The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging

Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons

{alex,coryan,schmidt,mk1,parsons}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130 *

Abstract

Historically, method-oriented middleware, such as Sun RPC, DCE, Java RMI, COM, and CORBA, has provided synchronous method invocation (SMI) models to applications. Although SMI works well for conventional client/server applications, it is not well-suited for high-performance or real-time applications due to its lack of scalability. To address this problem, the OMG has recently standardized an asynchronous method invocation (AMI) model for CORBA. AMI provides CORBA with many of the capabilities associated traditionally with message-oriented middleware, without incurring the key drawbacks of message-oriented middleware.

This paper provides two contributions to research on asynchronous invocation models for method-oriented middleware. First, we outline the key design challenges faced when developing the CORBA AMI model and describe how we resolved these challenges in TAO, which is our high-performance, real-time CORBA-compliant ORB. Second, we present the results of empirical benchmarks that demonstrate the performance benefits of AMI compared with alternative CORBA invocation models. In general, AMI based CORBA clients are more scalable than equivalent SMI based designs, with only a moderate increase in programming complexity.

1 Introduction

Motivation:

Historically, applications based on the standard CORBA [1] distributed object computing model have had to choose between three *invocation models*: one-way operations, synchronous two-way operations, and deferred synchronous operations using the dynamic invocation interface (DII). Unfortunately, these alternatives are often inappropriate for applications with stringent quality of service (QoS) requirements. For instance, one-way operations lack well-defined semantics [2], which reduces their portability and suitability for applications with non-trivial reliability requirements. Likewise, synchronous two-way operations are not scalable because they require a client thread for

* This work was supported by DARPA contract 9701516, NSF grant NCR-9628218, Siemens ZT, Sprint, Nortel, Boeing, SAIC, and Lucent.

each pending request/response invocation. Finally, the deferred synchronous model is inefficient and tedious to program due to its reliance on the DII [3], which allocates memory and copies data excessively.

To address these limitations, the OMG adopted a Messaging specification [4] for the CORBA standard. One of the key features in the CORBA Messaging specification is support for asynchronous method invocations (AMI).

Overview of CORBA AMI:

The CORBA AMI specification defines a *polling* model and a *callback* model, as described below:

• *Polling model:* In this model, each two-way AMI operation returns a `Poller valuetype` [5], which is very much like a C++ or Java class in that it has both data members and methods. Operations on a `Poller` are just local C++ method calls rather than remote CORBA operation invocations. The polling model is illustrated in Figure 1. The client can use the `Poller` methods to check the status of the request so it can

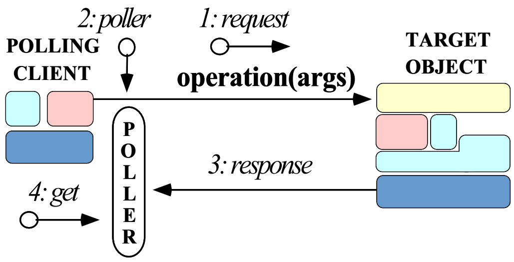

Fig. 1. Polling Model for CORBA Asynchronous Twoway Operations

obtain the server's reply. If the server hasn't replied yet, the client can either (1) block awaiting its arrival or (2) return to the calling thread immediately and check back on the `Poller` to obtain the `valuetypes` when it's convenient.

• *Callback model:* In this model, when a client invokes a two-way asynchronous operation on an object, it passes an object reference for a *reply handler* servant as a parameter. The reply handler object reference is not passed to the server, but instead is stored locally by the client ORB. When the server replies, the client ORB receives the response, and dispatches it to the appropriate callback operation on the reply handler servant provided by the client application, as shown in Figure 2.

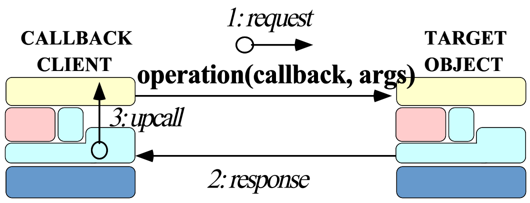

Fig. 2. Callback Model for CORBA Asynchronous Twoway Operations

Reply handler servants are accessed through normal object references. Therefore, it is possible for a client application to obtain an object reference for a remote reply handler servant and use that object reference to make AMI calls. In this case, replies for the asynchronous invocations will be handled in processes other than the client or the server involved in the original invocations. The most common use-case, however, is for the original client to process the response. In this case, therefore, client application developers must obtain, initialize, and activate reply handlers on a POA, which makes the application behave effectively as both a client and a server.

In general, the callback model is more efficient than the polling model because the client need not invoke method calls on a `valuetype` repeatedly to poll for results. Moreover, compared with CORBA's original invocation alternatives, the new AMI models provide the following benefits:

• *Simplified asynchronous programming model:* CORBA AMI allows operations to be invoked asynchronously using the *static invocation interface* (SII). Using SII for AMI eliminates much of the tedium, complexity, and inefficiency inherent in DII. In particular, DII requires programmers to allocate a new `Request` object explicitly and insert the operation parameters into a list of name value pairs, *i.e.*, an `NVList` pseudo-object. Conversely, in SII the IDL compiler can use an ORB's internal mechanisms to avoid extra memory allocations and data copies. Although deferred synchronous request implementations can exploit many AMI optimizations, such as better utilization of the network resources and improved parallelism, those improvements are hindered by DII's extra overhead, which often makes AMI a more attractive alternative.

• *Improved quality of service:* When implemented properly, AMI can improve the scalability of CORBA applications. For instance, it allows "pipelining" of two-way operations and minimizes the number of client threads that are otherwise required to perform two-way synchronous method invocations (SMI). In addition, AMI is important for real-time CORBA applications [6] because it helps to bound the amount of time a client spends blocking on two-way requests.

