the middleware framework; or by increasing the number of unique data points in the training sample.

From the presented results, the resource usage varied significantly on the given test platforms. The end-to-end latency results differed only a few microseconds on the desktop PCs, which essentially are more network bound than computationally bound. On the other hand, the result for the TINI boards differed by several milliseconds. Compared to the desktop PCs, it is evident that the TINI boards were limited by both the computational power and the network connection. Also, consider that the reported results are based on an application that only returns the value of its argument. Any embedded application that expend a few

seconds performing other tasks before a reply is sent back to the client would have a smaller performance impact in terms of percentage. It should also be taken into consideration that TINI are using a system clock at 40 MHz in contrast to the desktop PCs using a 2.4 GHz system clock. Besides the architectural differences in hardware, the low memory availability on TINI makes the Java Garbage collector run more frequently. The frequency of garbage collection has been measured to be about 5% for the desktop PCs running Linux, and 25% for the TINI boards [28][29].

Chapter 7

Related work

The presented research in this thesis is related to a number of different research areas. This chapter provides a brief overview of some of the most important related work. The first subsection presents related work in the area of procedural detection and internal sensor systems. Subsection two and three discuss related work in the area of embedded systems and middleware with emphasis on intrusion detection.

7.1 General Intrusion Detection Systems

The *Embedded Sensor Project* (ESP), which is a research project at Purdue University, developed a framework for intrusion detection using internal sensor and embedded detectors, and additionally proved their feasibility in both a network- and host-based environment [43][50]. The concepts and research behind

ESP is a fundamental building block for the EMDIS model and the MIDES framework presented in this paper. EMIDS further extends the model behind ESP by adding additional primitives and applying it to the middleware layer.

CylantSecure is a behavioral-based intrusion detection and intrusion prevention system that integrates internal sensors into the kernel of an operating system [8]. This approach analyzes the specific behavior of the operating system under use by a running application. The tight integration of the HIDS and the kernel of an operating system enabled CylantSecure to block behavior that can be categorized as unauthorized or malicious in only a few milliseconds. Procedural detection in CylantSecure and MIDES builds on the same fundamental building block as presented by Elbaum et al. [13]. In contrast to MIDES, the procedural based sensors used in CylantSecure were fitted retroactively into the kernel instead of being integrated as a part of the development process.

Internal sensors have also proven useful for detecting deviations in the system calls made by an application. Somayaji et al. [42], describes an IDS called *process Homeostasis* (pH), which is embedded into the kernel of an operating system and capable of automated responses such as time-delay embedding and aborting system calls. pH infers the application behavior based on the correlation of system calls and not the execution flow of an application as in MIDES.

There are very few application-based IDSs to date. Application-based intrusion detection has been described in a purely theoretical aspect by Robert S. Sielken [40]. Thus, recent advances in intrusion detection have put focus on application-based IDSs and their practicality, i.e. Almgren et al. [1] describes an application-level approach for data collection and intrusion detection integrated into the Apache web-server. Similarly, a commercial application-based IDS targeting web-servers is named AppShield and is available from Sanctum Inc [38]. These application-based IDSs targets a particular type of application intended for the high-end computing market in contrast to MIDES that augments a wide range of middleware-based applications for embedded systems.

Another interesting area where intrusion detection has made recent progress is Fraud Management Systems (FMSs) deployed by banks or telecom companies, [24]. These systems analyze the pattern in a customer's behavior, i.e. abnormal expenditures in billing- or bank- statements. For example, if your credit card is used to buy gas within a specified timeframe at two different gas stations hours away, it could respond by blocking the potential fraud. In many scenarios, a FMS would typically deploy application-based security policies at a higher level than MIDES, i.e. by looking at the billing- or bank- statements individually and not at the level of a single transaction as it would appear to MIDES.

7.2 Intrusion Detection for Embedded Systems

Cisco has embedded a set of mechanisms for network intrusion detection in their new generation of routers that can be easily managed and configured. This NIDS can be configured to take action such as dropping the packet, resetting the TCP connection and reporting to a centralized syslog or management server. The NIDS integrated in these routers incorporates a network-based sensor that performs a signature-based detection of packets as they are transmitted into or out of a network segment [7].

Arbor Networks designed and implemented Peakflow-X, which is an architecture including advanced embedded systems for a network-wide data collection, analysis, and anomaly detection [3]. As a part of this architecture, collectors and controllers are deployed at strategic places in the network. Peakflow-X is intended as a supplement for a signature driven NIDS, and is interoperable with other network components such as the previously described Cisco intrusion detection enabled routers for a more complete network protection.

These are both examples of embedded systems that perform intrusion detection to enhance the overall protection of a corporate network. In contrast to

MIDES that performs application level intrusion detection, these systems try to detect and prevent malicious behavior at the network level.

7.3 Middleware-Level Intrusion Detection

Applications that Participate in their Own Defense (APOD) [4], based on the *Quality of Objects* (QuO) middleware [51], uses a set of mechanisms to protect against network borne attacks. These mechanisms include elements from the area of intrusion detection such as the lightweight signature driven intrusion detection system Snort [41]. It additionally uses mechanisms such as TCP stack probes (Netstat) [37], IPTables [21], and various other mechanisms to enhance the overall level of intrusion tolerance. This approach uses a form of indirect monitoring where the IDS depends on other external applications or features provided by the operating system. Because of its external dependencies, it does not scale very well down to the smallest embedded systems. For example, TCP stack probes could be useful for the EMIDS model as it can detect and protect against a range of various known attacks, e.g. ARP cache poisoning. Thus, the MIDES was carefully designed not to incorporate any operating system level

mechanisms as they could potentially limit its portability and scalability to the smallest resource constrained embedded system.

Intrusion detection has previously been addressed by use of Byzantine fault detectors to solve the Byzantine problem and distributed consensus [24]. Group based systems can sense intrusions by use of these detectors and respond by expelling the member. There exist CORBA middleware frameworks that address the problem of intrusion detection and solve the Byzantine problem as an approach to intrusion tolerance [23][39]. Intrusion detection and fault tolerance as a combination is interesting and useful; however, the cost of intrusion tolerance is too high for the small networked embedded systems as targeted by MIDES.

The *Object Management Group* (OMG) has worked out various CORBA specifications. The Smart Transducer Interface Specification [34], and minimum CORBA specification [35], are the specifications that are most closely related to MicroQoSCORBA and the research presented in this paper. Smart Transducers are essentially small, single purpose devices such as sensors and actuators. None of these standards address intrusion detection and are not as configurable as MicroQoSCORBA.

Chapter 8

Discussion

This chapter focuses on some of the differences between a HIDS integrated at the operating systems level compared to an application-level IDS. It also provides a short discussion on false positives for the various mechanisms provided in MIDES.

The approach presented in this thesis raises the level of protection to include application-level intrusion across a distributed computing system instead of being limited to the scope of a single host, in a way that allows for reuse and richer policies. It is imperative that other means of security are well thought of before any release of the application. For example, any other concurrently running applications may contain security flaws that can be exploited by hackers to gain access to the systems. For larger embedded systems, it might even be useful to deploy a HIDS at the operating system level. Furthermore, the EMIDS model assumes that there is provided some protection at the network level. In other

words, it assumes a properly configured network. Perhaps, by using a network intrusion detection system such as the Cisco IDS enabled routers.

Host-based IDSs can terminate a specific application that compromises the security of the system or embed a time delay for each system call. Terminating the application can in certain cases prevent adversaries from getting root access to the system, but it can also result in denial of service for remote users that depend on this application. The philosophy of embedding a time-delay is to make an adversary believe that the attack did not work. However, a HIDS integrated into the operating system layer cannot easily distinguish which of the users of an application is the adversary. As a result of this, the response to an abnormal behavior could affect and possibly prevent other users from using the services provided.

In this case, a tighter integration between the application and the IDS is required. Distinguishing different parts of the application as they are used by local or remote users can be addressed if this integration between an IDS and application is handled at an early stage of development. Responses should in general not be made completely permanent, such as in the case of time-delay embedding where it can be used over a period of time to slow down and make the adversary believe the attack failed. Terminating the application that it is supposed

to be protecting is often ambiguous and therefore not a plausible response to an adversary.

Other IDSs have shown great effectiveness against various known host and network based attacks. Thus, very little is known about exploits that target embedded systems. Attacks against embedded systems are often easier illustrated by a brief description of the way they could compromise one or more of the CIA properties.

Until now, any discussion in regards to false positives has so far been omitted. As pointed out by Zamboni [50], embedded detectors have an accuracy of 100% since they look for specific signs of intrusions which require a particular condition to be true. However, this is different for the procedural-based detection mechanism. If a behavior is not incorporated into the behavioral model, it may either be considered as an intrusion or a false positive. In the latter case, it may be a result of improper training of the system [42]. In MIDES, the time it takes to build the profile can be configured to be any period of time from the time the model was last updated.

The case for interval-based sensors is a little different since they can both be modeled as an application constraint and as a profile of the system. For example, it is physically impossible to measure temperatures below the absolute zero; this

can be detected with 100% accuracy. Thus, application-based security policies such as the Maximum of Average can generate more false positives if the maximum distance to the moving average set by this policy is too small. In any case, thresholds have to be individually tested and tuned for the particular data set measured.

Chapter 9

Conclusion and Future Work

This thesis began with a general introduction to embedded systems, middleware, distributed systems and intrusion detection. Chapter 2 presents general background work in intrusion detection and CORBA. It additionally describes the MicroQoSCORBA middleware framework that was instrumented with the mechanisms for intrusion detection presented later in this thesis. Chapter 3 goes in depth on vulnerabilities and mistrust in embedded systems that are typically used in critical infrastructures. All of these chapters motivate and address the need for intrusion detection in embedded systems.

The model for Embedded Middleware-Level Intrusion Detection System designed for small embedded devices was presented in Chapter 4. This chapter also outlines possible methods of data collection and responses that can be very useful in a middleware based IDS. Chapter 5 presents MIDES, a prototype based on the EMIDS model that was integrated into MicroQoSCORBA providing

highly flexible and configurable support for middleware-level intrusion detection. This support for intrusion detection also incorporates a set of reusable application-based security policies and responses that are suitable for embedded system.

Chapter 6 presented the experimental evaluation of the provided mechanisms on two platforms. This evaluation gave an estimate of the resource overhead introduced by configuring any of the mechanisms for data collection with a different selection of application-based security policies. Furthermore, each of the data collection mechanisms were evaluated for their scalability in terms of increasing the number of sensors inserted into the application or middleware generated stubs and skeletons.

Related work as presented in Chapter 7 was divided into three subsections. These included related work in the area of intrusion detection, embedded systems and middleware with emphasis on intrusion detection. A discussion of general issues regarding application-level intrusion detection is provided in Chapter 8. This chapter discusses in particular how implementing responses in a traditional HIDS integrated into an operating system differ from implementing them into an application-based IDS.

In summary, the key contributions of this thesis are the following:

- A model for a middleware-level intrusion detection framework that can naturally infer some of the semantics of the distributed application.
- A fine grained design and implementation of a highly flexible and configurable framework for intrusion detection.
- A set of reusable application-based security policies based on the output of the data collection mechanism that is provided by this intrusion detection framework.
- An experimental evaluation of MIDES's performance on two hardware specific platforms.
- A scalability analysis in terms of resource usage of the provided mechanisms for data collection.

The model and framework for intrusion detection presented in this thesis are able to infer the application behavior through analyzing the execution flow of an application. This is a mechanism that has been proven effective against remote- and host- based attacks in other intrusion detection systems such as CylantSecure and pH [8][20].

Misuse or attacks against an application can be detected by using misuse-based detectors, which is a weaker form of embedded detectors. Internal- and

external- sensors as well as embedded detectors have proven useful in detecting both host- and network- based attacks [43][50].

This model and framework for application-based intrusion detection are able to analyze well defined data by scrutinizing arguments of function invocations between hosts in a distributed system. These possibly encrypted data are not very easily obtained by other lower-level IDSs. The nature of embedded systems and the type of data processed by them makes this mechanism inherently more interesting.

Furthermore, this thesis has shown the overhead of performing intrusion detection in a middleware framework on two platforms. The resource overhead on TINI, an embedded platform that was very resource constrained, was measured ranging from about 20% to 67%. For a system rich on resources, this was measured to be ranging from about 3% to 7%.

9.1 Future Work

Future work for EMIDS includes practical research experiments conducted empirically in a real-world application's development cycle to show the feasibility of the approach presented. Additionally, research could be performed to see if

application-based policies can be dynamically configured and made adaptable in a continuously changing environment. This research could further be extended to include research and development of mechanisms, techniques and responses for larger middleware frameworks targeting secure and dependable systems that is not specifically tailored for small embedded systems.

