NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF REAL-TIME MSHN USING ACE AND TAO

by

Panagiotis Papadatos

September 1999

Thesis Advisor:	Taylor Kidd
Second Reader:	Debra Hensgen

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: IMPLEMENTATION OF REAL-TIME MSHN USING ACE AND TAO	5. FUNDING NUMBERS
6. AUTHOR(S) Papadatos, Panagiotis	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Management System for Heterogeneous Networks (MSHN) project is a part of the DARPA/ITO QUORUM program. MSHN targets the execution of multiple, disparate tasks that use a set of shared, heterogeneous resources in a way that maximizes a collection of application-specific quality of service (QoS) measures.

This thesis examines some of the architectural requirements demanded of MSHN for it to be able to operate in a real-time environment, and presents an implementation of a MSHN communication schema using components designed for supporting real-time applications. This implementation is built over the Adaptive Communication Environment (ACE), a freely available, open-source, object-oriented (OO) framework for building concurrent communication. To support the communication between MSHN components, we used the Common Object Request Broker Architecture (CORBA), particularly The ACE ORB (TAO), a standards-based, CORBA middleware framework. Both ACE and TAO are being developed at the Washington University in St. Louis, MO.

In our experiments, we define and measure the latency (communication time required to start an application) and agility (communication time required to migrate an application given a platform failure). We find that MSHN has the potential for supporting certain types of real-time systems, such as vehicle control.

14. SUBJECT TERMS Resource Management Systems, MSHN, CORBA, ACE, TAO, Real-time, RMS	15. NUMBER OF PAGES 140
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFI-CATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

IMPLEMENTATION OF REAL-TIME MSHN USING ACE AND TAO

Panagiotis Papadatos
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

Author: [illegible]
Panagiotis Papadatos

Approved by: [illegible]
Taylor Kidd, Thesis Advisor

[illegible]
Debra Hensgen, Second Reader

[illegible]
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

The Management System for Heterogeneous Networks (MSHN) project is a part of the DARPA/ITO QUORUM program. MSHN targets the execution of multiple, disparate tasks that use a set of shared, heterogeneous resources in a way that maximizes a collection of application-specific quality of service (QoS) measures.

This thesis examines some of the architectural requirements demanded of MSHN for it to be able to operate in a real-time environment, and presents an implementation of a MSHN communication schema using components designed for supporting real-time applications. This implementation is built over the Adaptive Communication Environment (ACE), a freely available, open-source, object-oriented (OO) framework for building concurrent communication. To support the communication between MSHN components, we used the Common Object Request Broker Architecture (CORBA), particularly *The ACE ORB* (TAO), a standards-based, CORBA middleware framework. Both ACE and TAO are being developed at the Washington University in St. Louis, MO.

In our experiments, we define and measure the latency (communication time required to start an application) and agility (communication time required to migrate an application given a platform failure). We find that MSHN has the potential for supporting certain types of real-time systems, such as vehicle control.

TABLE OF CONTENTS

ACKNOWLEDGEMENT / DEDICATIONS

I would like to extend my sincere gratitude to my thesis advisors, Professor Taylor Kidd and Professor Debra Hensgen, for their patience and for providing continuous support and direction during this work. Additionally, I would like to thank my wife, Maria Vassilaki for leaving her work for more than two years so I could do these studies. And finally, I would like to dedicate this thesis to my son, Homer for his unconditional love and patience during this period.

I. INTRODUCTION

The Management System for Heterogeneous Networks (MSHN) is a Resource Management System (RMS) currently being designed and developed in the Heterogeneous Laboratory of the Naval Postgraduate School. MSHN aims to support the concurrent execution of different applications[1] in a heterogeneous environment. The intent of MSHN is to handle the Managed Environment[2] (ME) as a "virtual heterogeneous machine" [Ref. 8] that will select the most feasible resource configuration for the execution of any submitted job.

Although in the early development stages MSHN provided a "best effort" Quality of Service (QoS) to its applications [Ref. 1], the concept and design of MSHN have features, as it will be analytically presented later, intended for the support of real-time applications. In addition, MSHN itself comprises a real-time[3] application that must monitor and control a heterogeneous environment with performance specifications

[1] The terms "application", "process" and "job" will be used interchangeably to refer to a programming entity that needs to be serviced.

[2] By Managed Environment (ME) or Managed System (MS) we refer to the collection of applications that MSHN controls (and is influenced by), as well as the underlying OSs and platforms.

[3] "A real-time system must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure." [Ref. 49].

dictated by the requirements of its end users[4]. The performance characteristics that are required for MSHN to accomplish its goals compel a cautious selection both of its implementation structure and the communication means that will be used for the interaction of its components.

From this perspective, we chose in this research to create a MSHN-like communication scheme using a promising combination of framework and middleware[5], namely ACE and TAO, developed at the Washington University of St. Louis, MO (WUSTL). The specific characteristics of ACE and TAO, as well as the way these characteristics can influence the performance of MSHN, are presented in the following chapters. Also, as a result of our experimentation, we provide a performance analysis of this communication / implementation schema that we hope provides a basis for evaluating MSHN's applicability to a real-time environment.

A. MOTIVATION

Real-time systems have stringent QoS requirements. In order for MSHN to be able to support the execution of real-time applications, it must both provide the required functionality, and ensure that such functionality performs within specified limits.

[4] With the term "user" we refer to any entity (application or device) in the ME that requests some service.
[5] As far as the communication mechanism is concerned, the use of Commercial Off The Shelf (COTS) software, open systems architecture and middleware was proposed with a sound reasoning in [Ref. 9].

For the implementation of its components, MSHN desires to utilize a schema with enhanced portability and reuse features. The portability is dictated by MSHN's need to operate in a heterogeneous environment. Reuse (both from an object and design perspective) is a beneficial software engineering principle that we desire to utilize in our implementation [Ref. 12]. Other desirable features for an implementation of MSHN include robustness, high performance, high availability, scalability and extensibility. Of course, development and maintenance costs are always an issue.

The importance of the communication between MSHN components is also significant. As MSHN is targeting both local and distributed environments, we expect that the communication mechanisms and techniques used will play a major role in its performance.

We chose to use ACE and TAO for this experimental implementation for a number of reasons:

- Their features, as described in numerous papers such as [Ref. 13] and [Ref. 14], match our performance requirements.
- We wanted to familiarize ourselves with this framework, its ORB implementation, and its underlying design features.
- They are freely available.
- They are adopted for use in a large number of critical real-time applications in commercial projects as well as many academic and industrial research projects.

- Their underlying principles and mechanisms are well documented and, though they have a steep learning curve, once understood, their components can be effectively reused to create applications that are suitable for execution in a MSHN controlled environment.
- They can be used with a variety of operating systems and platforms.
- They are open source, offering a better understanding of their mechanisms and more full use of their capabilities.

In this thesis, we intend to study and present the characteristics and architecture of ACE and TAO, and use them in order to provide a sample implementation of a MSHN-like communication scheme that will be available for further experimentation. By measuring the time required for various operations performed by MSHN, we will be able to provide an estimate for the minimum delay required by those operations.

B. SCOPE OF THESIS

This thesis will explore two performance measures which we consider fundamental for the support both of real-time systems and of the QoS that MSHN will provide to its applications: latency and the agility.

"Latency" we define as the amount of time required for a newly submitted job to start executing. The term "agility" characterizes the time required by MSHN to respond to a critical change in the controlled environment, that critical change being related to the performance of an application being managed [Ref. 50].

We consider these two parameters of great significance for critical real-time applications. As an example, the delay of the start of the execution of a new job (latency) may be critical to the initiation of a new process that will track a target and provide data to a fire control system. Also, in the case of the failure of a computational unit in a tactical system, the time required for MSHN to react and re-initiate the process is a critical parameter which should be considered in evaluating the feasibility of MSHN in such a scheme.

The internal delay of its components is a major factor in the performance of MSHN. This is still an area of active research. As such, the delay associated with the internal function of the components is variable and depends on factors such as the implementation scheme, the OS and the platform. So, we will refrain from taking into account these internal delays and instead, measure the pure communication overhead imposed by real-time CORBA and the operating system. We argue that this is a worst case scenario from the communication perspective, especially in the event of heavy load, as communication delays will not be masked by the delays of the components. Elaboration on this perspective will be presented when we discuss our design in Chapter VI.

By measuring the communication latency and agility of MSHN in this particular implementation, we seek to establish an average minimum value for these factors. These values can be compared to the requirements of real-time applications in order to investigate the suitability of MSHN in such environments.

C. ORGANIZATION

In this thesis, we first present the architecture of MSHN. Next we briefly present the underlying theory of the tools we used for our sample implementation. We present the concepts of design patterns and frameworks, the motivation for using them and their utilization in ACE. Then we discuss the communication scheme used for our experimentation (real-time CORBA) and the special requirements that made us choose the particular implementation (TAO).

Finally, we present and explain the design and implementation of the experimental system, followed by a summary and propositions for future work.

II. ARCHITECTURE OF MSHN

In this chapter, we first introduce MSHN and then provide a brief description of the functionality of its components. Finally, we discuss the interaction of the various MSHN components, focusing on the exchange of messages, the interface exposed by each component, and the event handling.

A. PURPOSE

The Management System for Heterogeneous Networks (MSHN) is a Resource Management System (RMS) being designed and developed in the Heterogeneous Processing Laboratory at the Naval Postgraduate School. The project is supported by the QUORUM program, which is a project under DARPA / ITO.

The aim of MSHN is to determine an effective design for an RMS that can deliver, whenever possible, the required quality of service (QoS) to individual processes that are contending for the same set of distributed, heterogeneous resources [Ref. 1]. Some of the current active areas of research within the MSHN project include scheduling, application and system characterization, resource status determination, and security evaluation. The nature of MSHN's targeted environment imposes many challenges. MSHN must support the concurrent execution of many different applications (with a diversity of QoS requirements) varying from resource intensive computational applications with relaxed deadline requirements to critical real-time control systems. Clearly, satisfying such a variety of QoS requirements demands the support of a robust and flexible RMS. Moreover, not only these applications, but also the MSHN components themselves, are required to execute in a distributed and heterogeneous environment. This imposes an extra level of difficulty, including more stringent requirements on the exchange of messages and the handling of events in MSHN.

B. MSHN'S ARCHITECTURE

In this section, we will briefly introduce the architecture of MSHN in order to later elaborate, in detail, on its communication and interfacing issues. An extensive and detailed presentation of MSHN, its background, and its current and future research can be found in [Ref. 1].

Briefly, MSHN is composed of the following components (Figure 1):

- Client Library (CL),
- Resource Status Server (RSS),
- Resource Requirements Database (RRD),
- Scheduling Advisor (SA),
- Application Emulator (AE),
- MSHN Daemon.

The interactions are shown in Figure 2.

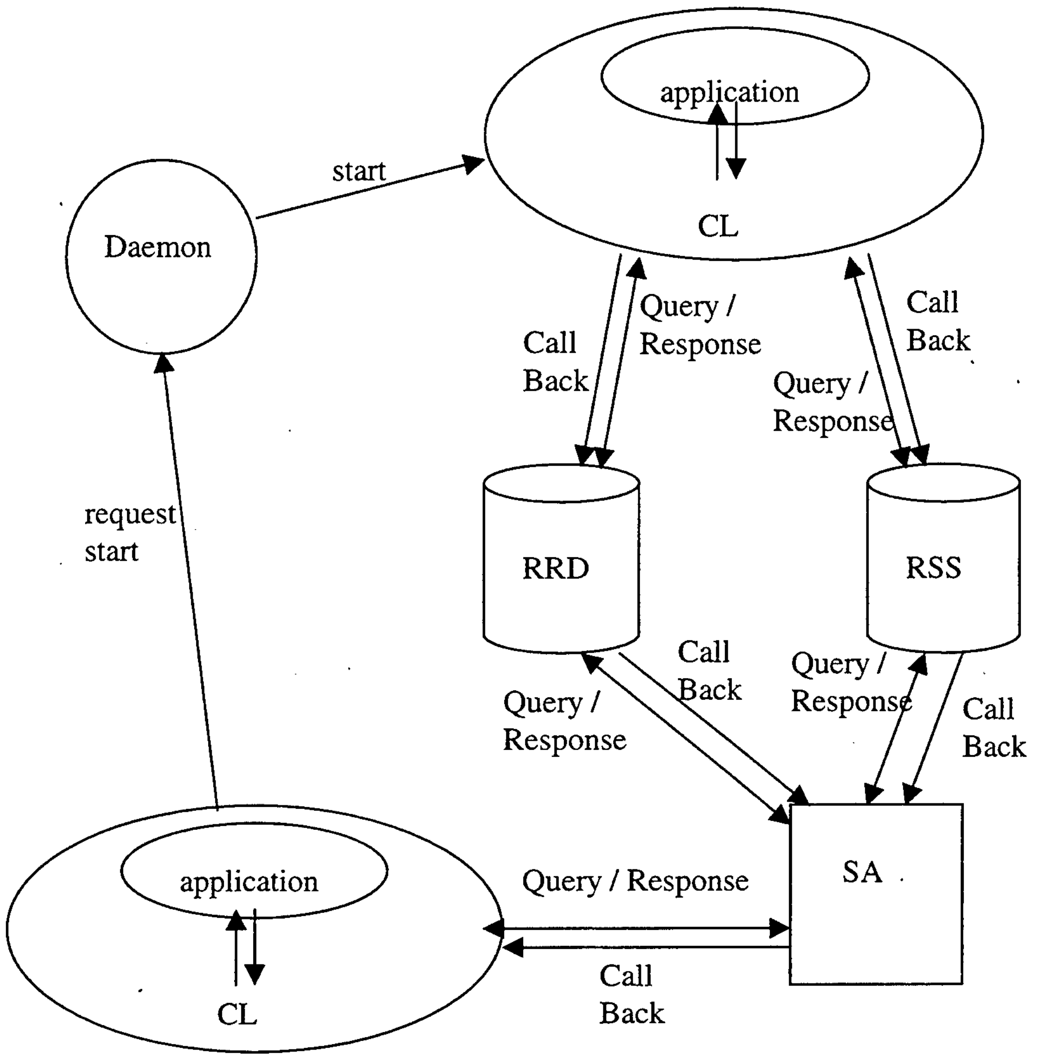

Figure 1. MSHN Architecture

1. The Client Library

The CL [Ref. 42] is linked with all the applications that use MSHN. The CL wraps the application, providing a proxy between the application and the environment. Once a new process or application is started, the CL queries the SA for an available platform suitable to execute this new task. Upon direction from the SA, if the platform specified is not the same as the platform of origin, the CL requests the daemon running on the assigned platform to initiate the application. The CL is responsible for updating the RSS and the RRD with information concerning the resource usage of the process, as well as the availability of the resources of the platform upon which the CL is running [Ref. 43].

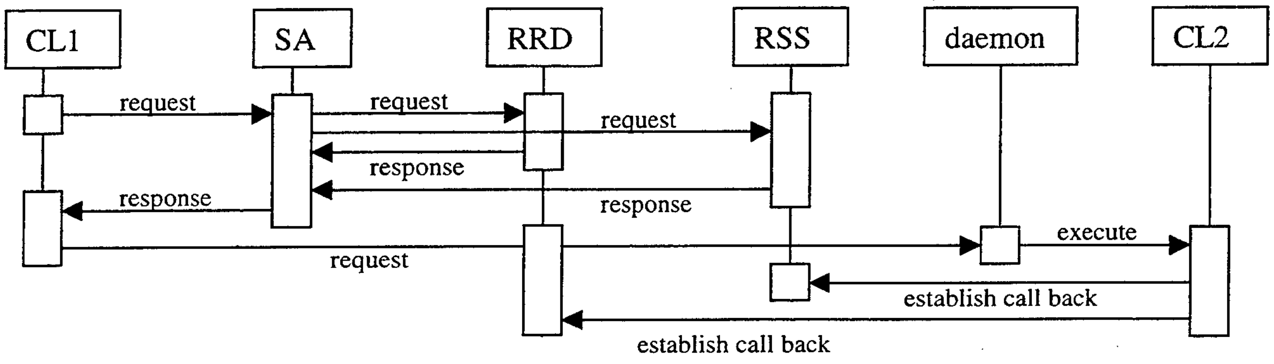

Figure 2. Interaction diagram

2. The Resource Status Server

The RSS maintains availability information for all the resources in the system. It is updated by the CL of every MSHN application that is running regardless of platform. The RSS provides information, upon request, to the SA. This information is then used, along with other information, to designate a platform for a new process. Also, it issues a callback message to the SA whenever any of the monitored resources exceed or fall below a predetermined threshold.

3. The Resource Requirements Database

The RRD [Ref. 44] keeps information concerning the resource requirements and resource usage of applications. It provides this information to the SA upon request or if there is a violation in an established threshold. The RRD is updated by the CL of every application running under MSHN.

4. The Scheduling Advisor

The SA [Ref. 45] allocates resources to applications by specifying on which platform the application should run. Upon request from the CL of the originating application, the SA queries the RRD and RSS, obtaining data about the requirements of the new process and the status of the resources of the system. Then the SA advises the requesting CL as to where to execute the new process [Ref. 46].

5. The Application Emulator

The AE [Ref. 47] has two functions: mimicing an application, and monitoring the availability of the resources on a platform. For its second function, when there is no CL running on the platform, the AE is initiated by a daemon in order to monitor the local resources. The AE is also used to simulate a given application in order to get an idea as to the behavior of both the application and MSHN were the actual application to be executed.

6. The MSHN Daemon

A MSHN Daemon executes on every platform available for use by MSHN. It starts any application requested by the CL of a remote machine, and is responsible for the initiation of AEs.

C. OVERVIEW OF COMPONENTS' INTERACTION

MSHN attempts to present to the application what appears as a diaphanous monolithic operating system [Ref. 1]. It does this by extending the functionality of the underlying OS. Doing so, it transparently intercepts any request that attempts to start executing an application. This interception is accomplished via the CL, which is linked with every MSHN application[6]. After interception, the CL will divert the request to the SA. The SA requests information about the system's resource status from the RSS, and the requirements of the particular application from the RRD. Based on this information, the SA implicitly assigns to the application specific resources by recommending that the original CL execute the application on a particular platform. At this point, the originating CL will request that the MSHN daemon on this particular platform execute the application. The daemon will launch the application, which is wrapped with a CL that will become the proxy of this new application with MSHN. During the life of the application, the CL will update the RSS with information about the availability of the resources of the local platform, and update the RRD about the usage of the resources by the application. The SA will direct the RSS and RRD on which callbacks they will issue to the SA and when they should occur.

