Are CORBA Services Ready to Support Resource Management Middleware for Heterogeneous Computing? *

Alpay Duman, Debra Hensgen, David St. John, and Taylor Kidd

Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

Abstract

The goal of this paper is to report our findings as to which CORBA services are ready to support distributed system software in a heterogeneous environment. In particular, we implemented intercommunication between components in our Management System for Heterogeneous Networks (MSHN[1]) using four different CORBA mechanisms: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), Untyped Event Services, and Typed Event Services. MSHN's goals are to manage dynamically changing sets of heterogeneous adaptive applications in a heterogeneous environment. We found these mechanisms at various stages of maturity, resulting in some being less useful than others. In addition, we found that the overhead added by CORBA varied from a low of 10.6 milliseconds per service request to a high of 279.1 milliseconds per service request on workstations connected via 100 Mbits/sec Ethernet. We therefore conclude that using CORBA not only substantially decreases the amount of time required to implement distributed system software, but it need not degrade performance.

1 Introduction

This paper describes the experiences we had using CORBA mechanisms to implement intercommunication in MSHN. MSHN's goal is to support the execution of multiple, disparate, adaptive applications[2] in a dynamic, distributed heterogeneous environment. To accomplish this goal, MSHN consists of multiple, distinct, and eventually replicated distributed components that themselves execute in a heterogeneous environment. These components have widely varying functionality, come in and out of existence, and communicate across heterogeneous networks. In addition to executing on different types of platforms, these components are also likely to be written in different programming languages. We can, of course, at the expense of a great deal of programmer's time, implement specialized naming services to locate the appropriate component at run-time, and specialized communication mechanisms to enable communication between the heterogeneous platforms upon which the components run. Alternatively, we can use a general tool, such as the Common Object Request Broker Architecture (CORBA), to achieve the same functionality while reducing our development time. Experience with generalized systems, such as CORBA, has revealed that the reduction in development time costs come at the expense of run-time performance, which can be critical in real-time applications. This research, therefore, investigates the utility and overhead of communication mechanisms, which are implemented according to the CORBA 2.2 specification, to support MSHN's inter-component communication.

We note to the reader that our interest lies in the CORBA mechanisms that support the development of (possibly real-time) resource management environments. This is a very specific realm where system overheads can have a significant impact on performance. We do not explore the many and varied capabilities of CORBA for the supporting of other environments, such as that of distributed general database services and video streaming. Our interest in CORBA is primarily as a tool to reduce the time/programming investment needed to implement our resource management system middleware. As the services and mechanisms provided by the CORBA 2.2 specification, particularly Static and Dynamic Invocation, and the Event Services, hold great promise in this regard, we performed the series of studies detailed in this paper.

CORBA specifies a standard to permit different pro-

*This research was supported by DARPA under contract number E583. Additional support was provided by the Naval Postgraduate School and the Institute for Joint Warfare Analysis.

[1] Pronounced "mission"

[2] This paper focuses on the use of CORBA mechanisms to support the components of MSHN, not the applications that MSHN itself supports. For more details concerning applications, please see the references or contact the the authors directly.

grams, executing on different computers, to request services from one another. CORBA's Naming Service and Object Request Brokers (ORBs) aid clients in locating appropriate servers. CORBA's static invocation enables a CORBA client to make a request of a server that is identified prior to compile time. It provides both reliable synchronous semantics and unreliable asynchronous semantics. In contrast, CORBA's dynamic invocation enables the client to locate a server that may not be known until run-time, and provides reliable synchronous and asynchronous semantics, as well as unreliable asynchronous semantics. CORBA's event services allow processes on one machine to place event notifications intended for processes on other machines into event queues so that the notifications can later be delivered to the serving processes. This service facilitates multicast. This paper will not cover CORBA in detail, but there are many other good references on the subject [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

The paper is organized as follows. We first briefly describe MSHN, concentrating on the type of intercommunication that is required by its components. A more complete description of MSHN can be found elsewhere [12]. Alternate designs for facilitating communication within MSHN itself and the implementation of these designs are presented. These designs are based upon, respectively, static invocation, dynamic invocation, untyped event service and typed event service. In this section, we also provide a qualitative assessment detailing the problems that we encountered while attempting to use these mechanisms within MSHN. In a subsequent section, we describe our experiments for evaluating these mechanisms within MSHN and present a quantitative analysis of each of the mechanisms. Finally, we summarize our findings.

2 The Management System for Heterogeneous Networks (MSHN)

In the Heterogeneous Processing Laboratory at the Naval Postgraduate School, we are designing, implementing, and testing a resource management system called the Management System for Heterogeneous Networks (MSHN). MSHN is designed as a general experimental platform for investigating issues relating to the design and construction of future resource management systems operating in heterogeneous environments. Though MSHN is used to explore a large number of such issues, our present research focuses on finding and developing (1) mechanisms for supporting adaptive applications, (2) mechanisms for supporting the satisfaction of user and system defined Quality of Service (QoS) requirements, and (3) mechanisms for acquiring and usefully aggregating measurements of both

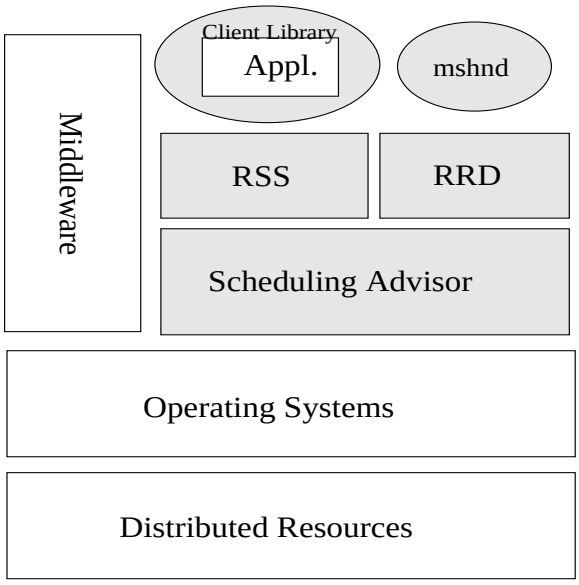

Figure 1: MSHN Conceptual Architecture

general resource availability and the resource usage of individual tasks. A thorough and complete description of MSHN can be found in Hensgen [12].