In some cases it is desirable for application-based policies to not take immediate effect. It may therefore be useful to model this as a stochastic process; this is also left out as future work.

Some work could also be done to optimize the process of embedding procedural sensors automatically. This would typically consist of a two-step operation where the sensors are embedded at all possible data points in an application, for then analyzing which part that of the generated trace can be eliminated.

Finally, parts of this research could be perfected by expanding the test application to contain functionality that better illustrate the goal and resource overhead of a real embedded application. In order to better demonstrate the feasibility of the approach taken, the test cases could be run on a third embedded platform that is more resourceful than the TINI boards.

Appendix A

A Refined Middleware Taxonomy

The large variety of embedded systems made it necessary to create a taxonomy that categorized some of the useful facets of embedded systems. This taxonomy, as presented in Table 5 [29], would enable a developer to early determine the applicable configuration options and the corresponding tradeoffs. The following four broad categories were of significant interest: embedded hardware, roles, software input/output, and IDL subsetting.

A priori knowledge of the hardware that is supported by the embedded application is critical to the middleware framework in order to appropriately constrain the code generation and other hardware specific optimizations. Typical options range from the level of hardware support for I/O operations and processing resources available to the device, to the higher level system composition of the distributed application. According to Table 5, this system composition can either be made asymmetric or symmetric. For example, the

number of severs in an distributed system are typically less than the number of clients, which allows the clients to be made by using less expensive hardware components.

The embedded system's role in a distributed system does affect the constraint and resource usage of a device. In this taxonomy, the definition of a role allows the middleware framework to precisely configure the application with only its needed functionality and no more. For example, a device such as a temperature sensor that only receives connection requests, but never initiate connections to other remote systems, can be created without the code for initiating remote connections.

The choice of communication support can also affects the constraints and resource usage of a device. Providing support for CORBA's Internet Inter-ORB Protocol (IIOP) [35] would be too costly for some embedded devices since this protocol relies on TCP. For this taxonomy, appropriate design choices given the level of communication support are outline in column 5 of Table 5.

CORBA's *Interface Definition Language* (IDL) is used to define an application's functional interfaces. This language specification often provides data types or data structures that are too resource demanding or complex for a resource constrained embedded system, i.e. 'Any's or composite data structures.

The choices of IDL-subsetting, as outlined in the last column in Table 5, are used to specify an application's requirement to support some of these functionalities.

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

Table 6. A refined middleware taxonomy for embedded systems

Appendix B

A Graphical Configuration Tool for MIDES

This appendix describes the *Graphical User Interface* (GUI) of the configuration tool used to configure MIDES. This GUI is made as a part of the configuration tool that is used to configure MicroQoSCORBA and its middleware specific options, e.g. options for protocols, transports, fault tolerance and encryption.

Figure 13 and 14 depicts the two main property sheets used to configure MIDES. From the property sheet illustrated in Figure 13, one can choose to either start the configuration wizard by clicking ‘Start Wizard’, view the composition of the configured data points, policies, profiles and responses by expanding the tree of configuration options, and at the same time observe the configured properties for each of the selected objects. If needed, a developer can select one of the debug options to output any messages generated by MIDES.

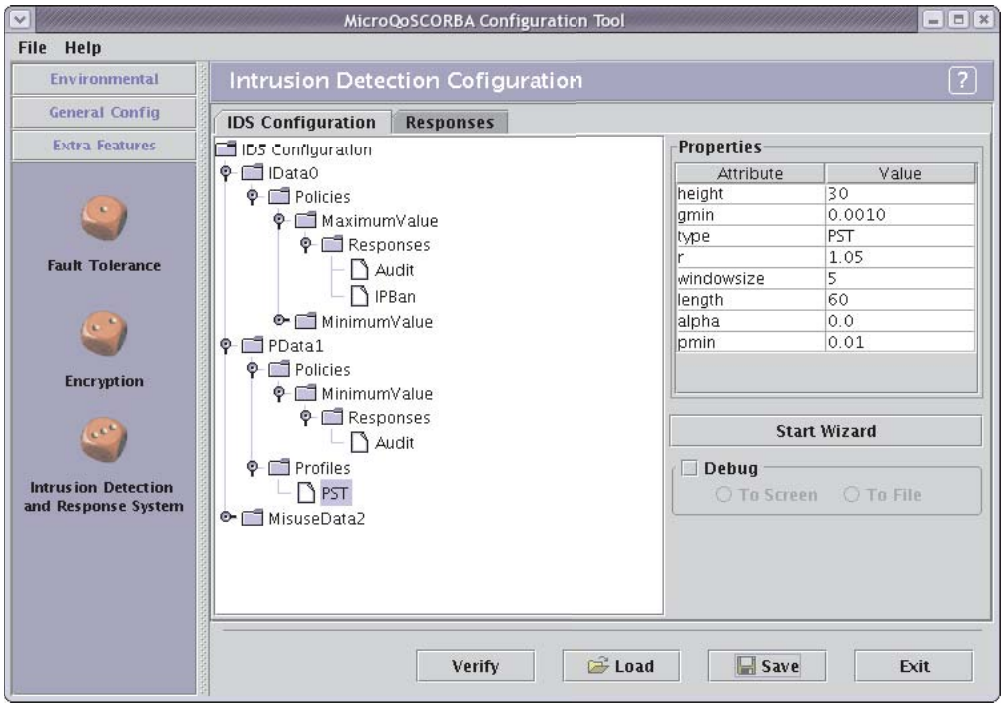

Figure 13. The configuration tool for MIDES

The objective of the configuration options provided in Figure 14 is to configure the responses that are used by any of the data points in Figure 13. Currently, the property sheet for the Audit response is shown. Nonetheless, each of the responses that can be configured is selected through the dropdown box. In addition to configure responses, a safe list can be globally configured to incorporate all permanently trusted devices, that is, all the statically configured devices that no response is to be activated.

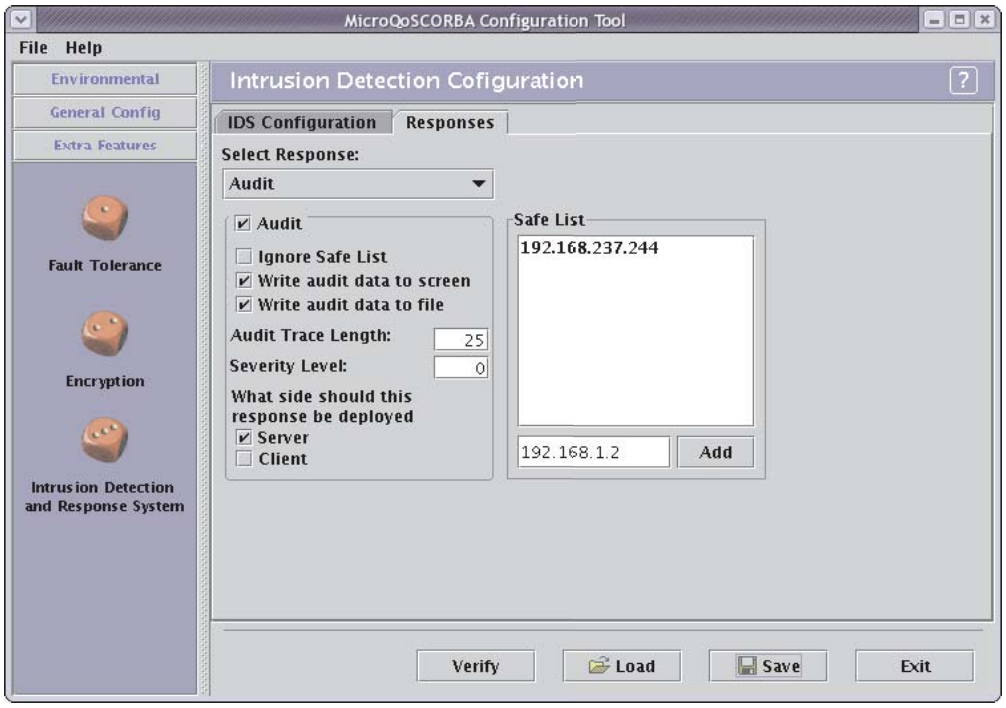

Figure 14. Configuring responses in MIDES

The ‘Start Wizard’ button available from the first configuration page invokes the ‘IDSConfigurationWizard’. This wizard provides an easy way to configure MIDES in 6 steps. Figure 15 illustrate the first step in this procedure where the type of mechanisms is selected for further configuration. For any of the interval-based mechanisms, the interface specification file commonly referred to as the ‘IDL-file’, needs to be loaded in order to specify the function or argument that is to be instrumented. The options for the procedural-based and misuse-based mechanisms need only to be checked before proceeding to the next step.

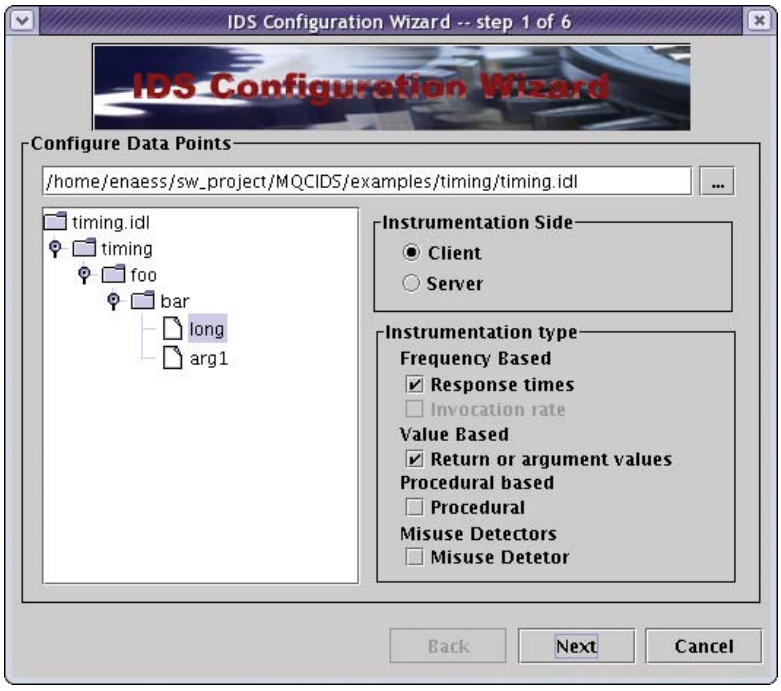

Figure 15. First step of the 'IDSConfigurationWizard'

The next step in the process of configuring the data points is to select the appropriate policies that are applicable to the specified mechanisms selected in step 1, see Figure 16. The policies that can be selected here are as described by Chapter 5. Starting from the top, the checkboxes enables policies for the maximum, minimum, delta value, maximum of moving average and the *Cumulative Distribution Function* (CDF). The textbox that specifies the step between each severity level is collectively used as an input to the policies in order

to calculate the severity levels of the violated policy. Step 2 is repeated for all the specified mechanisms in step 1, except the misuse-based mechanism.

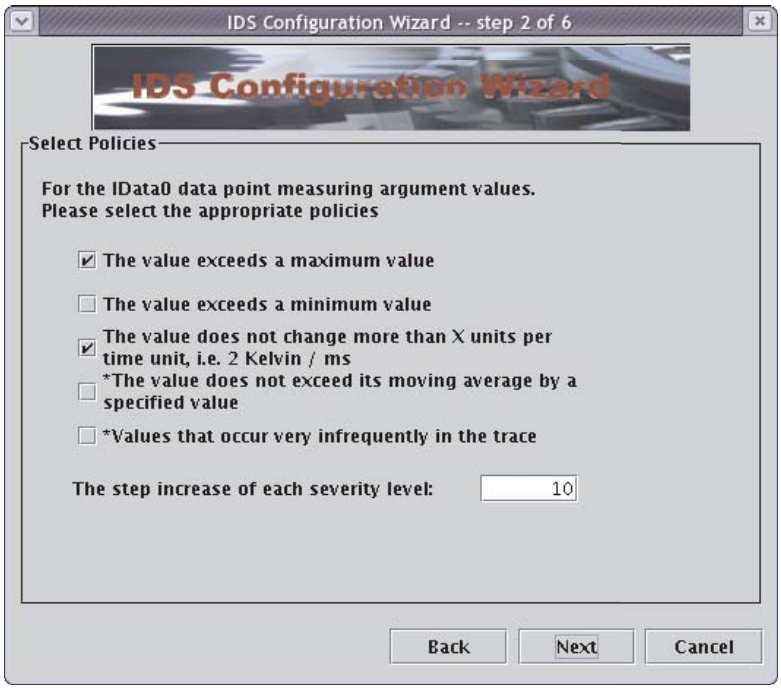

Figure 16. The second step of the 'IDSConfigurationWizard'

The application-level security policies for the misuse-based mechanism can be configured as shown in Figure 17. As described in Chapter 5, there are three security policies that can be configured within the middleware framework. For example, a configurable detector verifying the integrity of the application is implemented to use any of digest algorithms available in the security framework; this security framework for MicroQoSCORBA is explained in detail in [27][29].