[6] A MSHN application is a program linked with the MSHN wrapper libraries (the CL). This wrapper allows MSHN to intercept the interaction between the particular program and the operating system, enabling MSHN to monitor the behavior and control the execution of the application. Applications that are not wrapped affect the MSHN controlled environment as external factors, influencing the availability of resources. This is particularly important in a real-time environment, as is discussed in the next section.

THIS PAGE INTENTIONALLY LEFT BLANK

III. MSHN IN SUPPORT OF A REAL-TIME ENVIRONMENT

In this section, we examine and propose the functionality that the Management System for Heterogeneous Networks (MSHN) should have to support the execution of critical real-time applications.

A. INTRODUCTION

Real-time systems have stringent Quality of Service (QoS) requirements associated with them. As the resources of any given environment are limited, the introduction of a new application on an already loaded environment may not adversely affect that application, but the additional load can unacceptably degrade the overall system's performance. Non-real-time applications can tolerate delays, and non-critical applications can allow a certain risk of failure. In the case of critical real-time applications, though, the Resource Management System (RMS) must have tighter control of resources and applications; this is no longer a "best effort" environment, where delays and failure can be tolerated. Such critical applications require a certain degree of reliability. Also, there is a great need for information about the particular characteristics and status of the Managed Environment (ME). This need is imposed by the necessity to perform the scheduling as fast as possible and as accurately as practical, and by the requirement to minimize the amount of intervention required in the course of a job's execution. These proscribed requirements restrict the freedom of the users of a MSHN enhanced environment, but we argue here that if real-time critical applications are to be supported, then eventually the control of the ME presumed by MSHN should be stringent.

B. REAL-TIME ASPECTS AND FEATURES OF THE MANAGED ENVIRONMENT

A real-time RMS will try to best allocate the resources that it manages to the processes requesting its services. Its aim is to provide the QoS that is required both by the serviced processes and by the managed environment as a whole. In order to accomplish this, the RMS should have information on the following:

- The requirements of the applications already running on the managed platforms.
- The resources required by any new application and the actual availability of those resources on the managed platforms.
- The impact of the new application on the system.

Issues arising from the above have to do with the degree of knowledge that the RMS has about the managed environment (which consists of the environment's resources and the applications currently executing), and the privileges and authority that the RMS has with respect to the resources it manages. We will elaborate on these issues by examining the following topics:

- The autonomy of the managed system;
- The knowledge of the RMS on the behavior of the applications;
- The periodicity of the applications; and
- The state of the applications.

1. The Autonomy of the Managed System

The autonomy of the managed system has to do with whether all the applications running on this system are controlled by the RMS. Although it is desirable for an RMS to have a minimal impact on the environment it manages, this requirement is sometimes an unnecessary burden that limits and complicates the RMS's operation. If the RMS has no direct knowledge of, or relationship with, an application running in the ME, the RMS must rely upon its ability to sense the changes in the environment, and upon its ability to respond in a timely fashion to those changes. Since some applications do not initially need all the resources they eventually require, but instead request these resources gradually during their execution, the RMS needs the ability to respond to a continuously changing environment. This can be accomplished via having the appropriate information stored in the Resource Requirements Database (RRD) for each application. We assume that the application follows a consistent pattern of behavior. In cases where this is not so, MSHN has developed a methodology called Compute Characteristics [Ref. 51], which determines the behavior of the application by analysis (if the source code is available) or direct measurement. If no data exists on the behavior of the application, the Scheduling

Advisor (SA) will not know which resources to assign to the application, nor will it be able to predict the application's future resource demand. An expected consequence is that, when the required resources are not available, the application (or another competing with it for the same resources) may need to be migrated. If this is not possible, then the less critical application may need to be terminated. In any case, given that the ability of the RMS to predict the future behavior of the ME will be limited, the RMS will have to respond quickly for every application that enters the environment. This quick response will sometimes necessitate the making of poor decisions, while providing the same response agility to all the submitted applications regardless of the significance and required QoS of the applications.

2. The Knowledge of the RMS about the Behavior of the Applications

Certain applications follow specific behavioral patterns. Examples of these include when an application retrieves large amounts of data from a file at the beginning of its execution, and when applications have extended periods of calculation, communication or use of I/O. These behavioral patterns can be extracted by examining the design of the application, if available, or by examining and analyzing its resource utilization during execution. Subsequently, such pattern information can be stored in the RRD and provided to the RMS upon request in the RRD.

Applications that cannot tolerate a delay in the beginning of their execution (like the assignment of a target to a weapon control system prior to firing) should register their requirements with the RMS. Applications that can tolerate some short initial delay (such as a component that feeds a message to an automated message processing system) can be given a controlled entry point (admission control) into the system, designed to allow the RMS sufficient time to respond promptly.

The availability and utilization of such information is essential for the effective scheduling of the application by the SA. It allows the RMS to bind and have available resources when they are needed, avoiding undue delay in the execution of the submitted jobs and the underutilization of resources.

3. The Periodicity of the Applications

Some applications in a given ME execute periodically. Such applications may update a display or database, check the status of certain components, or provide data to a remote station. This periodicity can be utilized by the RMS to predict when certain resources will be required, facilitating the coexistence of applications competing for the same resources.

4. The State of the Applications

An application may be blocked at some point because a required resource is not available. For example, if a program is unable to send data because the associated buffer is full, instead of discarding or ceasing the generation of data, the program can explicitly inform the RMS of its status and request more resources.

C. CONTROL OF THE APPLICATIONS AND RESOURCES THEY USE

An application running in a managed environment can be controlled in the following ways: admission control, version selection, adaptation, suspension and premature termination.

1. Admission Control

Admission control is realized by calculating the impact of the initiation and subsequent execution of an application in the ME prior to its admission. The RMS must have the authority to delay, postpone or even deny the execution of the application, according to a contract negotiated during the application's submission.

Another aspect of admission control is the authority a user has to initiate a process. We can generally classify applications into three categories:

- Registered Applications
- System Applications
- Unregistered Applications

a) Registered Applications

Registered Applications are programs familiar to the RMS, with very well known behavioral patterns and resource requirements; in the case of multi-version applications, they must also have well-defined QoS profiles. These are either applications with a history (i.e., having run several times in the managed environment such that a satisfactory amount of data has been collected about its execution), or with specific entries in the RRD. Registered Applications are programs that the RMS knows how to manage and are expected to run in the ME. Admission control is easier for Registered Applications, as there are less security issues involved and scheduling is faster due to the familiar profile. Moreover, in case of applications with small variances in the required resources, the resource allocation margins are smaller due to well-defined resource requirements.

b) System Applications

System Applications are Registered Applications that either perform system functions or are automatically initiated by the managed system. They are usually either periodic in nature or their initiation can be anticipated by the RMS as part of a prescribed sequence of actions. In a non-pathological managed environment, there should always be either available resources or applications that are candidates for termination so that System Applications can be admitted with minimal delay.

c) *Unregistered Applications*

Unregistered Applications are those for which the RMS has minimal or no knowledge. The admission of Unregistered Applications in a critical real-time system must be handled carefully by the RMS. The user must provide authentication and QoS requirements. An Unregistered Application, though unknown, can be of critical nature, so the user must be able to choose from a selection of QoS parameters. Unregistered Applications should normally expect a larger delay than applications in the other categories we examined. They also face a higher rate of rejection than Registered Applications of the same profile, as the resource safety margins imposed by the RMS, due to the unavailability or low reliability of resources, will be significantly larger than those for the Registered Applications. In the case where these applications have a high degree of criticality, this large safety margin ties down a large amount of resources. Unregistered Applications should also expect a larger possibility of premature termination, especially if they have an unusual behavioral pattern and their resource demands unpredictably exceed those reserved or available to them.

2. Version Selection

Version selection (or external adaptation) is a means to allow the RMS to attempt to satisfy the QoS requirements of an application to an extent that imposes an acceptable load on a system with limited resources. This is feasible when the application is available in different versions, each having a different QoS profile and associated resource requirement footprint. The RMS will attempt to schedule for execution the version providing the most favorable QoS for the user; though, if needed, the RMS will eventually degrade the offered QoS to the lowest acceptable level that satisfies the overall managed system's QoS requirements by running the appropriate version.

3. Adaptation

We characterize an application as (internally) adaptable when it can adjust its behavior according to the available resources and its QoS requirements. Adaptability is achieved by having the application downgrade its own performance to an acceptable level. This can be accomplished by the application calling different forms of functions, transferring alternative file formats or performing calculations with varying granularity.

In order to be adaptable, an application must be implemented with a tighter bind to the RMS. This drawback comes with a reward though, as the adaptability provides to both the RMS and application an extra degree of freedom as the granularity of and sophistication with which available resources are used can be varied.

4. Suspension

A process may be suspended for some time depending upon its nature, its QoS requirements and its importance / criticality. The nature of resources that need to be freed-up will play a major role in determining both which application to suspend and the way the suspension will happen. For example, a process can be passivated for some duration, in order to release some memory or enable a system to avoid thrashing. Upon handling of the crisis, the RMS will restart / continue executing the terminated / suspended the application.

5. Premature Termination

Premature termination is the forced termination of a process before it has finished its execution regardless of the will of the user / initiator. An application can be forced to end when any of the following conditions hold:

- The application cannot be suspended (because it is neither feasible nor useful to do so).
- The application cannot adapt or run in another viable version.
- The system does not have (or is willing to allocate) enough resources to support the execution of the application at an acceptable QoS level.

Neither control of an application's admission nor its premature termination imply the modification of existing code. (Though an internally adaptive application, which implies an implementation with the RMS in mind, could be highly desired and would improve the efficiency of the RMS.) Pre-existing applications can be managed more efficiently if

- their operational characteristics and behavior are well known to the RMS,
- they have a well-defined QoS profile, and

- the RMS has the authority to enforce admission and termination rules based upon a specified priority scheme.

D. THE EFFECT OF UTILIZING AN RMS IN A REAL-TIME ENVIRONMENT

The utilization of MSHN in a real-time environment is expected to impose a certain level of complexity and overhead on the ME.

The complexity is introduced as a result of the information that the designers of MSHN desire to have about the applications and the control MSHN should have with regards to the execution of the applications. Both elements affect the effectiveness of MSHN. Extensive information about an application's behavior and characteristics, the availability of different versions for its execution, as well as provision for its adaptability allow MSHN to manage its environment very effectively.

The overhead results from the delay caused by the querying, schedule computation, and communication associated with assigning an application to a platform for execution, and to any costs relating to the intervention of MSHN in the execution of already running applications (such as in response to any extensive change in the resource availability of and demand on the ME).

Nevertheless, MSHN is expected to significantly increase the utilization of the available resources and improve the overall QoS of the ME. Although this is largely dependent on the degree of cooperation between the ME and MSHN, as described above, even in the case that the resource requirements of applications are not provided, the ability of MSHN to extract information about the behavior of the ME will allow MSHN to perform its functions.

Another benefit of using an RMS such as MSHN in critical real-time applications is the RMS's ability to provide fault tolerance to the managed environment. MSHN can detect, with a certain probability, the failure of resources and applications. It can attempt to recover from such a failure, depending on the nature and importance of the application and the availability of additional resources. It can also, using the past history of failures, determine the proper number of duplicate applications to run in order to reduce the future probability of failure below a certain value.

MSHN can predict and prevent a failure by detecting the incorrect behavior of or a severe degradation in performance of an application, and taking the appropriate action (such as suspending or migrating the process / processes). MSHN's ability to report detailed specific or statistical information about the execution and performance of an application as well as to provide to the user the timely notification of events affecting the application's performance will be additional features.

IV. PROGRAMMING TOOLS AND TECHNIQUES

In this section, we introduce the basic concepts and definitions underlying design patterns, frameworks and toolkits. We then present ACE, which realizes and implements the above concepts into an integrated suite to ease the development of critical, real-time applications.

A. DESIGN PATTERNS

1. Introduction

The design pattern is a concept introduced within the last two decades to Computer Science. Its origins possibly come from a pattern language for architecting buildings and cities, proposed and used by Christopher Alexander, an architect, and his colleagues. According to Christopher Alexander, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [Ref. 2]."

Although Alexander applied the concept of patterns to architecting buildings and towns, his method has found application in many aspects of both everyday life and science, including object-oriented design. The solutions proposed by CS researchers, such as the GoF[7] and those that followed them, are expressed in terms of objects and interfaces instead of walls and doors, but at the core of all these patterns is a solution to a problem within a context [Ref. 7].

We can view a design pattern as another form of documentation [Ref. 10]. It is used to systematically name, evaluate, explain and motivate a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when the solution is applicable, and the solution's consequences. It also provides implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. Applying the design pattern, the programmer customizes and implements this general solution to solve the problem in his particular context. GoF presents a formalized and broadly adopted specification for using design patterns, as well as a fundamental collection of such patterns [Ref. 3]. Recently, an increasingly large collection of patterns has emerged, especially from the conferences on Pattern Languages of Program Design (PLoP) [Ref. 4, 5, 6].

The theory underlying the design patterns concept originated from the observation that certain problems occur repeatedly in a particular context, and that their solutions generally follow a similar stereotype.

The motivation behind using design patterns is to facilitate the design of object-oriented software, and to assist in the creation of reusable software. The goal is to capture

[7] GoF: The Group of Four (also called the Gang of Four) consists of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Their work "Design Patterns: Elements of Reusable Object-Oriented Software" is now considered a classic in the design patterns programming paradigm.

design experience in a form that developers can use. Also, using design patterns makes it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes these techniques more accessible to the developers of new systems. Design patterns help developers choose design alternatives that make a system reusable while avoiding alternatives that might compromise that reusability. The use of design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions as well as underlying intent.

What is and is not a pattern is effectively determined by the point of view of the individual. Patterns discuss a problem at a certain level of abstraction; thus, they should neither reproduce simple designs (such as linked lists and hash tables), nor should they be complex, domain-specific designs for an entire application or subsystem. Patterns are descriptions of communicating objects and classes, customized to solve a general design problem in a particular context [Ref. 11].

2. Decomposition of a Design Pattern

A design pattern names, abstracts and identifies the key aspects of a common design structure, promoting the creation of a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of their responsibilities [Ref. 3]. A design pattern focuses on a particular object oriented design problem or issue, describing where the pattern applies, whether in view of other design constraints the pattern should be applied, and the consequences and trade-offs of the pattern's use. Usually, a design pattern provides sample code to illustrate its implementation.

Generally, a pattern has five essential elements:

- the pattern name
- the problem
- the context
- the solution
- the consequences

a) The Pattern Name

The pattern name is a handle for the design pattern and its elements. Using a specific name for a pattern allows the initial design to proceed at a higher level of abstraction. Following a practice long established in other sciences and professions, the use of specific common names for particular patterns allows the developers to achieve clarity in the communication, documentation, discussion and application of patterns.

b) The Problem

The problem element describes when the pattern can be applied, by explaining the problem and its context. It might include specific design subproblems, such as how to represent algorithms as objects, as well as detail object structures that are symptomatic of an inflexible design. Sometimes the problem element includes a list of conditions that must be met before the pattern can be applied.

c) The Context

The context is essentially those forces that must be carefully considered before applying the pattern [Ref 10]. In any particular setting of a problem, the impacting forces may have different influences, depending on the particular situation. In such cases, applying a particular solution will have dissimilar results and consequences depending on the situation.

d) The Solution

The solution element describes the components that make up the design, their relationships, responsibilities and collaborations. The solution element does not offer any particular concrete design or implementation because a pattern is essentially a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of components (classes and objects in our case) solves the problem.

e) The Consequences

The consequences element contains the results and trade-offs associated with applying the pattern [Ref. 3]. Consequences are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. Points that have to be considered concern space and time trade-offs, language and implementation issues, as well as the impact of the pattern on a system's flexibility, extensibility and portability.

B. FRAMEWORKS AND TOOLKITS

1. Toolkits

A toolkit is a collection of related and reusable classes designed to provide useful, general-purpose functionality. An example of a toolkit is the C++ I/O stream library. Toolkits do not dictate a particular design on an application; instead they provide functionality that assists in producing an implementation in an easier and safer way. This is accomplished by avoiding (1) the recoding of common functionality and (2) error prone and difficult-to-implement programming structures. Toolkits emphasize code reuse; they are incorporated as classes from one or more libraries and they are considered as the object-oriented equivalent of subroutine libraries.

2. Frameworks

A framework is a set of cooperating classes that make up a reusable design for a specific class of software [Ref. 3]. It provides architectural guidance by partitioning the design into abstract classes and defining the responsibilities and collaborations of the classes. A developer customizes the framework for a particular application by subclassing and composing instances of framework classes.

Two examples of the use of framework applications are (i) in the building of graphical editors for different domains (e.g., artistic drawing, music composition and mechanical CAD), and (ii) in the building of compilers for different programming languages and target machines.

The framework imposes an architecture on the specific application by predefining design parameters such as the overall structure, its partitioning into classes and objects, the key responsibilities for collaboration between classes and objects, and the thread of control. Thus, the designer/implementer can concentrate on the specifics of the application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses that can be utilized immediately.

Reuse on this level leads to an inversion of control between the application and the software on which the application is based. When a toolkit is used (or a conventional subroutine library), the developer writes the main body of the application and calls the code he wants to reuse. In contrast, when a framework is used, the main body is reused and the developer writes the code the main body calls. He has to write operations with particular names and calling conventions, but that reduces the design decisions that have to be made. Not only can applications be built faster as a result but they also have similar structures. They are easier to maintain, and seem more consistent to their users. On the other hand, some creative freedom is lost, since many design decisions have already been made for the developer.

Applications are dependent on the framework for their design and they are sensitive to changes in its interfaces. Applications should be loosely coupled with the framework, so as a framework evolves, they can evolve with it without major repercussions.

The design issues just discussed are most critical to framework design. A framework that addresses them using design patterns is far more likely to achieve high levels of design and code reuse than one that does not. Mature frameworks usually incorporate several design patterns. The patterns help make the framework's architecture suitable for many different applications without redesign. ACE, which will later be presented, is a representative framework heavily based upon patterns.

A framework which is documented with the design patterns it uses has an additional benefit. People who know the patterns can more readily understand the framework, while those who do not know the patterns can benefit from the structure they lend to the framework's documentation. As frameworks pose a steep learning curve that must be overcome before they are used, enhanced documentation is particularly important. Design patterns can make the learning curve less steep by making key elements of the framework's design more explicit.