MSHN's architecture consists of multiple instantiations of each of the components enumerated below:

- a Client Library (one for each executing application to be managed by MSHN),
- a Scheduling Advisor (hierarchically replicated),
- a Resource Requirement Database (hierarchically replicated),
- a Resource Status Server (hierarchically replicated), and
- a MSHN Daemon (when needed).

Figure 1, the MSHN Conceptual Architecture, shows all of the MSHN components (shaded) as **translucent layers** executing on distributed platforms. A translucent layer is one that can be bypassed by layers that are above or below it. For example, the MSHN Daemon (mshnd) can interact directly with the operating systems layer, bypassing the Resource Status Server, the Resource Requirement Database and the Scheduling Advisor. In the environment that MSHN supports, both MSHN and non-MSHN applications may be executing at any given time. Figure 2 illustrates how these components, along with various MSHN and non-MSHN applications, might actually be distributed among different heterogeneous machines.

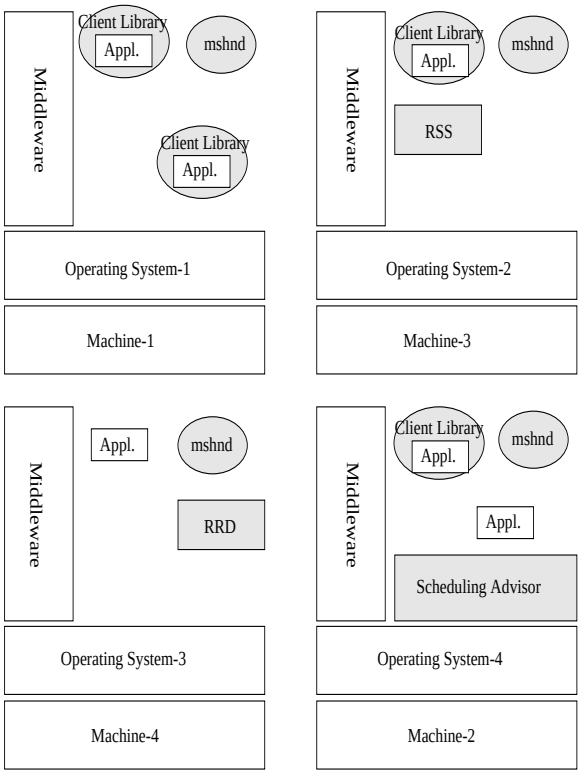

Figure 2: Example MSHN Physical Instantiation

This research investigates how communication between the components can be facilitated. As such, the MSHN description in the remainder of this section emphasizes that communication.

Figure 3, MSHN's Software Architecture, illustrates all of the interactions between the components. MSHN has a peer-to-peer architecture[3].

We now present two- and three-tier views to give a clear understanding of the interactions between the components. Generally, many applications, each linked with the MSHN Client Library, will be running at any given time. They will need to communicate with a Scheduling Advisor (SA) to request the appropriate resources needed to start new processes. They may also communicate with a MSHN Daemon when receiving their recommended schedule. Additionally, their Client Libraries update the Resource Requirement Database (RRD) and the Resource Status Server (RSS) with the expected resource requirements of the applications and current resource availability within the MSHN system.

[3]When callbacks are used the client and the server have a peer-to-peer relationship. In distributed systems, callbacks are useful as a mechanism for performing asynchronous communication. Callbacks transmit event notifications without blocking the event originator. Callbacks flow from the servers towards the clients.

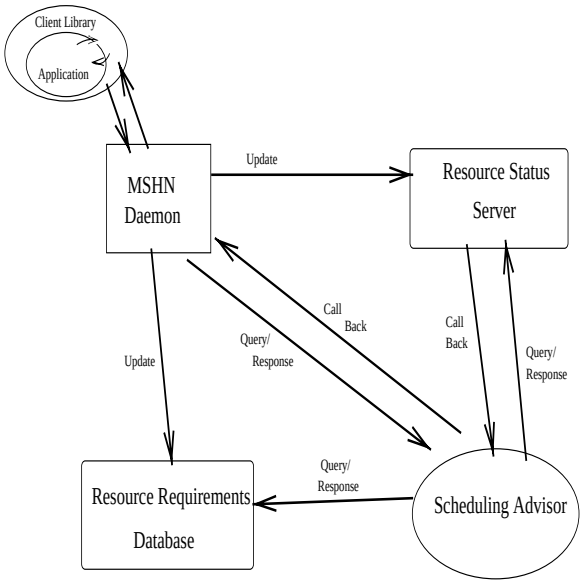

Figure 3: MSHN's Software Architecture

Figure 4 illustrates this updating interaction as a two-tiered client/server architecture. The arrows labeled "1" designate the Resource Requirements Database update path, and those labeled "2," the Resource Status Server update path. The update frequency of the Resource Status Server is expected to be high so that it, in turn, can supply the Scheduling Advisor with accurate and current information.

We anticipate that the frequency of the updates will load down the network, and cause a considerable processing load on the Resource Status Server and the Resource Requirement Database. To avoid these loads, MSHN's design includes proxy Resource Status Servers and Resource Requirement Databases that will come in and out of existence as required to minimize the number of updates. These proxies will filter gathered information and update the hierarchical Resource Status Server and the hierarchical Resource Requirement Database when necessary.

In one view, the Scheduling Advisor functionally resides between the information needed to create a schedule (the Resource Status Server and the Resource Requirement Database) and the requesters of schedules (applications linked with the Client Library). This indicates that there will be a high communication rate to and from the Scheduling Advisor. We can therefore also view MSHN as having three tiers, where the Scheduling Advisor is the second tier, and the Resource Status Server and the Resource Requirement Database are in the third tier (see Figure 5). When the Client Library (first tier) contacts the Scheduling Advisor for