Any limitations to the number of client's requests can be configured by the checkboxes and its corresponding textbox. Furthermore, proprietary misuse detectors can be enabled by specifying an identifier entered in the last textbox.

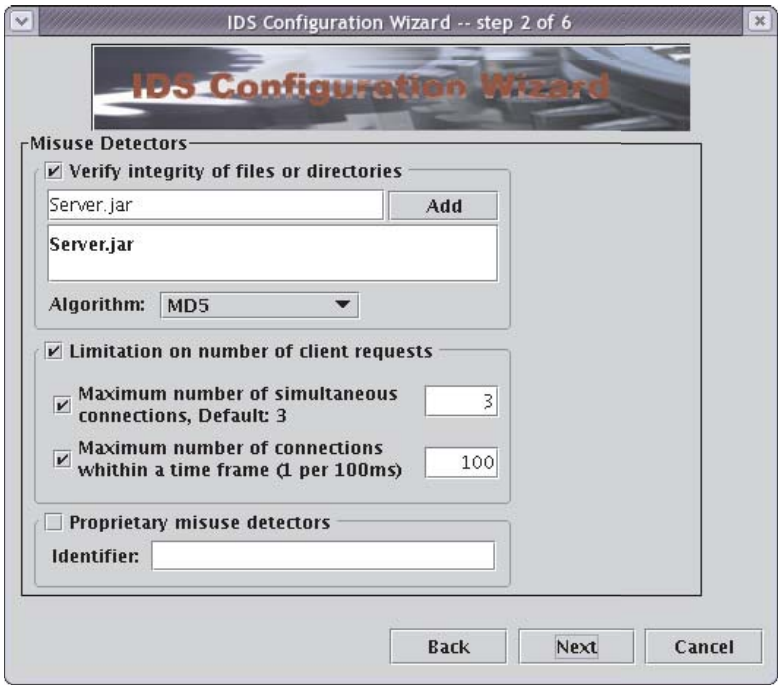

Figure 17. Configuration step 2 for misuse-detectors

Step 3 as depicted in Figure 18, is used to select the profile for each of the security policies that requires an associated profile. The three first checkboxes, which are toned-down in Figure 18, specify any of the profiles that can be associated with an interval-based type of data collection. For the procedural-based mechanism, it can be configured to use either a probabilistic suffix tree or a

sliding windows profile as a method of analyzing the collected data. The three textboxes specifies the length of the trace, window size, and the time from the last update before the profile becomes valid.

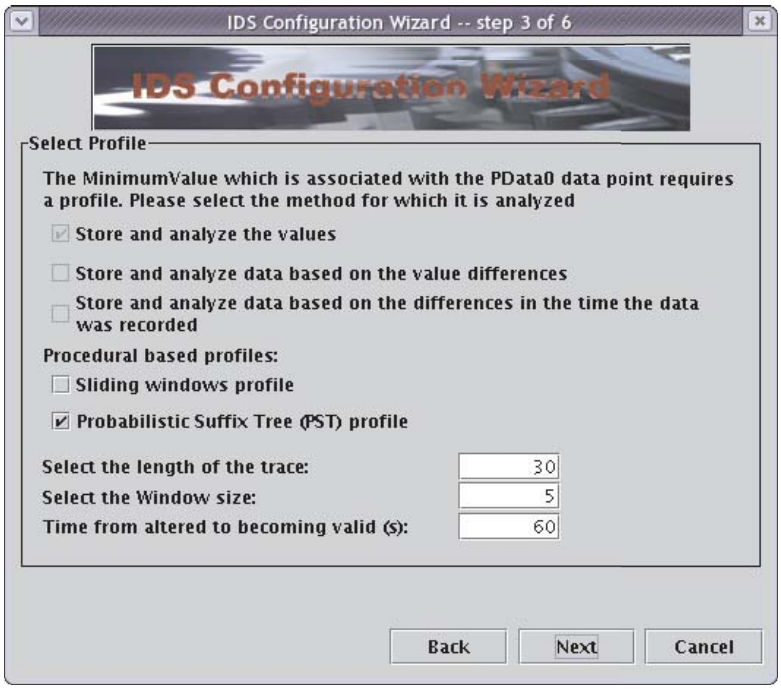

Figure 18. Specifying profiles required by the selected policies in step 2

Each of the policies selected in step 2 are configured in step 4, see Figure 19. The configuration is specified by a property sheet associated with the given policy. In Figure 19, the property sheet for the policies is shown to the left of the box specifying the range of validity. The range of validity provides a “safe-zone” for values that occur in a specified range. This is useful for policies that depend

on statistical methods e.g. the standard deviation. For example, given a set of values with very little or no difference, the standard deviation will approach a zero value. Any value with minor, but significant, deviation from the *n* previous values will violate the policy unless this range is given. This range of validity is based on the value's difference from the moving average. As illustrated by Figure 19, the range can be specified by a percentage or by set of constants describing the range, e.g. maximum and minimum.

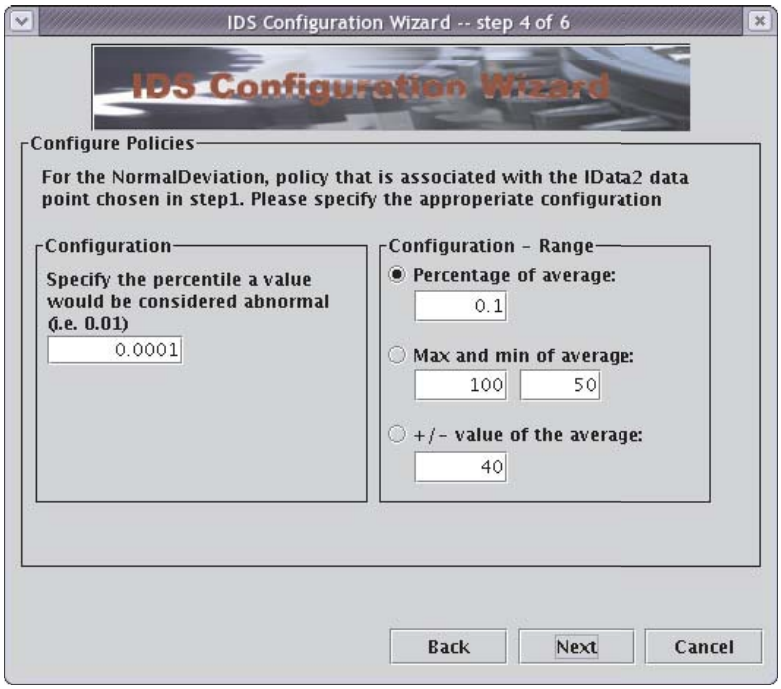

Figure 19. Configuring the selected policies in step 2

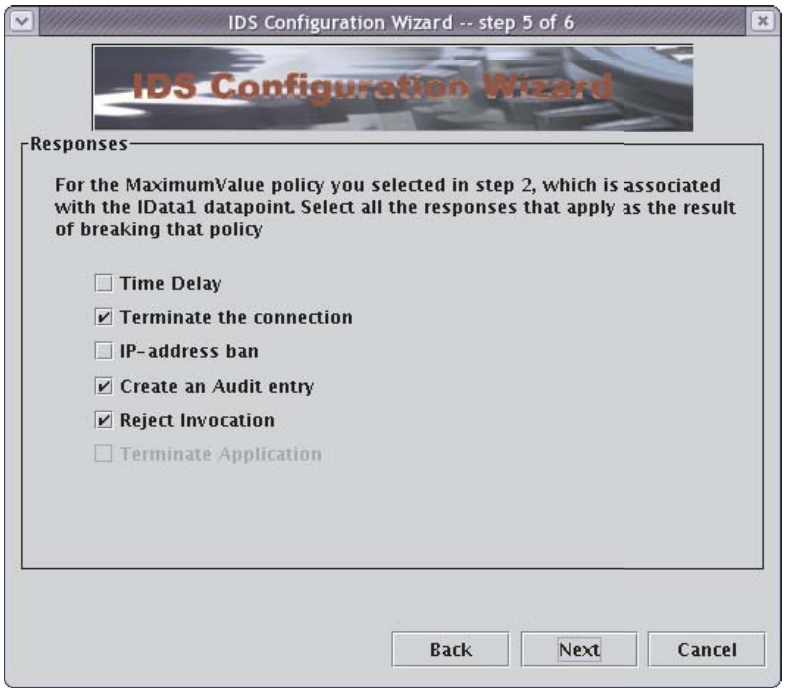

Figure 20. Specify the responses that can be configured for a policy

As presented in Figure 20, one or more responses must be configured for each policy. Dependent on the given policy, a range of responses can be associated with this specific policy as described in Chapter 5. In brief, responses that can be configured with a policy or detector are as follows: Audit, Time-Delay, Termination or ban of any connection hold by a specific IP-address, reject and invocation, and termination of the application. The last response can only be configured for extreme cases of misuse, e.g. when failing the integrity test.

The final step of the ‘IDSConfigurationWizard’ is to confirm the appropriate configuration, see Figure 21. After this final step, the configurations are merged back into the tree as shown in Figure 13.

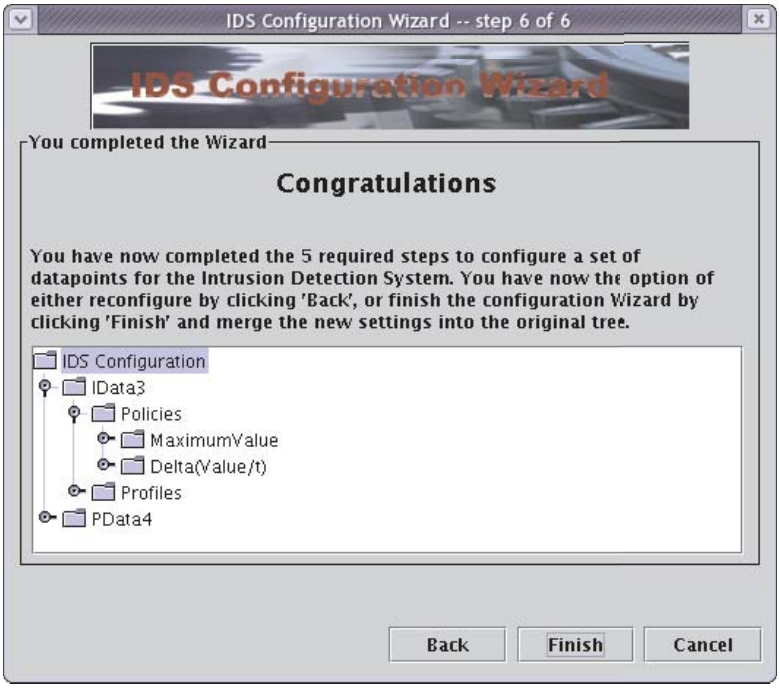

Figure 21. The final step of the configuration process

As demonstrated in this section, MIDES is a highly configurable framework for intrusion detection. The accompanied configuration tool can possible alleviate some of the configuration overhead associated with deploying an IDS in an easy and fashionable way.

Appendix C

Source Code and Configuration Files

This section includes source code and configuration files from the testbed used to measure the results as presented in Chapter 6. There are listed eight files in this appendix; the first file is a XML file that collectively describes the configuration settings of various IDS mechanisms. The next two files describe the source code for the client and server respectively. Also listed are the client and server IDS specific configuration files that are compiled into the respective implementation. In addition to this, the stub and skeleton containing the IDS specific code as generated by the IDL compiler are also presented. Finally, the last file listed is the server implementation that implements the object's API. These source listings are appended in consecutive order after section C.8.

C.1 Timing.xml

Rather than listing several configuration files, the timing.xml listed in source listing 2 contains the data points for the various scenarios as used to measure the overhead in Chapter 6. In other words, it collectively illustrates the various data points that were enabled.

For example, lines 9 through 23 configures a Maximum value policy for a value based data point on the client side using the return value of function ‘bar’ in interface ‘foo’, which is a member of module ‘timing’. Furthermore, lines 23 through 149 describe data points using various mechanisms for data collection, policies, profiles and responses. The rest of this configuration file configures the options available to the middleware framework.

C.2 Client.java

The source code for the client application that was used for the measurements described in Chapter 6 is listed in source listing 3. This code also contains the event filtering mechanism as discussed early in Chapter 6.