Patterns and frameworks are different in three major ways:

1. Design patterns explain the intent, trade-offs and consequences of a design. They are more abstract than frameworks and they have to be implemented each time they are used. Examples of patterns, when provided, assist in the understanding and the effective application of the patterns. Frameworks can be written down in programming languages and not only studied but also executed and reused directly.
2. Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns.
3. Design patterns are less specialized than frameworks, and unlike frameworks, they do not impose a particular application architecture.

C. ACE – THE ADAPTIVE COMMUNICATION ENVIRONMENT

In this section, having presented the underlying theory and concepts behind Design Patterns, frameworks and toolkits, we will discuss ACE[8], the object oriented framework and toolkit that we use for our implementation.

---

[8] ACE is currently being developed at the Center for Distributed Object Computing of the Department of Computer Science of Washington University in St. Louis, MO.

ACE targets the application area of communication software, implementing core concurrency and several distribution patterns. It provides an extensive set of reusable C++ wrappers and framework components, enabling the designer to implement communication software tasks on a variety of platforms.

1. Introduction

In the development of communication software and applications that must run on different OS and platforms, reuse and portability are features of special interest. Rewriting common components for different applications and adapting software to use in different environments is both error-prone and time consuming. ACE provides a collection of common components and architectural blocks, which are reused repeatedly in the domains of network and systems programming. Components of ACE can be used in the following applications [Ref. 15]:

- Concurrency and Synchronization
- Interprocess Communication
- Memory Management
- Timers
- Signals
- File System Management
- Event Demultiplexing and Handler Dispatching
- Connection Establishment and Service Initialization

- Static and Dynamic Configuration and Reconfiguration of Software
- Layered Protocol Construction and Stream-based Frameworks
- Distributed Communication Services

ACE is structured in three basic layers, each one providing particular features [Ref. 15]:

- The Operating System (OS) Adaptation Layer
- The C++ Wrapper Layer
- The Frameworks and Patterns layer

2. The OS Adaptation Layer

In order to make applications written using ACE platform independent, there exists a thin layer of code, the OS Adaptation Layer, which shields the higher layers of ACE from the underlying platform. This code lies between the native OS APIs and ACE. By setting parameters and including the appropriate header files, ACE can be built on several platforms. Using ACE, an application developer with a small amount of effort can move his application to a different supported OS and / or platform. Because of this Adaptation Layer, the ACE framework is available for many OS and platforms, both real-time and conventional, including most versions of UNIX, Win32 and MVS OpenEdition. Also, this variety of implementations of ACE provides the applications written with ACE extensive portability.

3. The C++ Wrappers Layer

The C++ wrapper classes included in this layer can be used to build portable and typesafe C++ applications. Currently there are C++ classes that provide the following functionality:

- Concurrency and Synchronization
- Inter - Process Communication (IPC)
- Memory Management
- Timer Functions
- Containers
- Signal Handling
- Filesystem Functions
- Thread Management

a) Concurrency and Synchronization

ACE provides components that wrap primitives such as Semaphores, Locks, Barriers and Condition Variables for both threads and processes [Ref. 21].

b) Inter-Process Communication (IPC)

The C++ wrapper classes provided with ACE make it easier to use different inter-process communication (IPC) mechanisms on different operating systems. ACE provides a main class, ACE_IPC_SAP, which has the common functionality of all supported IPC mechanisms. From this class four others are derived, which inherit this functionality providing specific implementations for different environments [Ref. 26]. The ACE_SOCK class, which we are using for our implementation, contains functions that are common to the BSD sockets programming interface.

c) Memory Management

ACE provides functionality for the dynamic management and inter-process sharing of memory. As real-time systems require extreme granularity in memory management, ACE provides a mechanism to pre-allocate all the dynamic memory and then manage it locally.

d) Timer Functions

ACE implements timer functions by providing classes for various timers as well as wrapper classes for the high-resolution timers available on some platforms. The high-resolution timer used for our experimentation is discussed in Section VI.

e) Containers

ACE provides a number of standard container classes, such as Map, Hash_Map, Set and List [Ref. 41]. These classes use the same interface, making them portable across many platforms.

f) Signal Handling

The wrapper classes of ACE provide a common signal handling interface for all supported OSs, with functionality such as the installation and removal of signal handlers and the installation of several handlers for one signal.

g) Filesystem

ACE provides classes which wrap the filesystem API, including file I/O, locking, streams, connection and asynchronous file I/O.

h) Thread Management

ACE contains classes that wrap the OS threading APIs providing functionality for creating and managing threads.

4. The ACE Framework Components

The ACE framework provides components based on several communication software design patterns. These components can be used in both the design and implementation of a system. The following components are included [Ref. 15]:

- Event Handling
- Connection / Service Initialization
- Stream
- Service Configuration

a) Event Handling

ACE provides functionality for the efficient de-multiplexing, dispatching and handling of events in the Reactor component [Ref. 4]. The Reactor is a design pattern that handles requests that are delivered concurrently to an application by one or more clients. Each service in the application may consist of several methods and is represented by a separate event handler responsible for dispatching service-specific requests. Dispatching of event handlers is performed by an initiation dispatcher, which manages the registered event handlers. Demultiplexing of service requests is performed by a synchronous event demultiplexer.

b) Connection or Service Initialization

In order to separate the initialization of a connection from the actual service that is to be performed by the application after the connection has been established, ACE provides the Connector and Acceptor components [Ref. 6] and service handlers. A Connector actively establishes a connection with a remote Acceptor component, which is passively waiting for connection requests from remote Connectors. Once the connection is established, they both initialize a service handler to process the data that will be exchanged from the connection. The processing specific to the application is performed by these initialized service handlers, which communicate using the connection previously established by the Connector and the Acceptor.

The Acceptor-Connector are very efficient when there are many connection requests being initialized and then handled by different handling routines. They are also used in the ORB Core layer in the ACE ORB (TAO) [Ref. 17], which is a real-time implementation of CORBA [Ref. 16] and discussed in Chapter V, to passively initialize server object implementations when clients request ORB services.

c) Streams

The Streams component is used for the development of software that is layered or hierarchic in nature, where user-level protocol stacks are composed of several interconnected layers developed independently of each other. This allows the re-use or replacement of the layers with minimal effort.

d) Service Configuration

The Service Configurator offers functionality for the dynamic manipulation of services that an application provides, either at installation or at run-time.

D. SUMMARY

In this section, we introduced the basic concepts of design patterns, frameworks and toolkits. We then briefly presented the structure and functionality of ACE, a framework and toolkit based on design patterns, which we used for our implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

V. MIDDLEWARE

For the communication between the Management System for Heterogeneous Networks (MSHN) components, the use of Commercial Off The Shelf (COTS) software, an open systems architecture and middleware was proposed with a sound reasoning in [Ref. 9]. In this chapter, we present the middleware that we chose for our implementation (CORBA) and its main characteristics and structure. Also, we present a standards-based CORBA middleware framework, *The ACE ORB (TAO)*, which we used for our implementation.

A. COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)

CORBA is an industry-wide standard for creating distributed object systems defined by the Object Management Group (OMG) [Ref. 20], a non-profit consortium of over 800 companies. The goal of this group is to provide definitions of standards for interoperable software components. CORBA specifies how software components distributed over a network can work together to perform a task without regard to the operating systems and programming languages used. OMG facilitates only the definitions of these standards, without dealing with implementation issues. Software that is produced in accordance with these standards should be interoperable with other software that follows the same standards. CORBA also specifies an extensive set of services for creating and deleting objects, accessing them by name, storing them in persistent stores, externalizing their states, and defining ad-hoc relationships between them. [Ref. 18]

We chose CORBA as superior to other middleware solutions for application in MSHN. Besides the argumentation in [Ref. 9], we present a rough comparison with alternative technologies at the end of this section. Some of the benefits CORBA offers are summarized below:

- Open standards: CORBA is based on open, published specifications and is implemented by different vendors on different hardware platforms and operating systems using various programming languages. This gives the user flexibility in choosing and upgrading a Client / Server system.
- Interoperability: CORBA objects should be fully interoperable even when they are developed by different vendors who have no previous knowledge of each other's objects because they communicate using a common protocol, the Internet Inter-ORB protocol, and an agreed common interface.
- Modularity: Every CORBA-compliant object has a well-defined interface, which it will use to communicate with other CORBA-compliant objects. Changing the implementation of an object does not require changes in other objects as long as the interface of that object remains the same.
- Coexistence with legacy systems: CORBA enables a legacy application to be encapsulated in a CORBA wrapper that defines an interface to the legacy code. This interface makes the application interoperable with other objects in the distributed environment.

- Portability: A CORBA object written on one platform can be deployed on any other platform that supports CORBA.
- Security: CORBA provides security features such as encryption, identification and authentication of the entities in the distributed system, and also controls access to objects and their published services.

1. Object Management Architecture

The OMG published the Object Management Architecture Guide (OMA Guide) in 1990. It was revised in 1992 and 1995. The OMA Guide is the highest level specification that covers all constituents of the Common Object Request Broker Architecture. The five parts of the architecture are provided in the Figure 3.

a) Object Request Broker (ORB):

A CORBA Object Request Broker (ORB) is a middleware that handles interactions between objects. A client entity can invoke a method on a server entity that can be on the same machine or across a network using the ORB. The ORB intercepts this call, locates the object that is offering the services, provides supplied parameters to methods, and returns the results to the caller. The calling object does not need to know the server object's location, the programming language the server was written in, or the operating system it is running on. The ORB separates the client and the server from the underlying communication infrastructure and the protocol stack. The protocol stack is replaceable as migration occurs from one implementation of CORBA to another. This

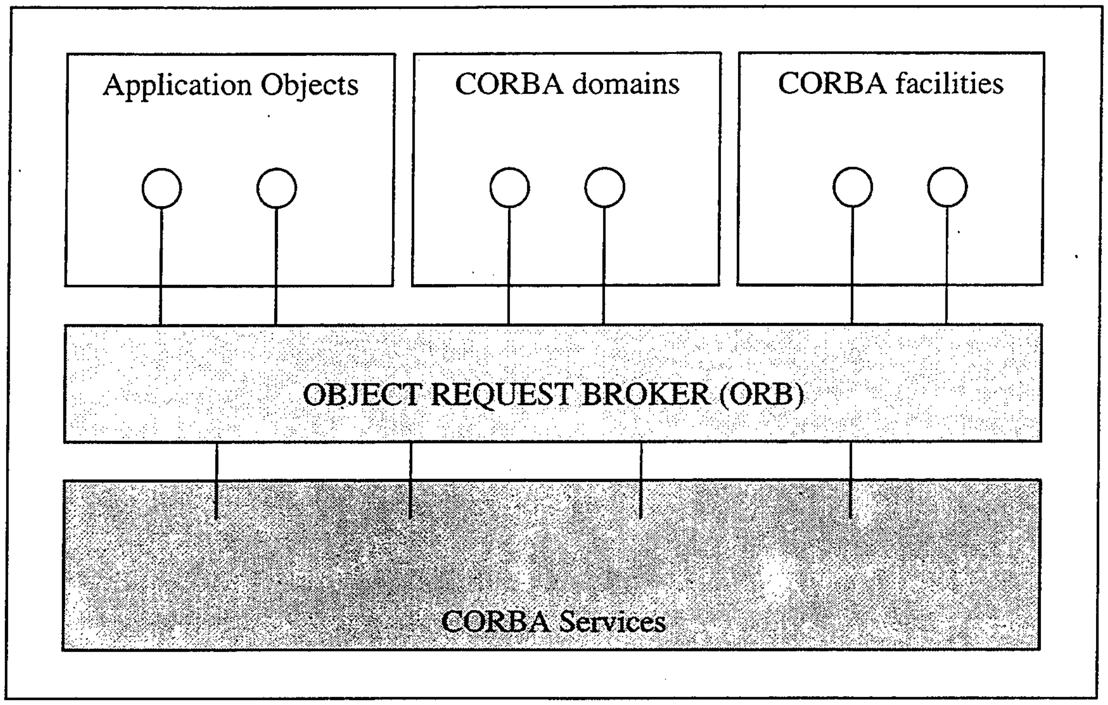

Figure 3. Object management architecture [Ref. 22]

provides flexibility for application architectures and simplifies the distributed computing model [Ref. 23]. The client and server roles are dynamic, so an object can act as a client to a published service of another object on one occasion and can itself offer services on another [Ref. 19]. Figure 4 shows the client and server interacting through the ORB.

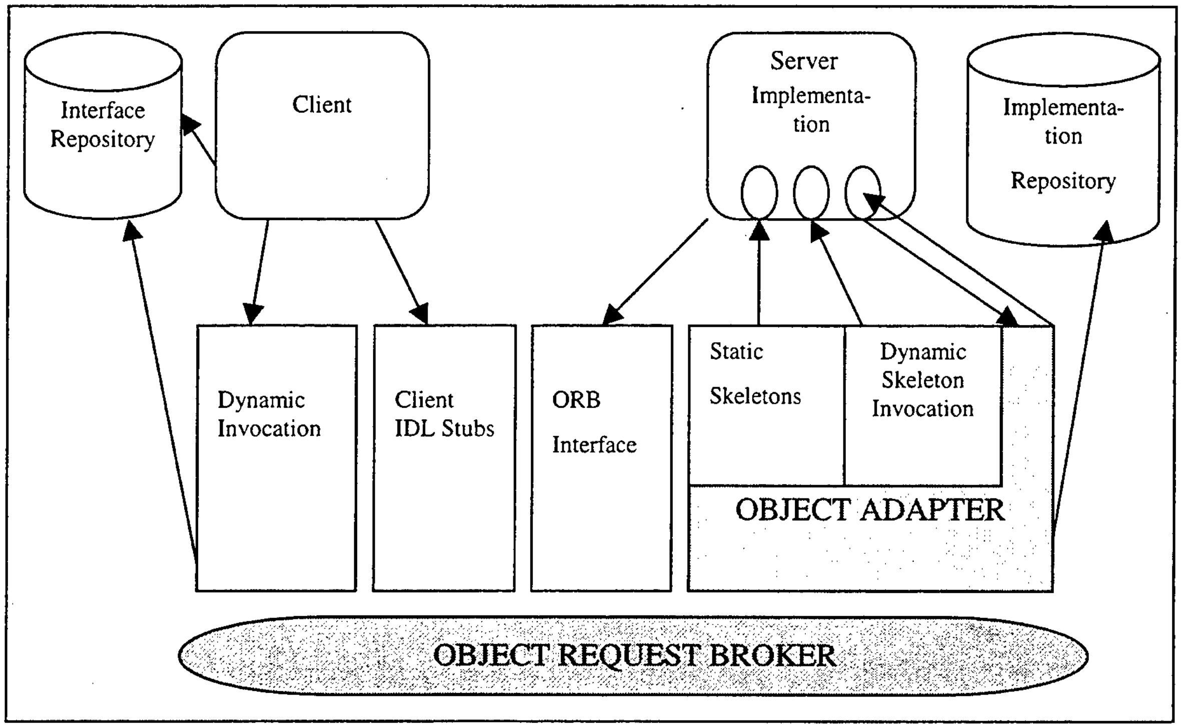

Figure 4. The structure of a CORBA 2.0 ORB [Ref. 18]

CORBA provides both static and dynamic interfaces to the client. Static interfaces are defined at compile-time and provide a robust and efficient way to publish the services provided by an object. Static accesses also provide the fastest access at run-time. Dynamic interfaces lack this robustness and speed, but enable the client to discover and use the services of server objects at run-time. The following is a brief discussion of the parts that make up the ORB.

- Client IDL Stubs: An interface for an object consists of named operations and the parameters required by those operations. All interfaces in CORBA are defined using Interface Definition Language (IDL). Client IDL stubs are generated by an IDL compiler and provide static interfaces to services provided by an object. A client must have a stub for the services it wants to use on the server. The stub performs the job of packaging parameters into a message format for transfer over the network, an operation referred to as marshalling.

- Dynamic Invocation Interface: This interface allows clients to discover and invoke services offered by an object at run-time.

- Interface Repository (IR): The IR is a database for storing persistent references to objects that are registered with the ORB. The IR contains enough information for the ORB to locate and activate implementations that correspond to an entry in this database.

- The ORB Interface: This consists of services that may be used by an application such as converting an object to a string representation.

- Object Adapter: This is a logical set of services that enable the ORB and the implementation of the server object to communicate with each other.

- Static Skeletons: Static skeletons are created by compiling the IDL definition of a server object using an IDL compiler for a specific language. Each service supported by a server has a corresponding skeleton that handles the marshalling of the parameters.

- Dynamic Skeleton Invocation: These services find the object that offers the service requested by a client by inspecting the parameters and name of the method. This provides maximum flexibility in a rapidly changing environment or an environment with different ORB implementations that have no previous knowledge of each other.

- The Implementation Repository: This is a database that can be used to keep track of the server objects and the services they offer.

b) CORBA Services

CORBA services include services to store, manage and locate objects, to enforce relationships between objects and to provide the infrastructure for building licensing and security services.

CORBA SERVICES

Naming Service	Query Service
Event Service	Licensing Service
Persistence Service	Security Service
Life Cycle Service	Time Service
Concurrency Control Service	Trader Service
Transaction Service	Collections Service
Relationship Service	

Figure 5. CORBA Services in the object management architecture

OMG has defined a set of common CORBA services as shown in Figure 5. The following is a list of some of the more important services [Ref. 27]:

- Naming Service: The Naming Service is used to associate a human-readable name with a CORBA object reference. The name of the object is bound to the object relative to a naming context, in which each name is unique. Naming service enables CORBA to find another object by resolving the provided name in a naming context.

- Event Service: This service allows objects to register and unregister their interest in specific events. A common (virtual) bus named the event channel is used to transfer messages from the generator of an event to objects that have expressed their desire (subscribed) to receive the event.

- Persistent Object Service: This service provides a set of common interfaces for storing objects in persistent storage. The storage can range in type from a text file to a Relational or Object DBMS.

- Life Cycle Service: This service includes operations for the creation, copying, moving and deletion of objects from the ORB. Factory objects, which can be used to create CORBA objects, are defined in this service.

- Concurrency Control Service: This service enables multiple clients to coordinate their access to shared resources. This is achieved by placing a lock on an object to provide atomic access.

- Transaction Service: Distributed applications need to have certain properties to function properly. The four vital properties are atomicity, consistency, isolation and durability. This service is used for the enforcement of these properties in a distributed system that uses CORBA as its architecture.