Synopsis of research contributions:

Our previous research has examined many dimensions of high-performance and real-time ORB endsystem design, including static [7] and dynamic [8] scheduling, event processing [9], I/O subsystem [10] and pluggable protocol [11] integration, ORB Core architectures [12], systematic benchmarking of multiple ORBs [13], patterns for ORB extensibility [14] and ORB performance [15]. This paper focuses on a previously unexplored dimension in the high-performance and real-time ORB endsystem design space: *the design and optimizations used to implement the standard CORBA asynchronous method invocation (AMI) callback model*.

The vehicle for our research on high-performance and real-time CORBA is TAO [7]. TAO is an open-source[1], CORBA-compliant ORB designed to address applications with stringent quality of service (QoS) requirements. In addition to being the first ORB with a standard Portable Object Adapter [15], TAO was the first ORB to implement the standard CORBA AMI callback model.

[1] The source code and documentation for TAO can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

Related work:

The AMI polling model stems from research on programming language support for distributed computing. For instance, Futures [16] and Promises [17] are language mechanisms that decouple method invocation from method return values passed back to the caller when a method finishes executing. As with AMI `Pollers`, calls are invoked asynchronously, clients can rendezvous with a Future/Promise to obtain reply values when they become available.

Previous research on *method-oriented middleware* [9, 18, 19] has examined how the CORBA Event Service can be used to perform asynchronous communication between CORBA applications. However, the CORBA AMI specification provides a different programming model than the CORBA Event Service. For instance, since the CORBA Event Service allows single-point-to-multi-point and anonymous communication models, application developers must devise their own means to send replies from event consumers back to event suppliers. In contrast, AMI applications can receive replies that include multiple IDL types. Moreover, CORBA Event Service participants communicate using a single `Any` argument. Although Anys can send all IDL types, they incur significant marshaling and message footprint overhead. In contrast, AMI clients can send and receive multiple IDL types and IDL compilers [20] can generate efficient marshaling and demarshaling code for them.

Message-oriented middleware (MOM), such as the Isis [21] Message Distribution System, TIBCO Information Bus, and IBM's MQSeries, provide mechanisms that allow suppliers to reliably transmit messages asynchronously to one or more consumers. MOM systems typically consist of additional "router" processes that store and forward messages on behalf of application processes. If a consumer happens to be unavailable due to scheduled downtime, a site crash, or a network partition, the router will attempt to deliver the message periodically until the consumer becomes available. The OMG Message specification defines similar routing capabilities via its Time-Independent Invocation (TII) feature [22, 4]. Both the TII and MOM asynchrony mechanisms are too heavyweight, however, for many high-performance and real-time applications. Moreover, the message-oriented invocation mechanisms of MOM systems can be harder to program correctly due to the lack of strong typechecking.

The remainder of this paper is organized as follows: Section 2 outlines the general structure and dynamics an ORB requires to support AMI callbacks; Section 3 describes key design challenges faced when implementing the CORBA AMI callback model and explains how TAO resolves these challenges; Section 4 empirically analyzes the performance of AMI callbacks in TAO [7] and compares it with alternative communication models; and Section 5 presents concluding remarks that summarize the lessons learned from implementing AMI callbacks in TAO.

2 ORB Architectural Support for AMI Callbacks

This section outlines the general structure and dynamics an ORB requires to support AMI callbacks.

2.1 AMI Callback Features

To support AMI callbacks, an ORB should implement the following functionality:

1. AMI stubs: For each two-way operation in the IDL interface, an ORB's IDL compiler [20] should generate an *AMI stub* that applications can use to issue asynchronous operations. Each AMI stub is responsible for (1) setting up state in the ORB to receive the reply and dispatch it to the appropriate reply handler, (2) marshaling the `in` and `inout` arguments provided by the application, and (3) using the ORB Core to send the message to a remote ORB. High-quality IDL compilers should provide an option to suppress the generation of AMI stubs to reduce the footprint of applications that do not use them.

2. Manage pending invocations: The client ORB must store reply handler object references for all asynchronous invocations. If the reply handler servant is collocated with the client, the application developer must activate the reply handler implementation with the client's ORB POA. When a reply returns, the client ORB locates the reply handler servant and invokes the callback method on it. The client ORB delivers this new request to the reply handler servant using its regular invocation path, which allows an ORB's collocation optimizations [23] to be used to minimize dispatching overhead.

3. Explicit event loop methods: An ORB must implement the standard CORBA `work_pending` and `perform_work` operations. Clients can use these operations to invoke the CORBA event loop in a client explicitly. In addition, if asynchronous replies arrive while a client is blocked waiting for a synchronous reply, the ORB can use the blocked thread to dispatch the asynchronous reply.

2.2 Collaborations Between ORB Components for Asynchronous Invocation

After an OMG IDL compiler generates the AMI callback stubs, the generated code must collaborate with internal ORB components to send and receive asynchronous invocations. To demonstrate how this works, Figure 3 depicts the general sequence of steps involved when an asynchronous two-way `get_quote` operation is executed.[2] As shown in this figure, the interactions between client ORB components for an asynchronous invocation consist of the following steps:

- The client application invokes the `sendc_get_quote` method on the `Stub` to issue the asynchronous operation **(1)**. The client passes the `AMI_QuoterHandler` object reference, along with the name of the stock we're interested in, *e.g.*, IBM.
- The `Stub` marshals its string argument into a buffer and instantiates an `Invocation` **(2)**, which is a facade that delegates to internal ORB components that establish connections **(3)** & **(4)** with a remote server (if necessary), the ORB stores the `AMI_QuoterHandler` object **(5)**, and send the requests **(6)** & **(7)** to the server.
- After the request is sent, `Invocation` returns control to the `Stub` **(8)**, which itself returns control to the client **(9)**.