C.3 Server.java

The server implementation that is used for the experiments conducted in Chapter 6 is listed in source listing 4. The ‘ShowMemory’ function located at line 68 in the source file prints the memory usage on the server side to the screen. This function is invoked repeatedly in three seconds intervals for the test cases measuring the memory.

C.4 IDSClientConfig.java

This IDL generated configuration file, as listed in source listing 5, is compiled into the client side of the distributed application. The data points as configured here is the result of the ‘timing.xml’ file as shown in source listing 2.

The ‘configure’ function at line 26 are given a reference to an analyzer object that is to be configured. This analyzer object is also known as the kernel in MIDES. Nonetheless, this function starts by declaring all the responses that is to be used for this particular configuration. Lines 34 through 41 create the appropriate sensors and register them with the IDS kernel. The data points are

separately configured with the corresponding policies and responses before they are registered with the IDS-kernel.

The ‘createProfiles’ function is used by the IDS-kernel when it is necessary to create a set of new profiles. This is typically required when a new connection is made from the device. Furthermore, the ‘getIntegrityFiles’ function located at lines 83 to 89, returns a vector of files that is used to verify an application’s integrity and its external dependencies.

This file configures three specific data points, one interval-based sensor integrated into the client side stub ‘_fooStub.java’, see section C.6; one procedural-based sensor using a sliding windows profile, and one misuse-based detector used to verify the integrity of the client application and its external dependencies.

C.5 IDSServerConfig.java

The IDL-generated counterpart to the client’s IDS-configuration is listed in source listing 6. This file configures four data points that are used to configure the server application. These data points are: two interval-based sensors embedded into ‘fooPOA.java’, see C.7; a procedural based sensor using a Probabilistic Suffix

Tree profile, and a misuse-based detector used to detect resource starvation by repeated client requests.

C.6 timing/_fooStub.java

Source listing 7 shows the IDL-generated client stub as specified by source listing 1. Lines 34, 36, 37 and 38, demonstrate the use of an interval-based sensor recording the response-time. In other words, the time it takes for an invocation to be sent, processed, and returned by the server. Lines 43 to 45 demonstrate the use of an interval-based sensor recording the values of a function parameter.

C.7 timing/fooPOA.java

The source listed in source listing 8 shows the IDL-generated server skeleton as defined by the IDL specification in source listing 1. The lines from 30 to 39 demonstrate the use of an interval-based sensor measuring the invocation-rate. This file additionally shows an interval-based sensor that records the value of the first argument given by the client implementation. Now, this sensor also

demonstrates the use of a reject invocation response as it can be configured for interval based sensors.

C.8 fooImpl.java

The file listed in source listing 9 is the server side implementation of the object ‘foo’ as specified in the source listing 1. This file is also a part of the experiments conducted in Chapter 6. There is embedded one procedural-based sensor in this file that simulates an application behavior on a per invocation basis by altering the order which functions are virtually invoked. It does so by creating data nodes with different identifiers that corresponds to a unique class and function.

timing.xml 1/3

```
<?xml version="1.0"?>

<MQCConfiguration>
   <MQCFAULTTOLERANCE>
      <!-- Configuration for MQCFaultTolerance Plugin -->
   </MQCFAULTTOLERANCE>
   <MQCIDSCONFIG>
      <!-- Configuration for MQCIDSConfiguration Plugin -->
      <DATA name="IData0" typeid="2" function="bar"
            client="true" interface="foo"
            module="timing" type="VALUEBASED"
            server="false" returnvalue="long">
         <POLICIES>
            <POLICY name="MaximumValue" typeid="0" requireprofile="false"
              type="MAXPOLICY" step="10.0"
              maximum="100.0">
               <RESPONSES>
                  <RESPONSE name="Terminate" typeid="1" type="TERMINATE" />
                  <RESPONSE name="Timedelay" typeid="3" type="TIMEDELAY" />
               </RESPONSES>
            </POLICY>
         </POLICIES>
      </DATA>
      <DATA name="IData1" typeid="0" function="bar"
            client="true" interface="foo"
            module="timing" type="RESPONSETIME"
            server="false">
         <POLICIES>
            <POLICY name="MaximumMovingAverage" typeid="3" requireprofile="true"
              type="MAXAVERAGE" step="10.0"
              profileId="0" maximum="100.0">
               <RANGE name="Range" typeid="0" type="PERCENTAGE"
             Percentage="0.1" />
               <RESPONSES>
                  <RESPONSE name="Terminate" typeid="1" type="TERMINATE" />
                  <RESPONSE name="Timedelay" typeid="3" type="TIMEDELAY" />
               </RESPONSES>
            </POLICY>
         </POLICIES>
         <PROFILES>
            <PROFILE name="Value" typeid="0" type="VALUE"
               length="30" />
         </PROFILES>
      </DATA>
      <DATA name="PData2" typeid="3" client="true"
            type="PROCEDURAL" server="false">
         <POLICIES>
            <POLICY name="MaximumValue" typeid="0" requireprofile="true"
              type="MAXPOLICY" step="10.0"
              profileId="3" maximum="0.85">
               <RESPONSES>
                  <RESPONSE name="Audit" typeid="0" type="AUDIT" />
                  <RESPONSE name="Timedelay" typeid="3" type="TIMEDELAY" />
               </RESPONSES>
            </POLICY>
         </POLICIES>
         <PROFILES>
            <PROFILE name="SlidingWindows" typeid="3" type="SLIDINGWINDOWS"
                  windowsize="5" length="20" timebeforevalid="60" />
         </PROFILES>
      </DATA>
      <DATA name="MisuseData3" typeid="4" client="true"
            algorithm="MD5" identifier="integrity"
            type="MISUSE" files="Client.class;"
            server="false">
         <RESPONSES>
            <RESPONSE name="Exit" typeid="4" type="EXIT" />
         </RESPONSES>
      </DATA>
      <DATA name="IData4" typeid="1" function="bar"
            client="false" interface="foo"
            module="timing" type="RATEINVOCATION"
            server="true">
         <POLICIES>
            <POLICY name="MaximumValue" typeid="0" requireprofile="false"
              type="MAXPOLICY" step="10.0"
              maximum="100.0">
               <RESPONSES>
                  <RESPONSE name="IPBan" typeid="2" type="IPBAN" />
               </RESPONSES>
            </POLICY>
         </POLICIES>
```

Source Listing 2. timing.xml

```
      </DATA>
      <DATA name="IData5" typeid="2" function="bar"
            client="false" interface="foo"
            module="timing" type="VALUEBASED"
            argument="arg1" server="true"
            rejectinvocation="true">
         <POLICIES>
            <POLICY name="NormalDeviation" typeid="4" percentile="1.0E-4"
              requireprofile="true" type="NORMALDEV"
              step="10.0" profileId="0">
               <RANGE name="Range" typeid="1" Minimum="50.0"
             type="MAXMIN" Maximum="100.0" />
               <RESPONSES />
            </POLICY>
         </POLICIES>
         <PROFILES>
            <PROFILE name="Value" typeid="0" type="VALUE"
              length="30" />
         </PROFILES>
      </DATA>
      <DATA name="PData6" typeid="3" client="false"
            type="PROCEDURAL" server="true">
         <POLICIES>
            <POLICY name="MinimumValue" typeid="1" requireprofile="true"
              type="MINPOLICY" step="10.0"
              minimum="0.0010" profileId="4">
               <RESPONSES>
                  <RESPONSE name="Audit" typeid="0" type="AUDIT" />
                  <RESPONSE name="Timedelay" typeid="3" type="TIMEDELAY" />
               </RESPONSES>
            </POLICY>
         </POLICIES>
         <PROFILES>
            <PROFILE name="PST" typeid="4" height="3"
              gmin="0.0010" type="PST"
              r="1.05" windowsize="3"
              length="500" alpha="0.0"
              pmin="0.01" />
         </PROFILES>
      </DATA>
      <DATA name="MisuseData7" typeid="4" maxconnections="3"
            client="false" identifier="resourcestarvation"
            type="MISUSE" maxconnectionspersecond="100"
            server="true">
         <RESPONSES>
            <RESPONSE name="Audit" typeid="0" type="AUDIT" />
            <RESPONSE name="Timedelay" typeid="3" type="TIMEDELAY" />
         </RESPONSES>
      </DATA>
      <IDSRESPONSES>
         <TERMINATE boolean="true" typeid="1" server="true"
                client="true" severitylevel="3" />
         <IPBAN boolean="true" typeid="2" permanent="false"
             client="true" server="true"
             severitylevel="2" numbanbeforepermanent="3"
             severity="1000" />
         <TIMEDELAY boolean="true" typeid="3" severity_app="40"
                severity_conn="1000" server="true"
                client="true" severitylevel="1" />
         <EXIT boolean="true" typeid="4" server="true"
            client="false" severitylevel="5" />
         <AUDIT boolean="true" typeid="0" auditlength="20"
             ignoresafelist="true" tofile="false"
             toscreen="true" server="true"
             client="true" severitylevel="0" />
      </IDSRESPONSES>
   </MQCIDSCONFIG>
   <MQCTRANSPORT>
      <!-- Configuration for MQCTransport Plugin -->
      <CLIENT_TRANSPORT value="TCPIP" />
      <CLIENT_PROTOCOL value="GIOP" />
      <SERVER_TRANSPORT OneWire="false" Serial="false" TCPIP="true"
                        UDP_Unreliable="false" Go_Back_N_UDP="false"
                        Stop-N-Wait_UDP="false" />
      <SERVER_PROTOCOL GIOP="true" MQC-IOP="false" GIOP-Lite="false" />
      <GIOPVERSION value="12" />
   </MQCTRANSPORT>
   <MQCENCRYPTION>
      <!-- Configuration for MQCEncryption Plugin -->
   </MQCENCRYPTION>
   <MQCDATATYPES>
      <!-- Configuration for MQCDataTypes Plugin -->
```

Source Listing 2. timing.xml

```
165:       <PRIMARYTYPES float="false" unsigned_short="false" double="false"
166:                     char="false" short="false"
167:                     long="true" octet="false"
168:                     wchar="false" long_long="false"
169:                     long_double="false" unsigned_long_long="false"
170:                     unsigned_long="false" boolean="false" />
171:       <COMPLEXTYPES array="false" union="false" struct="false"
172:                     enum="false" sequence="false"
173:                     wstring="false" string="false" />
174:       <EXCEPTIONTYPES user="false" system="false" />
175:    </MQCDATATYPES>
176:    <MQCMISC>
177:       <!-- Configuration for MQCMisc Plugin -->
178:       <DEBUG>
179:          <!-- Debug Settings -->
180:          <MEMORY boolean="true" />
181:          <TIMING boolean="true" />
182:          <DEBUGLEVEL boolean="false" />
183:          <LEVEL integer="0" />
184:       </DEBUG>
185:       <HARDWARE>
186:          <!-- Hardware settings -->
187:          <CLDC boolean="false" />
188:          <FORCE_ENDIANESS boolean="false" />
189:          <ENDIANESS value="0" />
190:          <HETEROGENEITY value="homogenious" />
191:       </HARDWARE>
192:       <MISC>
193:          <!-- Miscellaneous Settings -->
194:          <COMPRESS boolean="false" />
195:          <MARSHALL value="proxy-marshalling" />
196:          <MAXPAYLOAD integer="0" />
197:          <MAXMETHODS integer="0" />
198:          <MAXINTERFACE integer="0" />
199:       </MISC>
200:    </MQCMISC>
201: </MQCConfiguration>
```