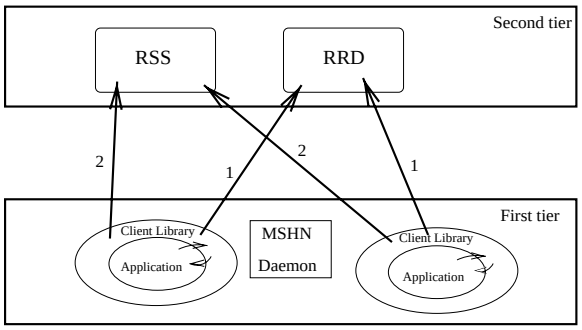

Figure 4: Two-tiered Architectural View of MSHN Architecture

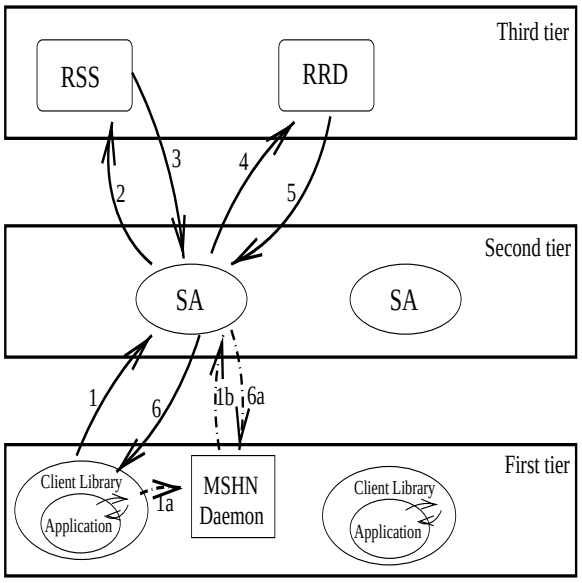

Figure 5: Three-tiered View of MSHN

a schedule, either directly or via the MSHN Daemon (the arrows labeled "1" and "1a"), the Scheduling Advisor queries both the Resource Status Server (arrows "2" and "3"), and the Resource Requirement Database (arrows "4" and "5") before it computes its schedule and sends it to the MSHN Daemon or client library depending upon which is more appropriate (arrows "6" and "6a")

Although the Client Libraries are the initiators of many of the communication chains through the MSHN system, other chains are initiated by the Resource Status Server. For example, in the case where a violation of a deadline occurs because of a change in resource availability, the Resource Status Server will trigger the Scheduling Advisor to reschedule processes that would not otherwise meet their deadline. The Scheduling Advisor will adapt to the new situation by either changing

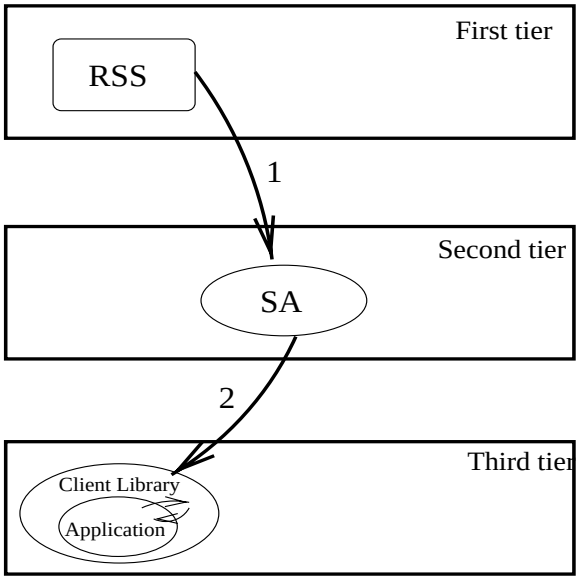

Figure 6: Alternate Three-tiered View of MSHN

the format[4] of the process or restarting it on a different resource, possibly via the MSHN Daemon. This interaction is the reverse of the previously described communication chain and can be used to define another version of a three-tiered view. (See Figure 6.)

Although we have shown several two and three tier views of MSHN, the reader should understand that these are only examples. Much larger chains will actually exist when the various components are hierarchically replicated.

3 Use of CORBA Services in MSHN and Problems Encountered

Our goal is to determine both (1) how we can best facilitate efficient communication between the components in our architecture using mechanisms from the CORBA 2.2 specification, and (2) to determine the runtime overhead of each of those mechanisms. Our justification for choosing a particular mechanism included extensibility, scalability, portability, flexibility, and efficiency.

MSHN consists of multiple, eventually replicated, distinct distributed components that execute in a heterogeneous environment. These components will have widely varying functionality, will come in and out of existence, will communicate via heterogeneous networks, and will execute on different platforms. To facilitate the interactions between MSHN's components, we identified four mechanisms from the CORBA 2.2 specifica-

[4] We use the term "format" to refer to a mechanism we have developed to support adaptive applications [13].

tion that had particular promise: the Typed Event Service, the Untyped Event Service, the Static Invocation Interface (SII), and the Dynamic Invocation Interface (DII). After settling on these four mechanisms, we implemented a prototype of MSHN's communication infrastructure using each of them. First we describe how the MSHN architecture would benefit from the both the Typed and Untyped Event Service, the Static Invocation Interface (SII), and the Dynamic Invocation Interface (DII). Then we discuss how we use the Naming Service within MSHN to obtain object references. In this section, since part of the objective of this paper is to make recommendations with regards to additions and improvements to the evolving CORBA specification, we describe and justify each of our designs, the problems we encountered, and the solutions to which we arrived.

3.1 Selection of a CORBA ORB

At the beginning of this research, we explored various implementations of the CORBA standard. Figures 7 and 8 present a summary of the results of that exploration[5]. Based upon various requirements, including the cost of some of the implementations, the time required to implement comparative tests, and the duration of this study, we had to limit ourselves to one CORBA implementation. We chose the implementation that seemed, at that time, to have the most mature features relevant to MSHN. Our assumption was that once such an implementation was found, other implementations would typically have similar difficulties and comparable performance. As such, we based our studies around IONA's Orbix, the implementation that best fit this requirement.

3.2 Event Service

Event Service allows multiple suppliers and multiple consumers to deliver and receive notifications for a set of events. An Event Channel transparently permits (1) suppliers to send notifications of events and (2) consumers to receive these notifications, all without knowledge of the existence of one another. Hence, the Event Service will support the transparent replication of MSHN system components for reliability and dependability. Event Service will enable Client Libraries, linked with different concurrent applications, to communicate with other MSHN components seamlessly. Finally, Event Service supports a standard Application Programming Interface (API) (e.g., for the Push-Push Model, a single operation `push()` taking a variable of type `any` as a parameter) which eases the development of MSHN system components.

Though there are four models for Event Service, there were only two available in relatively robust industrial implementations when we performed our experiments: the Push-Push Model and the Pull-Pull Model [14]. Using the Pull-Pull Model creates an additional load on the consumers. Because our servers, the consumers in this case, must minimize their use of computing resources even when there is no event to be delivered on the Event Channel, we chose to use only the Push-Push Model.

3.2.1 Using Event Service in MSHN

Figure 9 illustrates the use of Event Service to organize communication in the MSHN architecture. In this approach, the components of MSHN must register themselves as both a consumer and a supplier to the Event Channel. The Event Channel acts as the glue between all of the components and delivers notifications to each of them.