[2] The names of certain objects in this discussion are specific to TAO, though the general flow of control and behavior should generalize to other ORBs that implement AMI callbacks.

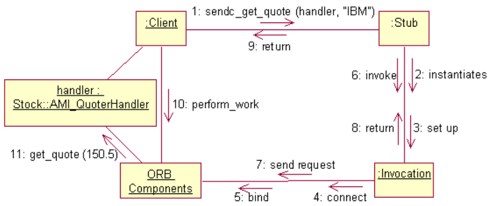

Fig. 3. Interactions Between Client ORB Components for Asynchronous Invocation

- When a client application is prepared to handle callbacks, it calls the ORB's `work_pending` and `perform_work` **(10)** methods to receive and dispatch replies associated with asynchronous invocations.
- When the reply arrives, the ORB demarshals the reply and demultiplexes it to the callback method on the reply handler servant that was passed in by the application when the AMI method was invoked originally **(11)**.

Section 3.2 revisits these steps in more detail after we've explained the components in TAO's ORB architecture.

3 The Design of TAO's AMI Callback Architecture

To make the discussion of ORB architectural support for AMI in Section 2 more concrete, this section describes our resolutions to key design challenges encountered when implementing TAO's AMI-enabled ORB architecture. Section 4 then illustrates the performance characteristics of TAO's AMI implementation compared to alternative SMI and DII deferred synchronous communication models.

3.1 Design Challenges and Resolutions

To assist developers of distributed object systems in making informed choices among alternative ORB middleware solutions, they should understand how the ORBs are implemented. Below, we (1) outline the key design challenges we faced when implementing AMI in TAO and (2) explain the patterns and components we used to resolve these challenges.

Challenge: How to Process Asynchronous Replies Efficiently

Context: Early TAO implementations supported only the Synchronous Method Invocation (SMI) model. In SMI, the calling thread that makes a two-way invocation blocks awaiting the server's reply. The client ORB can use the calling thread to process the response. For example, consider the *Leader/Followers* thread pool concurrency model [12] illustrated in Figure 4. TAO uses this concurrency model to support multi-threaded client applications efficiently, as follows:

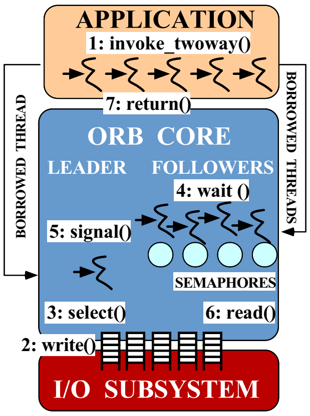

Fig. 4. Processing Synchronous Two-way Client Invocations using the Leader/Followers Concurrency Model

- Each calling thread that invokes a two-way synchronous method **(1)** uses a connection to send the request **(2)**.
- The client ORB designates one of the waiting threads the *leader* and the other threads as the *followers*. The leader thread blocks on the `select` operation **(3)**; the follower threads block on semaphores **(4)**.
- When a reply arrives on a connection, the leader thread returns from `select`. If the reply belongs to the leader, it continues to process the reply after first promoting the next follower to become the new leader. If the reply belongs to one of the followers, however, the leader signals the corresponding semaphore to wake up the follower thread **(5)**.
- The awakened follower thread reads the reply **(6)**, completes the two-way invocation **(7)**, and returns to its caller.

Problem: Although the Leader/Followers thread pool model described above works well for SMI, it does not work without modification for AMI. The problem stems from the fact that the calling stub goes out of scope as soon as the request is sent and control returns to client application code. Thus, the ORB must be prepared to process an asynchronous reply in another context, possibly within another client thread. Moreover, to complete the processing of server replies to asynchronous invocations, the ORB must maintain certain state information, such as reply handler object reference and a function to demarshal the reply (the so-called *reply-stub*).

Forces: The mechanisms provided to support AMI replies should add no significant run-time overhead to the existing SMI mechanisms.

Solution $\rightarrow$ *Strategizing the reply dispatching mechanisms:* The problem of processing asynchronous replies can be solved by *strategizing* the reply processing and dispatching mechanisms used for AMI and SMI calls. Figure 5 illustrates the components in TAO's `Reply Dispatcher` hierarchy. A `Synchronous Reply Dispatcher` is created by an `Invocation` object during a synchronous invocation on the local stack activation record. When the reply is received, the reply buffer, *i.e.*, TAO's `InputCDR`

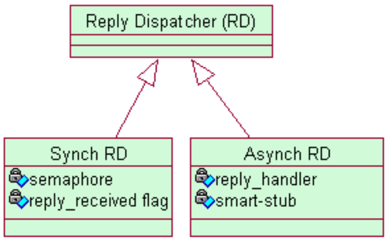

Fig. 5. Reply Dispatching Strategy

object, is placed in the dispatcher and control returns first to the `Invocation` object and then to the `Stub`. At this point, the `Stub` obtains the reply buffer from the `Invocation` object, demarshals the reply, and completes the invocation. Each `Reply Dispatcher` object maintains a `reply_received` flag that indicates if the reply has been received. This flag is set when the reply is dispatched to this object and the thread waiting for the reply returns to the `Stub`.

During an AMI call, an `Invocation` object creates an `Asynchronous Reply Dispatcher` on the heap[3] because the activation record where the `Invocation` object is created is exited before the reply is received. The AMI stub, *i.e.*, the `sendc_*` operation, stores the reply handler object reference provided by the client in the `Asynchronous Reply Dispatcher` object. In addition, the AMI stub stores the pointer to the appropriate reply-stub method in this object.