- Relationships Service: This is a general-purpose service for establishing relationships between objects. The expression of a relation in the form of an object makes abstract concepts such as entities and relationships explicitly representable in the distributed architecture. One of the relationship types between objects, for example, is containment relationship, which is represented by a relationship between the container object and the contained objects.

- Externalization Service: This service defines protocols and conventions for recording the state of an object in a stream of data that can be saved to a file or transported across the network. This process is called externalization. The externalized object can be restored by reversing the process, which is called internalization.

- Security Service: This service includes features that can be used to provide a framework for a distributed object system. The issues addressed are the identification and authentication of principals in the system, confidentiality and integrity of messages sent over the ORB, ensuring availability of resources, providing access control to objects based on the identity and privileges of the requesting object and auditing the actions of a principal.

- Trading Service: The functionality of this service is similar to a matchmaking service for objects in the system. An object that provides a service advertises itself by exporting information about the service it provides, the parameters it expects and a reference to itself that can be used by a client to invoke operations on the advertised services.

- Licensing Service: Licensing service includes interfaces to protect the intellectual property of developers. Licensing services can be used to control software licenses in a distributed system.

- Time Service: This service can be used for the synchronization of different components.

CORBA services are primitive, general and fundamental building blocks that can be used in a distributed object system. They are useful for all kinds of applications and are domain-independent in that they are intended to be reused and specialized by applications [Ref. 20]. An application developer can achieve the functionality desired in an implementation by inheriting from multiple corresponding services. Not all CORBA services are available at this time, but all ORB vendors provide a subset of existing CORBA services.

c) *CORBA Facilities*

CORBA facilities are higher-level services that are common to multiple domains and aim to establish application-level interoperability. They provide defined frameworks written in Interface Definition Language (IDL) that are of use to application objects. The common facilities that are being built by OMG members include facilities to handle user interface management, information management, systems management and task management. CORBA facilities can reuse services provided by the CORBA services or they can inherit and extend them. Figure 6 illustrates the relationship between CORBA services, CORBA facilities and CORBA domains.

d) *CORBA Domains*

CORBA domains are a business-specific standardization that considers the interoperability needs of specialized areas such as healthcare, manufacturing or telecommunications. CORBA domains do not answer the common needs of multiple domains, which are handled by CORBA facilities. CORBA domains might use or inherit from the services provided by CORBA facilities or CORBA services as needed.

e) *Application Objects*

The last category of objects in the CORBA distributed architecture is that for the application objects themselves. These objects perform specific tasks for users.

They can use or inherit from the standard interfaces provided by the OMG, including CORBA services, CORBA facilities and CORBA domains, or they can provide their own interfaces. Reuse of the services provided by the OMG enables the rapid design and employment of distributed systems and enforces some conformity to standards.

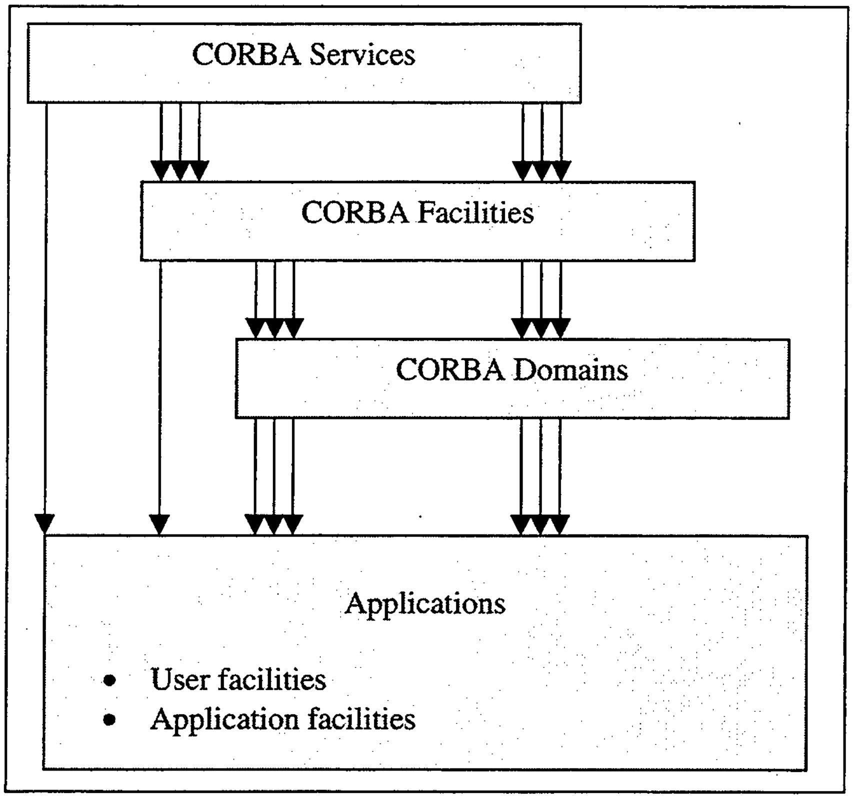

Figure 6. Reuse of OMG specifications [Ref. 20]

2. Internet Inter-ORB Protocol (IIOP)

The CORBA 2.0 specification requires ORB vendors to implement the Internet Inter-ORB (IIOP) protocol or to provide half-bridges to it. The goal of this requirement is to ensure communication between different ORB implementations. IIOP makes it possible for a client of one vendor's ORB to invoke operations transparently on an object in a different ORB. The General Inter-ORB protocol (GIOP) defines a set of message formats and common data representations for communication between different ORBs. GIOP is designed to work on any transport protocol. IIOP is a specialization of GIOP that uses TCP/IP as its transport layer.

3. Interface Definition Language

Interface Definition Language (IDL) is a standard language that defines defines interfaces used by CORBA objects. IDL is a part of the CORBA specification and is independent of any programming language. Language independence gives application developers the freedom to choose the language they want to use in the distributed object system. Developers can choose a language that provides better performance, or one in which they have a significant investment. They can even use different languages for different parts of the system. It is also possible to retain and use legacy applications by creating an IDL wrapper for them. IDL mappings exist for a number of languages including C++, Ada95 and Java.

IDL is a declarative language that is syntactically a subset of the ANSI C++ standard. IDL is considered a very useful tool for software designers, because it separates the implementation of an object from its specification. IDL is used to describe an object's attributes, the services the object provides, the classes the object inherits from, the exceptions the object raises and the name that can be used to locate the implementation in a distributed architecture. IDL compilers generate client stubs and server skeletons by processing an IDL file. These are generated in the form of header and source code files, and they form part of the actual implementation of the applications.

4. CORBA and alternatives

There are several protocols in use today, which provide alternative solutions for the communication between objects. The Common Gateway Interface (CGI) protocol has been the dominant model for Client/Server applications using the TCP/IP as a medium. CGI is a slow, stateless protocol that is not suited for distributed object applications. Also, CGI launches a new process to service each client request. Many vendors attempt to overcome the weaknesses of CGI by providing server extensions, but these extensions are non-standard and some of them are platform specific. CGI is not a long-term solution for a heterogeneous environment.

Remote Procedure Call (RPC) is a mechanism that enables programs running on one machine to make calls to functions on another machine connected to the Internet. Remote calls are blocking, which means that the calling application can not proceed until it gets the results of the remote invocation. This imposes a performance penalty. CORBA method calls can be declared as one way, thus transforming the calls into asynchronous messages. RPC is not object-oriented, so it cannot take advantage of features like encapsulation, inheritance and polymorphism.

There are other competing distributed object models that are fully object-oriented [Ref. 25]. We will mention the three most notable ones here, namely Java's Remote Method Invocation (RMI) by Sun Microsystems, Distributed Component Object Model (DCOM) by Microsoft, and Open Software Foundation's (OSF) Distributed Computing Environment (DCE). RMI does not provide language-neutral messaging services. An object written for RMI needs to be written in Java and can operate only with objects that are implemented in the same language. RMI does not support dynamic invocations and interface repositories, nor does it define the protocols for services like transactions and security. DCOM has serious limitations as well. CORBA objects have unique and persistent references and they have state, where DCOM objects do not maintain their state between connections. This creates a problem in environments such as the Internet when there are numerous faulty connections. In addition, it is very difficult to configure and run DCOM applications on non-Windows platforms. DCOM does not support a universal naming service, which severely limits scalability. DCE is designed to support distributed procedural programming, while CORBA is designed to support distributed object-oriented programming, which is the programming paradigm of our choice [Ref. 28].

The limitations of alternative approaches pinpoint CORBA as the leading tool for providing the communication mechanism for our implementation of a MSHN-like communication scheme. The ORB implementation that we used was TAO 1.0, whose features will be discussed in the next section.

B. TAO

Middleware like CORBA [Ref. 31] and DCOM [Ref. 32] can only satisfy the communication QoS requirements of best-effort applications. In addition, ORB middleware is not suitable for distributed real-time applications with high performance requirements. Generally, the conventional ORBs lack the following characteristics [Ref. 33]:

- QoS specification interfaces
- QoS enforcement
- Real-time programming features
- Performance optimizations

TAO [Ref. 34] was developed in order to address the above limitations. It is a high-performance, real-time ORB endsystem, intended for use with applications which may have both deterministic and statistical QoS requirements, as well as best effort requirements.

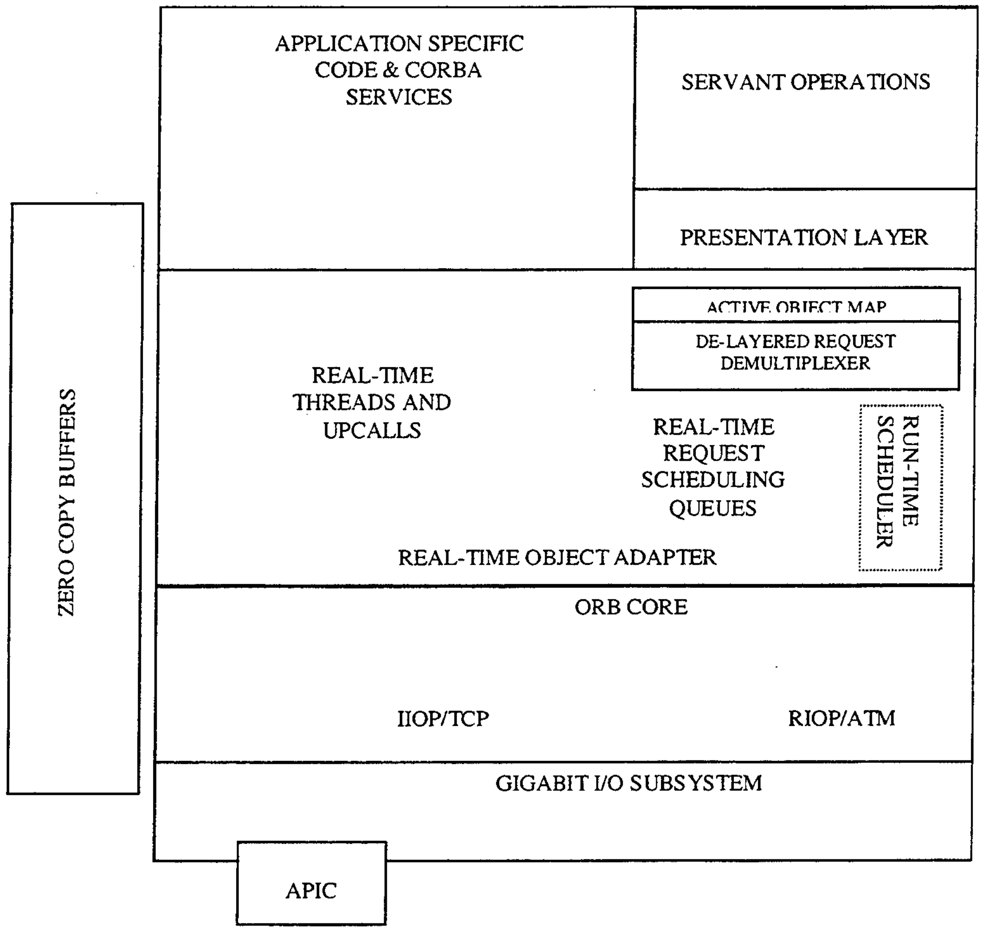

Figure 7. Architectural components of the TAO real-time ORB endsystem.

The TAO ORB endsystem contains the network interface, OS, communication protocol and features shown in Figure 7. TAO supports the standard OMG reference model defined in [Ref. 31], with the following enhancements that intend to overcome the limitations of the conventional ORBs that we mentioned above:

1. Optimized IDL Stubs and Skeletons / Presentation Layer

The conversions in the presentation layer transform application-level data into a portable format that masks the differences in byte order, alignment and word length. The IDL compiler provided with TAO generates stubs and skeletons that can selectively use highly optimized compiled and / or interpretive marshaling and demarshaling [Ref. 39]. The compiled marshaling code is efficient but requires large amounts of memory. On the contrary, the interpreted marshalling code is slower but more compact. Using TAO, the developers are able to selectively optimize for time or space.

2. Real-time Request Demultiplexing and Dispatching / Real-time Object Adapter

Conventional ORBs demultiplex and dispatch incoming client requests to the appropriate operation of a servant at multiple layers, including the network interface, the protocol stack, the user/kernel boundary and the ORB's Object Adapter. This demultiplexing and dispatching results in an increased performance overhead and a potential for priority inversion, which scale with the number of operations that appear in the IDL interface and the number of servants managed by the ORB. To overcome these limitations, TAO's real-time Object Adapter [Ref. 35] uses perfect hashing [Ref. 36] and active demultiplexing. Perfect hashing is a two-step layered demultiplexing strategy that sequentially uses an automatically-generated perfect hashing (using GNU *perf*) to locate the servant and then to locate the operation. In the worst case, these lookups require constant time. The keys to be hashed must be known in advance, a requirement which is easily fulfilled as servants and operations in real-time systems are configured statically. In the active demultiplexing strategy, the client passes a handle that directly addresses the servant and operation in O(1) time. The client obtains this handle when the servant's object reference is registered with a naming service.

3. Run-time Scheduler

The applications' QoS requirements, such as end-to-end latency, are mapped with a real-time I/O class [Ref. 37] to ORB endsystem and network resources, such as the CPU, the memory, the network connections and the storage devices. Once a thread of the real-time I/O class is admitted by the OS, the scheduler will compute the thread's priority relative to other in the class and dispatch the thread periodically so that it will meet its deadlines.

4. Admission Controller

In order to guarantee that the application meets its QoS requirements, TAO enforces admission control for the real-time scheduling class. In this way, the OS either guarantees the specified computation time or it will refuse to admit the thread.

5. Real-time ORB Core

TAO's real-time ORB Core [Ref. 34] uses a multi-threaded preemptive priority based connection and concurrency architecture for delivering client requests to the Object Adapter and for returning any responses.

6. Memory Management

Data copying consumes a significant amount of CPU, memory and I/O bus resources. Also, the dynamic memory management has a significant performance penalty caused by heap fragmentation and locking overhead. In order to minimize the data copying and dynamic memory allocation, multiple layers in the ORB endsystem must collaborate. These layers include the network adapters, the I/O subsystem protocol stacks, the Object Adapter and the presentation layer [Ref. 40]. TAO uses a zero-copy memory management mechanism, which minimizes dynamic memory allocation and data copying.

7. Real-time I/O Subsystem

TAO's Real-time I/O (RIO) subsystem extends the support for CORBA to the OS by assigning priorities to real-time I/O threads. In this way, it is possible to enforce the schedulability of the application components and the ORB endsystems resources. Although TAO runs efficiently on conventional I/O systems lacking advanced QoS features, when used with advanced hardware, such as the TAO I/O subsystem's high speed network interface, TAO is able to perform early demultiplexing of I/O events into prioritized kernel threads, thus avoiding thread-based priority inversion. TAO can also maintain distinct priority streams to avoid packet-based priority inversion as well.

8. High Speed Network Interface

The TAO I/O subsystem is designed to cooperate with a network interface consisting of one or more ATM Port Interconnect Controller (APIC) chips [Ref. 38]. The APIC is designed to sustain an aggregate bidirectional data rate of 2.4 Gbps by using zero-copy buffering optimization to avoid data copying across endsystem layers.

C. SUMMARY

In this chapter, we discussed the main characteristics and structure of CORBA, the middleware we chose for our implementation. Then we presented TAO, the real-time ORB we used for our experiments.

VI. MSHN IMPLEMENTATION USING ACE AND TAO

In order to support efficient communication between its components and to perform its operation in a heterogeneous distributed environment, MSHN uses CORBA [Ref. 9]. As MSHN's intent is to provide support for mission-critical, real-time applications over a variety of operating systems and platforms, we are investigating the adoption of ACE and TAO as candidates to provide the communication infrastructure underlying MSHN's implementation. In the previous chapters, we analyzed some of the unique features of this development toolkit. In this chapter, we present the design and implementation of an ACE and TAO based application built to emulate the interaction between the different components of MSHN. The emulation's purpose is to obtain the first order performance characteristics of a MSHN-managed system based upon TAO, and to provide a framework for further experimentation.

A. DESIGN

For our implementation, we created a typed event channel (EC), by utilizing the functionality of the Notifier object provided in the TAO ORB middleware. All the components of MSHN were implemented using objects that are both Consumers and Suppliers [Ref. 19], as those components concurrently require the functionality of both.

Once instantiated, the EC registers itself with the Naming Service (NS). If it cannot find a NS, it becomes a NS itself. The Consumers consult the NS in order to locate the EC. They then register with the EC for the events they wish to receive.

This implementation realizes a typed EC which has the benefit of avoiding the overhead associated with having to send every event to (and subsequently be received by) all consumer objects. In this way, we use point-to-point communication between the objects instead of broadcast. The EC uses the push-push model [Ref. 29]. The MSHN components push events to the EC, and the EC demultiplexes the events received and pushes them to the appropriate addressees.

For our experiment, we focused on three schemes:

1. The communication between two objects using the event channel
2. The initiation of a process under the control of MSHN
3. The migration of an application due to either a shortage of a critical resource or as the result of a failure

These schemes are illustrated in Figures 8, 9 and 10 which follow.