3.2.2 Problems with Initial Approach

Although this approach helps to organize MSHN's communication, providing transparent reliability and scalability, some problems can be seen involving both performance and the CORBA 2.2 specification. Some of the problems with this approach are identical to the problems identified by Schmidt and Vinoski in the analysis of their stock market application [11]. We first summarize their findings in the first two items below, Loss of Events in the System, and Problems with the Untyped Event Service. Then we enumerate additional problems that are particular to using CORBA within the MSHN architecture. Lastly, we look at how to implement a component that is both a supplier and a consumer.

Loss of Events in the System. Event Service guarantees delivery of notifications to all registered consumers as long as the Event Service process does not fail[6]. However, in the Event Service specification, persistency of events in the Event Channel is not required. Therefore, if an Event Service process does fail, undelivered notifications in the system may be lost.

The loss of notifications is fatal for MSHN because we are creating an environment for mission-critical applications. The obvious solution to this problem is to

[5] The capabilities of the various implementations of CORBA evolve very quickly. The content of these figures present the state of some of the implementations at the time this research was performed. As the capabilities of most CORBA implementations can quickly change, the reader is recommended to do his own similar exploration.

[6] Although there are many definitions of failure, we specifically mean that if the Event Service does not fail, then all consumers receive the correct value. This agrees with Lamport's definition of failure [15].

Vendor	Naming	Life Cycle	Event	Trading	Identity	Relationships
Expersoft	yes		yes			yes
Sun	yes	yes	yes		yes	yes
IONA	yes		yes	yes		
Visigenic	yes		yes			
BEA						
ICL			yes			
HP	yes	yes	yes	yes		
IBM	yes	yes	yes		yes	
Chorus						
OOT	yes			yes		
Electra	yes	yes	yes			
Xerox						
BBN	yes	yes				

Figure 7: Available Services

Vendor	Concurrency	Externalization	Persistency	Transactions	Security
Expersoft					
Sun					
IONA				yes	
Visigenic				yes	
BEA					yes
ICL				yes	yes
HP				yes	
IBM	yes	yes	yes	yes	
Chorus					
OOT		yes			
Electra					
Xerox					
BBN			yes		

Figure 8: Available Services (Continued)

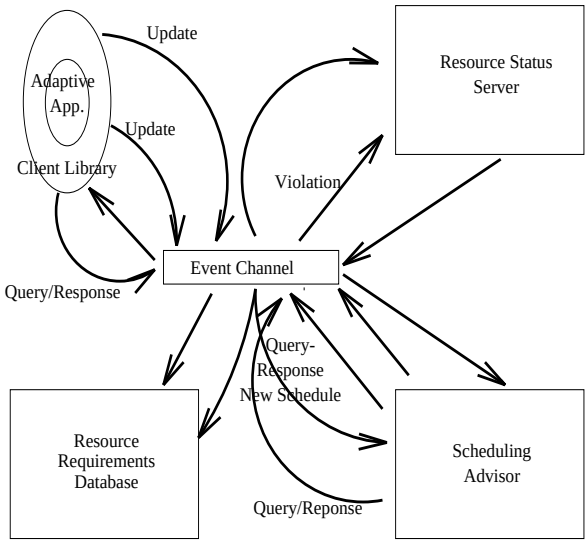

Figure 9: Using Event Service in MSHN

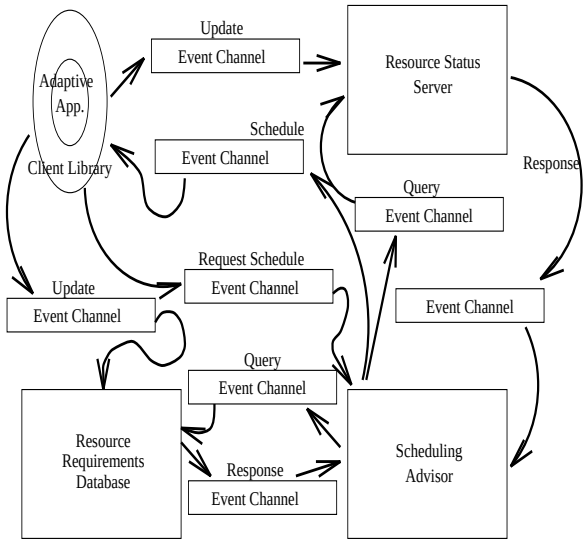

Figure 10: Using UntypedEvent Service

redefine the Event Service specification to include persistency for the undelivered notifications in the Event Channel. The OMG has been defining this requirement in the Notification Service specification [7]. However, no vendors had implemented this new specification at the time of this research.

Problems with Untyped Event Service. The Untyped Event Service does not specify any way to filter notifications. Therefore when using this service, all notifications are received by all registered consumers.

Passing all of these notifications in MSHN, many of which will be discarded by any particular consumer, through the network will increase the network load between the Event Channel and the consumer. Additionally, the consumers must filter events and convert the parameters that have type `any` to the type that is expected. In this case, there is an additional and unwanted load on the consumers to process all the events received. Finally, when more suppliers, in particular more applications, register with the Untyped Event Channel, more events will be generated in the system. Since the Untyped Event Channel delivers each event to all of the registered consumers and the consumers will filter all the events, the network load and consumer load will increase rapidly.

To handle this problem, we can use Typed Event Channels which filter the notifications according to their type. With this solution, the consumers receive only the notifications for which they register, decreasing the network traffic. In this solution, one Event Channel processes all of the notifications and delivers them only to the corresponding consumers. This also lightens the loads on the consumers because they avoid having to examine and discard events not meant for them. However, we note that it increases the computational load on the Event Channel. Later, we compare the run-time performance of Typed Event Channel to Untyped Event Channel using this approach in the MSHN architecture.

Alternatively, since we only have five different types of components in MSHN, we could use different channels for each connection between these components. In this approach, each Event Channel will only support one notification type. For example, for the Client Library - Scheduling Advisor Event Channel, we will have the Client Library as a supplier, the Scheduling Advisor as a consumer, and the possible client scheduling requests as the types of the notifications. Each MSHN component may be replicated by registering the additional (identical) components to the same Event Channel. This solution is shown in Figure 10.

Obviously, some combination of these two solutions may be best. That is, the Typed Event Channel itself can become a bottleneck in the first solution. Therefore, replication of Typed Event Channels may better fit MSHN's requirements. In this paper, we focused on the careful analysis of individual solutions rather than empirically exploring the exponentially sized solution space that combining these two techniques will create.

