Designing an Efficient and Scalable Server-side Asynchrony Model for CORBA

Darrell Brunsch, Carlos O'Ryan, and Douglas C. Schmidt
{brunsch, coryan, schmidt}@uci.edu
Department of Computer Engineering
University of California, Irvine
Irvine, CA 92612, USA

Abstract

When the Asynchronous Method Invocation (AMI) model was introduced into the CORBA specification, client applications benefited from the ability to invoke non-blocking two-way requests. In particular, AMI improved the scalability of clients by removing the restrictions associated with Synchronous Method Invocations (SMI). Server request handling remained synchronous, however, which minimized the benefits of AMI for middle-tier servers, such as firewall gateways and front-end database servers.

This paper describes our strategy for implementing a scalable server-side asynchrony model for CORBA. We first outline the key design challenges faced when developing an Asynchronous Method Handling (AMH) model for CORBA and then describe how we are resolving these challenges in TAO, our high-performance, real-time CORBA ORB. In general, AMH-based CORBA servers provide more scalability than existing concurrency models, with only a moderate increase in programming complexity. Although targeted for CORBA, similar techniques can also be used in other method-oriented middleware, such as COM+ and Java RMI.

1. Introduction

Background

Most first-generation implementations of distributed object computing middleware, such as CORBA [OMG 2000a], DCOM [Box 1997], and Java RMI [Wollrath et al. 1996], supported reliable communication solely through S*ynchronous Method Invocation* (SMI) models. In an SMI model, a client invokes an operation through a local proxy, which:

1. Sends the parameters to the target object and blocks waiting for the reply
2. Returns control to the client thread along with the response

Figure 1 illustrates a typical two-way SMI interaction between a client and server.[1]

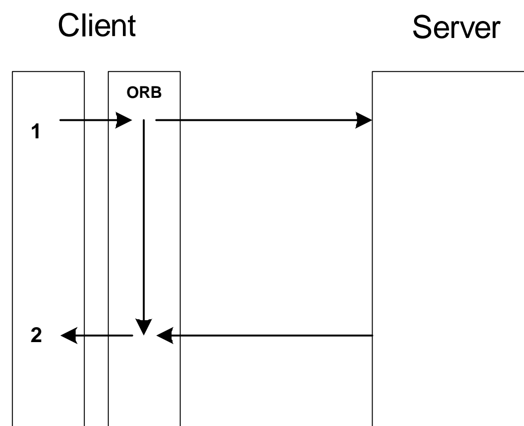

Figure 1: The SMI Model. (1) The client invokes the operation and the ORB blocks. (2) After the response is returned, the ORB returns control to the client application thread that invoked the operation.

Although SMI is a common invocation model, it does not scale well due to its need for a separate client thread to execute for each concurrent two-way SMI call. Moreover, SMI is not well suited for real-time or interactive applications that must remain responsive while waiting for replies to arrive from servers [Arulanthu et al. 2000a].

Recent changes to the CORBA specification [OMG 2000a] have added a new type of invocation model known as A*synchronous Method Invocation* (AMI) [Arulanthu et al. 2000b]. By separating requests from responses in time and space, AMI enables more scalable and responsive clients compared with conventional SMI clients. The two variants of AMI are *polling* and *callbacks*, as shown in Figure 2 and Figure 3.

The polling model shown in Figure 2 is characterized by a `Poller` object the client can use to check the status of the request.

[1] In the context of this paper, we assume all operations are two-way operations.

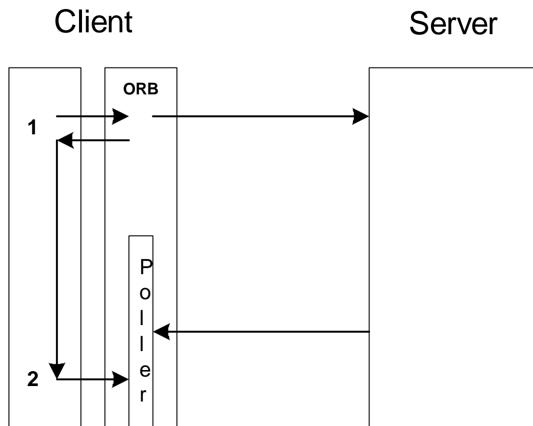

Figure 2: The AMI Polling Model. (1) The client invokes the operation and the call returns immediately. (2) It later checks with the collocated Poller object to retrieve the response.