A Leader/Followers implementation using TAO's `Reply Dispatcher` architecture is illustrated in Figure 6 and behaves as follows:

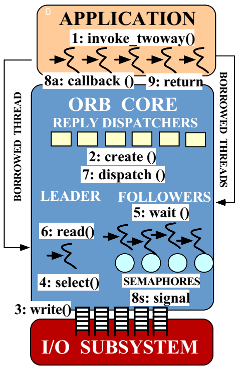

Fig. 6. TAO's AMI-enabled Leader/Followers Implementation

– When application threads make two-way invocations **(1)**, a `Reply Dispatcher` object is created for each invocation **(2)** and the request is sent **(3)**.

[3] As an optimization, an ORB could use a pre-allocated pool to allocate these objects, thereby alleviating heap fragmentation [15].

– The leader thread then blocks on the `select` call **(4)** and the follower threads block on the semaphores **(5)**.
– When a reply arrives on a connection, the leader thread itself reads the complete reply **(6)** and calls the `Reply Dispatcher` object that was created for that invocation to dispatch the reply **(7)**.
– For SMI calls, the `Synchronous Reply Dispatcher` signals **(8s)** the thread waiting for that reply and completes the invocation **(9)**. For AMI calls, however, the `Asynchronous Reply Dispatcher` object invokes the callback method in the reply handler servant **(8a)**.

Challenge: How to Minimize Connection Utilization

Context: Early implementations of TAO supported only a *non-multiplexed* connection model [12], which is well-suited for hard real-time applications whose QoS requirements include highly predictable response times. In this model, a connection cannot be reused for another two-way request until the reply for the previous request is received. Figure 7 illustrates TAO's non-multiplexed connection model, where five threads make

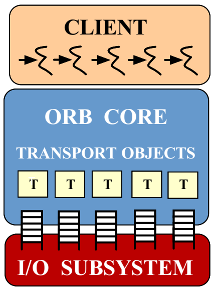

Fig. 7. One Outstanding Request Per-Connection

two-way invocations to the same server, which creates five connections. TAO represents connections using a `Transport` object that provides a uniform interface to the TAO's `pluggable protocols framework` [11], this framework abstracts various underlying transport mechanisms, such as TCP, UNIX-domain sockets, and VME, implemented by TAO. TAO's pluggable protocols framework uses key patterns and components provided by ACE [24].

Problem: Non-multiplexed connection models are inefficient for CORBA AMI because client applications can issue hundreds or thousands of asynchronous requests before waiting for the replies. Thus, a non-multiplexed connection model would use a correspondingly large number of connections.

Forces:

1. An ORB should implement connection multiplexing so that multiple outstanding requests required to support the AMI model can be processed efficiently.

2. When multiple threads access a connection simultaneously, they should be synchronized so that requests are sent one-by-one and not corrupted through intermingled I/O calls.
3. To accommodate various use-cases and QoS requirements, applications should be able to configure multiplexed and non-multiplexed connection behavior both statically *and* dynamically.

Solution → Strategize the transport multiplexing mechanisms: To overcome the scalability limitations of a non-multiplexed connection architecture, we extended TAO to support a multiplexed connection option for both SMI and AMI. In this design, many requests can be sent simultaneously over the same connection, even when replies are pending for earlier requests. In general, multiplexing yields better use of connections and other limited OS resources [12], such as memory buffers.

To implement this design in TAO, we applied the Strategy pattern [25] and defined a new strategy called `Transport Mux Strategy` that supports both multiplexed and the non-multiplexed connections. The components in this design are illustrated in Figure 8.

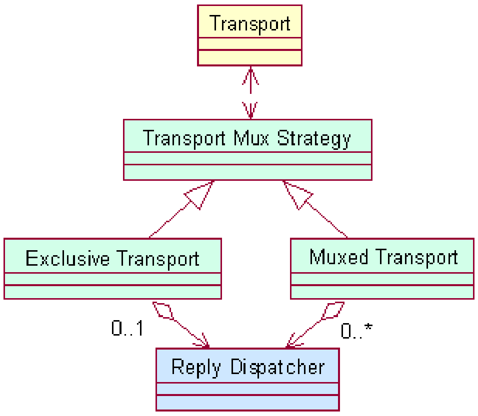

Fig. 8. Transport Mux Strategy

The `Exclusive Transport Strategy` implements the non-multiplexed connection strategy by holding a reference to a single `Reply Dispatcher` object. This strategy is "exclusive" because only one outstanding request at a time can pend on each connection. In contrast, the `Muxed Transport Strategy` uses a hash table that stores multiple `Reply Dispatchers`, each representing a request sent on the connection. As shown in Figure 8, the `Transport Mux Strategy` base class provides a common interface for these two different implementations. TAO uses the Service Configurator pattern [26] to allow applications to select between these two strategies and thereby configure TAO's `Transport Mux Strategy` either statically or dynamically.

To synchronize access to a multiplexed connection among multiple threads, the `Transport` object for that connection is marked as "busy" while one thread is sending a request. If during that time another thread tries to send a request, either a cached connection is recycled or a new connection is created. After the request is sent, the `Transport` object is marked as "idle" and is cached so it can be reused to send subsequent requests.