How to implement a component that is both a supplier and a consumer in a system in order to minimize the run-time overhead. All components of MSHN are both consumers and suppliers. Also,

and perhaps particular to MSHN, when a component receives a notification, it usually becomes a supplier by generating another notification and delivering it to the appropriate Event Channel. Figure 11 shows the process of passing notifications from the Client Library to the Scheduling Advisor using the `push()` operation. It reveals how the Scheduling Advisor changes from a consumer to a supplier. In the Untyped Event Service's Push-Push Model, the supplier (here the Client Library) invokes a default `push()` operation on the Event Channel which in turn invokes a `push()` operation supplied by the developer of the consumer (here the Scheduling Advisor). In the `push()` operation that the developer supplied for the Scheduling Advisor (as a consumer), the developer of the Scheduling Advisor invokes the default push operation on the Scheduling Advisor – Resource Requirement Database (SA – RRD) Event Channel (which of course, invokes the `push()` operation supplied by the developer of the Resource Requirements Database).

The design issue here is to determine how to supply the Interoperable Object Reference (IOR) of the SA – RRD Event Channel to the `push()` operation of the Scheduling Advisor. We want to avoid using the Naming Service every time the `push()` operation (here the push operation of the Scheduling Advisor) is invoked. Instead, the developer can locate the SA–RRD Event Channel in the servant implementation. That is, the servant implementation will obtain the IOR for the SA–RRD Event Channel, stringify the IOR, and storing it in a file. The `push()` operation implementation can retrieve these IORs from their files, as needed, and deliver generated events, thereby pushing the corresponding notifications to the channel.

Therefore in the Untyped Event Service, to react to the notification (here a request for a schedule) that the consumer receives, the developer of the consumer (here the Scheduling Advisor) must override the default `push()` operation between the Event Channel and the consumer. For example, when the Scheduling Advisor receives an event from the Client Library requesting a schedule, it will generate a query notification for the Resource Requirement Database and deliver it to the SA – RRD Event Channel. In this case, the Scheduling Advisor becomes a supplier and is required to locating the SA – RRD Event Channel. To avoid locating the Event Channel to which the supplier will deliver the notification, via the Naming Service inside the `push()` operation, the developer can locate the Event Channel in the servant implementation and obtain IORs of it. Then, the servant implementation can stringify these IORs and store them in files.

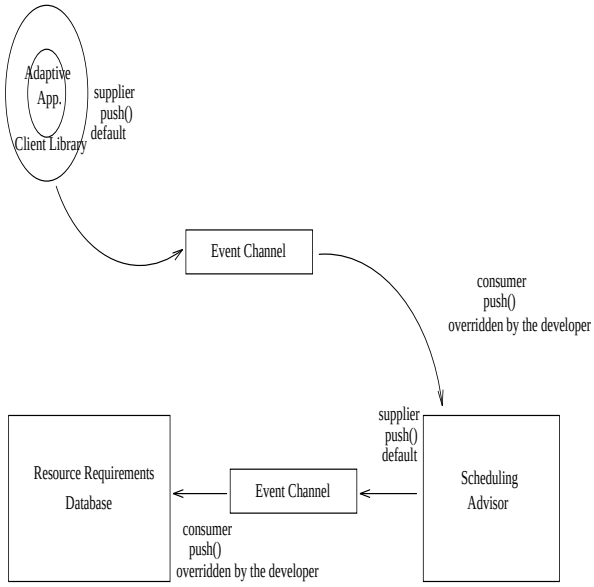

Figure 11: Using `push()` Operation

3.3 Remote Invocations

In this section, we discuss using remote invocations to coordinate the interactions of MSHN's components. Since both the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII) have similar remote invocation mechanisms, we first define the general problems encountered with both, and then enumerate any additional ones that are specific to the DII.

The same functionality described above using the Event Service can be implemented using remote invocation. The most important difference is that the replication of the components is not as easy as it is using Event Service. To support replication using remote invocation, clients must make multiple invocations rather than just the one needed in Event Service.

3.3.1 General Approach using Remote Invocation

Figure 12 shows our approach that uses remote invocations (i.e., either the Static Invocation Interface (SII) or the Dynamic Invocation Interface (DII)) to establish inter-component communication in the MSHN architecture. We chose from two communication methods available in both the SII and DII: one-way invocation and synchronous invocation, depending upon whether reliable communication is required.

When using the SII, a component requires compile-time knowledge of the Interface Description Language (IDL) interface of the target component from which it will request a service. In contrast, the same component, using the Event Service, makes its request via a

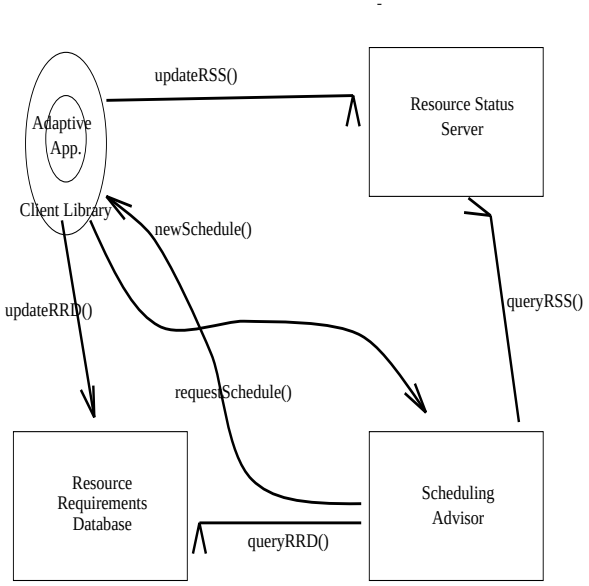

Figure 12: Using Remote Invocations in MSHN

standard API that is independent of the target component and its functionality. However, when using the DII, the components of MSHN can invoke operations on other components without requiring precompiled stubs. Thus, we may substitute different instantiations of such components without requiring a re-linking. Additionally, using the DII allows us to invoke objects using deferred synchronous invocation. Such invocation is not available from the SII within the current CORBA 2.2 specification. With deferred synchronous invocation, the clients may continue their computation instead of waiting for the results of the previously invoked operations to be delivered.

3.3.2 Problems with Using the Initial Remote Invocation Approach

We now enumerate some problems with our initial remote invocation approach.