The client uses the `Poller` object to check for a returned response or to block until the response is available. The callback model shown in Figure 3 differs in that the client will pass a *reply handler object* with the request. This object will be called back by the ORB to notify the application of an available response [Arulanthu et al. 2000b].

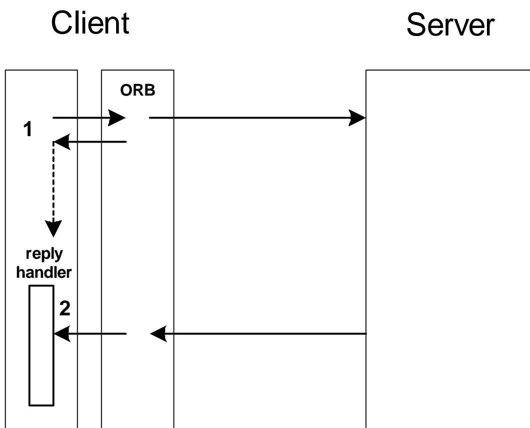

Figure 3: The AMI Callback Model. (1) The client invokes the operation and the call returns immediately. (2) The ORB later invokes the callback when the response arrives.

These two AMI models allow a client to make multiple requests on remote objects residing on one or more servers without blocking synchronously for responses. Moreover, AMI allows clients to initiate many long-running method invocations concurrently, without incurring the overhead associated with using a separate thread for each request.

An interesting aspect of the CORBA AMI specification is that it does not affect server design or behavior. AMI only affects the behavior of clients by decoupling the sending of the request and the receiving of the response. This design allows a client to make requests asynchronously, but does not (by itself) enable a server to handle the requests asynchronously.

Limitations with CORBA AMI for Middle-tier Servers

For many types of systems, CORBA AMI improves concurrency, scalability, and responsiveness significantly. Since AMI allows a client to invoke multiple two-way requests without waiting for responses, the client can use the time normally spent waiting for replies to perform other useful work. These capabilities are sufficient for a wide range of applications, particularly two-tier client/server applications.

There is another important class of systems where the standard CORBA AMI capabilities are insufficient. These systems have multi-tier architectures, such as the three-tier structure found in many distributed business systems [Eckerson 1995]. In a multi-tier system, one or more "middle-tier" servers are placed between a *source client* and a *sink server*. A source client's two-way request may visit multiple middle-tier servers before it reaches its sink server; the result then flows in reverse through these intermediary servers before arriving back at the source client.

Figure 4 illustrates a multi-tier architecture involving a middle-tier server and several source clients and sink servers.

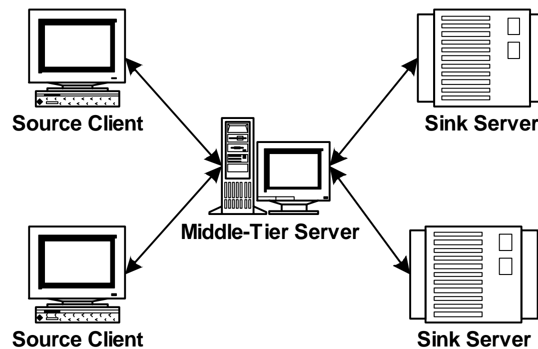

Figure 4: A Three-tier Client/Server Architecture

The three-tier client/server architecture shown in this figure can be used for many types of systems. For example, the middle-tier server could be a firewall gateway that validates requests from external clients before forwarding them to sink servers. Likewise, the middle-tier server could be a load balancer that distributes access to a group of database

servers [Othman et. al. 2001]. In both cases, the middle-tier servers act as *intermediaries* that accept requests from a client and then pass the requests on to another server or external data source. When an intermediary receives a response, it sends its own response back to the source client. The general behavior of a middle-tier server is summarized in Figure 5.

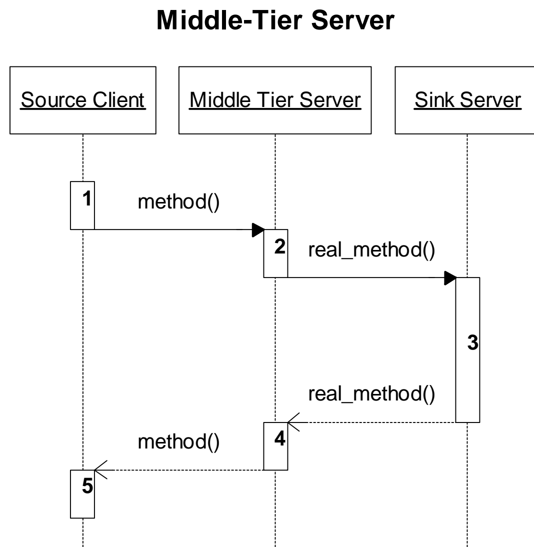

Figure 5: Typical Middle-tier Server Steps: (1) client sends request, (2) middle-tier processes the request and sends a new request to a sink server, (3) sink server processes and returns data, and (4) middle-tier returns data to (5) the client.