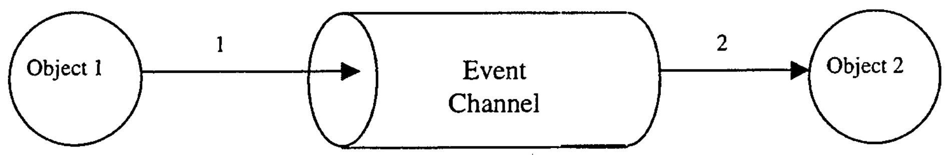

Figure 8. Communication of objects using an event channel

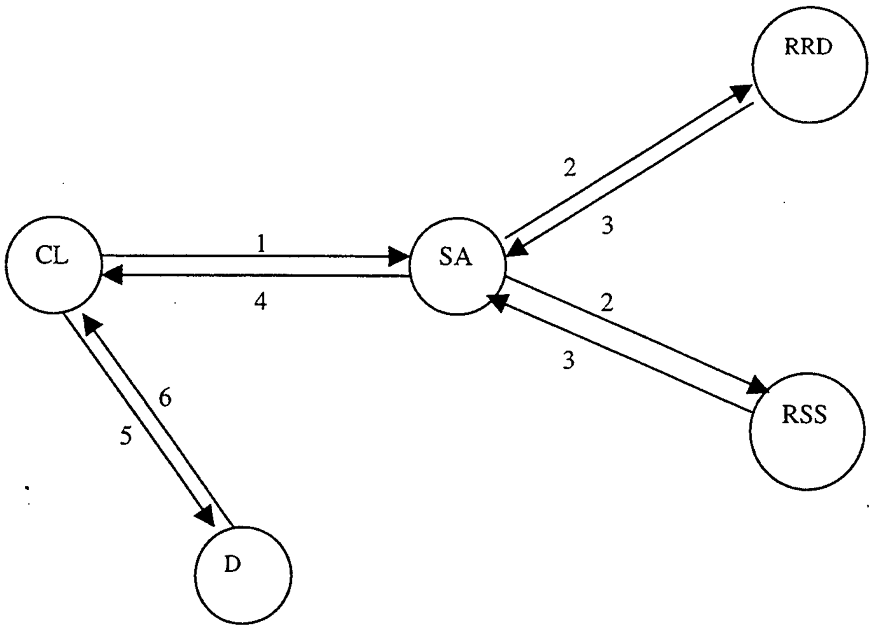

Figure 9. Initiation of a process under MSHN

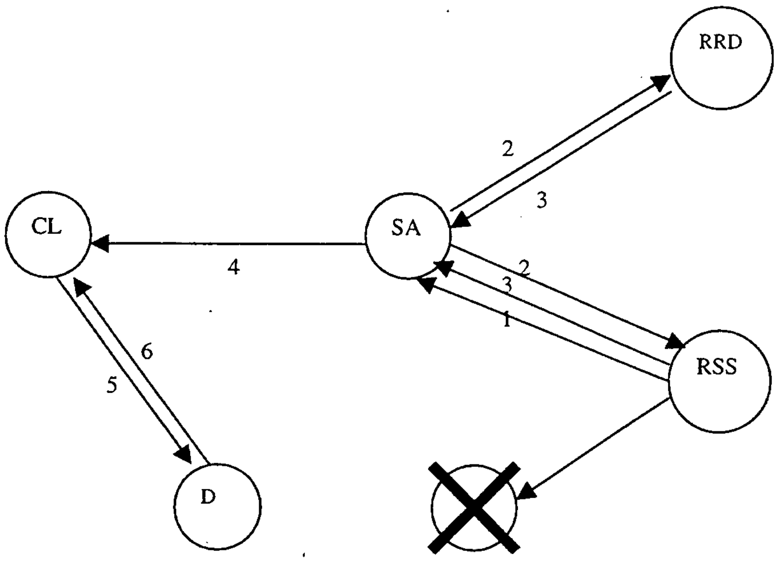

Figure 10. Migration of a process under MSHN

B. IMPLEMENTATION

For our implementation, we use the Windows NT operating system running on platforms with 400 MHz Pentium II Intel processors. The machines are equipped with 128 MB RAM, and are connected via a 100Mbps Ethernet LAN.

We compiled ACE v5.0 and TAO v1.0 using Microsoft Visual C++ v5.0. As ACE and TAO are products undergoing constant evolution and improvement, we installed several beta versions, (which were released on the average every three or four weeks during our development). Although the directions in the accompanying documentation were precise, the frequent reinstallation caused several problems and delays.

ACE and TAO have extensive documentation on the design patterns they are based on, and thorough theoretical coverage of their concepts and foundations. They come with a series of tutorials; also there are two active mailing lists which provided useful answers to all of our questions directly from the people who created ACE and TAO. As the wrapper classes and APIs are not as yet formally and completely documented, we had to overcome a very steep learning curve, which is common when working with frameworks and toolkits.

1. Basic Functionality

The emulated MSHN objects can be viewed as consumers of the EC. The role a Consumer is to play is established upon its initialization by a parameter in the command line. This parameter specifies the identity of each Consumer object (i.e., which MSHN component it is to become), and defines the filtering criterion. The filtering criterion specifies the events which an object is interested in receiving.

The EC is implemented as a Notifier object. Each **new** Consumer that registers with the Notifier is added to a hash table. A Consumer is considered a duplicate, and as such is not added to this table, under the following circumstances:

1. It has the same object reference and the same filtering criterion as a Consumer already registered with the Notifier.
2. It has the same object reference and its filtering criterion is "" (the wildcard)[9].

ACE provides a means to directly implement the filtering criterion based upon a regular expression. Since Win32 platforms do not support the REGEXP functions, such as `<compile>` and `<step>`, which ACE uses to perform its filtering, we modified the event channel to support typed events.

Each Consumer subscribes to the Notifier, passing as parameters its role and a reference pointer to itself. Its role is used as a filtering criterion by the Notifier who routes the events to the appropriate Consumer.

The EC is instantiated as an object and registers itself with the Naming Service (NS) by sending a multicast request. If the EC cannot find a NS on the host machine or on the network, it becomes a NS itself. The Consumers consult the NS to locate the EC. The

[9] When no filtering criterion is specified, the Consumer will be notified of all events.

EC's identity is hard wired into the Consumer implementation as a string, so that the Consumers can simply query the NS for a server with the identity "Notifier". Once the Consumers locate the EC, they obtain a reference to the Notifier object, and register themselves for the events they want to receive. Part of this registration is the declaration of their filtering criterion. The Consumers also have a callback functionality, which is only used here to implement a graceful shutdown, but could easily be extended at some future time to meet the full functional requirements of the MSHN components, using code already in the MSHN v2 implementation.

When the Consumers are first instantiated, the user provides a command line argument that defines the role of the Consumer object in the MSHN architecture. The components use this role identification to register with the event channel as described above. When a MSHN component sends an event to another component, it provides the event with an identification string (the `event_tag`). The EC will forward this event only to the subscribers whose roles match the tag of the event.

2. The Push-push Model

In order to implement the push-push model, the Notifier and the MSHN components provide a `push` function in their interfaces. When a component generates an event, it calls the Notifier's `push` function, "pushing" the event to the EC. The implementation of the `Notifier::push` function is provided below in Figure 11.

```
void Notifier_i::push (const Event_Comm::Event &event,
   CORBA::Environment &ACE_TRY_ENV)
   ACE_THROW_SPEC ((CORBA::SystemException))
{
   ACE_DEBUG ((LM_DEBUG,
      "in Notifier_i::send_notification = s\n",
      (const char *) event.tag_));
```

// iterator to the hash table

```
   MAP_ITERATOR mi (this->map_);
```

// counter for valid addresses

```
   int count = 0;
```

// Notify all the consumers.
// For every entry in the map ...

```
   for(MAP_ENTRY *me =0; mi.next (me) != 0; mi.advance ())
   {
      Event_Comm::Consumer_ptr consumer_ref = me->int_id_->consumer ();

      ACE_ASSERT (consumer_ref != 0);

      #if defined (ACE_HAS_REGEX)

         char *regexp = ACE_const_cast (char *, me->int_id_->regexp ());

      ACE_ASSERT (regexp);

      const char *criteria = me->int_id_->criteria ();

      ACE_ASSERT (criteria);
```

// Do a regular expression comparison to determine matching.

```
      if (ACE_OS::strcmp ("", criteria) == 0 )
```
// Everything matches the wildcard.
```
         || ACE_OS::step (event.tag_, regexp) != 0)

      #endif // #if defined (ACE_HAS_REGEX)
```

// if ACE_HAS_REGEX has not been defined, go through the switch / case.
```
      {
         ACE_DEBUG ((LM_DEBUG,
            "string %s matched regexp \"%s\" for client %x\n",
            const char *) event.tag_, me->int_id_->criteria (),
            consumer_ref));

         const char *criteria = me->int_id_->criteria ();

         char *cr = (char *)criteria;
         char *et = (char *)(const char *)event.tag_;
```

//* ... if the destination (event flag) matches the current ...

Figure 11. Implementation of the `Notifier::push` Function (part 1 of 2)

```
         int flag = 0;
            for(int ix=0;ix<1;ix++)
            {
               if((int)cr[ix]==(int)et[ix])
               flag = 1;
            }

            if(flag)
            {
               cout << "message for " << cr << endl;
```

// ... send the event to the current customer entry in the map

```
            ACE_TRY
            {
               consumer_ref->push (event,ACE_TRY_ENV);
               ACE_TRY_CHECK;
            }
            ACE_CATCHANY
            {
               ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,
                  "Unexpected exception\n");
               continue;
            }
            ACE_ENDTRY;
            count++;
         }
         flag = 0;
      }
   }

   if (count == 1)
      ACE_DEBUG ((LM_DEBUG,
         "there was 1 consumer\n"));
   else
      ACE_DEBUG ((LM_DEBUG,
         "there were %d consumers\n",count));
}
```

Figure 11. Implementation of the `Notifier::push` Function (part 2 of 2)

Inside the `push` function, the Notifier iterates through the objects that have subscribed for events, comparing the destination of the event with the filtering criteria of the subscribed objects. When they match, the Notifier pushes the event to this object by calling the Consumer's `push` function. This is illustrated in Figure 12.

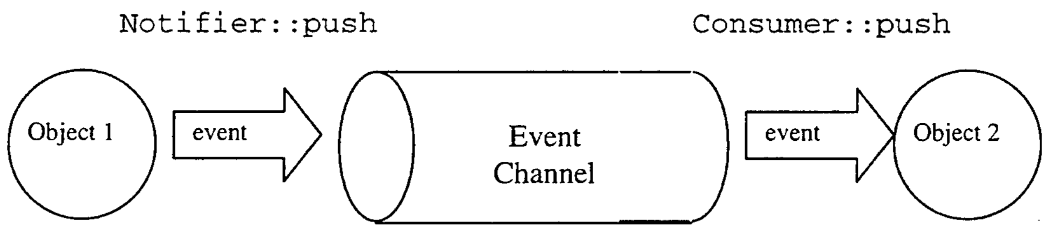

Figure 12. The Push-push Model

The Consumer's push function processes the incoming event. We extended the functionality of the Consumer's `push` function, giving the implementation of the MSHN components a polymorphic nature. Thus, we used the filtering criterion, which is passed as command-line parameter in the instantiation of a MSHN component, to determine the "type" of events received by each object. Using this role, the Consumer's `push` function performs a `switch / case` operation based on the identity of the event, actuating the part of the `push` function relevant to the role of the component. Once the role is selected, the event is further demultiplexed using the current state of the object as an entry to another `switch / case` operation. In this way, the object's operation depends on both the incoming event and the object's current state. By using the same class to instantiate any of the MSHN components, we provide a flexible implementation.

```
void
Consumer_i::push (const Event_Comm::Event &event,
   CORBA::Environment &)
ACE_THROW_SPEC ((CORBA::SystemException))
{
```

Figure 13. Implementation of the `Consumer::push` function (part 1 of 5)

```
static int state1;
const char *tmpstr = event.tag_;
ACE_DEBUG ((LM_DEBUG,
   "**** got notification = %s\n",
   tmpstr));

static ACE_Profile_Timer ptimer;
static ACE_Profile_Timer::ACE_Elapsed_Time eltime;
static double time = 0;

ACE_TRY_NEW_ENV
{
```

// consumer received event and processes it

```
char *et = (char *)(const char *)event.tag_;

switch (et[0])                                // what is our role?
{
   case '1' :                                 // scheduling advisor
      switch (state1)                      // what is our state?
      {
         case 0 :                          // idle
            state1++;  // received SA<-CL. sending SA->RSS
            {
               Event_Comm::Event event2;
               event2.tag_ = (const char *)"2";
```

// ... send the event to the current customer entry in the map

```
               this->notifier_i->push (event2, ACE_TRY_ENV);
            }
            {
               Event_Comm::Event event2;
               event2.tag_ = (const char *)"3";
```

// ... send the event to the current customer entry in the map

```
               this->notifier_i->push (event2, ACE_TRY_ENV);
            }
         break;

         case 1 :                          // waiting reply from RSS
            state1++;  // received SA<-RSS. Waiting reply from RRD
         break;

         case 2 :                          // waiting reply from RSS
            state1 = 0;  // received SA<-RRD. sending SA->CL
            {
               Event_Comm::Event event2;
               event2.tag_ = (const char *)"4";
```

// ... send the event to the current customer entry in the map

```
               this->notifier_i->push (event2, ACE_TRY_ENV);
            }
         break;
      }
   break;
```

Figure 13. Implementation of the `Consumer::push` function (part 2 of 5)

```
        case '2' :                              // RSS
           switch (state1)                      // what is out state?
           {
              case 0 :                          // idle
                 state1++;  // received RSS<-SA. querying DB
                 state1--;  // sending responce RSS->SA
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"1";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
           }
        break;

        case '3' :                                 // RRD
           switch (state1)                      // what is out state?
           {
              case 0 :                          // idle
                 state1++;  // received RRD<-SA. querying DB
                 state1--;  // sending responce RRD->SA
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"1";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
           }
        break;

        case '4' :                                 // CL
           switch (state1)                      // what is out state?
           {
              case 0 :                          // idle
                 ptimer.start ();
                 state1++;  // sending CL->SA.
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"1";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
              case 1 :                  // waiting schedule from SA
                 state1++;  // receiving CL<-SA.
                 state1++;  // sending CL->D
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"5";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
```

Figure 13. Implementation of the `Consumer::push` function (part 3 of 5)

```
        case 3 :                      // the job is running!
           state1=0;  // receiving CL<-D
           ptimer.stop ();
           ptimer.elapsed_time (eltime);
           time = eltime.real_time;
           ACE_DEBUG ((LM_DEBUG,
              "Latency is %.0f usec\n",time * 1e6));
        break;
     }
  break;
  case '5' :                              // MSHN Daemon
     switch (state1)                  // what is out state?
     {
        case 0 :                      // idle
           state1++;  // received D<-CL. Executing application
           state1--;  // sending notification D->CL
           {
              Event_Comm::Event event2;
              event2.tag_ = (const char *)"4";
// ... send the event to the current customer entry in the map
              this->notifier_i->push (event2, ACE_TRY_ENV);
           }
        break;
     }
  break;

// Additional functionality for Object to Obect communication  overhead measurement.

  case '6' :                              // Object_1
     switch (state1)                  // what is out state?
     {
        case 0 :                      // idle
           ptimer.start ();
           state1++;
           cout << "NEW state1 = " << state1 << endl;
           cout << "sending Object_1->Object_2. " << endl;
           {
              Event_Comm::Event event2;
              event2.tag_ = (const char *)"7";
// ... send the event to the current customer entry in the map
              this->notifier_i->push (event2, ACE_TRY_ENV);
           }
        break;

        case 1 :              // waiting responce from Object_2
           state1--;  // receiving Object_1<-Object_2
           ptimer.stop ();
           ptimer.elapsed_time (eltime);
           time = eltime.real_time;
           ACE_DEBUG ((LM_DEBUG,
              "Latency is %.0f usec\n",time * 1e6));
        break;
     }
  break;
```

Figure 13. Implementation of the `Consumer::push` function (part 4 of 5)

```
// Additional functionality for Object to Obect communication  overhead measurement.
// Object_2.

      case '7' :                                    // Object_2
         switch (state1)                         // what is out state?
         {
            case 0 :                             // idle
               state1++;  // received Object_2 <- Object_1
                          // Executing application
               state1--;  // sending Object_2 -> Object_1
               {
                  Event_Comm::Event event2;
                  event2.tag_ = (const char *)"6";
// ... send the event to the current customer entry in the map
                  this->notifier_i->push (event2, ACE_TRY_ENV);
               }
            break;
            }
         break;

         default:
         cout << "Error! default found. "  << endl;
      }

      ACE_TRY_CHECK;
   }
   ACE_CATCHANY
   {
      ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION, "Unexpected exception\n");
   }
   ACE_ENDTRY;
}
```

Figure 13. Implementation of the `Consumer::push` function (part 5 of 5)

3. The Finite State Machine Approach

For this thesis' emulation of the functionality of MSHN, we utilized a Finite State Machine (FSM) approach [Ref. 30]. Each Consumer object keeps a persistent internal state for the duration of its life, which changes according to the object's previous state and the event that the object receives. The sequence of states and the actions of every

Consumer for a given state depends on the role that the Consumer has in the MSHN environment.

The FSM discussed below reflects our experimental setup. It does not implement the full functionality of the MSHN components, such as the issuing and handling of callbacks and the concurrent handling of multiple events in different states. This would complicate the experiment and would add functionality irrelevant to our measurements. We anticipate documenting the entire functionality of MSHN using a FSM in future work.

In the FSM model of the MSHN components, each component assumes an initial idle state indicated by "0". Next, we discuss the FSM of each of the MSHN components.

a) The Scheduling Advisor

The SA changes state in our model upon receiving a request (+req) for a schedule. This request can be issued either from the CL, for a new process, or from the RSS, due to either the unavailability of a resource or a failure. Next, the SA performs a query to the RSS (-rss) and the RRD (-rrd). In the current model, the order in which the requests are issued is irrelevant and modeled by showing both possible sequences. In future implementations, where the actual performance of the RRD and RSS will be known, the sequence of the queries can be a factor for optimizing performance. In any case, the SA will be, after the queries, in state 4 awaiting a response. We model the arrival of the data from the RRD (+rrd) and RSS (+rss) as randomly ordered events.

Being in state 7, the SA normally calculates the schedule and sends it to the CL. If there are pending requests, the SA returns to state 1, otherwise it returns to idle (the default in our implementation).