Lack of a Standard Thread Mechanism. Our first design decision was to implement the remote invocations with threads, i.e., handling each invocation of a component using a different thread. Using threads would avoid any data synchronization problems and support fairness for each schedule request. However, the CORBA 2.2 specification does not define how the threads must be implemented. Therefore, each vendor has come up with their own solution, leading to applications that are non-portable. For example, if you use IONA's Orbix as your development environment, and IONA's Filters to implement your threads, you cannot use the same implementation on Inprise's Visibroker because Inprise's solution for handling threads uses Interceptors.

We avoided non-compliant extensions of the vendor when implementing our prototypes. Therefore, we were unable to use threads for any of our prototypes, although the usage of threads would have improved the throughput of schedule requests.

Best-Effort Semantics. One-way invocation has best-effort semantics. Thus, there is no guarantee that the requested method is actually invoked. In this mechanism, the client continues its processing immediately after initializing the request and never synchronizes with the completion of the request. Hence, one-way invocation is not a good mechanism for most of the MSHN system because it is not reliable.

However, using one-way invocations for frequent short-term updates could be cost effective in some cases in MSHN. There are two advantages to selectively using best-effort asynchronous semantics between MSHN's Client Library and Resource Status Server. First, the Client Library can continue its computation immediately without blocking. Second, we expect that the Resource Status Server will be updated very frequently. Therefore, we can afford the delay needed to get the accurate status of a resource with the next update instead of forcing the use of a more reliable transmission mechanism.

3.3.3 Problems with Our Initial Approach that are Specific to using DII

We now enumerate some problems with our initial approach that are specific to using DII.

The Additional Overhead of the DII. A straight forward DII approach requires 5-6 method invocations in order to invoke a single remote method: looking up the interface name, getting the operation identifier/parameters, and creating the request (which may also be remote). This would add a lot of overhead to run-time performance, which would be unacceptable in MSHN's architecture.

In MSHN however, we know the interface of the components, i.e., the operation identifier, the parameters and the return type, when we are developing the client applications. Thus, we can obtain the flexibility and benefits the DII's deferred synchronous invocation, without having to pay the overhead of querying the Interface Repository for the interface information. We do note that if a deferred synchronous invocation, such as Promises [16], had been specified as part of CORBA's static invocation interface, the use of DII would not be necessary in this case. We compare the performance of the SII and DII in the results section.

3.4 Using the Naming Service

We used the Naming Service to obtain object references in each of our prototypes. For the static and dynamic invocation interfaces, all components must resolve names only once, when they are instantiated, to obtain IORs via the Naming Service. References within all components, except the Client Library, are stored in files for future use as we described previously. The components do not use the Naming Service unless the IORs that they have are no longer valid. We use the exception handling mechanism in CORBA to catch non-valid IORs, and then use the Naming Service to obtain new valid ones.

To improve the run-time performance of the Event Service implementations, we registered each component with the appropriate Event Channel. We resolve the Event Channel references using the Naming Service. Then we query the Event Channels to obtain the references for the Proxy Push Suppliers, stringify them, and then store them in files. When a component receives an event, and generates another event in response to the one it received, that component reads the appropriate file to obtain the stringified reference and uses this reference to push the event to the corresponding Event Channel.

4 Quantitative Results

We described our design decisions for implementing our prototypes in the previous section. In this section, we discuss the performance results of these different prototypes. First, we describe our test bed. Then we explain our tests and enumerate their results.

4.1 Hardware and Software Used in the Test Bed

As discussed earlier at the beginning of this research, we surveyed the available implementations of CORBA to determine what services were supported. (See Figure 7 and 8.) Based upon the robustness and availability of services, particularly the Typed Event Service, we chose IONA Technologies' CORBA implementation, specifically OrbixMT2.3c, OrbixNames1.1c, OrbixEvent1.0c (Untyped Event Service) and OrbixEvent1.0b (Typed Event Service) built using the SunSparc C++ Compiler 4.1.

We ran our tests on SunSparc Station 10 hosts with 300MHz CPUs and 128 MB of RAM each, running the Solaris 2.6 operating system. The hosts were connected via a 100 Mbits/sec Ethernet LAN.

To obtain correct results in the tests utilizing the network, we used the Network Time Protocol to synchronize the system clocks of the hosts. We found that the system clock on the SunSparc 10 has a skew of approximately 3 milliseconds every 15 minutes. Therefore in order to minimize the difference between the various system clocks, we synchronized the clocks every 5 minutes and ran the tests immediately after the synchronization.

4.2 Experiments

We determined the overhead of each CORBA mechanism on a single machine, and then measured the response times over the network of the various mechanisms, that is, the total time required to service 1000 scheduling requests. This interval begins when the Client Library requests a schedule from the Scheduling Advisor and includes all processing up until the time that the Client Library receives a response. This duration includes the time spent querying the Resource Requirement Database and the Resource Status Server. At the time of this testing, we did not have a fully functional Scheduling Advisor, so we emulated its execution by having the thread that was computing a schedule pause for .5 seconds. We chose this duration based upon the average execution time of a set of 11 scheduling algorithms proposed for MSHN's repertoire by Siegel [17].

To assess the overhead of CORBA, we included one non-CORBA test. This base case consists of an application linked with all the MSHN components and executing as a single process on a single host. This non-CORBA test uses local method invocation to perform MSHN component intercommunication. In order to assess CORBA's overhead, we performed two sets of tests. In the first set, we compared this base case against test cases where we ran all the MSHN components on the same machine and had them communicate via CORBA mechanisms. In the second, we compared the latter tests against ones where the MSHN components are distributed across different machines.

With the exception of the non-CORBA base case, we ran all tests both on a single machine and over the network using different workstations to execute each of the Client Library, the Resource Status Server, the Resource Requirements Database and the Scheduling Advisor.

All single machine CORBA tests were executed using four different processes. The non-CORBA single machine tests executed completely in a single process, with all MSHN calls being implemented as ordinary C++ function calls. In implementing both static invocation and dynamic invocation for a single machine, we used synchronous semantics.

The average inter-arrival rate of schedule requests varies with the facility and time of day. Therefore, we ran all of our tests for two different circumstances. In the first, the inter-arrival rate of the requests is less than the service time, i.e., each request is completed

by the system before the next request arrives on average. The second represents the situation that exists in the middle of a burst. In this case, the inter-arrival rate of the requests is greater than the service time, i.e., some requests must be queued to be handled later. The first case is important in determining performance under normal conditions, but it is equally important for us to determine that the system neither (1) fails completely when heavily loaded, nor (2) incurs overhead that varies exponentially with the number of requests pending. Indeed, no typed event service that we have tested to date could pass the above stress tests.