Unlike ordinary source clients, middle-tier servers must communicate with multiple source clients and sink servers. They must therefore be highly scalable to avoid becoming a bottleneck. Due to the cost of thread creation, context switching, synchronization, and data movement, it may not be scalable to dedicate a separate thread for each outstanding client request.

The overhead of threads motivates the need for another way to increase middle-tier server scalability. Unfortunately, these servers cannot leverage the benefits of AMI fully since AMI only provides asynchrony on the client side of a request. In a middle-tier server, therefore, outgoing requests can use AMI to return program control from the ORB quickly, but the servant for the incoming request must remain on the stack of activation records (*i.e.,* control cannot return back to the ORB) until a response can be returned to the source client.

Properties of an Ideal Solution for Middle-tier Servers

Figure 5 shows the sequence of steps performed per-request by a middle-tier server. The point where asynchrony can provide the most benefit is step 3, *i.e., send a new request to another server or data source and wait to receive the response*. Depending on the speed of the sink server or data source, considerable time could be spent blocked waiting for a response. This waiting reduces the scalability of the middle-tier server since it must dedicate a thread for the duration of the request/response cycle.

An ideal solution to this problem should have the following properties:

- **Request throughput**: A solution should provide high throughput for a client, *i.e.,* it should be able to handle a large number of requests per unit time, *e.g.,* per second or per "busy hour." A solution that serializes or minimizes incoming requests can degrade throughput.
- **Latency/Jitter**: A solution should minimize the request/response processing delay (latency), as well as the variation of the delay (jitter).
- **Scalability**: A solution should take advantage of multiple sink servers and handle many aggregate requests/responses.
- **Portability**: Ideally, little or no changes and non-portable features should be required to implement a scalable solution.
- **Simplicity**: Compared with existing SMI designs, the solution should minimize the amount of work needed to implement scalable middle-tier servers. In addition, any ORB features required by the solution should be easy to implement.

Although it is hard to achieve all these properties in a single design, we present this ideal solution as an archetypical baseline for comparison.

Paper Organization

The remainder of this paper is organized as follows: Section 2 introduces our solution—*Asynchronous Method Handling* (AMH)—and outlines its benefits; Section 3 describes how we are addressing the key design challenges of AMH in *The ACE ORB* (TAO); Section 4 compares AMH with related work; and Section 5 presents concluding remarks.

2. An Overview of Asynchronous Method Handling for Middle-tier Servers

The difficulty of implementing a scalable middle-tier server is rooted in the tight coupling between a server's receiving a request and returning a response in the same activation record. This tight coupling limits a middle-tier server's ability to handle incoming requests and responses efficiently. In particular, each request needs its own activation record, which effectively restricts a request/response pair to a single thread in standard CORBA.

This restriction also affects the design of servers that depend on long-running asynchronous hardware tasks. These servers cannot take advantage of asynchrony of the hardware because the ORB and servants must handle requests synchronously.

A solution to this problem is to extend the same benefits of AMI to the server by employing Asynchronous Method Handling. By allowing the server to asynchronously return responses, we decouple the existing 1-to-1 association of an incoming request to the run-time stack originating from the activation record that received the request *without* incurring the overhead of multi-threading. This design can be seen in Figure 6, where the out parameters must be set when the servant upcall returns control to the ORB.

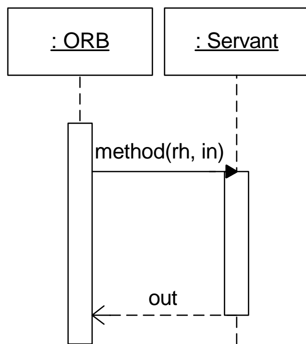

Figure 6: Current Method Handling Model

Adding AMH to a middle-tier server that also uses the AMI callback model yields a server that can take full advantage of asynchrony. Such an AMI/AMH server can act much like a message-oriented middleware (MOM) server, which is well suited for use in middle-tier servers. A middle-tier server's main task is to forward requests from source clients and responses from sink servers. A server using AMI and AMH can treat a method call as two separate messages: a request and a response. This type of architecture results in a more efficient design that does not require as much state in a middle-tier server as a method-oriented design.