Source Listing 2. timing.xml

```
 1: /*
 2:  * Copyright (c) 2003 David E. Bakken, his research students, and Washington State University.
 3:  * Please see the file LICENSE.pdf for more details on terms of use.
 4:  */
 5:
 6: //
 7: /* Do NOT edit this file--It was autogenerated by m4 from a *.java.m4 file */
 8:
 9: import mqc.*;
10: import mqc.holders.*;
11: import mqc.Config;
12:
13: import jni.JNITimer;
14:
15: public class Client
16: {
17:    static final int REPEAT_CNT = 3; // number of times to repeat the main timing loop
18:    static long dt_cnt = 0; // count of dt[] elements
19:    static long dt_sum = 0; // E[x]  of dt histogram (unnormalized by cnt)
20:    static long dt_sum2 = 0; // E[x2] of dt histogram (unnormalized by cnt)
21:    static float dt_avg = 0; // avg time based upon dt
22:    static double dt_stdev = 0.0; // standard deviation
23:    static double dt_stdev2 = 0.0; // standard deviation squared
24:    static double STDEV_ERR = 99.9; // value to set stdev is an error occurs
25:    static long[] r_cnt = new long[REPEAT_CNT]; // r_cnt[i]   == i_th dt_cnt
26:    static float[] r_avg = new float[REPEAT_CNT]; // r_avg[i]   == i_th dt_avg
27:    static double[] r_stdev = new double[REPEAT_CNT]; // r_stdev[i] == i_th stdev
28:    static final int DT_SIZE = 500; // 500 for linux w/TR,5000 for SaJe w/TR,1000 for TINI w/TR
29:    static final int DTW_SIZE = 300; // 300 for linux w/TR,1000 for SaJe w/TR,100 for TINI w/TR
30:    static final int[] dt = new int[DT_SIZE]; // stores either individual event times or bins of event counts
31:    static final int[] dtw = new int[DTW_SIZE]; // stores either individual event times or bins of event counts
32:    static int[] dt_b;
33:    static int dt_offset = 0; // dt[] offset,ie w/ binning dt[i] correspond to time i+dt_offset
34:    static boolean dt_binEvents; // false -> ind. event times,true -> event cnt bins in dt[]
35:    static float nSigma = 3.5f; // default width of the peak (in stdev)
36:    static int timeoutSec = 600; // timeout,in seconds,for the main timing loop
37:    static int MAX_PEAK_WIDTH = 75; // the timing peak should be found within the first MAX_PEAK_WIDTH bins
38:    static double MAX_OK_STDEV = 4.0; // maximum value of an ok/good peak stdev
39:
40:    static String info;
41:
42:    public static void main(String[] args)
43:    {
44:       int i;
45:       int maxIterations;
46:       String propertyStr;
47:       int ticksPerMilliSecond = 1;
48:       long dt_time, dt_time0;
49:       int dt_delta;
50:       long dtw_time, dtw_time0;
51:       int dtw_delta;
52:
53:       //showMemory();
54:       System.gc();
55:
56:       // check to see if the user wants a custom peak width
57:       propertyStr = System.getProperty("nSigma");
58:       if (propertyStr != null)
59:       {
60:          nSigma = 0.01f * Integer.parseInt(propertyStr);
61:          System.out.println("nSigma: " + formatFloat(nSigma));
62:       }
63:
64:       // check to see if the user wants a custom timeout
65:       propertyStr = System.getProperty("TO");
66:       if (propertyStr != null)
67:       {
68:          timeoutSec = Integer.parseInt(propertyStr);
69:          System.out.println("timeoutSec: " + timeoutSec);
70:       }
71:
72:       // initialize the timer (actual calls and values supplied by the following macros)
73:       JNITimer.init(1000000);
74:       ticksPerMilliSecond = 1000;
75:
76:       Object object;
77:       C_ORB orb = new C_ORB();
78:
79:       // Find the server to connect to (via a corbaloc)
80:       //
81:       propertyStr = System.getProperty("corbaloc");
82:       if (propertyStr != null)
```

Source Listing 3. Client.java

```
   83:       {
   84:
   85:          object = orb.corbaloc_to_object(propertyStr);
   86:       } else
   87:       {
   88:
   89:          object = orb.corbaloc_to_object(args[0]);
   90:          //object = orb.string_to_object(args[0]);
   91:       }
   92:
   93:       // Get the number of iterations
   94:       //
   95:       maxIterations = 1000; // The default number of iterations
   96:       propertyStr = System.getProperty("cnt");
   97:       if (propertyStr != null)
   98:       {
   99:          maxIterations = Integer.parseInt(propertyStr);
  100:       }
  101:       System.out.println("Iteration 'cnt' set to " + maxIterations);
  102:       if (maxIterations > DT_SIZE)
  103:       {
  104:          dt_binEvents = true; // dt[i] == # of events that took i ms to complete
  105:       } else
  106:       {
  107:          dt_binEvents = false; // dt[i] == delta time of event i
  108:       }
  109:
  110:       timing.foo fooObj = timing.fooHelper.narrow(object);
  111:
  112:       // configID contains info about the current MQC settings/options
  113:       String configID = "cfg:p" + Config.intPacketSize + ":";
  114:
  115:       configID += "::";
  116:
  117:       long ssStartTime = 0; // steady state start time
  118:       long ssTotalTime = 0; // steady state total time
  119:       long startTime = 0; // start of "real" timing loop time
  120:       long totalTime = 0; // total time for the "real" timing loop
  121:
  122:       //
  123:       // STEADY STATE TIMING LOOP
  124:       //-------------------------
  125:       //
  126:       //showMemory();
  127:       System.gc();
  128:
  129:       System.out.println("---Steady State Begin---");
  130:
  131:       // init variable needed to compute the stdev of time deltas of each call
  132:       for (i = 0; i < DT_SIZE; i++)
  133:       {
  134:          dt[i] = 0;
  135:       }
  136:       for (i = 0; i < DTW_SIZE; i++)
  137:       {
  138:          dtw[i] = 0;
  139:       }
  140:
  141:       ssStartTime = JNITimer.currentTime();
  142:       dt_time0 = JNITimer.currentTime();
  143:
  144:       int dt_min = 999999999;
  145:       int ss_min;
  146:       int ss_max;
  147:       int ss_upperLimit = 999999999;
  148:
  149:       int ss_nLoop = 3;
  150:       int ss_jcnt = 15000;
  151:       for (int j = 0; j < ss_nLoop; j++)
  152:       {
  153:          dt_sum = 0;
  154:          dt_sum2 = 0;
  155:          dt_cnt = 0;
  156:          ss_min = 999999999;
  157:          ss_max = 0;
  158:          dt_time0 = JNITimer.currentTime();
  159:
  160:          for (i = 0; i < ss_jcnt; i++)
  161:          {
  162:             try
  163:             {
  164:                fooObj.bar(1);
```

Source Listing 3. Client.java

```
165:             } catch (Exception e)
166:             {}
167:             dt_time = JNITimer.currentTime();
168:             dt_delta = (int) (dt_time - dt_time0);
169:             if (dt_delta < ss_min)
170:             {
171:                ss_min = dt_delta;
172:             }
173:             if (dt_delta > ss_max)
174:             {
175:                ss_max = dt_delta;
176:             }
177:             if (dt_delta < ss_upperLimit)
178:             {
179:                dt_sum += dt_delta;
180:                dt_sum2 += (long) dt_delta * (long) dt_delta;
181:                dt_cnt++;
182:             }
183:             dt_time0 = dt_time;
184:
185:          }
186:          if (ss_min < dt_min)
187:          {
188:             dt_min = ss_min;
189:          }
190:          // semi-fragile code,probably should check that dt_cnt > 1.  -ADM
191:          dt_avg = (float) dt_sum / dt_cnt;
192:          dt_stdev2 = (double) (dt_sum2 - (dt_sum * dt_sum) / dt_cnt) / (dt_cnt - 1);
193:          if (dt_stdev2 >= 0.0)
194:          {
195:             dt_stdev = Math.sqrt(dt_stdev2);
196:          } else
197:          {
198:             System.err.println("Oops! Likely integer overflow -- dt_stdev2= " +
199:                                formatDouble(dt_stdev2) + " < 0.0 -- Setting stdev to " +
200:                                formatFloat( (float) STDEV_ERR) + "!");
201:             dt_stdev = STDEV_ERR;
202:          }
203:          ss_upperLimit = (int) (dt_avg + nSigma * dt_stdev);
204:          System.out.println(" - " + (j + 1) * ss_jcnt + ":   avg: " + formatFloat(dt_avg) +
205:                             "  stdev: " + formatDouble(dt_stdev) +
206:                             "  min/max: " + ss_min + "-" + ss_max + "  upL: " +
207:                             ss_upperLimit);
208:
209:       }
210:
211:       showMemory();
212:       showMemory();
213:       showMemory();
214:
215:       float dt_ssAvg = (float) (dt_avg / ticksPerMilliSecond);
216:       double dt_ssStdev = dt_stdev / ticksPerMilliSecond;
217:       long dt_ssCnt = dt_cnt;
218:
219:       ssTotalTime = (JNITimer.currentTime() - ssStartTime);
220:       float ssAvgTime = (float) (ssTotalTime / ticksPerMilliSecond) / (ss_nLoop * ss_jcnt);
221:
222:       System.out.println("---Steady State Info---");
223:
224:       System.out.println("Overall:  " + formatFloat(ssAvgTime) + "    n/a     " +
225:                          (ss_nLoop * ss_jcnt) + "  " +
226:                          formatFloat( (float) ssTotalTime / (1000 * ticksPerMilliSecond)) +
227:                          "s");
228:       System.out.println("  ->end:  " + formatFloat(dt_ssAvg) + "  " +
229:                          formatDouble(dt_ssStdev) + "  " + dt_ssCnt);
230:
231:       System.out.println("---Steady State End---");
232:
233:       int dd = (int) (5.0 * (dt_ssStdev * ticksPerMilliSecond)) + 10;
234:       if (dd > (int) (.2 * DT_SIZE))
235:       {
236:          dd = (int) (.2 * DT_SIZE);
237:       }
238:       dt_offset = dt_min - dd;
239:       if (dt_offset < 0)
240:       {
241:          dt_offset = 0;
242:       }
243:
244:       //
245:       // MAIN TIMING LOOP
246:       //------------------
```

Source Listing 3. Client.java

```
247:       //
248:       int iterations = 0;
249:       long stop_time;
250:       for (int r = 0; r < REPEAT_CNT; r++)
251:       {
252:          i = -1;
253:          dt_delta = -1;
254:          dtw_delta = -1;
255:          try
256:          {
257:
258:             // init variable needed to compute the stdev of time deltas of each call
259:             for (i = 0; i < DT_SIZE; i++)
260:             {
261:                dt[i] = 0;
262:             }
263:             for (i = 0; i < DTW_SIZE; i++)
264:             {
265:                dtw[i] = 0;
266:             }
267:
268:             startTime = JNITimer.currentTime();
269:             dt_time0 = startTime;
270:             dtw_time0 = dt_time0 / 1000;
271:             stop_time = dt_time0 + timeoutSec * 1000 * ticksPerMilliSecond;
272:
273:             int d;
274:             iterations = maxIterations;
275:             for (i = 0; i < iterations; i++)
276:             {
277:                try
278:                {
279:                   fooObj.bar(1);
280:                } catch (Exception e)
281:                {}
282:                dt_time = JNITimer.currentTime();
283:                dt_delta = (int) (dt_time - dt_time0);
284:                dt_time0 = dt_time;
285:                dtw_time = dt_time / 1000;
286:                dtw_delta = (int) (dtw_time - dtw_time0);
287:                dtw_time0 = dtw_time;
288:                if (dt_binEvents)
289:                {
290:                   d = dt_delta - dt_offset;
291:                   if (d < DT_SIZE)
292:                   {
293:                      dt[d]++;
294:                   } else
295:                   {
296:                      dt[DT_SIZE - 1]++;
297:                   }
298:                } else
299:                {
300:                   // store raw (unbinned) times into dt[]
301:                   dt[i] = dt_delta;
302:                }
303:                // wide events are always binned
304:                if (dtw_delta < DTW_SIZE)
305:                {
306:                   dtw[dtw_delta]++;
307:                } else
308:                {
309:                   dtw[DTW_SIZE - 1]++;
310:                }
311:                if (dt_time > stop_time)
312:                {
313:                   iterations = i;
314:                   break;
315:                }
316:
317:             }
318:             totalTime = (JNITimer.currentTime() - startTime);
319:             float avgTime = (float) (totalTime / ticksPerMilliSecond) / iterations;
320:             float ssDelta = ssAvgTime / avgTime;
321:
322:             dt_stdev = findStdev(false, iterations); //warning computes dt_cnt,dt_avg,dt_sum,dt_sum2
323:             dt_avg /= ticksPerMilliSecond;
324:             dt_stdev /= ticksPerMilliSecond;
325:
326:             info = "@@@   Loop" + r + ":  " +
327:                 (int) (totalTime / (1000 * ticksPerMilliSecond)) + "  " +
328:                 formatFloat(avgTime);
```