Unfortunately, the system clocks had insufficient granularity to measure precisely the total time to process a single request in our non-CORBA implementation. We therefore first read the system clock. We then generate a request and await its response, repeating this 1000 times. Lastly, we read the clock again, and determine the total time (for 1000 consecutive request-response pairs). Because requests are generated consecutively, and because each request uses synchronous semantics to make the invocations, we call this set of tests, the **consecutive synchronous tests**.

To simulate the case where many requests occur within a short time frame, we generated requests every .06 seconds, on average, in our base case. For this set of tests, we used asynchronous calls within the application to start the schedule request chain in the DII and SII implementations. Event Service is meant to be used asynchronously, so there was no special programming required to implement these cases. We call this set the **bursty asynchronous** tests because during such a burst, the requests arrive faster than the expected required service time and queue up for the Scheduling Advisor.

For another of our projects, Schnaidt and Duman implemented a fully optimized version of an application using sockets and compared it to an equivalent CORBA implementation to determine CORBA's overheads when running over the network [18]. As such, we did not implement such a socket implementation of MSHN. In the following paragraphs, we draw some conclusions based both on the Schnaidt-Duman experiments and those reported here.

4.3 Results

We summarize our quantitative results in Figure 13. The times shown are the actual execution times, in seconds, for 1000 requests. We have included a scheduling time of .5 seconds per request and have not simulated the execution time of the application.

In order to fully understand these results, we must first explain some anomalies that we observed in the Unix calls we used to emulate the Scheduling Advisor

Config.	Communication Mechanism	Local	Network
Consec. Synch.	Non-CORBA	500.1	N/A
	SII	511.4	520.0
	DII	530.1	530.4
	Untyped Event	607.4	593.9
	Typed Event	580.5	779.2
Bursty Asynch.	Non-CORBA	500.1	N/A
	SII	510.8	510.8
	DII	521.2	520.2
	Untyped Event	592.8	564.4
	Typed Event (for 100 requests)	64.7	63.6

Figure 13: Results of the Generic Experiments for 1000 Requests

(`select()`) and the request generation inter-arrival rate (`ualarm()`). The average of the actual `select()` times was 125 microseconds more than the requested .5 seconds. We also observed an average of 10 milliseconds error for the `ualarm()` requests of 60 milliseconds.

As expected, there is significant overhead in using CORBA for communication, and therefore across more than one address space, as compared to local invocations within a single address space. In our earlier project, we noted similar results as well as substantial overhead when an optimized non-CORBA local socket implementation was compared to a local CORBA implementation [18]. The efficiency of the socket implementation on a single machine is due to its use of shared memory. However, even if a CORBA implementation used shared memory, comparable performance would not be obtained. Unfortunately, the CORBA specification requires all parameters of a request to be converted to an external, machine independent data representation, even if the target object resides on the same machine. Also, in that earlier project, we noted that a networked CORBA implementation, which required less than 5% of the time to implement as compared to the socket implementation, had only 20% more runtime overhead. Since our results are comparable here, and because we did not implement a highly optimized MSHN socket implementation, we will limit the remainder of our remarks to comparing the performance of various CORBA implementations of MSHN.

Static invocation is generally the fastest intercommunication mechanism available in CORBA [1]. Even though dynamic invocation is generally much slower, we see that the performance of dynamic invocation, when we know the interfaces at development time, is close

Communication Mechanism	Added Overhead
SII	10.5
DII	20.0
Untyped Event	64.2
Typed Event	13.5

Figure 14: Added Overhead for Bursty Asynchronous Test Case over the Network

to that of static invocation. However, we note that the most efficient implementation would likely be available from a deferred synchronous Static Invocation Interface. We recommend that such semantics, similar to those in Promises [16], be considered for adoption into the CORBA specification.

The comparison between the consecutive synchronous and bursty asynchronous tests seems surprising at first glance. One would normally expect that a system loaded with bursty requests would not perform better than an unloaded system. To understand the reason for this performance improvement, we must further elaborate on the client application's use of the Naming Service. In the consecutive case, the Client Library obtains the reference of the Scheduling Advisor from the Naming Service immediately prior to making each request. However, in the bursty asynchronous case, the Client Library obtains all of the references asynchronously. Thus in the bursty asynchronous case, obtaining these references overlaps with the actual computation. Unfortunately, we will only expect to see this improvement in the actual MSHN implementation if the Scheduling Advisor is executing on a dual processor machine. In our experiments, the emulated Scheduling Advisor is actually blocked while the Naming Service is resolving addresses.

In the 4-machine network tests, the number of context switches required between MSHN's components and the Object Request Broker is substantially reduced. Multiple components actually execute simultaneously, and thus run-times were smaller.

As seen in Figure 13, the Untyped Event Service adds more overhead than either static or dynamic invocation because the Event Service process is the bottleneck in the system. Of course in an overall evaluation, this additional overhead must be balanced against the reduced cost with which information can be delivered to replicated system components.

In addition to the tests described above, we replicated the Untyped Event Service to see whether any speedup could be obtained by distributing the load of the Event Service process. First we created two Event Service processes, one on the same host as the application and the other on the same host as the Scheduling Advisor, in an attempt to achieve some speed up. This approach performed worse than the single Event Service process. Upon analysis, we determined that it introduced unnecessary network communication and placed the Event Service processes on the busiest hosts. Then we moved the Event Service processes to the same hosts as the Resource Requirements Database and the Resource Status Server. Figure 15 shows the speedup we observed with this configuration. We also ran tests using four distributed Event Service processes. Unfortunately, probably because of the excessive amount of communication, this approach performed no better than using a single Event Service process.

In MSHN's Typed Event Service implementation, all of the communication passes through a single process. The CORBA implementations that we used[7] failed in this bursty asynchronous case. In Figure 13, we include the time required to process 100 requests for the bursty asynchronous case. Since the current implementations of Typed Event Service do not allow replication, we could not run a replicated test with the Typed Event Service as we did with the Untyped Event Service. Hence, we believe that the Typed Event Service is not ready to be used in middleware to support heterogeneous distributed computing.