Much as AMI allows a client application to provide a reply handler object to an ORB, AMH allows an ORB to provide a server application's servant with a response handler object that contains the context information needed to return a response to the appropriate client. The server application can return a response via this response handler either during the servant upcall or at some later point during the server's execution.

Figure 7 shows how a servant upcall in a middle-tier server receives a response handler object and "in" parameters.

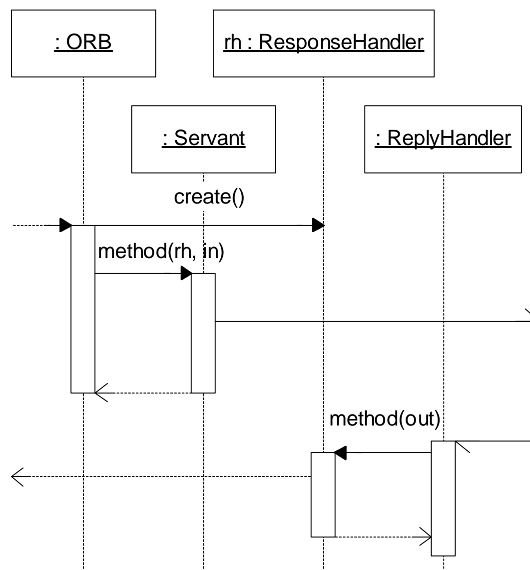

Figure 7: Asynchronous Method Handling Model

Using AMH, this middle-tier server can invoke an AMI call to a sink server and return control to the ORB without specifying the "out" parameters. When the response from the sink server becomes available, it is dispatched to the AMI callback handler, which can use the context information stored in the response handler to specify the "out" parameters and return a response to the client.

Below we evaluate the pros and cons for this AMI/AMH model in terms of the ideal solution we outlined towards the end of Section 1.

- **Request throughput**: The middle-tier server can provide very high throughput by handling multiple incoming requests from a client asynchronously.

- **Latency/Jitter:** When a request arrives, it is handled quickly, and when the response returns from the sink server, a reply can be sent back immediately. Latency should be relatively low since no additional threads need be created to handle requests and wait for responses. However, more state is required than in the simple single-threaded case, resulting in more context stored on the heap.
- **Scalability**: Scalability can be very high since the upcall for requests and callbacks on `ReplyHandler` objects need not block. Moreover, performance can be enhanced to take advantage of multiple CPUs by combining the AMI/AMH model with a thread pool [Schmidt 1998].
- **Portability**: AMH is not yet defined in a CORBA specification, nor is it implemented by many ORBs. TAO's implementation will include a proof-of-concept prototype.
- **Simplicity**: The server application becomes more complicated because application code uses both AMH and AMI. The ORB and IDL compiler also become more complicated because request lifetimes are decoupled from the lifetime of a servant upcall.

3. The Design of TAO's Asynchronous Method Handling Architecture

The design of TAO's AMH support is modeled after its AMI callback model implementation [Arulanthu et al. 2000b]. In the AMI callback model, the client creates a custom `ReplyHandler` object that it passes to the ORB when it uses the callback version of an operation. Methods on a `ReplyHandler` are then invoked by the client ORB when a reply is returned from the server.

TAO's AMH design uses a `ResponseHandler` that behaves much like an AMI `ReplyHandler`. When a server ORB invokes an upcall on a servant's method, it passes the `ResponseHandler` to the servant. The servant upcall method then has the following choices:

- *Immediate response* -- It can use this `Response Handler` during the upcall to send a reply immediately (which is akin to conventional two-way server behavior) or
- *Deferred response* -- It can save the `Response Handler` and defer the reply to a more suitable point in time, *i.e.,* after the original upcall returns back to the skeleton.

As with AMI, AMH requires changes to both IDL compilers and ORB internals. For example, for each IDL interface an IDL compiler must generate a POA skeleton class that contains signatures that take the appropriate `ResponseHandler` parameter. Likewise, an ORB must be extended to handle both the immediate and deferred response use-cases outlined above.

Below, we describe the challenges we encountered during the design of TAO's AMH support and outline our solution approach.[2]

Challenge 1: How to Process Asynchronous Responses Efficiently

Context: To support SMI-based servant upcalls, an ORB can make assumptions about the lifetime of objects needed for method handling. For example, ORBs can use automatic variables to control the lifetime of objects associated with method handling.