Source Listing 3. Client.java

```
329:             info += "  " + formatDouble(dt_stdev) + "  " + iterations + "  @@  r" +
330:                 formatFloat(ssDelta);
331:
332:             info += "  @@  " + configID;
333:
334:             dt_stdev = findStdev(true, iterations); //warning computes dt_cnt,dt_avg,dt_sum,dt_sum2
335:             dt_avg /= ticksPerMilliSecond;
336:             dt_stdev /= ticksPerMilliSecond;
337:             float dt_ssDelta = dt_ssAvg / dt_avg;
338:
339:             System.out.println(info); // print non-GC info
340:
341:             System.out.print("@@@ gcLoop" + r + ":  " +
342:                              (int) (totalTime / (1000 * ticksPerMilliSecond)) + "  " +
343:                              formatFloat(dt_avg) + "  " + formatDouble(dt_stdev) +
344:                              "  " + dt_cnt + "  @@  r" + formatFloat(dt_ssDelta));
345:             System.out.println("  @@  " + configID);
346:
347:          } catch (java.lang.ArrayIndexOutOfBoundsException e)
348:          {
349:             System.err.println("i: " + i + " dt_offset/dt/dtw: " + dt_offset +
350:                                "  " + dt_delta + "  " + dtw_delta);
351:             dt_cnt = -1;
352:             dt_avg = 999999;
353:             dt_stdev = STDEV_ERR;
354:          }
355:
356:          r_cnt[r] = dt_cnt;
357:          r_avg[r] = dt_avg;
358:          r_stdev[r] = dt_stdev;
359:
360:       }
361:
362:       System.out.println("---Statistical Info---");
363:       System.out.print("Loop s:  " + formatFloat(ssAvgTime));
364:       System.out.print("  " + formatDouble(dt_ssStdev) + "  " + dt_ssCnt);
365:       System.out.println();
366:
367:       dt_cnt = r_cnt[0];
368:       dt_avg = r_avg[0];
369:       dt_stdev = r_stdev[0];
370:       for (int r = 0; r < REPEAT_CNT; r++)
371:       {
372:          System.out.println("Loop " + r + ":  " + formatFloat(r_avg[r]) +
373:                             "  " + formatDouble(r_stdev[r]) + "  " + r_cnt[r]);
374:          if (r_avg[r] < dt_avg)
375:          {
376:             dt_cnt = r_cnt[r];
377:             dt_avg = r_avg[r];
378:             dt_stdev = r_stdev[r];
379:          }
380:       }
381:       System.out.println("@@@ Summary: " + formatFloat(dt_avg) + "  " +
382:                          formatDouble(dt_stdev) + "  " +
383:                          dt_cnt + " @@- " + configID);
384:
385:       showMemory();
386:       showMemory();
387:       showMemory();
388:
389:    }
390:
391:    /**
392:     * formatFloat -- used to print a float with four decimal digits
393:     * (This routine is needed because CLDC does not support printing floats).
394:     *
395:     * @param f  floating point number to print
396:     */
397:    static String formatFloat(float f)
398:    {
399:       f += 0.00005;
400:       long fint = (long) f;
401:       long ffra = (long) (10000 * ( (f + 1) - fint));
402:       StringBuffer ffraStr = new StringBuffer(String.valueOf(ffra));
403:       ffraStr.setCharAt(0, '.');
404:       return String.valueOf(fint) + ffraStr;
405:    }
406:
407:    /**
408:     * formatDouble -- used to print a double with six decimal digits
409:     * (This routine is needed because CLDC does not support printing doubles).
410:     *
```

Source Listing 3. Client.java

```
411:     * @param d  double to print
412:     */
413:    static String formatDouble(double d)
414:    {
415:       d += 0.0000005;
416:       long dint = (long) d;
417:       long dfra = (long) (1000000 * ( (d + 1) - dint));
418:       StringBuffer dfraStr = new StringBuffer(String.valueOf(dfra));
419:       dfraStr.setCharAt(0, '.');
420:       return String.valueOf(dint) + dfraStr;
421:    }
422:
423:    /**
424:     * findStdev -- find the standard deviation of a set of point.
425:     *
426:     * @param nonGC   if true,computate the stdev of only the
427:     *                non-Garbage Collected,if false,then use all events
428:     * @param iCnt    number of non-binned events in dt[] (see dt_binEvents)
429:     * @return double the raw standard deviation (ie, non-scaled value).
430:     */
431:    static double findStdev(boolean nonGC, int iCnt)
432:    {
433:       //WARNING: global variables dt_cnt,dt_sum,dt_sum2 are all modified
434:       //within this routine (yes--an ugly hack...)
435:
436:       int i, iStart, iPeakStart, iStop;
437:       int dt_max;
438:       int dt_b[]; // bins
439:       double stdev2, stdev;
440:       String dt_info;
441:
442:       if (dt_binEvents)
443:       {
444:          // bins already computed,just alias dt[]
445:          dt_b = dt;
446:       } else
447:       {
448:          // need to compute number of bins needed
449:          dt_offset = 3600000; // an hour (in ms)
450:          dt_max = 0;
451:          for (i = 0; i < iCnt; i++)
452:          {
453:             if (dt[i] < dt_offset)
454:             {
455:                dt_offset = dt[i];
456:             }
457:             if (dt[i] > dt_max)
458:             {
459:                dt_max = dt[i];
460:             }
461:          }
462:          dt_offset--;
463:          dt_max++;
464:          if (dt_max > dt_offset + 2 * DT_SIZE)
465:          {
466:             dt_max = dt_offset + 2 * DT_SIZE;
467:          }
468:          // create the bins
469:          dt_b = new int[dt_max - dt_offset + 1];
470:          // stuff the bins
471:          for (i = 0; i < iCnt; i++)
472:          {
473:
474:             if (dt[i] - dt_offset < 2 * DT_SIZE)
475:             {
476:                dt_b[dt[i] - dt_offset]++;
477:             } else
478:             {
479:                dt_b[2 * DT_SIZE]++;
480:             }
481:          }
482:
483:       }
484:
485:       // Initially,use the first MAX_PEAK_WIDTH non-zero channels to compute
486:       // the average and stdev of the timing peak
487:       iStart = 0;
488:       while (dt_b[iStart] == 0 && iStart < dt_b.length - 1)
489:       {
490:          iStart++;
491:       }
492:       iPeakStart = iStart + 1;
```

Source Listing 3. Client.java

```
493:        while (dt_b[iPeakStart] < 2 && iPeakStart < dt_b.length - 1)
494:        {
495:           iPeakStart++;
496:        }
497:
498:        iStop = dt_b.length;
499:        if (nonGC)
500:        {
501:           // the last bin contains overflow values--discard it
502:           iStop--;
503:        }
504:        if (iStop > iPeakStart + MAX_PEAK_WIDTH)
505:        {
506:           iStop = iPeakStart + MAX_PEAK_WIDTH;
507:        }
508:        if (nonGC)
509:        {
510:           System.err.println("iStart/iPeakStart/iStop: " + (iStart + dt_offset) +
511:                              "  " + (iPeakStart + dt_offset) + "  " + (iStop + dt_offset));
512:        }
513:        int iStop0;
514:        long dtb, dti;
515:        double x, x2;
516:
517:        do
518:        {
519:           iStop0 = iStop;
520:           // sum up the dt[] bins
521:           dt_cnt = 0;
522:           dt_sum = 0;
523:           dt_sum2 = 0;
524:           for (i = iStart; i < iStop; i++)
525:           {
526:              dt_cnt += dt_b[i];
527:              dtb = dt_b[i];
528:              dti = dt_offset + i;
529:              dt_sum += dtb * dti;
530:              dt_sum2 += dtb * dti * dti;
531:              // check for overflowed values (they will cause dt_sum2 to go negative)
532:              if (dt_sum2 < 0)
533:              {
534:                 System.err.println("Oops! Likely integer overflow problem.");
535:                 System.err.println("dt_sum2 < 0: " + dt_sum2 + " at i=" + (dt_offset + i));
536:              }
537:           }
538:
539:           // compute the average and its standard dev.
540:           dt_avg = (float) dt_sum / dt_cnt;
541:           x = (double) dt_sum;
542:           x2 = (double) dt_sum2;
543:           stdev2 = (x2 - (x * x) / dt_cnt) / (dt_cnt - 1);
544:           if (stdev2 >= 0)
545:           {
546:              stdev = Math.sqrt(stdev2);
547:           } else
548:           {
549:              System.err.println("Oops! Likely integer overflow -- stdev2= " +
550:                                 formatDouble(stdev2) + " < 0.0 -- Setting stdev to " +
551:                                 formatFloat( (float) STDEV_ERR) + "!");
552:              stdev = STDEV_ERR;
553:           }
554:           if (nonGC)
555:           {
556:              iStop = (int) (dt_avg + (nSigma * stdev) + 0.5) - dt_offset;
557:              if (iStop >= dt_b.length)
558:              {
559:                 iStop = dt_b.length - 1;
560:              }
561:              if ( (iStop >= iStop0) && (stdev > MAX_OK_STDEV))
562:              {
563:                 // Compute an alternate upper limit for the first (nonGC)
564:                 // peak. The value (dt_avg - iStart) is *assumed* to be the
565:                 // bottom (lower-time) tail of the nonGC peak. This means
566:                 // that (dt_avg - iStart) should be another approximation
567:                 // to the value of 3.5 * nSigma (of the nonGC peak). If this
568:                 // value gives a smaller iStop use it.
569:                 int iStop2 = (int) ( (dt_avg - dt_offset) +
570:                                      ( (dt_avg - dt_offset) - iPeakStart));
571:                 System.err.println("avg,iStop,iStop2: " + formatFloat( (float) dt_avg) +
572:                                    "  " +
573:                                    (dt_offset + iStop) + "  " + (dt_offset + iStop2));
574:                 if (iStop2 < iStop)
```

Source Listing 3. Client.java

```
575:                {
576:                   iStop = iStop2;
577:                }
578:             }
579:             if (iStop < iStart + 1)
580:             {
581:                iStop = iStart + 1;
582:             }
583:             System.err.println("x-y,avg,stdev,iStop: " +
584:                                (dt_offset + iStart) + "-" + (dt_offset + iStop0) + "  " +
585:                                formatDouble(dt_avg) + "  " + formatDouble(stdev) +
586:                                "  " + (dt_offset + iStop));
587:          }
588:       } while (iStop < iStop0);
589:
590:       // Pretty-Print the data
591:       if (nonGC)
592:       {
593:          System.out.println("--- " + (dt_offset + iStart) + "  " +
594:                             (dt_offset + (iStop0 - 1)) + " ---non-GC---");
595:          printArray(dt_b, dt_offset);
596:          System.out.println("---Wide bins---");
597:          printArray(dtw, 0);
598:       }
599:
600:       if (!dt_binEvents)
601:       {
602:          dt_b = null; // release the dt_b array--it is no longer needed
603:       }
604:
605:       return stdev;
606:    }
607:
608:    /**
609:     * printArray -- a pretty printer for the dt/dtw arrays
610:     *
611:     * @param a      an array to print
612:     * @param offset  index offset of the array
613:     */
614:    static void printArray(int a[], int offset)
615:    {
616:       int lastZero = 0;
617:       boolean inZeros = true;
618:       System.out.println(offset + "  " + a[0]);
619:       for (int i = 1; i < a.length; i++)
620:       {
621:          if (a[i] > 0)
622:          {
623:             if (inZeros)
624:             {
625:                inZeros = false;
626:                if (i - 1 > lastZero)
627:                {
628:                   System.out.println( (offset + i - 1) + "  " + a[i - 1]);
629:                }
630:             }
631:             System.out.println( (offset + i) + "  " + a[i]);
632:          } else
633:          {
634:             if (inZeros)
635:             {
636:                // do nothing
637:             } else
638:             {
639:                inZeros = true;
640:                lastZero = i;
641:                System.out.println( (offset + i) + "  " + a[i]);
642:             }
643:          }
644:       }
645:    }
646:
647:    static Runtime runtime = Runtime.getRuntime();
648:
649:    /**
650:     * showMemory -- print our the free/used memory
651:     */
652:    static void showMemory()
653:    {
654:       long tMem, fMem, uMem, fMem1, uMemMax, fMemMax;
655:
656:       tMem = runtime.totalMemory();
```

Source Listing 3. Client.java

```
657:        fMem = runtime.freeMemory();
658:        fMemMax = fMem;
659:        if (tMem == fMem)
660:        {
661:           // TINI hack,tMem and fMem are reported as equal
662:           tMem = 334240; // MAGIC NUMBER appears to be the proper value
663:        }
664:        uMem = tMem - fMem;
665:
666:        System.gc();
667:
668:        for (int i = 0; i < 100; i++)
669:        {
670:           fMem1 = runtime.freeMemory();
671:           ;
672:
673:           if (fMem1 > fMemMax)
674:           {
675:              fMemMax = fMem1;
676:           } else
677:           {
678:              if (i > 1)
679:              {
680:                 break;
681:              }
682:           }
683:           try
684:           {
685:              Thread.sleep(500);
686:           } catch (InterruptedException ie)
687:           {}
688:        }
689:        uMemMax = tMem - fMemMax;
690:
691:        System.out.println("@@@ c.Memory (total/free/util/deltautil): \t"
692:                           + tMem + " \t" + fMemMax + " \t" + uMemMax + " \t" +
693:                           (uMemMax - uMem));
694:     }
695:
696: }
```