Challenge: How to Implement Scalable Reply Processing Mechanisms

Context: High-quality CORBA implementations should support "nested upcalls", in which an ORB processes incoming requests while it waits for replies. This support can be implemented using `select` to wait for both the reply and any incoming requests. This implementation can add unnecessary overhead, however, to "pure" clients that do not receive any incoming requests from servers. Therefore, TAO provides the following three reply processing strategies that allow developers to select the most appropriate mechanism for their application QoS requirements:

- Wait-on-Read: In this strategy, the calling thread blocks on `read` to receive the reply. This is a very efficient strategy for pure clients that need not receive requests or nested upcalls while waiting for server replies.
- Wait-on-Reactor: The Reactor [27] is a framework implemented in ACE [24] that provides event demultiplexing and event handler dispatching. In this strategy, a single-threaded `Reactor` is used to dispatch events, such as reply arrivals and upcalls. This strategy supports single-threaded client applications efficiently by having the waiting thread run the event loop of the Reactor to check for server replies. When there is input on a connection, the `Transport` object is notified and it reads the input message and dispatches the reply. The Wait-on-Reactor strategy also works with multi-threaded applications that use a Reactor-per-thread to minimize contention and locking overhead [12].
- Wait-on-Leader/Followers: If the application is multi-threaded and several threads are sharing the same Reactor, only one of them can run the Reactor's event loop at a time. Therefore, this strategy uses the Leader/Followers pattern [12] to synchronize access to the Reactor. In this pattern, the leader thread runs the event loop of the Reactor. All other threads wait on a semaphore. When a reply is available, the leader thread reads and dispatches the complete reply. If the reply is for an AMI request, it is dispatched to the callback method in the reply handler servant. For synchronous replies, the reply buffer is transferred to the `Synchronous Reply Dispatcher` from the `Transport` object. If a reply belongs to the leader thread, it selects another thread as the leader and returns from the event loop. If the reply belongs to another thread, however, it signals this thread so it can wake up from the semaphore, return to its stub, and process the reply.

Problem: Pre-AMI-enabled versions of TAO implemented the three reply processing strategies described above as `Connection Handlers` within TAO's pluggable protocols framework, as shown in Figure 9. However, every `Transport` mechanism,

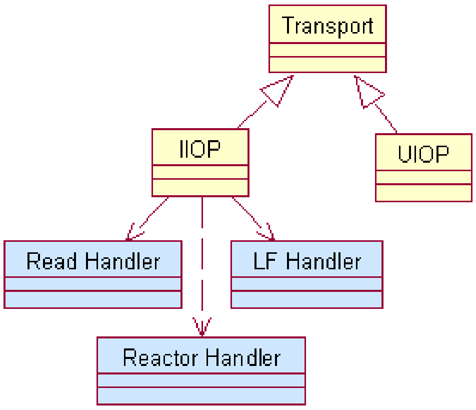

Fig. 9. Initial Design of TAO's Reply Processing Mechanisms

such as IIOP and UNIX-domain sockets (UIOP), in TAO's pluggable protocols framework [11] required three `Connection Handler` implementations to support all the reply wait strategies in its `Transport` implementation. Not surprisingly, this approach did not scale up effectively when TAO incorporated additional transport mechanisms, such as VME, Fibrechannel, or TP4. TAO's original design also complicated the integration of the AMI callback model because changes to the reply wait mechanisms were necessary for *each* `Transport` implementation.

Forces: The semantics of the existing wait mechanisms, as well as the existing optimizations, must be maintained while integrating the AMI callback model. Moreover, applications should be able to configure TAO's reply wait mechanism according to their particular needs.

Solution → Refactor reply wait strategies: As part of our enhancement to TAO, we moved the reply wait mechanisms from the `Connection Handlers` to the new `Wait Strategy` and decoupled it from the underlying `Transport` and the `Connection Handler` objects. TAO's new Wait Strategy architecture is illustrated in the UML class diagram in Figure 10. In TAO's enhanced architecture, each `Transport`

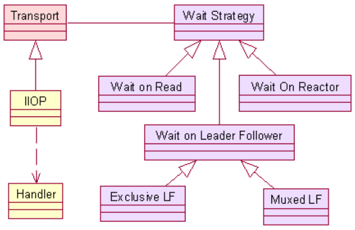

Fig. 10. Enhanced Design of TAO's Reply Processing Strategies

implements only one `Connection Handler`. Due to the patterns-based OO design [11] used in TAO, this modification required changes only to its `Transport` and `Connection Handler` implementations; no other ORB components were affected.

In addition to refactoring the wait strategies, a variation of the Leader/Followers implementation has been integrated into TAO's Wait-on-Leader/Followers strategy. This change was necessary because the original Leader/Followers implementation assumed non-multiplexed connections, *i.e.*, only one request at a time was sent per-connection. Therefore, state variables, such as semaphores, were kept in the `Transport` and the `Connection Handler` objects, which are per-connection objects. Although this implementation works for the `Exclusive Transport` strategy, it is unsuitable for `Muxed Transport`, where multiple threads may wait simultaneously for replies on a single connection.

To address the multiplexing problem, we enhanced the Leader/Followers model described earlier to create a variation called Muxed-Wait-on-Leader/Followers strategy. This new strategy uses the Thread-Specific Storage pattern [28] to store a per-ORB-per-thread condition variable. This condition variable is created on-demand just once, by a factory method in TAO's ORB Core. This factory method provides a facade [25] to all ORB strategies, helper classes, and global or thread-specific resources.