Problem: The introduction of AMH violates most of the SMI assumptions. In the above example, the lifetime of an object must be extended until a `ResponseHandler` is used to return a response (which can occur long after the upcall returns). Since assumptions like this are key to optimizations involving memory allocations and thread synchronization, new techniques must be devised to handle these issues efficiently.

Solution: Use the Strategy pattern [Gamma et al. 1995] to encapsulate different algorithms and interchange them easily. To minimize the impact of AMH support, the TAO ORB mechanisms that process incoming requests are controlled by a strategy. The ORB will normally be configured without support for AMH, thereby minimizing the performance and footprint impact of this extension.

Challenge 2: How to Minimize the Time/Space Cost of AMH Support on non-AMH Applications

Context: Only certain types of applications, such as middle-tier servers, will require AMH capabilities.

Problem: AMH requires changes to the ORB's use of servant upcalls. As described in Challenge 1, this change can affect the optimizations currently in place. As a result, non-AMH calls can also be affected. Ideally, only servants that make use of AMH should have to pay any penalties, such as additional dynamic memory allocators or footprint enlargement, related to this feature.

Solution: Use the Component Configurator pattern [Schmidt et al. 2000], which allows middleware or application developers to delay their configuration

[2] Note to reviewers: the final version of this paper will contain the results of benchmarks that will illustrate the benefits of AMH empirically.

decisions until run-time. In TAO, we use this pattern to dynamically load the AMH support strategies only when needed. For example, a different implementation of a server ORB's request-processing strategy can be loaded dynamically to handle the special requirements of AMH request processing. Applications that do not require these features need not load the AMH strategy, thereby avoiding any time/space penalties it incurs.

Challenge 3: How to Leverage AMI Stub Generation in TAO's IDL compiler for AMH Skeleton Generation

Context: TAO's IDL compiler currently has the ability to generate AMI stubs. AMH requires the IDL compiler to also generate AMH skeletons.

Problem: As noted earlier, many similarities exist between the `ReplyHandler` class generated for AMI stubs and the `ResponseHandler` class needed for AMH skeletons. Instead of adding new code, it should be possible to reuse existing AMI generation code to generate AMH skeletons.

Solution: TAO's IDL compiler design is based on the Visitor pattern [Gamma et al. 1995], which represents operations that are performed and members of an object structure. The use of this pattern allows us to reuse the components that generate the `ReplyHandler` declarations. However, the implementation of the new server-side `ReplyHandler` requires the addition of other visitors.

Challenge 4: How to Minimize or Remove All Blocking I/O Operations from the ORB

Context: There are many opportunities for the ORB to block due to flow control while reading or writing data to the network.

Problem: Efficient use of the AMH model relies on the ability of a single thread to do other work while waiting for a response to outstanding requests. Whenever the ORB blocks unnecessarily, the application is prevented from handling other requests, which can degrade the performance of middle-tier servers significantly.

Solution: Recently, support for non-blocking I/O for both sending and receiving data in the ORB was added to TAO. This feature is fundamental for the efficient use of AMH in a server. The ORB configuration must be extended to help enable the appropriate configurations for optimal ORB behavior when AMH is enabled.

Challenge 5: How to Handle Multithreaded Issues with AMH

Context: TAO supports multiple threading models, ranging from one thread (*i.e.,* a reactive model [Schmidt et al. 2000]) to thread-per connection, to various forms of thread pools [Schmidt 1998].

Problem: The separation of the lifetime of the servant upcall to the lifetime of the request causes situations where the thread receiving the request may not be the thread sending back the response. These situations must be handled in an efficient and scalable manner.

Solution: The nature of AMH modifies some seldom-used properties of CORBA servers. For example, an application can ordinarily consult the `POACurrent` to obtain incoming request information, such as its target ObjectID. In AMH, this information cannot be obtained from a thread-specific object, and must therefore be represented explicitly. To support this capability, therefore, we are adding a new `AMHCurrent` to represent all information normally contained in the thread activation. This object can be obtained during the upcall to the AMH skeleton, and used by the application until the response is sent, or it can be passed as an extra argument in the AMH operation.

Likewise, CORBA ORBs must follow strict threading rules in the invocation of interceptors and similar mechanisms; those rules cannot be followed for AMH requests. We expect that applications designed for AMH will simply take these changes into consideration, however, and normal servants should not be affected by the relaxation of these rules.