Source Listing 3. Client.java

Server.java

```
 1: /*
 2:  * Copyright (c) 2003 David E. Bakken, his research students, and Washington State University.
 3:  * Please see the file LICENSE.pdf for more details on terms of use.
 4:  */
 5:
 6: //
 7: /* Do NOT edit this file--It was autogenerated by m4 from a *.java.m4 file */
 8:
 9: import mqc.*;
10: import mqc.holders.*;
11:
12: import java.io.*;
13:
14: public class Server
15: {
16:    public static void main(String[] args)
17:    {
18:       //ShowMemory();
19:       System.gc();
20:
21:       System.gc();
22:       S_ORB orb = new S_ORB();
23:
24:       POA rootPOA = new POA(orb, "RootPOA");
25:       fooImpl test = new fooImpl();
26:
27:       Object object = rootPOA.servant_to_reference(test);
28:
29:       //--Generate the corbaloc
30:       String corbaloc = orb.object_to_corbaloc(object);
31:
32:       System.out.println(corbaloc);
33:
34:       try
35:       {
36:          FileWriter out = new FileWriter(new File("timing.corbaloc"));
37:          out.write(corbaloc);
38:          out.close();
39:       } catch (IOException e)
40:       {
41:          System.out.println("Error writing IOR/corbloc");
42:          return;
43:       }
44:
45:       orb.run();
46:
47:       // Give the ORB time to start running/waiting for client connections
48:       try
49:       {
50:          Thread.sleep(500);
51:       } catch (InterruptedException ie)
52:       {}
53:       ShowMemory();
54:
55:       for (; ; )
56:       { // continuous loop of ShowMemory values
57:          try
58:          {
59:             Thread.sleep(3000);
60:          } catch (InterruptedException ie)
61:          {}
62:          ShowMemory();
63:       }
64:
65:    }
66:
67:    static Runtime runtime = Runtime.getRuntime();
68:
69:    static void ShowMemory()
70:    {
71:       long tMem, fMem, uMem, fMem1, uMemMax, fMemMax;
72:
73:       tMem = runtime.totalMemory();
74:       fMem = runtime.freeMemory();
75:       fMemMax = fMem;
76:       if (tMem == fMem)
77:       {
78:          // TINI hack,tMem and fMem are reported as equal
79:          tMem = 334240; // MAGIC NUMBER appears to be the proper value
80:       }
81:       uMem = tMem - fMem;
82:
```

Source Listing 4. Server.java

```
 83:       System.gc();
 84:
 85:       for (int i = 0; i < 100; i++)
 86:       {
 87:          fMem1 = runtime.freeMemory();
 88:          ;
 89:
 90:          if (fMem1 > fMemMax)
 91:          {
 92:             fMemMax = fMem1;
 93:          } else
 94:          {
 95:             if (i > 1)
 96:             {
 97:                break;
 98:             }
 99:          }
100:          try
101:          {
102:             Thread.sleep(500);
103:          } catch (InterruptedException ie)
104:          {}
105:       }
106:       uMemMax = tMem - fMemMax;
107:
108:       System.out.println("@@@ s.Memory (total/free/util/deltautil): \t"
109:                          + tMem + " \t" + fMemMax + " \t" + uMemMax + " \t" +
110:                          (uMemMax - uMem));
111:    }
112:
113: }
```

Source Listing 4. Server.java

IDSClientConfig.java

```
 1: /** This is an automatically generated file,
 2:  * please do not make changes to this file
 3:  */
 4: package mqc.ids;
 5:
 6: import mqc.ids.algorithm.*;
 7: import mqc.ids.sensor.*;
 8: import mqc.ids.policy.*;
 9: import mqc.ids.policy.range.*;
10: import mqc.ids.response.*;
11: import mqc.ids.profile.*;
12: import java.util.Hashtable;
13: import java.util.Vector;
14:
15: public class IDSClientConfig implements IDSConfig
16: {
17:    public Policy policy0 = null;
18:    public Policy policy1 = null;
19:    public Policy policy2 = null;
20:
21:    public boolean isServerSide()
22:    {
23:       return false;
24:    }
25:
26:    public void configure(Analyzer analyzer)
27:    {
28:       Response terminate = new TerminateConnection(3);
29:       Response ipban = new IPBan(2);
30:       Response timedelay = new TimeDelay(1);
31:       Response exit = new Exit(5);
32:       Response audit = new Audit(0);
33:
34:       IntervalSensor is = new IntervalSensor(analyzer);
35:       analyzer.addSensor(Analyzer.SEN_INTERVAL, is);
36:
37:       ProceduralSensor ps = new ProceduralSensor(analyzer);
38:       analyzer.addSensor(Analyzer.SEN_PROCEDURAL, ps);
39:
40:       MisuseDetector md = new MisuseDetector(analyzer);
41:       analyzer.addSensor(Analyzer.DET_MISUSE, md);
42:
43:       Data IData0 = is.createDataNode(new Location(0, 0, 1));
44:       policy0 = new MaxPolicy(100.0, 10.0);
45:       policy0.addResponse(terminate);
46:       policy0.addResponse(timedelay);
47:       analyzer.addPolicy(IData0, policy0);
48:
49:       Data IData1 = is.createDataNode(new Location(0, 1, 2));
50:       ValidRange range1 = new ValidRangePercentage(0.1);
51:       policy1 = new MaxAveragePolicy(100.0, 10.0, range1);
52:       policy1.addResponse(terminate);
53:       policy1.addResponse(timedelay);
54:       analyzer.addPolicy(IData1, policy1);
55:
56:       Data PData2 = ps.createDataNode(new Location(0, 0, 0));
57:       policy2 = new MaxPolicy(0.85, 10.0);
58:       policy2.addResponse(audit);
59:       policy2.addResponse(timedelay);
60:       analyzer.addPolicy(PData2, policy2);
61:
62:       Data MisuseData3 = md.createDataNode("integrity", "no reason");
63:       analyzer.addMisuseResponse(MisuseData3, exit);
64:    }
65:
66:    public Hashtable createProfiles(boolean procedural)
67:    {
68:       Hashtable retval = new Hashtable();
69:
70:       if (procedural)
71:       {
72:          Profile sdprofile20 = new SlidingWindowsProfile(20, 5, 60);
73:          retval.put(policy2, sdprofile20);
74:       } else
75:       {
76:          Profile vprofile10 = new ValueProfile(30);
77:          retval.put(policy1, vprofile10);
78:       }
79:
80:       return retval;
81:    }
82:
```

Source Listing 5. IDSClientConfig.java

```
83:     public Vector getIntegrityFiles()
84:     {
85:
86:        Vector retvec = new Vector();
87:        retvec.addElement("Client.class");
88:        return retvec;
89:     }
90:
91: }
92:
93: /* End Of File */
```

Source Listing 5. IDSClientConfig.java

```
 1: /** This is an automatically generated file,
 2:  * please do not make changes to this file
 3:  */
 4: package mqc.ids;
 5:
 6: import mqc.ids.algorithm.*;
 7: import mqc.ids.sensor.*;
 8: import mqc.ids.policy.*;
 9: import mqc.ids.policy.range.*;
10: import mqc.ids.response.*;
11: import mqc.ids.profile.*;
12: import java.util.Hashtable;
13: import java.util.Vector;
14:
15: public class IDSServerConfig implements IDSConfig
16: {
17:    public Policy policy0 = null;
18:    public Policy policy1 = null;
19:    public Policy policy2 = null;
20:
21:    public boolean isServerSide()
22:    {
23:       return true;
24:    }
25:
26:    public void configure(Analyzer analyzer)
27:    {
28:       Response terminate = new TerminateConnection(3);
29:       Response ipban = new IPBan(2);
30:       Response timedelay = new TimeDelay(1);
31:       Response exit = new Exit(5);
32:       Response audit = new Audit(0);
33:
34:       IntervalSensor is = new IntervalSensor(analyzer);
35:       analyzer.addSensor(Analyzer.SEN_INTERVAL, is);
36:
37:       ProceduralSensor ps = new ProceduralSensor(analyzer);
38:       analyzer.addSensor(Analyzer.SEN_PROCEDURAL, ps);
39:
40:       MisuseDetector md = new MisuseDetector(analyzer);
41:       analyzer.addSensor(Analyzer.DET_MISUSE, md);
42:
43:       Data IData4 = is.createDataNode(new Location(0, 0, 3));
44:       policy0 = new MaxPolicy(100.0, 10.0);
45:       policy0.addResponse(ipban);
46:       analyzer.addPolicy(IData4, policy0);
47:
48:       Data IData5 = is.createDataNode(new Location(0, 1, 4));
49:       ValidRange range1 = new ValidRangeConstant(100.0, 50.0);
50:       policy1 = new NormalPolicy(1.0E-4, 10.0, range1);
51:       analyzer.addPolicy(IData5, policy1);
52:
53:       Data PData6 = ps.createDataNode(new Location(0, 0, 0));
54:       policy2 = new MinPolicy(0.0010, 10.0);
55:       policy2.addResponse(audit);
56:       policy2.addResponse(timedelay);
57:       analyzer.addPolicy(PData6, policy2);
58:
59:       Data MisuseData7 = md.createDataNode("resourcestarvation", "no reason");
60:       analyzer.addMisuseResponse(MisuseData7, audit);
61:       analyzer.addMisuseResponse(MisuseData7, timedelay);
62:    }
63:
64:    public Hashtable createProfiles(boolean procedural)
65:    {
66:       Hashtable retval = new Hashtable();
67:
68:       if (procedural)
69:       {
70:          Profile pstprofile20 = new PSTProfile(3, 500, 3, 0.01, 0.0, 0.0010, 1.05);
71:          retval.put(policy2, pstprofile20);
72:       } else
73:       {
74:          Profile vprofile10 = new ValueProfile(30);
75:          retval.put(policy1, vprofile10);
76:       }
77:
78:       return retval;
79:    }
80:
81:    public Vector getIntegrityFiles()
82:    {
```

Source Listing 6. IDSServerConfig.java

```
83:
84:         return null;
85:     }
86:
87: }
88:
89: /* End Of File */
```

Source Listing 6. IDSServerConfig.java

_fooStub.java

```
 1: package timing;
 2:
 3: import mqc.*;
 4: import mqc.ids.*;
 5: import mqc.ids.sensor.*;
 6: import java.io.IOException;
 7:
 8: public class _fooStub extends mqc.ObjImpl implements foo
 9: {
10:    IntervalSensor is = null;
11:
12:    public _fooStub()
13:    {
14:       is = (IntervalSensor) C_ORB.m_idskernel.getSensor(Analyzer.SEN_INTERVAL);
15:    }
16:
17:    public int bar(int arg1) throws IOException
18:    {
19:       byte[] _msg = null;
20:       int _result = 0;
21:       mqc.holders.IntHolder ptr = new mqc.holders.IntHolder();
22:       _msg = this._request("bar", 4, true, ptr);
23:       ptr.value = (ptr.value + 3) & ~3;
24:       if (_msg.length < (ptr.value + 4))
25:       {
26:          byte[] tmp = new byte[_msg.length + 4 + 10];
27:          System.arraycopy(_msg, 0, tmp, 0, _msg.length);
28:          _msg = tmp;
29:       }
30:       _msg[ptr.value++] = (byte) (arg1 >> 24);
31:       _msg[ptr.value++] = (byte) (arg1 >> 16);
32:       _msg[ptr.value++] = (byte) (arg1 >> 8);
33:       _msg[ptr.value++] = (byte) arg1;
34:       long _timex = System.currentTimeMillis();
35:       _msg = this._invoke(_msg, ptr, true);
36:       IntervalData data012 = (IntervalData) is.createDataNode(new Location(0, 1, 2));
37:       data012.setValue(System.currentTimeMillis() - _timex);
38:       is.recordData(data012);
39:       ptr.value = (ptr.value + 3) & ~3;
40:       _result = (int) ( ( (_msg[ptr.value++] & 0xFF) << 24) |
41:                         ( (_msg[ptr.value++] & 0xFF) << 16) |
42:                         ( (_msg[ptr.value++] & 0xFF) << 8) | (_msg[ptr.value++] & 0xFF));
43:       IntervalData data001 = (IntervalData) is.createDataNode(new Location(0, 0, 1));
44:       data001.setValue(_result);
45:       is.recordData(data001);
46:       return _result;
47:    }
48: };
49: // End of file.
50:
```