Challenge: How to Minimize Stub Footprint

Context: Earlier, we discussed the ORB components used by the client stub to set up the connection, create the `Reply Dispatchers`, send the request, keep track of the `Reply Dispatchers` and *reply-stubs*, wait for and process replies, and deliver the replies to target threads or reply handler servants. A stub can either invoke methods on these ORB components directly, or it can use helper classes that can be implemented as part of the ORB. Helper classes can interact with various ORB components on behalf of the stub and execute all functionality outlined above.
Problem: If stubs interact with the internal ORB components directly, the code size of the stub increases. In turn, this increases the footprint of the generated C++ code because TAO's IDL compiler creates stubs for each operation in the IDL interface.
Forces: There is a tradeoff between code size and performance [29]. In general, stubs could inline all the code required to complete their task [30]. However, inlining can cause unacceptably large memory footprint. Conversely, stubs could simply pass parameter data to a shared interpreter, such as a DSI/DII engine [31]. In this case, however, system performance would suffer.
Solution → Optimized invocation helper facades: To reduce memory footprint, stubs should use helper classes to factor out common code from the stubs into reusable ORB Core components. In TAO, these helper classes are called `Synchronous Invocation` and `Asynchronous Invocation`. They provide stubs with facades that encapsulate the details of various features implemented internally to the ORB to support both AMI and SMI.

When called by a stub on behalf of a client, the `Synchronous Invocation` class establishes a connection[4] to the remote host, sends the request, waits for a reply, receives the reply, and returns control to the stub once the reply is received. The

[4] TAO uses connection caching [12] to avoid establishing new connections if one is already open to a particular ORB endpoint.

`Asynchronous Invocation` class is similar, but it returns control to the stub as soon as it sends the request. Thus, the `Synchronous Invocation` object creates the `Synchronous Reply Dispatcher` on its local stack activation record, whereas the `Asynchronous Invocation` object creates the `Asynchronous Reply Dispatcher` on the heap.

As illustrated in Figure 11, TAO's synchronous and asynchronous variants inherit

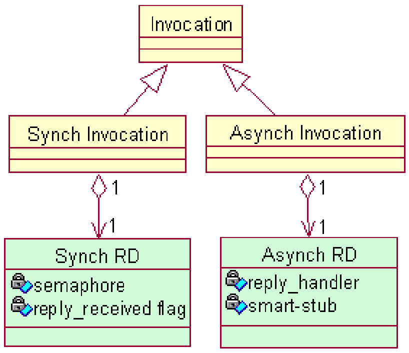

Fig. 11. Invocation Interface

from a common `Invocation` class, which provides a uniform interface to other components in the ORB. Both classes delegate the tasks described above to other ORB components we discussed earlier.

3.2 Collaborations Between Components in TAO's AMI-enabled Architecture

Now that the preceding sections described TAO's ORB architecture components that process synchronous and asynchronous requests, we can present the overall AMI-enabled ORB architecture of TAO, which is shown by the UML class diagram in Figure 12. Moreover, Figure 13 reexamines the sequence of steps that occur when an application issues an AMI or SMI call. Each of these steps is described below:

– The `Client` calls the `Stub` to invoke an operation. In the case of an AMI call, it passes a reference to a reply handler servant **(1)**.
– The stubs generated by TAO's IDL compiler are different for the SMI and AMI calls. In particular, the SMI and AMI stubs instantiate their corresponding `Invocation` objects **(2)**.
– The `Invocation` object creates a `Synchronous` or `Asynchronous Reply Dispatcher`, depending on the type of the request **(3)**. The `Invocation` object then binds the `Reply Dispatcher` object with the `Transport Mux Strategy` object (**4** & **5**).
– The `Invocation` object calls the `Transport` object, which in turn uses TAO's pluggable protocols framework [11] and ACE [24] to send the request (**6** & **7**).

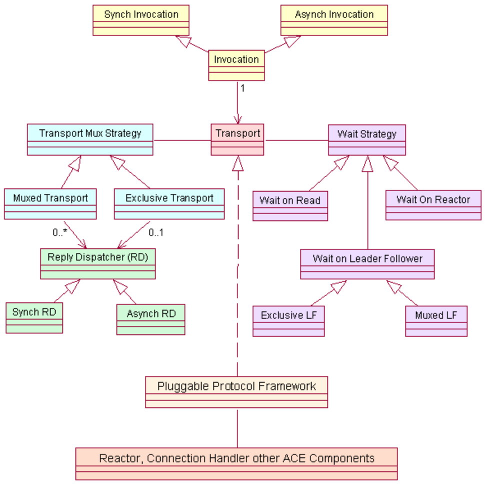

Fig. 12. AMI-enabled TAO ORB Architecture

- In the AMI model, the stub returns control to the application at this point. Later, the `Client` can wait for the server's reply. In the SMI model, conversely, the `Invocation` object calls the `Transport` to wait for the reply, which delegates this task to the `Wait Strategy` **(8)**.
- When the reply arrives, the `Transport` object is notified to read the reply **(9)**. It reads the complete reply and calls the `Transport Mux Strategy` to dispatch the reply **(10)**. The `Transport Mux Strategy` uses the correct `Reply Dispatcher` object created for that invocation and calls its dispatch method **(11)**.
- If a `Synchronous Reply Dispatcher` is used, it simply stores the reply buffer, sets the state variables within the object to indicate that the reply has been received, and then returns. Conversely, the `Asynchronous Reply Dispatcher` invokes the reply stub stored in the object, passing in the reply handler servant and the reply buffer, and dispatches the reply **(12)**.

4 Evaluating the Performance of TAO AMI Callbacks

4.1 Overview

As discussed in Section 1, AMI can help improve the scalability of CORBA applications by minimizing the number of client threads required to perform two-way invocations. In this section, we present empirical results that show how TAO's AMI implementation helps to increase application scalability by minimizing the number of client threads. We demonstrate the efficiency of the implementation by comparing both the latency and operation throughput of SMI and AMI two-way invocations in TAO.