4. Related Work

[Draves et al. 1991] outlines the use of continuations in an operating system's kernel. Continuations allow a thread to discard its call stack and provide a high-level representation of its execution state. This capability is similar to AMH in that AMH also encapsulates state in a `Response Handler` that is later used to return a response to a client.

The AMH model is based on the design of the AMI callback model [Arulanthu et al. 2000a]. The idea of a `ReplyHandler` was extended for use within the server as a `ResponseHandler`. In addition to allowing a client to use callbacks when the ORB receives a response from a server with AMI, the server application can now use a callback into the ORB to send a response back to the client.

Some other examples of work on this subject include Futures [Halstead et al. 1985] and Promises [Liskov et al. 1998], which are language mechanisms that decouple method invocation from method return values passed back

to the caller when a method finishes executing. While mainly dealing with the client side of an operation, their use can be extended into the method itself to provide further decoupling.

The TAO AMH design is influenced by the Proactor pattern [Schmidt et al. 2000], which allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations. The Proactor pattern separates (1) long-duration operations that execute asynchronously and (2) completion handlers that process the results of these operations to achieve the performance benefits of concurrency without incurring certain of its liabilities.

5. Concluding Remarks

Recent CORBA specifications [OMG 2000a] have standardized an Asynchronous Method Invocation (AMI) mechanism, which helps improve the scalability of client applications. Within the context of middle-tier servers, however, the utility of AMI is restricted by the fact that the activation records of threads in the servers cannot perform useful work while waiting for a response. A potentially better solution, therefore, is to apply a combination of AMI and the Asynchronous Method Handling (AMH) mechanism described in this paper.

AMH allows servants in middle-tier server applications to store response handlers in a container maintained by the servant application code and return control to the ORB immediately. This design allows the ORB to start handling a new request while it is waiting for a response from a sink server *without* requiring a thread for each request/response pairing in a middle-tier server. Any response from a sink server can invoke an immediate response to the client. Similarly, AMH can be applied to servers that block for a long time, *e.g.,* waiting for I/O, communicating over wide-area and other high latency networks, or long computations. Such applications can take advantage of AMH to improve predictability, latency and throughput, without the overhead of multi-threaded ORB server concurrency models [Schmidt 1998].

The open-source software, documentation, tests, examples, and related papers pertaining to TAO are available from http://www.cs.wustl.edu/~schmidt/TAO.html.

References

[Arulanthu et al. 2000a] A.B. Arulanthu, C. O'Ryan, D.C. Schmidt, M. Kircher: *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging*, in J. Sventek, G. Coulson (eds.): Middleware 2000, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Springer, 2000, ACM/IFIP, Lecture Notes in Computer Science, Springer, 2000

[Arulanthu et al. 2000b] A.B Arulanthu, C. O'Ryan, D.C Schmidt, M. Kircher: *Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks*, C++ Report, vol. 12, March 2000

[Box, 1997] D. Box: *Essential COM*, Addison-Wesley, Reading, MA. 1997

[Draves et al. 1991] R.P. Draves, B.N. Bershad, R.F. Rashid, R.W. Dean: *Using Continuations to Implement Thread Management and Communication in Operating Systems*, Proceedings of the 13th Symposium on Operating System Principles (SOSP), October 1991

[Eckerson 1995] W.W. Eckerson: *Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications*, Open Information Systems, vol. 10, no. 1, January 1995

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

[Halstead et al. 1985] R.H. Halstead, Jr.: *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Trans. Programming Languages and Systems, vol. 7, pp. 501-538, October 1985

[Liskov et al. 1998] B. Liskov and L. Shrira: *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, in Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp 260-267, June 1998

[OMG 2000a] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., October 2000.

[OMG 2000b] Object Management Group: *Real-Time CORBA*, formal/00-10-28, October 2000

[Othman et al. 2001] O. Othman, C. O'Ryan, D.C. Schmidt: *The Design and Performance of an Adaptive CORBA Load Balancing Service*, IEEE Distributed Systems Online, vol. 2, no. 4, April 2001

[Schmidt 1998] D.C. Schmidt: *Evaluating Architectures for Multi-threaded CORBA Object Request Brokers*, Communications of the ACM, special issue on CORBA, Krishnan Seetharaman (ed.), vol. 41, no. 10, ACM, October 1998

[Schmidt et al. 2000] D.C Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*, Wiley & Sons, New York, NY. 2000

[Wollrath et al. 1996] A. Wollrath, R. Riggs, and J. Waldo: *A Distributed Object Model for the Java System*, in Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies (COOTS), Toronto, Canada, 1996