5 Conclusions

In this paper, we described our experiences using mechanisms of the CORBA 2.2 specification to facilitate communication in a resource management system that is both designed to manage distributed heterogeneous applications, and is itself distributed and heterogeneous. In our qualitative assessment of CORBA 2.2, we found several minor problems and recommended the addition of deferred asynchronous semantics to CORBA's Static Invocation Interface. We found that both CORBA's static invocation and dynamic invocation, when used solely to obtain asynchronous semantics, were efficient enough to support distributed heterogeneous resource management systems. We found that substantial work is needed to provide implementations of Typed Event Services that can handle the loads placed on them when requests occur in a bursty fashion. We also determined that while Untyped Event Services add substantial overhead as compared to static invocation, they may still be desirable in the case where multicast of requests is desired, particularly if they are replicated and themselves wisely allocated to machines in the system. In summary, many of the

[7] Typed Event Service is new in the CORBA 2.2 specification and not many CORBA products have this service available as yet.

	Replication Mechanism	All	SA and Client Hosts	RRD and RSS Hosts
Bursty Asynch.	Two Event Pro.	N/A	574.38	561.44
	Four Event Pro.	560.98	N/A	N/A
Consec. Synch.	Two Event Pro.	N/A	599.05	593.82
	Four Event Pro.	593.82	N/A	N/A

Figure 15: Results of the Untyped Event Service Special Cases

existing CORBA services can be quite useful in implementing resource management systems for heterogeneous computing, and other CORBA services hold substantial promise for the future.

6 Acknowledgements

The authors would like to thank Ted Lewis for his suggestions during this research as well as for sharing his broad understanding of the motivation behind the CORBA specifications.

References

[1] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. John Wiley, New York, 1997.

[2] Robert Orfali, Dan Harkey, and Jeri Edwards. *Distributed Objects*. John Wiley, New York, 1997.

[3] Sean Baker. *CORBA Distributed Objects Using Orbix*. Addison Wesley Longman Limited, Essex, 1997.

[4] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide*. John Wiley, New York, 1997.

[5] IONA Technologies PLC. *Orbix Programmer's Reference Manual*, October 1997.

[6] IONA Technologies PLC. *Orbix Programmer's Guide*, October 1997.

[7] Object Management Group. *CORBA 2.2 Specification*, February 1998.

[8] Object Management Group. *Naming Service Specification*, November 1997.

[9] Object Management Group. *Event Service Specification*, November 1997.

[10] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[11] Douglas C. Schmidt and Steve Vinoski. Overcoming drawbacks in the OMG events service (column 10). *SIGS C++ Report Magazine*, June 1997.

[12] Debra A. Hensgen, Taylor Kidd, Matthew C. Schnaidt, David St. John, Howard Jay Siegel, Tracy D. Braun, Muthucumaru Maheswaran, Shoukat Ali, Jong-Kook Kim, Cynthia Irvine, Tim Levin, Roger Wright, Richard F. Freund, Michael Godfrey, Alpay Duman, Paul Carff, Shirley Kidd, Viktor Prasanna, Prashanth Bhat, and Ammar Alhussaini. An overview of MSHN: A Management System for Heterogeneous Networks. In *8th. IEEE Workshop on Heterogeneous Computing Systems (HCW'99)*, San Juan, Puerto Rico, April 1999. IEEE, IEEE. invited.

[13] John Kresho. Quality network load information improves performance of adaptive applications. Master's thesis, Naval Postgraduate School, September 1997.

[14] IONA Technologies PLC. *OrbixEvents Programmer's Guide*, December 1997.

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[16] B. Liskov and L Shirira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *Proceedings SIGPLAN'88 Conference Programming Design and Implementation*, 1988.

[17] Tracy D. Braun, Muthucumaru Maheswaran, Howard Jay Siegel, Noah Beck, Ladislau Boloni, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, and Bin Yao. A taxomony for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. *Proceedings of the Workshop on Advances in Parallel and Distributed Systems (ADAPS)*, 1998.

[18] Matthew Schnaidt and Alpay Duman. A comparison of Unix sockets and CORBA in a distributed communication intensive application. Technical Report 3, Naval Postgraduate School, 1998.

Alpay Duman is LTJG in Turkish Navy. He graduated from the Turkish Naval Academy getting his BS in Operations Research with honor degree. He received his Ms.Cs. degree in the area of Systems Design and Architecture from the Naval Postgraduate School. He investigated the use and runtime overhead of CORBA

in DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) with Dr. Debra Hensgen and Dr. Ted Lewis. He is currently a systems engineer at Turkish Navy Software Development Center working on a CORBA based communication infrastructure for Command Control Systems. His area of interest are fault tolerant, real-time, CORBA based distributed systems.

Debra Hensgen is an Associate Professor in the Computer Science Department at The Naval Postgraduate School. She received her PhD in the area of Distributed Operating Systems from the University of Kentucky. She is currently a Principal Investigator of the DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) and a co-investigator of the DARPA-sponsored Server and Active Agent Management (SAAM) Next Generation Internet project. Her areas of interest include active modeling in resource management systems, network rerouting to preserve quality of service guarantees, visualization tools for performance debugging of parallel and distributed systems, and methods for aggregating sensor information. She has published numerous papers concerning her contributions to the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems

David St. John is the head of staff at the Heterogeneous Network and Computing Laboratory. He plays a strong role in development of the MSHN prototype and in directing students' research at the Naval Postgraduate School. He has over six years experience in object-oriented software development primarily for process control, sensor collection, and Internet transaction processing systems. He is a member of IEEE and IEEE Computer Society. He was a recipient of the Chancellor's Fellowship and an MS degree in Engineering from the University of California, Irvine in 1994. He received his BS degree in Mechanical Engineering from the University of Florida in 1992.

Taylor Kidd obtained his Ph.D. from the University of California at San Diego in 1991. He is an active researcher in theoretical and applied distributed computing. As part of the SmartNet Team at NRaD he led the SmartNet Research Team. Recently he joined Debra Hensgen as co-director of the Heterogeneous Network and Computing Laboratory. While part of the SmartNet Team, he instigated a number of important advances to the SmartNet Scheduling Framework, including enhancing compute characteristic learning by using Student-T techniques, performing experiments and simulations exploring the performance of different RMS's in homogeneous and heterogeneous environments, and fundamentally changing the way SmartNet learns and schedules by recognizing the basic predictable uncertainty of job run times. He has served on the program committees of several conferences and has worked as an acting subject area editor for JPDC. Most recently, he is working under DARPA's Quorum program as a co-PI for MSHN and as a co- investigator on the SAAM Project.