All experiments were performed on two 400 Mhz quad-CPU Dell 6300 computers running Linux 2.2 and connected by a 100 Mbps Fast Ethernet. Each computer has 1

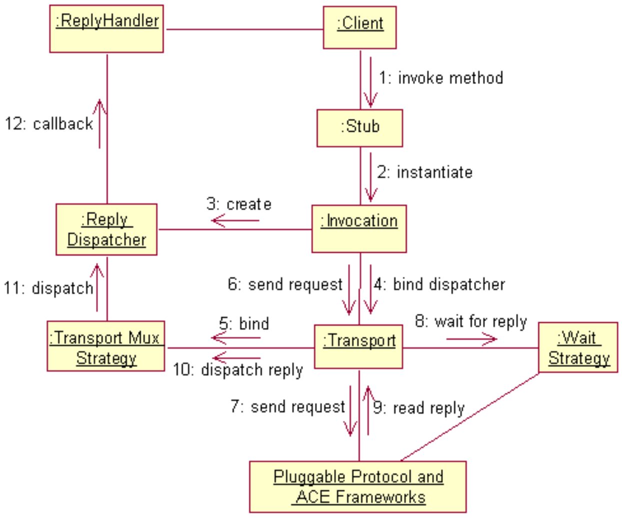

Fig. 13. Sequence of Steps in TAO's SMI & AMI Invocations

GB of RAM. The benchmarks were compiled using the GCC v. 2.95 compiler with the highest level of optimization.

The server implementation is held constant in all our benchmarks. Moreover, to minimize the overhead on the server, we use a simple interface that accepts a single argument and returns it. The argument is a 64-bit `unsigned long` that the client uses to send timestamps to the server to measure round-trip delays. To minimize jitter, all client and server benchmarking processes were run in the Linux real-time scheduling class.

4.2 Empirical Results

Two-way latency benchmark: In our first experiment, we compared the round-trip latency of 10,000 two-way calls in single-threaded applications using three different invocation models: (1) SMI using the SII, (2) AMI using the SII, and (3) deferred SMI using the DII. For the DII and AMI benchmarks we sent the request and immediately waited for the asynchronous reply.

Table 1 compares the latency for the three invocation models. The best results are obtained using AMI requests, though the difference with respect to SMI is small (3%). This difference is within the error margins defined by the jitter measurements and is not significant. Compared to SMI, a larger amount of jitter was observed for AMI, resulting from the extra locking overhead required to dispatch the reply-stub. In contrast, the worst performance is obtained using the deferred synchronous model, which averaged 20% slower than AMI because it incurs additional DII processing overhead.

Test	Minimum	Average	Maximum	Jitter
SMI	455	497	684	2.7%
AMI	447	479	1,859	3.0%
DII	499	573	2,652	9.6%

Table 1. μsecond Latency Results for Different Invocation Models

Operation throughput benchmark: In this experiment, we compared the throughput (in number of requests per second) of the different invocation models. To simulate asynchronous communication using ORBs without AMI support, applications have traditionally spawned additional threads. To compare this approach with an AMI application, therefore, the client process creates a new thread for each two-way SMI call, up to an OS imposed limit of 220.[5] The benchmark sends 10,000 requests on each thread.

In contrast to the heavily threaded SMI client, the AMI client uses only two threads. One thread sends as many two-way requests as required and the other thread runs the ORB event loop to dispatch replies to the appropriate reply handler. To match the number of calls performed by the SMI client, therefore, the AMI client performs 2,200,000 calls. Finally, we perform the same test using DII deferred synchronous requests.

The results of this experiment are shown in Figure 2. As shown by these results,

Test	Average Calls/sec.
SMI (220 threads)	1914
SMI (7 threads)	7080
AMI	8524
DII	3816

Table 2. Operation Throughput Results for Different Invocation Models

the AMI client not only provides a more scalable design than the multi-threaded SMI client, but also shows a significant performance improvement. This improvement stems from the fact that (1) the TCP/IP stack can send larger data packets containing multiple AMI requests, (2) the two threads in the AMI client can overlap request invocations and response processing, and (3) the AMI client fully utilizes the network resources, *i.e.*, it can completely fill TCP/IP windows because it can "pipeline" the two-way invocations.

In addition to scalability problems, the use of hundreds of threads in the SMI client also increases its synchronization overhead. Table 2 shows how reducing the number of threads in the SMI client test from 220 to 7 improved performance significantly. This solution has the adverse affect of reducing the number of simultaneous two-way calls,

[5] Note that we were unable to create more than 220 threads before running out of resources on Linux. This illustrates one of the drawbacks of using threads to simulate asynchronous communication.

however, which increases average latency. In contrast, the AMI client do not suffer from this tradeoff.

Finally, note that that deferred synchronous requests can sometimes achieve better performance than a naively designed, heavily-threaded SMI client. It is unlikely, however, that the performance of deferred synchronous DII could ever rival that of AMI, due to the inherent overhead of memory allocation and data copying. Moreover, DII's invocation model is more tedious and error-prone to program.

4.3 Summary of Results

The latency and operation throughput results presented above can be interpreted as follows:

- For simple applications that require few request-response interactions, SMI is almost as effective as AMI, with an insignificant difference in latency within the error margins. In addition, SMI has slightly less jitter because its implementation uses fewer locks.
- For more demanding applications, AMI applications can exhibit a measurable (20%) improvement in operation throughput compared with the best SMI results. These performance improvements illustrate how AMI clients can leverage network resources and inherent parallelism in distributed systems more effectively than SMI clients.

5 Concluding Remarks

Asynchronous method invocations (AMI) are an important feature that has been integrated into CORBA via the OMG Messaging specification [4]. A key aspect of AMI is that operations can be invoked asynchronously, while still using the static invocation interface (SII). The use of SII eliminates much of the complexity and inefficiency inherent in the dynamic invocation interface (DII)'s deferred synchronous model.