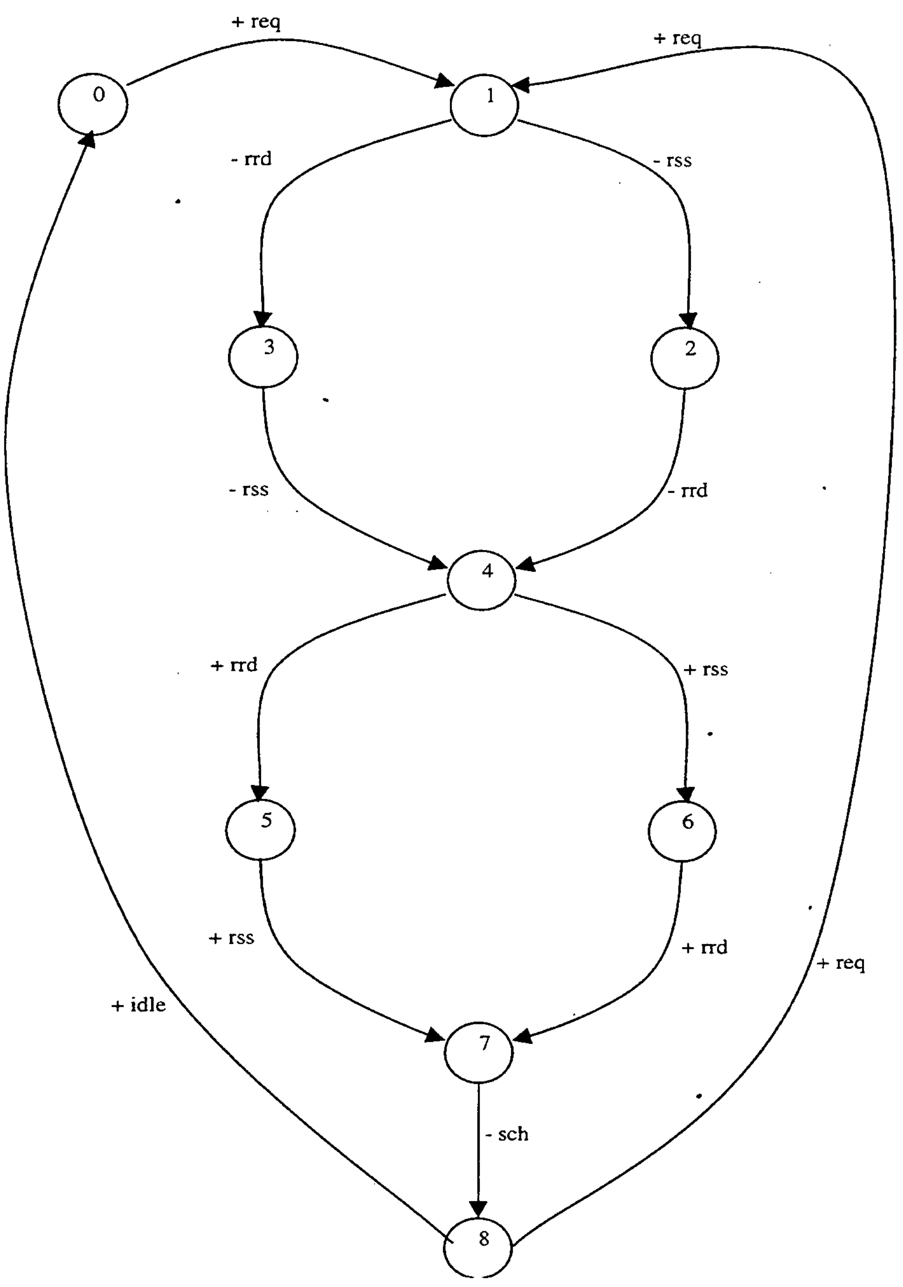

Figure 14. Scheduling Advisor FSM

b) The Resource Requirements Database and Resource Status Server FSM

In our model, the RSS and the RRD have identical functionality, and are modeled identically. They remain at state 0 when idle or when performing an update. When a request is received (+query), they go to state 1, where they would actually perform the query, and then return to state 0 while sending an event (-response) back to the SA. This is illustrated in Figure 15.

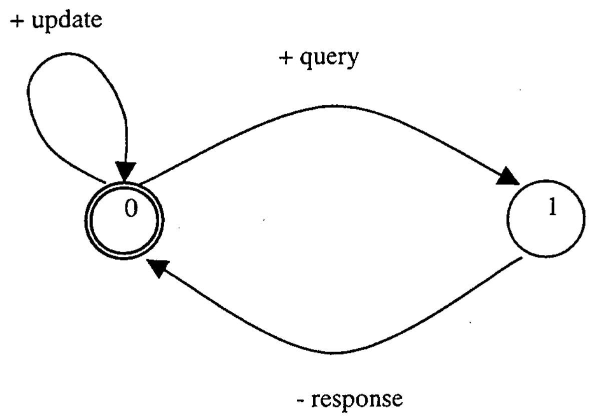

Figure 15. The Resource Requirements Database and Resource Status Server FSM

c) The Client Library FSM

The CL will assume state 1 when the application it wraps causes a request (-req) to be sent to the SA. Upon receiving the schedule (+sch), the CL will move to state 2. In state 2, a request is sent to a (usually) remote daemon to execute the job (-

exec_req). In state 3, and upon reception of the results of its request (+req_res), the CL assumes the idle state (0).

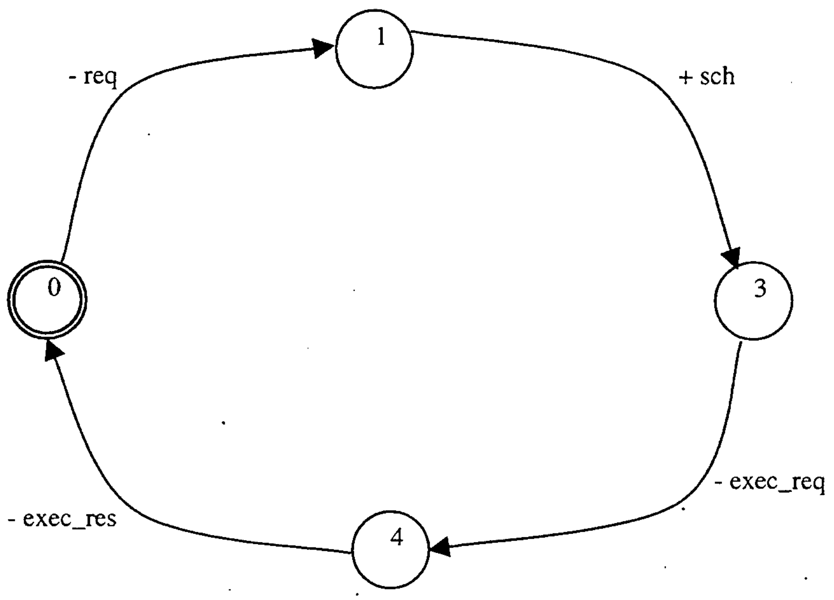

Figure 16. The Client Library FSM

d) The MSHN Daemon FSM

The MSHN Daemon assumes state 1 upon the reception of a request to execute a job (+exec_req). In this state, the Daemon performs its normal functionality (- exec_app), and then transitions to state 2. Then the Daemon returns to state 0, sending an event to the CL with information about the job (- exec_res).

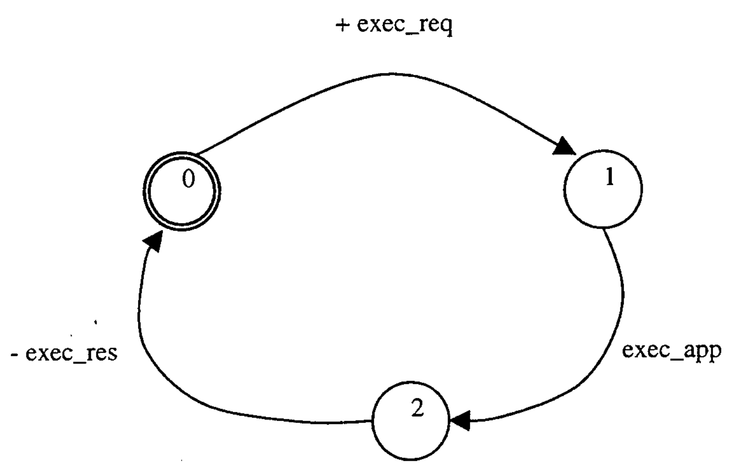

Figure 17. The MSHN Daemon FSM

C. PERFORMANCE MEASUREMENTS

1. Measurement Methodology

In our measurements, we did not take into account the internal delays in the components. We measured the pure communication overhead imposed by TAO, ACE and the operating system under various configurations and functions. As MSHN is still evolving and the implementation of the actual components' functionality is still under development, the actual component delays are not completely known. Nevertheless, the finite state machine approach we used allows the eventual insertion of a function call that can invoke an emulator designed to impose a predetermined overhead.

As MSHN only uses TAO and ACE as a communication infrastructure, our interest is in determining the communication overhead of TAO. By measuring only the communication overhead of the MSHN emulation, we are able to obtain data about the minimum value of this overhead and the response time of a real-time MSHN implementation on top of TAO. This provides a lower bound for the delay introduced by the execution of an application via MSHN on top of TAO. Such a delay affects many of the QoS parameters that MSHN may provide to its users.

2. Experimental Setup

In making our measurements, we studied four configurations:

1. Two Consumer objects on the same platform (Figure 18).
2. One Consumer on one platform, and the EC and the other Consumer on another (Figure 19).
3. Initiation of a new job in a MSHN-like environment where all the components of MSHN are on the same platform as in Figure 9.
4. Initiation of a new job in a MSHN-like environment where the SA, RSS and RRD are on one platform, and the initiating CL and the MSHN Daemon are on two others (Figure 20).

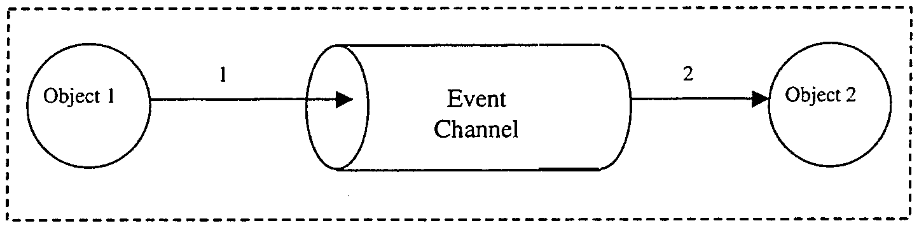

Figure 18. Communication between objects on the same platform

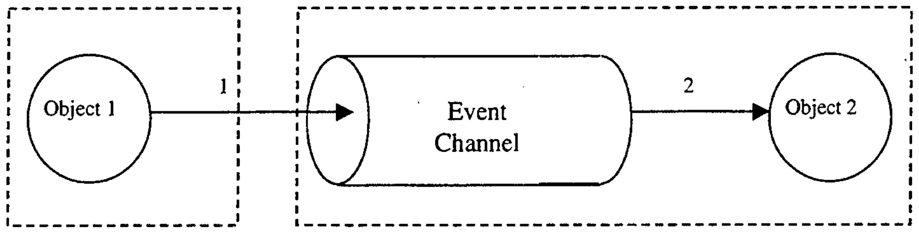

Figure 19. Communication between objects on different platforms

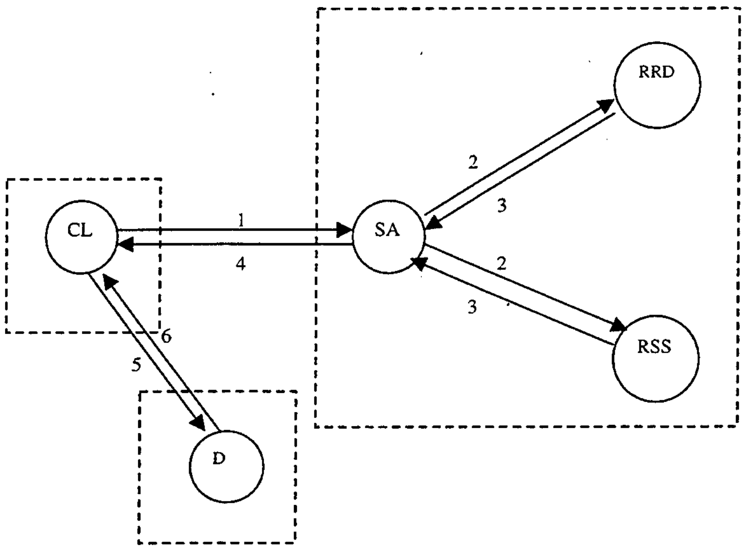

Figure 20. Initiation of a new job in a MSHN-like environment where the SA, RSS and RRD are on one platform, and the initiating CL and the MSHN Daemon are on two others.

D. MEASUREMENTS

For Latency, we measured the difference between the time a job is first submitted and the time that the job would actually start executing on a local or remote platform. This is the delay that the user experiences due to the intervention of MSHN in the execution of his application.

For the Agility, we measured the difference between the time that the RSS is first notified by the CL wrapping the process of a shortage of a resource, and the time the process restarted execution on another platform.

We found out that for this particular implementation, where no delays in the components were involved, the experimental setup for Agility resulted in the same number of events sent between the components, giving us the same communication overhead as Latency.

To measure the overhead for a (two-way) exchange of events between two objects using ACE and TAO, the objects were first located on the same and then on different platforms. One set of experiments used the same EC for all measurements, while another used a new EC for every measurement.

Platform	Same		Different	
Event Channel	Same	Different	Same	Different
Average	36.9	43	44.2	47.2
Standard deviation	2.6	1.9	6.5	2.7
Maximum	44.5	46.4	49.8	54.5
Minimum	34.6	40.8	28.3	44.0

Table 1. Latency for the two-way communication between objects (msec)

For the first Latency / Agility experiments, we implemented the MSHN communication scheme in only one platform, thus having no network overhead but with the cost of using the same OS for all objects and event routing. For the next set of experiments, we performed our measurements where the CL, the MSHN Daemon and the MSHN core (SA, RSS and RRD) were distributed across three different platforms. The results are shown in the following Table 2.

Platform	Same	Different
Average	154.7	132.3
Standard deviation	48.0	332.3
Maximum	281.9	109.1
Minimum	119.9	40.2

Table 2. Communication latency for initiation of a new process using MSHN (msec)

E. RESULTS

In order to be able to support real-time applications, MSHN must perform its functions in a finite, preferably short, amount of time. Two fundamental functions that are expected to be frequently performed are **the initiation (initial execution) of a job** and **the migration of a process**. The performance of MSHN during the execution of these functions is of great significance, as it is one of the main factors determining the suitability of this system for managing real-time applications.

1. Initiation of a Job

The functions of the various components are part of the system's performance equation, which determines if MSHN at any given time is able to provide the required latency (which is an aspect of QoS) to a new application. The delay caused by the SA can vary depending upon the granularity and the accuracy of its scheduling, as well as upon its use of priorities when servicing the various applications. It is possible that this variability can be used to alter the operational characteristics of MSHN in such a way that the QoS requirements of a new application are satisfied. For example, a job might be scheduled by the SA with less granularity, in order to accommodate excessive communication delays or an anticipated the response from the RSS.

The initial execution time (initial latency) T_{IL} can be expressed by the following equation:

$$T_{IL} = T_{CL1} + 2EC_{CLSA} + T_{SA} + T_{CL1}' + EC_{CLd} + T_d + T_{JL} \qquad (1)$$

where:

T_{CL1} = Delay of the Client Library on platform 1 from the time of submission

		of the new job until the request for scheduling is sent to the SA
EC_{CLSA}	=	Latency of the event channel between CL1 and SA
T_{SA}	=	Latency resulting from the Scheduling Advisor's processing
T_{CL1}'	=	Delay of the Client Library on platform 1 from the time that the response is received from the SA until the request for remote execution is sent to the MSHN Daemon on platform 2
EC_{CLd}	=	Latency of the event channel between CL1 and the daemon on platform 2
T_d	=	Delay of the daemon on platform 2 from the time of submission of the request for the execution of the job from CL1 until the initiation of the new process on platform 2
T_{JL}	=	Delay on platform 2 from the time of the initiation of the new process from the MSHN Daemon on this platform, until the job is actually started

Further, the delay in the scheduling advisor T_{SA} can be further decomposed in the actual processing time of the scheduling advisor (T_{SA}'), the delay (back and forth) in the event channels between the SA and the RRD and RSS (noted as T_{EC}), as well the time required for the responses to the queries to the RRD (T_{RRD}) and RSS (T_{RSS}).

$$T_{SA} = T_{SA}' + 2T_{EC} + T_{RRD} + T_{RSS} \qquad (2)$$

In our experiments, we measured the total communication overhead (L) for this operation.

$$L = 2EC_{CLSA}+T_{SA}+T_{CL1}'+EC_{CLd}+ T_{EC} \qquad (3)$$

The values we obtained for L provide an idea of the minimum delay associated with the execution of our application due to the intervention of MSHN. In order to obtain the overall delay, we add the above values for the internal delay of the components.

2. Migration of a Process

The time required to migrate a process (T_M) is the interval starting from the time that the amount of the available resources drop below a predetermined threshold until the application is executing on a new platform. This process is shown in Figure 18.

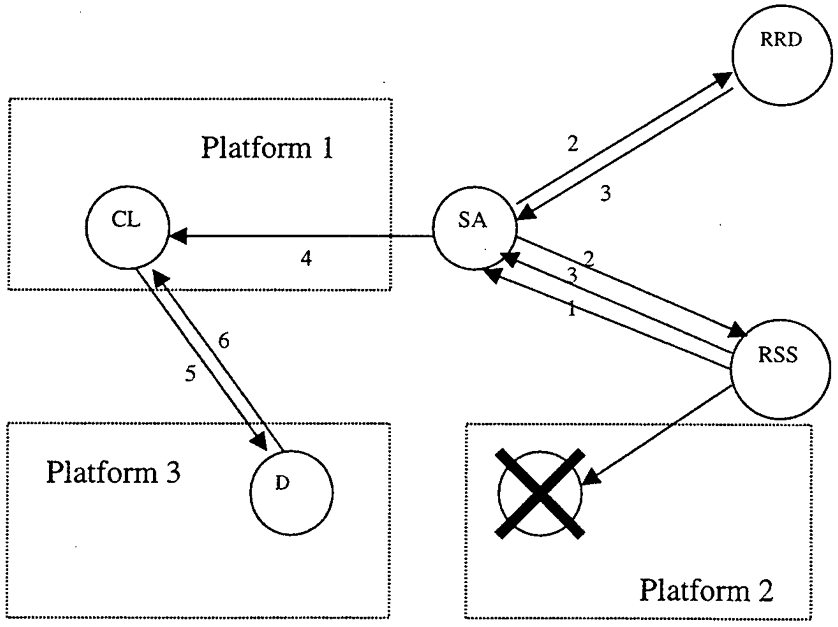

Figure 18. Migration of a Process

$$T_M = T_{CL2} + EC_{CL2\text{-}RSS} + T_{RSS} + EC_{RSS\text{-}SA} + T_{SA} + EC_{SA\text{-}d1} + T_{d1} + EC_{d1\text{-}d3} + T_{d3} + T_p \quad (4)$$

where

T_{CL2}	=	the detection / sampling time of the CL of the current platform
$EC_{CL2\text{-}RSS}$	=	the event channel latency for the communication between the CL2 and the RSS
T_{RSS}	=	the response time of the RSS
$EC_{RSS\text{-}SA}$	=	the Event Channel latency for the communication between the RSS and the SA
T_{SA}	=	the delay of the Scheduling Advisor as specified in Equation 2.
$EC_{SA\text{-}d1}$	=	the event channel latency between the SA and the daemon in platform 1
T_{d1}	=	the latency of the daemon d1 in the target platform 1
$EC_{d1\text{-}d3}$	=	the event channel latency between the daemons in platforms 1 and 3
T_{d3}	=	the latency of the daemon d3 in the target platform 3
T_p	=	the duration from when the daemon d3 first initiates the process until the start of the execution of the job on that platform.