Source Listing 7. _fooStub.java

fooPOA.java

```
 1: package timing;
 2:
 3: import mqc.*;
 4: import mqc.ids.*;
 5: import mqc.ids.sensor.*;
 6: import java.io.IOException;
 7:
 8: public abstract class fooPOA extends mqc.Servant implements timing.fooOperations
 9: {
10:
11:    IntervalSensor is = null;
12:
13:    public fooPOA()
14:    {
15:       is = (IntervalSensor) S_ORB.m_idskernel.getSensor(Analyzer.SEN_INTERVAL);
16:    }
17:
18:    public String getID()
19:    {
20:       return "IDL:timing/foo:1.0";
21:    }
22:
23:    public byte[] _invoke(mqc.protocols.Reply handler, String method, byte[] _msg,
24:                          mqc.holders.IntHolder ptr) throws IOException
25:    {
26:
27:       /* bar */
28:       if (method.equals("bar"))
29:       {
30:          long l = S_ORB.m_idskernel.getLastInvocationTime();
31:          if (l == 0)
32:          {
33:             S_ORB.m_idskernel.setLastInvocationTime();
34:          } else
35:          {
36:             IntervalData data003 = (IntervalData) is.createDataNode(new Location(0, 0, 3));
37:             data003.setValue(S_ORB.m_idskernel.setLastInvocationTime() - l);
38:             is.recordData(data003);
39:          }
40:          ptr.value = (ptr.value + 3) & ~3;
41:          int arg1;
42:          arg1 = (int) ( ( (_msg[ptr.value++] & 0xFF) << 24) |
43:                         ( (_msg[ptr.value++] & 0xFF) << 16) |
44:                         ( (_msg[ptr.value++] & 0xFF) << 8) | (_msg[ptr.value++] & 0xFF));
45:          IntervalData data014 = (IntervalData) is.createDataNode(new Location(0, 1, 4));
46:          data014.setValue(arg1);
47:          if (is.recordData(data014))
48:          {
49:             throw new RuntimeException("IDS executed an REJECT INVOCATION response");
50:          }
51:          int _result;
52:          _result = bar(arg1);
53:          ptr.value = 0;
54:          ptr.value = (ptr.value + 3) & ~3;
55:          if (_msg.length < (ptr.value + 4))
56:          {
57:             byte[] tmp = new byte[_msg.length + 4 + 10];
58:             System.arraycopy(_msg, 0, tmp, 0, _msg.length);
59:             _msg = tmp;
60:          }
61:          _msg[ptr.value++] = (byte) (_result >> 24);
62:          _msg[ptr.value++] = (byte) (_result >> 16);
63:          _msg[ptr.value++] = (byte) (_result >> 8);
64:          _msg[ptr.value++] = (byte) _result;
65:          return _msg;
66:       }
67:       return null;
68:    }
69: };
70: // End of file.
71:
```

Source Listing 8. fooPOA.java

fooImpl.java

```
 1: /*
 2:  * Copyright (c) 2003 David E. Bakken, his research students, and Washington State University.
 3:  * Please see the file LICENSE.pdf for more details on terms of use.
 4:  */
 5:
 6: import mqc.S_ORB;
 7: import mqc.holders.*;
 8: import mqc.ids.*;
 9: import mqc.ids.sensor.*;
10:
11: public class fooImpl extends timing.fooPOA
12: {
13:    /** The procedural based sensor */
14:    protected ProceduralSensor ps = null;
15:
16:    /** Array holding the combinations of datapoints */
17:    protected static int[] array =
18:        {0, 6, 1, 3, 2, 6, 6, 1, 6, 1, 2, 4, 5, 6, 6, 3, 2, 3, 2, 6, 2, 1, 0, 4, 1, 2, 0,
19:        0, 0, 5, 2, 6, 0, 3, 0, 0, 5, 2, 3, 6, 3, 4, 6, 0, 5, 2, 2, 1, 4, 0};
20:
21:    /** Index into the array */
22:    protected static int index = 0;
23:
24:    /**
25:     * Default Constructor
26:     */
27:    public fooImpl()
28:    {
29:       ps = (ProceduralSensor) S_ORB.m_idskernel.getSensor(Analyzer.SEN_PROCEDURAL);
30:    }
31:
32:    /**
33:     * The implemented function
34:     */
35:    public int bar(int arg1)
36:    {
37:       if (arg1 == -1) // hack to stop the server by setting the keepRunning flag to false
38:       {
39:          S_ORB.keepRunning = false;
40:       }
41:
42:       index = (index + 1) % array.length;
43:       ps.recordData(ps.createDataNode(new Location(0, array[index])));
44:
45:       return arg1;
46:    }
47: }
```

Source Listing 9. fooImpl.java

Bibliography

[1] M. Almgren, and U. Lindqvist. “Application-Integrated Data Collection for Security Monitoring”. In *Proceeding of Recent Advances in Intrusion Detection (RAID)*, LNCS, pages 22-26, Davis, CA, October 2001, Springer.

[2] Anderson economic group. Mars 2004. See *http://www.andersoneconomicgroup.com/Publications/articles_pressreleases/ blackout_AEGwp2003-2.pdf*

[3] Arbor Networks Inc. April 2004. See website *http://www.arbor.com/*.

[4] M. Atighetchi, P. P. Pal, C. Jones, P. Rubel, R. E. Schantz, J. P. Loyall, and J. A. Zinky. “Building Auto-Adaptive Distributed Applications: The QuO-APOD Experience”. *The 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems*, in conjunction with the *23rd*

International Conference on Distributed Computing Systems, May 19-22, 2003, Providence, Rhode Island, USA.

[5] R. G. Bace. “Intrusion Detection”. Macmillan Technical Publishing, 201 West 103rd Street, Indianapolis, IN 46290.

[6] A. S. Brown. “SCADA vs. the hackers – Can freebie software and a can of Pringles bring down the US power grid?”. December 2002, Mechanical Engineering Magazine.

[7] Cisco. “Self Defending-Network”, April 2004. See *http://www.cisco.com/warp/public/cc/so/neso/vpn/vpne/csdnq_wp.pdf*.

[8] CylantSecure. April 2004. See website *http://www.cylant.com*/.

[9] R. F. Dacey. “Challenges in Securing Control Systems”. United States General Accounting Office Critical Infrastructure Protection, October 2003. See *http://www.gao.gov/cgi-bin/getrpt?GAO-04-140T*.

[10] Dallas Semiconductor and Maxim Integrated Products Inc. April 2004. See *http://www.maxim-ic.com/1-Wire.cfm*.

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

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

[13] S. Elbaum and J. Munson. “Intrusion Detection through Dynamic Software Measurement”, USENIX Workshop on Intrusion Detection and Network Monitoring, USENIX, 41-50, April 1999.

[14] E. Eskin, W. Lee, and S. J. Stolfo. “Modeling System Calls for Intrusion Detection with Dynamic Window Sizes”. In *Proceedings of DARPA Information Survivabilty Conference and Exposition II (DISCEX II)*. Anaheim, CA: June 12-14 2001.

[15] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. “A sense of self for Unix processes”. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, IEEE Computer Press, 1996.

[16] B. Gellman. “Cyber-Attacks by Al Qaeda Feared”. Washington Post Thursday, June 27, 2002; page A01.

[17] N. Gibbs. “Lights Out. Time Magazine”. Monday, August 25, 2003; page 30.

[18] O. Haugan. “Configuration and code generation tools for middleware targeting small, embedded devices”. Master’s thesis, Washington State University, December 2001. See *http://microqoscorba.eecs.wsu.edu/Haugan-Thesis*.

[19] R. Heady, G. Luger, A. Maccabe, and M. Servilla. “The Architecture of a Network Level Intrusion Detection System”. Technical Report CS90-20, University of New Mexico, Department of Computer Science, August 1990.

[20] S. Hofmeyr, S. Forrest, and A. Somayaji. "Intrusion Detection Using Sequences of System Calls". *Journal of Computer Security,* Vol. 6, pp. 151-180 (1998).

[21] IPTables. April 2004. See website *http://www.netfilter.org/*.

[22] H. S. Javitz, A. Valdez. “The NIDES Statistical Component: Description and Justification”. Technical report, SRI International, March 1993.

[23] K. P. Kihlstrom, P. Narasimhan. ”The Starfish System: Providing Intrusion Detection and Intrusion Tolerance for Middleware Systems”. *IEEE Workshop on Object-oriented Realtime Dependable Systems,* Guadalajara, Mexico, January 2003.

[24] E. Lundin. “Aspects of employing fraud and intrusion detection systems” Thesis, Technical Report No 2L, School of Computer Science and Engineering, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2002.

[25] D. Malkhi, M. Reiter, AT&T Labs. “Unreliable Intrusion Detection in Distributed Computations”. *10th Computer Security Foundations Workshop (CSFW '97),* June 10 - 12, 1997, Rockport, Massachusetts.

[26] G. Mazeroff, V. De Cerqueira, J. Gregor, and M. Thomason. “Probabilistic trees and automata for application behavior modeling”. In *Proceedings of the 41st Annual ACM Southeast Conference*, 2003 (pp. 435-440). Savannah, GA.

[27] D. A. 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.

[28] D. A. McKinnon, D. E. Bakken, and J. C. Shovic. “A configurable security subsystem in a middleware framework for embedded systems”. *Computer Networks*, Submitted for publication.

[29] D. A. McKinnon. “Supporting Fine-Grained Configurability with Multiple Quality of Service Properties in Middleware for Embedded Systems”. PhD thesis, Washington State University, Pullman, WA, September 2003.

[30] MicroQoSCORBA, April 2004. See website *http://microqoscorba.net/*.

[31] J. Munson and S. Winner. “CylantSecure: The Missing Piece of the Security Puzzle”. *17th Annual Computer Security Applications Conference*, December 10-14, 2001, New Orleans, Louisiana.

[32] P. Oman, E. Schweitzer, and D. Frincke. “Concerns about Intrusions into Remotely Accessible Substation Controllers and SCADA Systems”. *27th Annual Western Protective Relay Conference*, Paper #4, (October 23-26, Spokane, WA), 2000.

[33] P. Oman, E. Schweitzer, and J. Roberts. “Safeguarding IEDs, Substations, and SCADA Systems against Electronic Intrusions”, In *Proceedings of the 2001 Western Power Delivery Automation Conference*, Paper No. 1, (April 9-12, Spokane, WA), 2001.

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

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

[36] D. Ron, Y. Singer, and N. Tishby. "The power of amnesia: Learning probabilistic automata with variable memory length". *Machine Learning*, 25:117-150, 1996.

[37] E. Siever et al. *Linux in a Nutshell*. O'Reilly, 2000.

[38] Sanctum Inc. April 2004, See website: *http://www.sanctuminc.com/*.

[39] D. Sames, B. Matt, B. Niebuhr, G. Tally, B. Whitmore, D. E. Bakken. "Developing a Heterogeneous Intrusion Tolerant CORBA System". *International Conference on Dependable Systems and Networks (DSN'02)*, June 23 - 26, 2002, Washington, D.C., USA.

[40] R. S. Sielken. "Application intrusion detection", Technical Report CS-99-17, Department of Computer Science, University of Virginia, June 1999.

[41] Snort. April 2004. See website *http://www.snort.org/*.

[42] A. Somayaji and S. Forrest. "Automated response using system-call delays". In *Proceedings of the 9th USENIX Security Symposium*, August 2000. See *http://cs.unm.edu/~forrest/publications/uss-2000.ps.*

[43] E. Spafford, D. Zamboni. "Data collection mechanisms for intrusion detection systems". CERIAS Technical Report 2000-08, CERIAS, Purdue University, 1315 Recitation Building,West Lafayette, IN, June 2000.

[44] Sygate Technologies. April 2004. See website: *http://www.sygate.com/*.

[45] D. Tennenhouse. “Embedding the Internet: Proactive Computing”, *Communications of the ACM*, May, 2000.

[46] TINI. April 2004. See website *http://www.ibutton.com/TINI/*.

[47] Tripwire. April 2004. See website *http://www.tripwire.com/*.

[48] J. Turley. “The Essential Guide to Semiconductors”. Prentice Hall, 2003, Professional Technical Reference, Upper Saddle River, NJ 07458, www.phptr.com.

[49] J. A. Whittaker and H. H. Thompson. “How to Break Software Security”. 1st edition, Addison Wesley 2003.

[50] D. Zamboni. “Using Internal Sensors for Computer Intrusion Detection”. PhD thesis, Purdue University, West Lafayette, IN, August 2001, CERIAS TR 2001-42.

[51] J. A. Zinky, D. E. Bakken, and R. E. Schantz. “Architectural Support for Quality of Service for CORBA Objects”. *Theory and Practice of Object Systems*, vol. 3, num. 1, April, 1997.