This paper explains how ORBs can be structured to support the CORBA AMI callback model efficiently and scalably. The following is a synopsis of the lessons learned developing TAO's AMI callback implementation:

AMI requires a scalable ORB architecture: An ORB should implement the AMI and SMI reply handling in a flexible and scalable manner. For instance, to support many simultaneous AMI requests efficiently, connection multiplexing optimizations should be supported in the ORB Core.

Optimizations should be guided by empirical measurements: AMI and SMI enhancements should be guided by systematic blackbox benchmarks and whitebox profiling so that existing optimizations in the ORB are preserved, while allowing applications to configure the ORB based on their specific QoS requirements. For example, during the validation phase of our AMI changes, we discovered that the SMI model was performing one memory allocation more than it did before the AMI changes. The problem was easily fixed, but it illustrates that careful, repeated whitebox analysis of the system and application of optimization principle patterns [15] is required to ensure and maintain its quality.

The ORB should adapt readily to different use-cases: Design patterns should be applied to configure ORBs with policies and mechanisms appropriate for particular application use-cases, while still preserving key optimizations necessary to support stringent QoS requirements. In particular, we repeatedly applied the Strategy pattern [25] to TAO's AMI implementation to support scalable connection multiplexing strategies, while retaining configurations that ensure the determinism required for hard real-time applications. Applications can select AMI or SMI strategies using the Service Configurator pattern [25], which makes the TAO framework dynamically configurable and therefore highly flexible.

Both AMI and SMI are important invocation models: Enhancements needed to support AMI should not add overhead to the ORB's SMI processing. Patterns like Strategy and Service Configurator can be used to make any additional overhead optional for applications that do not require it.

Programming AMI clients requires application developers to make design decisions: While developing our tests for the AMI implementations, we recognized that the AMI model, while more intuitive and easier to use than the DII deferred synchronous model, is more complex than simple SMI applications. For instance, client developers must decide how to handle the replies, *e.g.*, by using a separate thread, waiting for replies after a fixed number of requests, or adaptively waiting for replies. Developers must also decide how to connect the reply with the original request, *e.g.*, by using a different reply handler servant for each one, returning some kind of request id from the server, or using the POA dynamic activation mechanisms to distinguish between all the requests. Finally, client developers must be prepared to handle "inversion of control" in their applications, *i.e.*, by using a callback to handle the incoming reply.

These challenges should not be viewed as insurmountable problems, however. After developers master the appropriate patterns and idioms, AMI can be significantly easier to program than the CORBA deferred synchronous model. Moreover, it offers significant performance improvements over both SMI and DII calls. Thus, CORBA AMI is an important addition to the CORBA family of features and specifications.

References

1. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.
2. D. C. Schmidt and S. Vinoski, "Introduction to CORBA Messaging," *C++ Report*, vol. 10, November/December 1998.
3. D. C. Schmidt and S. Vinoski, "Programming Asynchronous Method Invocations with CORBA Messaging," *C++ Report*, vol. 11, February 1999.
4. Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.
5. Object Management Group, *Objects-by-Value*, OMG Document orbos/98-01-18 ed., January 1998.
6. Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.
7. D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

8. C. D. Gill, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-Time CORBA Scheduling Service,” *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, to appear 2000.
9. T. H. Harrison, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-time CORBA Event Service,” in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.
10. F. Kuhns, D. C. Schmidt, and D. L. Levine, “The Design and Performance of a Real-time I/O Subsystem,” in *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.
11. C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, “The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware,” in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.
12. D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, “Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers,” *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.
13. A. Gokhale and D. C. Schmidt, “Measuring the Performance of Communication Middleware on High-Speed Networks,” in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.
14. D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible ORB Middleware,” *IEEE Communications Magazine*, vol. 37, April 1999.
15. I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, “Applying Optimization Patterns to the Design of Real-time ORBs,” in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.
16. R. H. Halstead, Jr., “Multilisp: A Language for Concurrent Symbolic Computation,” *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.
17. 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 1988.
18. Y. Aahlad, B. Martin, M. Marathe, and C. Lee, “Asynchronous Notification Among Distributed Objects,” in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.
19. C. Ma and J. Bacon, “COBEA: A CORBA-Based Event Architecture,” in *Proceedings of the* 4^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, Apr. 1998.
20. A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, and M. Kircher, “Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks,” *C++ Report*, vol. 12, Mar. 2000.
21. K. Birman, “The Process Group Approach to Reliable Distributed Computing,” *Communications of the ACM*, vol. 36, pp. 37–53, December 1993.
22. C. O'Ryan and D. C. Schmidt, “Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation,” in 5^{th} *International Workshop on Object-oriented Real-Time Dependable Systems*, (Monterey, CA), IEEE, Nov 1999.
23. N. Wang, D. C. Schmidt, and S. Vinoski, “Collocation Optimizations for CORBA,” *C++ Report*, vol. 11, November/December 1999.
24. D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
26. P. Jain and D. C. Schmidt, “Dynamically Configuring Communication Services with the Service Configurator Pattern,” *C++ Report*, vol. 9, June 1997.

27. D. C. Schmidt, “The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2),” *C++ Report*, vol. 5, September 1993.
28. D. C. Schmidt, T. Harrison, and N. Pryce, “Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently,” *C++ Report*, vol. 9, November/December 1997.
29. A. Gokhale and D. C. Schmidt, “Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems,” *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.
30. E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, “Flick: A Flexible, Optimizing IDL Compiler,” in *Proceedings of ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.
31. A. Gokhale and D. C. Schmidt, “Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance,” in *Hawaiian International Conference on System Sciences*, January 1998.