As we mentioned earlier in Section VII - B, the number of events exchanged (and consequently the measured delay) for both the initiation and migration of a job are the same in our experiments.

VII. SUMMARY, RESULTS AND FUTURE WORK

A. SUMMARY

In this thesis, we examined the aspects and features of a real-time environment, and in particular, how these would interact with MSHN. We also discussed and proposed the control MSHN should have over the applications and the resources they use, in order for MSHN to be able to more effectively support real-time applications.

Next we presented some of the concepts underlying the use of design patterns, frameworks and toolkits, in particular, their implementation in ACE, a freely available, open source, object-oriented framework for the development of concurrent communication software. We also presented CORBA, and particularly the features of TAO, a real-time implementation of CORBA based on ACE.

Finally, we implemented a MSHN-like communication schema using ACE and TAO. We defined and measured two types of communication overhead between the MSHN components, the **latency** and the **agility**. **Latency** is defined as the delay due to communication for a given MSHN operation. The term **agility** refers to the reaction time of MSHN to some event. For example, MSHN's agility could be measured given a dramatic change in the availability of the resources that an application needs, and resulting in process migration or termination. In this case, agility is a measure of the time that a service (that this application offers) will be unavailable due to the application being migrated.

B. APPLICABILITY OF MSHN FOR REAL-TIME SYSTEMS

In our experiments, we found the latency and agility built of MSHN on top of the current TAO implementation to have an average value of 132 msec. Although this value includes only the communication overhead between MSHN components, optimization in an actual implementation should trade-off with and compensate for the time taken for their actual execution.

We found that there exist several real-time systems that can tolerate the latency and agility introduced by MSHN. Automotive control is one such system, where a rapid throttle transient is on the order of a second [Ref. 52]. The processing of a track across radar frames in [Ref. 53] is another such system, where frames are processed starting at one every fifteen seconds, and latency deadline requirement is in the order of 3 seconds. Our research also identified real-time systems whose latency and agility requirements were much more stringent than that possible using MSHN on top of TAO, e.g. 12.5 msec in [Ref. 54] and 33 msec in [Ref. 55]. Even for these systems, MSHN has a potential role if their applications were to be MSHN aware.

C. FUTURE WORK

In our experiments, the current FSM implementation is limited to handling only one job at a time; we have not provided support for multiple states in each component. In order to embody the full functionality of the MSHN components, we will expand in our future work the capability of the components to concurrently handle multiple states and applications. After this is achieved, we plan to measure the scalability and throughput of such an implementation of MSHN, as well as the performance of the event channel under heavy load. Other aspects worth further elaboration are the utilization of multiple event channels for higher performance and redundancy, and on the activation of event channels and components on demand.

APPENDIX A. THE EVENT_COMM IDL

```
/* -*- C++ -*- */
// Event_Comm.idl,v 1.6 1999/07/19 16:20:08 pradeep Exp
//==========================================================================
//
// = LIBRARY
//    EventComm
//
// = FILENAME
//    Event_Comm.idl
//
// = DESCRIPTION
//    The CORBA IDL module for distributed event notification.
//
// = AUTHOR
//    Douglas C. Schmidt (schmidt@cs.wustl.edu) and
//    Pradeep Gore (pradeep@cs.wustl.edu)
//
//==========================================================================

#if !defined (_EVENT_COMM_IDL)
#define _EVENT_COMM_IDL

module Event_Comm
{
  // = TITLE
  //   The CORBA IDL module for distributed event notification.

  struct Event
  {
    // = TITLE
    //   Defines the interface for an event <Event>.
    //
    // = DESCRIPTION
    //   This is the type passed by the Notifier to the Consumer.
    //   Since it contains an <any>, it can hold any type.  Naturally,
    //   the consumer must understand how to interpret this!

    string tag_;
    // Tag for the event.  This is used by the <Notifier> to compare
    // with the <Consumer>s' filtering criteria.

    any value_;
    // An event can contain anything.

    Object object_ref_;
    // Object reference for callbacks.
  };
```

```
interface Consumer
{
  // = TITLE
  //   Defines the interface for a <Consumer> of events.

  void push (in Event event);
  // Inform the <Consumer> that <event> has occurred.

  void disconnect (in string reason);
  // Disconnect the <Consumer> from the <Notifier>,
  // giving it the <reason>.

};

interface Notifier
{
  // = TITLE
  //   Defines the interface for a <Notifier> of events.

  exception CannotSubscribe
  {
    // = TITLE
    //   This exception in thrown when a <subscribe> fails.

    string reason_;
  };

  exception CannotUnsubscribe
  {
    // = TITLE
    //   This exception in thrown when a <unsubscribe> fails.

    string reason_;
  };

  // = The following operations are intended for Suppliers.

  void disconnect (in string reason);
  // Disconnect all the receivers, giving them the <reason>.

  void push (in Event event);
  // Send the <event> to all the consumers who have subscribed and
  // who match the filtering criteria.

  // = The following operations are intended for Consumers.

  void subscribe (in Consumer Consumer,
     in string filtering_criteria) raises (CannotSubscribe);
  // Subscribe the <Consumer> to receive events that match the
  // regular expresssion <filtering_criteria> applied by the
  // <Notifier>.  If <filtering_criteria> is "" then all events are
  // matched.

  void unsubscribe (in Consumer Consumer,
     in string filtering_criteria) raises (CannotUnsubscribe);
  // Unsubscribe the <Consumer> that matches the filtering criteria.
  // If <filtering_criteria> is "" then all <Consumers> with the
```

```
        // matching object references are removed.
    };
};

#endif /* _EVENT_COMM_IDL */
```

APPENDIX B. THE EVENT CHANNEL IMPLEMENTATION

```
// Original by Douglas C. Schmidt
// Modified and extended to support MSHN functionality
// by Panagiotis Papadatos

Notifier_i::Notifier_i (size_t size)
  : map_ (size)
{
// if platforms (such as win32) do not support the REGEXP functions
// such as <compile> and <step> then warn the user that the regular
// expression feature is not available.
#ifndef ACE_HAS_REGEX
  ACE_DEBUG ((LM_DEBUG, "\n WARNING: This platform does not support \
the functions for regular expressions.\n\
The filtering criteria will not work.\n"));
#endif //#ifndef ACE_HAS_REGEX
}

// Add a new consumer to the table, being careful to check for
// duplicate entries.  A consumer is considered a duplicate under the
// following circumstances:
//
//   1. It has the same object reference and the same filtering
//      criteria.
//   2. It has the same object reference and its filtering criteria is
//      "" (the wild card).

void
Notifier_i::subscribe (Event_Comm::Consumer_ptr consumer_ref,
             const char *filtering_criteria,
             CORBA::Environment &ACE_TRY_ENV)
  ACE_THROW_SPEC ((
                   CORBA::SystemException,
                   Event_Comm::Notifier::CannotSubscribe
                   ))
{
  ACE_DEBUG ((LM_DEBUG,
     "in Notifier_i::subscribe for %x with filtering criteria \"%s\"\n",
     consumer_ref,
     filtering_criteria));

  MAP_ITERATOR mi (this->map_);

  // Try to locate an entry checking if the object references are
  // equivalent.  If we do not find the entry, or if the filtering
  // criteria is different that is good news since we currently do not
  // allow duplicates...  @@ Should duplicates be allowed?

  for (MAP_ENTRY *me = 0; mi.next (me) != 0; mi.advance ())
    {
     Consumer_Entry *nr_entry = me->int_id_;

     // The <_is_equivalent> function checks if objects are the same.
```

```
      // NOTE: this call might not behave well on other ORBs since
      // <_is_equivalent> isn't guaranteed to differentiate object
      // references.

      // Check for a duplicate entry.
      if (consumer_ref->_is_equivalent (me->ext_id_)
         && (ACE_OS::strcmp (filtering_criteria,"") == 0
         || ACE_OS::strcmp (filtering_criteria,
             nr_entry->criteria ()) == 0))
      {
      // Inform the caller that the <Event_Comm::Consumer> * is
      // already being used.

     ACE_THROW (Event_Comm::Notifier::CannotSubscribe ("Duplicate
consumer and filtering criteria found.\n"));
     }//end if
  }//end for

  // If we get this far then we didn't find a duplicate, so add the
  // new entry!
  Consumer_Entry *nr_entry;
  ACE_NEW (nr_entry,
           Consumer_Entry (consumer_ref,
            filtering_criteria));

  // Try to add new <Consumer_Entry> to the map.
  if (this->map_.bind (nr_entry->consumer(), nr_entry) == -1)
    {
      // Prevent memory leaks.
      delete nr_entry;
      ACE_THROW (Event_Comm::Notifier::CannotSubscribe ("Failed to add
Consumer to internal map\n"));
    }
}

// Remove a consumer from the table.

void
Notifier_i::unsubscribe (Event_Comm::Consumer_ptr consumer_ref,
          const char *filtering_criteria,
          CORBA::Environment &ACE_TRY_ENV)
  ACE_THROW_SPEC ((
                   CORBA::SystemException,
                   Event_Comm::Notifier::CannotUnsubscribe
                   ))
{
  ACE_DEBUG ((LM_DEBUG,
             "in Notifier_i::unsubscribe for %x\n",
         consumer_ref));

  Consumer_Entry *nr_entry = 0;
  MAP_ITERATOR mi (this->map_);
  int found = 0;

  // Locate <Consumer_Entry> and free up resources.  @@ Note, we do not
  // properly handle deallocation of KEYS!
  for (MAP_ENTRY *me = 0;
       mi.next (me) != 0;
       mi.advance ())
    {
      nr_entry = me->int_id_;

      // The <_is_equivalent> function checks if objects are the same.
      // NOTE: this call might not behave well on other ORBs since
      // <_is_equivalent> isn't guaranteed to differentiate object
      // references.

      // Look for a match ..
      if (consumer_ref->_is_equivalent (me->ext_id_)
     && (ACE_OS::strcmp (filtering_criteria, "") == 0
         || ACE_OS::strcmp (filtering_criteria,
             nr_entry->criteria ()) == 0))
   {
     ACE_DEBUG ((LM_DEBUG,
                  "removed entry %x with criteria \"%s\"\n",
                  consumer_ref,
                  filtering_criteria));
     found = 1;
     // @@ This is a hack, we need a better approach!
     if (this->map_.unbind (me->ext_id_,
             nr_entry) == -1)
       ACE_THROW (Event_Comm::Notifier::CannotUnsubscribe
         ("Internal map unbind failed."));
     else
       delete nr_entry;
     }
  }

  if (found == 0)
    ACE_THROW (Event_Comm::Notifier::CannotUnsubscribe ("The Consumer
and filtering criteria were not found."));
}

// Disconnect all the consumers, giving them the <reason>.

void
Notifier_i::disconnect (const char *reason,
         CORBA::Environment &ACE_TRY_ENV)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  ACE_DEBUG ((LM_DEBUG,
              "in Notifier_i::send_disconnect = %s\n",
              reason));

  MAP_ITERATOR mi (this->map_);
  int count = 0;

  // Notify all the consumers, taking into account the filtering
  // criteria.

  for (MAP_ENTRY *me = 0;
       mi.next (me) != 0;
       mi.advance ())
    {
      Event_Comm::Consumer_ptr consumer_ref =
        me->ext_id_;

      ACE_ASSERT (consumer_ref != 0);
      ACE_DEBUG ((LM_DEBUG,
                  "disconnecting client %x\n",
                  consumer_ref));
      ACE_TRY
        {
          consumer_ref->disconnect (reason,
                                    ACE_TRY_ENV);
          ACE_TRY_CHECK;
        }
      ACE_CATCHANY
        {
     ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION, "Unexpected exception\n");
        }
      ACE_ENDTRY;

      delete me->int_id_;
      count++;
    }

  this->map_.close ();

  if (count == 1)
    ACE_DEBUG ((LM_DEBUG,
                "there was 1 consumer\n"));
  else
    ACE_DEBUG ((LM_DEBUG,
                "there were %d consumers\n",
                count));
}

//* This push is triggered / called from the Consumers.
// Notify all consumers whose filtering criteria match the event.

void
Notifier_i::push (const Event_Comm::Event &event,
        CORBA::Environment &ACE_TRY_ENV)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  ACE_DEBUG ((LM_DEBUG,
              "in Notifier_i::send_notification = %s\n",
         (const char *) event.tag_));
  MAP_ITERATOR mi (this->map_);
/**/
  int count = 0;

  // Notify all the consumers.
  // For every entry in the map ...

  for (MAP_ENTRY *me = 0; mi.next (me) != 0; mi.advance ())
    {
      Event_Comm::Consumer_ptr consumer_ref = me->int_id_->consumer ();
      ACE_ASSERT (consumer_ref != 0);
```

```
#if defined (ACE_HAS_REGEX)
      char *regexp = ACE_const_cast (char *, me->int_id_->regexp ());
      ACE_ASSERT (regexp);

      const char *criteria = me->int_id_->criteria ();
      ACE_ASSERT (criteria);

      // Do a regular expression comparison to determine matching.
      if (ACE_OS::strcmp ("", criteria) == 0 // Everything matches the
wildcard.
     || ACE_OS::step (event.tag_, regexp) != 0)
#endif // #if defined (ACE_HAS_REGEX)
        // if ACE_HAS_REGEX has not been defined,
        // let everything through.
   {
     ACE_DEBUG ((LM_DEBUG,
                      "string %s matched regexp \"%s\" for client %x\n",
             (const char *) event.tag_,
             me->int_id_->criteria (),
             consumer_ref));

/**/
   const char *criteria = me->int_id_->criteria ();

   char *cr = (char *)criteria;
   char *et = (char *)(const char *)event.tag_;

//* ... if the destination (event flag) matches the current
//* map entry->criteria ....


      int flag = 0;
      for(int ix=0;ix<1;ix++) // useful for more characters
      {
         if((int)cr[ix]==(int)et[ix])
            flag = 1;
      }

      if(flag)
      {
            cout << "message for " << cr << endl;

//* ... send the event to the current customer entry in the map


         ACE_TRY
         {
            consumer_ref->push (event,
               ACE_TRY_ENV);
            ACE_TRY_CHECK;
         }
         ACE_CATCHANY
            {
            ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,
                                 "Unexpected exception\n");
            continue;
```

```
                }
            ACE_ENDTRY;
            count++;

            }

        flag = 0;


        }
    }

  if (count == 1)
    ACE_DEBUG ((LM_DEBUG,
                "there was 1 consumer\n"));
  else
    ACE_DEBUG ((LM_DEBUG,
                "there were %d consumers\n",
                count));
}
```

APPENDIX C. THE MSHN COMPONENTS' IMPLEMENTATION

```
// Original by Douglas C. Schmidt
// Modified and extended to support MSHN functionality
// by Panagiotis Papadatos

Consumer_i::Consumer_i (void)
  : shutdown (0)
{
}

Consumer_i::~Consumer_i (void)
{
}

// Inform the <Event_Comm::Consumer> that <event> has
// occurred.

void
Consumer_i::push (const Event_Comm::Event &event,
   CORBA::Environment &)
ACE_THROW_SPEC ((CORBA::SystemException))
{
   static int state1;
   const char *tmpstr = event.tag_;
   ACE_DEBUG ((LM_DEBUG,
      "**** got notification = %s\n",
      tmpstr));

   static ACE_Profile_Timer ptimer;
   static ACE_Profile_Timer::ACE_Elapsed_Time eltime;
   static double time = 0;

   ACE_TRY_NEW_ENV
   {
      char *et = (char *)(const char *)event.tag_;

      switch (et[0])              // switch depending on the role
      {
         case '1' :               // scheduling advisor
         switch (state1)          // switch depending on the state
         {
            case 0 :              // idle
               state1++;          // received SA<-CL. sending SA->RSS
               {
                  Event_Comm::Event event2;
                  event2.tag_ = (const char *)"2";
// ... send the event to the current customer entry in the map
                  this->notifier_i->push (event2, ACE_TRY_ENV);
               }
               {
                  Event_Comm::Event event2;
                  event2.tag_ = (const char *)"3";
// ... send the event to the current customer entry in the map
```

```
                  this->notifier_i->push (event2, ACE_TRY_ENV);
                  }
              break;

              case 1 :              // waiting reply from RSS
                 state1++;  // received SA<-RSS. Waiting reply from RRD
              break;

              case 2 :              // waiting reply from RSS
                 state1 = 0;        // received SA<-RRD. sending SA->CL
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"4";
// ... send the event to the current customer entry in the map
                 this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
          }
      break;

      case '2' :                    // RSS
          switch (state1)           // // switch depending on the state
          {
              case 0 :              // idle
                 state1++;          // received RSS<-SA. querying DB
                 state1--;          // sending responce RSS->SA
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"1";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
          }
      break;

      case '3' :                    // RRD
          switch (state1)           // // switch depending on the state
          {
              case 0 :              // idle
                 state1++;          // received RRD<-SA. querying DB
                 state1--;          // sending responce RRD->SA
                 {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"1";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                 }
              break;
          }
      break;

      case '4' :                    // CL
          switch (state1)           // switch depending on the state
          {
              case 0 :              // idle
                 ptimer.start ();
```

```
                state1++;         // sending CL->SA
                {
                   Event_Comm::Event event2;
                   event2.tag_ = (const char *)"1";
// ... send the event to the current customer entry in the map
                   this->notifier_i->push (event2, ACE_TRY_ENV);
                }
             break;

             case 1 :             // waiting schedule from SA
                state1++;         // receiving CL<-SA
                state1++;         // sending CL->D
                {
                   Event_Comm::Event event2;
                   event2.tag_ = (const char *)"5";
// ... send the event to the current customer entry in the map
                   this->notifier_i->push (event2, ACE_TRY_ENV);
                }
             break;
             case 3 :             // the job is running
                state1=0;         // receiving CL<-D
                ptimer.stop ();
                ptimer.elapsed_time (eltime);
                time = eltime.real_time;
                ACE_DEBUG ((LM_DEBUG,
                   "Latency is %.0f usec\n",time * 1e6));
             break;
          }
       break;

       case '5' :                 // MSHN Daemon
          switch (state1)         // switch depending on the state
          {
             case 0 :             // idle
                state1++;  // received D<-CL. Executing application
                state1--;  // sending notification D->CL
                {
                   Event_Comm::Event event2;
                   event2.tag_ = (const char *)"4";
// ... send the event to the current customer entry in the map
                   this->notifier_i->push (event2, ACE_TRY_ENV);
                }
             break;
          }
       break;

       case '6' :                 // Object_1
          switch (state1)         // switch depending on the state
          {
             case 0 :             // idle
                ptimer.start ();
                state1++;         // sending Object_1 -> Object_2.
                {
                   Event_Comm::Event event2;
                   event2.tag_ = (const char *)"7";
// ... send the event to the current customer entry in the map
                   this->notifier_i->push (event2, ACE_TRY_ENV);
                }
            break;

            case 1 :                    // waiting responce from Object_2
                state1--;               // receiving Object_1 <- Object_2
                ptimer.stop ();
                ptimer.elapsed_time (eltime);
                time = eltime.real_time;
                ACE_DEBUG ((LM_DEBUG,
                    "Latency is %.0f usec\n",time * 1e6));
            break;
        }
    break;

    case '7' :                          // Object_2
        switch (state1)                 // switch depending on the state
        {
            case 0 :                    // idle
                state1++;               // received Object_2 <- Object_1
                                        // Executing application
                state1--;               // sending Object_2 -> Object_1
                {
                    Event_Comm::Event event2;
                    event2.tag_ = (const char *)"6";
// ... send the event to the current customer entry in the map
                    this->notifier_i->push (event2, ACE_TRY_ENV);
                }
            break;
        }
    break;

    default:
    cout << "Default reached. "  << endl;

    }
    ACE_TRY_CHECK;
  }
  ACE_CATCHANY
  {
    ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION, "Unexpected exception\n");
  }
  ACE_ENDTRY;
}

// Disconnect the <Event_Comm::Consumer> from the
// <Event_Comm::Notifier>.

void
Consumer_i::disconnect (const char *reason,
         CORBA::Environment &)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  ACE_DEBUG ((LM_DEBUG,
     "**** got disconnected due to %s\n",
     reason));
```

```
   ACE_ASSERT (shutdown != 0);
   shutdown->close ();
}

void
Consumer_i::set (ShutdownCallback *_shutdown)
{
   shutdown = _shutdown;
}


//* Keep a pointer to the notifier here so we will be able to resend
events

void
Consumer_i::set_notifier_i (Event_Comm::Notifier_var notifier_i_temp)
{
   this->notifier_i = notifier_i_temp;
}
```

APPENDIX D. MEASUREMENT OF TIME USING ACE

In order to measure the time span between events in our experiments, we used the ACE_High_Res_Timer, one of the numerous components of the ACE toolkit. The ACE_High_Res_Timer is a class wrapper that encapsulates OS-specific high-resolution timers, such as those found on Solaris, AIX, Win32/Pentium, and VxWorks.

The ACE_High_Res_Timer uses a native high-resolution timer if one is available, such as gethrtime( ) on SunOS. Otherwise, it uses the tick counter on supported CPUs, such as Pentium and PowerPC.

The interface provided by the ACE_High_Res_Timer is as follows:

```
#include <ace/High_Res_Timer.h>

class ACE_High_Res_Timer
{
   public:
   static void global_scale_factor (ACE_UINT32 gsf);
      static ACE_UINT32 global_scale_factor (void);
      static int get_env_global_scale_factor (
         const char *env = "ACE_SCALE_FACTOR"
       );
      static ACE_UINT32 calibrate (
         const ACE_UINT32 usec = 500000,
         const u_int iterations = 10
       );
      ACE_High_Res_Timer (void);
      ~ACE_High_Res_Timer (void);
      void reset (void);
      void start (
      const ACE_OS::ACE_HRTimer_Op = ACE_OS::ACE_HRTIMER_GETTIME
        );
      void stop (
         const ACE_OS::ACE_HRTimer_Op = ACE_OS::ACE_HRTIMER_GETTIME
      );
      void elapsed_time (ACE_Time_Value &tv) const;
      void elapsed_time (ACE_hrtime_t &nanoseconds) const;
      void elapsed_time (struct timespec &) const;
      void elapsed_microseconds (ACE_hrtime_t &usecs) const;
      void start_incr (const ACE_OS::ACE_HRTimer_Op =
         ACE_OS::ACE_HRTIMER_GETTIME
          );
      void stop_incr (
         const ACE_OS::ACE_HRTimer_Op = ACE_OS::ACE_HRTIMER_GETTIME
      );
    void elapsed_time_incr (ACE_Time_Value &tv) const;
    void elapsed_time_incr (ACE_hrtime_t &nanoseconds) const;
    void print_total (
       const char *message,
       const int iterations = 1,
       ACE_HANDLE handle = ACE_STDOUT
      ) const;
    void print_ave (
       const char *message,
       const int iterations = 1,
       ACE_HANDLE handle = ACE_STDOUT
      ) const;
    void dump (void) const;
       ACE_ALLOC_HOOK_DECLARE;
       ACE_OS::ACE_HRTIMER_GETTIME
 );
    static void hrtime_to_tv (
        ACE_Time_Value &tv,
       const ACE_hrtime_t hrt
      );
    static ACE_UINT32 get_cpuinfo (void);
 private:
    ACE_OS::ACE_HRTIMER_GETTIME);
 ACE_hrtime_t start_;
    ACE_hrtime_t end_;
    ACE_hrtime_t total_;
    ACE_hrtime_t start_incr_;
    static ACE_UINT32 global_scale_factor_;
    static int global_scale_factor_status_;
};
```

The global scale factor is required for platforms having high-resolution timers returning units other than microseconds (e.g. clock ticks). It is represented as a static `u_long`, can only be accessed through static methods, and is used by all instances of `High_Res_Timer`. The member functions that return or print times use the global scale factor. They divide the "time" that they get from `ACE_OS::gethrtime( )` by `global_scale_factor_` to obtain the time in microseconds. Its units are therefore 1/microsecond. On Solaris, a scale factor of 1000 should be used because its high-resolution timer returns nanoseconds. However, on Intel platforms, we use RDTSC (read-time stamp counter) instruction of Intel architecture, which returns the number of clock ticks since system boot. The time-stamp counter keeps an accurate count of every cycle that occurs on the processor. The Intel time-stamp counter is a 64-bit MSR (model specific register) that is incremented every clock cycle. On reset, the time-stamp counter is set to zero. As the time-stamp counter measures "cycles" and not "time", thus two hundred million cycles for example on a 200 MHz processor is equivalent to one second of real time, while the same number of cycles on a 400 MHz processor is only one-half second of real time. Thus, comparing cycle counts only makes sense on processors of the same speed. To compare processors of different speeds, the cycle counts should be converted into time units, where: # seconds = # cycles / frequency

So, for our 400MHz cpu, each clock tick is 1/400 of a microsecond; the global_scale_factor_ should therefore be 400.

The calibration of the global scale factor is performed using the following function:

```
static ACE_UINT32 calibrate (
    const ACE_UINT32 usec = 500000,
    const u_int iterations = 10
);
```

It sets (and returns, for info) the global scale factor by sleeping for `usec` and counting the number of intervening clock cycles. Average over iterations of `usec` each. On Pentium platforms, this is called automatically during the first `ACE_High_Res_Timer` construction with the default parameter values. An application can override that by calling calibrate with any desired parameter values _prior_ to constructing the first ACE_High_Res_Timer instance.

The `ACE_High_Res_Timer` has nanosecond resolution.

APPENDIX E. ABBREVIATIONS

CGI – Common Gateway Interface

CL – Client Library

DARPA - Defense Advanced Research Project Agency

DCOM – Distributed Component Object Model

EC – Event Channel

FSM – Finite State Machine

GIOP – General Inter-ORB protocol

IDL – Interface Definition Language

IIOP – Internet Inter-ORB

IPC – Interprocess Communication

IR – Interface Repository

ME – Managed Environment

MS – Managed System

MSHN – Management System for Heterogeneous Networks

ORB – Object Request Broker

OS - Operating System

QoS - Quality of Service

RMI – Remote Method Invocation

RPC – Remote Procedure Call

RRD	Resource Requirements Database
RSS	Resource Status Server
SA	Scheduling Advisor
SAP	Service Access Point
WUSTL	Washington University of St. Luis

LIST OF REFERENCES

[1] Debra Hensgen, Taylor Kidd, David St. John, Matthew C. Schnaidt, H. J. Siegel, Tracy Braun, Jong-Kook Kim, Shoukat Ali, Cynthia Irvine, Tim Levin, Viktor Prasanna, Prashanth Bhat, Richard Freund, and Mike Gherrity. An Overview of the Management System for Heterogeneous Networks (MSHN). 8th Workshop on Heterogeneous Computing Systems (HCW '99). San Juan, Puerto Rico. Apr. 1999.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel et al. A Pattern Language. Oxford University Press, New York, 1977.

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

[4] James Coplien, Douglas Schmidt. Pattern Languages of Program Design vol 1. Addison – Wesley, Reading MA, 1995.

[5] John Vlissides, James Coplien, Norman Kerth. Pattern Languages of Program Design vol 2. Addison – Wesley, Reading MA, 1996.

[6] Robert Martin, Dirk Riehle, Frank Buschmann. Pattern Languages of Program Design vol 3. Addison – Wesley, Reading MA, 1998.

[7] John Vlissides. Pattern Hatching: Design Patterns Applied. Addison – Wesley, Reading MA, 1998.

[8] Roger Wright, David J. Shifflett, Cynthia E. Irvine. Security Architecture for a Virtual Heterogeneous Machine. *Proceedings of Fourteenth Computer Security Applications Conference*. Phoenix, AZ. pp 167—77, December 1998.

[9] Alpay Duman. The Use and Run-time Overhead of CORBA in MSHN Project. Master's thesis. Naval Postgraduate School, Monterey, CA. September 1998.

[10] Linda Rising, ed. The Patterns Handbook. Cambridge University Press. Cambridge, UK. 1998.

[11] James Coplien. Generating Pattern Languages: An emerging direction of software design. Proceedings of the 5th Annual Borland International Conference. Orlando, FL. June 1994.

[12] Roger S. Pressman. Software Engineering. A practitioner's approach. McGraw-Hill. New York, 1997.

[13] Douglas C. Schmidt. The Adaptive Communication Environment. Proceedings of the 12th Sun User Group Conference. San Francisco, CA. June 1993.

[14] Douglas C. Schmidt. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*. Summer, 1997.

[15] Umar Syyid. The Adaptive Communication Environment. A Tutorial. Hughes Network Systems. Germantown MD. October 1998.

[16] Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.0 ed., July 1995.

[17] Douglas C. Schmidt. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*. vol.21, April 1998.

[18]. Robert Orfali, Dan Harkey. Client Server Programming with Java and CORBA. Wiley. New York, NY. 1998.

[19] Michi Henning, Steve Vinoski. Advanced CORBA Programming with C++. Addison – Wesley, Reading MA, 1999.

[20] Thomas J. Mowbray, Ron Zahavi. The Essential Corba. Object Management Group. Wiley. New York, NY. 1995.

[21] Douglas C. Schmidt, Nanbor Wang. An OO Encapsulation of Lightweight Concurrency Mechanisms in the ACE Toolkit. Technical Report WUCS-95-31. Washington University, St Luis, MI. February, 1999.

[22] Thomas J. Mowbray, William A. Ruh. Inside CORBA. Addison Wesley, Reading MA. 1997.

[23] Doug Pedrick, Jonathan Weedon, Jon Goldberg, Eric Bleifield. Programming with Visibroker. Wiley. New York, NY. 1998.

[24] Jeremy L. Rosenberger. Teach Yourself Corba in 14 Days. Sams Publishing. Devon, UK. January, 1998.

[25] Andreas Vogel, Keith Duddy. Java Programming with CORBA. Wiley. New York, NY. February, 1998.

[26] Douglas C. Schmidt. IPC SAP. C++ Wrappers for Efficient, Portable and Flexible Network Programming. *C++ Report*. November / December 1992.

[27] Object Management Group. CORBA Services: Common Object Services Specification, December 1998.

[28] MITRE Document MP 95B-93. March 1995.

[29] John Siegel. CORBA Fundamentals and Programming. Object Management Group. Wiley. New York, NY. 1996.

[30] John E.Hopcroft, Jeffrey D. Ullman. Introduction to Automata Theory, Languages and Computation.Addison Wesley, Reading MA. 1979.

[31] Object management Group. The Common Object Request Broker: Architecture and Specification. 2.2 ed., February 1998.

[32] Don Box. Essential COM. Addison-Wesley. Reading, MA. 1997.

[33] Douglas C. Schmidt, Andy Gokhale, T.Harisson and G.Parulkar. A High Performance Endsystem Architecture for Real-time CORBA. IEEE Communications Magazine, vol. 14. IEEE. February 1997.

[34] Douglas C. Schmidt, David L.Levine, and Sumedh Mungee. The Design and Performance of Real-time Object request Brokers. Computer Communications, vol. 21. IEEE. April 1998.

[35] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishan Kachroo and Andy Gohkale. Applying Optimization Patterns to the Design of Real-time ORBs. Proceedings of the 5th Conference on Object-Oriented technologies and Systems. USENIX Association. May 1999.

[36] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. Proceedings of the 2nd C++ Conference. USENIX Association. April 1990.

[37] Object Management Group. Real-time CORBA 1.0 Joint Submission. OMG Document orbos/98-12-05 ed. December 1998.

[38] Zubin D. Dittia, G.M. Parulkar, and J.R.Cox, Jr. The APIC Approach to High Performance Network Interface Design: Protected DMA and other Techniques. Proceedings of INFOCOM '97. IEEE. April, 1997.

[39] P.Hoschka. Automating Performance Optimization by Heuristic Analysis of a Formal Specification. Proceedings for Joint Conference for Formal Description

Techniques (FORTE) and Protocol Specification, Testing and Verification (PSTV). Kaiserlautern. 1996.

[40] G. Copalakrishnan and G. Paarulkar. Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing. Proceedings of the SIGMETRICS Conference. ACM. May 1996.

[41] David R. Musser, Atul Saini. STL Tutorial and Reference Guide. Addison-Wesley. Reading, MA. 1996.

[42] Matthew Schnaidt. Design, Implementation, and Testing of MSHN's Application Resource Monitoring Library. Masters Thesis. Naval Postgraduate School. Monterey, CA. December 1998.

[43] Matthew Schnaidt, Debra Hensgen, David St. John, Taylor Kidd, and John Falby. Passive Domain-Independent, End-to-End Message Passing Performance Monitoring to Support Adaptive Applications in MSHN. 8th International Symposium on High Performance Distributed Computing (HPDC). August 1999.

[44] John Kresho, Debra Hensgen, Taylor Kidd, and Geoffry Xie. Determining the Accuracy Required in Resource Load Prediction to Successfully Support Application

Agility. Proceedings of the 2nd IASTED International Conference of European Parallel and Distributed Systems. July 1998.

[45] Prashanth B. Bhat, C.S. Raghavendra, and Viktor K. Prasanna. Efficient Collective Communication in Distributed Heterogeneous Systems. The 19th International Conference on Distributed Computing Systems (ICDCS), 1999.

[46] Howard Jay Siegel and Muthucumaru Maheswaran. Mapping Tasks onto Heterogeneous Computing Systems. IX Simposio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho (SBAC-PAD '97) (IX Brazilian Symposium on Computer Architectures - High Performance Computing). Sao Paulo, Brazil. October 1997.

[47] Paul Carff. When is a Simple Model Adequate For Use in Scheduling in MSHN? . Master's Thesis. Naval Postgraduate School. Monterey, CA.

[48] John Cresho. Quality Network Load Information Improves performance of Adaptive Applications. Masters Thesis. Naval Postgraduate School. Monterey, CA. September 1997.

[49] Phillip A. Laplante. Real-time Systems Design and Analysis. IEEE Computer Society Press. New York, NY. 1997.

[50] Brian Noble, M.Satyanarayanan, Dushyanth Narayanan, James E. Tilton, Jason Flinn and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. Proceedings of the 16th Symposium on Operating Systems. 1997.

[51] Taylor Kidd. Unpublished Work. June and July 1999.

[52] Robert W. Weeks, John J. Moskwa. Automotive Engine Modeling for Real-time Control Using MATLAB/SIMULINK. Society of Automotive Engineers (SAE) 1995 International Congress and Exposition. SAE. Detroit, MI. April 1995.

[53] Brian Van Voorst, Luiz Pires, Rakesh Jha. A Real-time Parallel Benchmark Suite. Available at http://www.htc.honeywell.com/projects/rtpbs/ on 10th September 1999.

[54] Jeffrey M. Maddalon, Jeff I. Cleveland II. A Study of Workstation Computational Performance for Real-time Flight Simulation. Langley Research Center.

[55] Nelson Weiderman. Hartstone: Synthetic Benchmark Requirements for Hard Real-time Applications. Technical Report CMU/SEI-89-TR-23. Carnegie Mellon University. Pittsburgh, PA. June 1989.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center...2
   8275 John J. Kingman Road, Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library..2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Naval Attache Office...3
   Embassy of Greece
   2228 Massachusetts Avenue
   Washington NW 20008

4. Chairman, Code CS...1
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

5. Taylor Kidd, Code CS/Kt...1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

6. Debra Hensgen, Code CS/Hd...1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

7. Panagiotis Papadatos...1
   Hellenic Navy General Staff
   Greece