Object Interconnections

Scalable and Efficient Architecture
for CORBA Asynchronous Messaging
(Column x)

Alexander B. Arulanthu, Carlos O'Ryan, and Douglas C. Schmidt
{alex,coryan,schmidt}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

This column will appear in the XXXX 1999 issue of the SIGS C++ Report magazine.

1 Introduction

To make informed choices amoung middleware alternatives, distributed object computing developers should understand how CORBA ORBs implement key features. Our last column explored the design and performance of alternative collocation strategies [1]. In this column, we describe how the new OMG Asynchronous Method Invocation (AMI) callback model can be implemented scalably and efficiently by CORBA ORBs.

As we discussed in earlier columns [2, 3], the CORBA AMI callback model is an important feature that has been introduced into CORBA via the CORBA Messaging specification [4]. AMI allows operations to be invoked asynchronously using the *static invocation interface* (SII), thereby eliminating much of the complexity inherent in the *dynamic invocation interface* (DII)'s deferred synchronous model. When implemented properly, AMI helps improve the scalability of CORBA applications because it minimizes the number of client threads required to perform two-way invocations.

The CORBA Messaging specification defines two AMI programming models, the *Polling* model and the *Callback* model, which are outlined below:

Polling model: In this model, each asynchronous two-way invocation 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 and not distributed CORBA operation invocations. This model is illustrated in Figure 1. The client can use the `Poller` methods to check the status of the request and to obtain the value of the reply from the server. If the server hasn't replied yet, the client can either (1) block awaiting its arrival or (2) return to the calling thread immediately

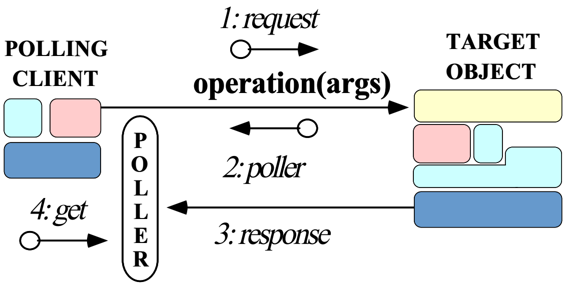


Figure 1: Polling Model for CORBA Asynchronous Twoway Operations

and check on the value of the `Poller` when it's convenient.

Callback model: As illustrated in Figure 2, in this model

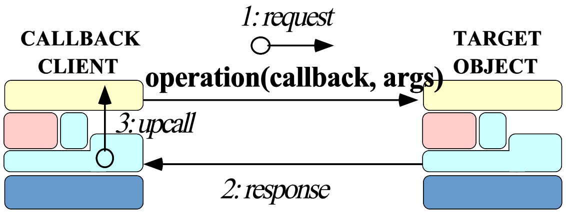


Figure 2: Callback Model for CORBA Asynchronous Twoway Operations

the client passes an object reference for a `ReplyHandler` object as a parameter when it invokes a two-way asynchronous operation on an object reference to a server. When the server replies, the client ORB receives the response and dispatches it to the appropriate operation on the `ReplyHandler` servant, where the client then processes the reply. In other words, the client ORB turns the response into a request on the client's `ReplyHandler`.

In general, the callback model can be more efficient than the polling model because the client need not repeatedly invoke method calls on the ORB to poll for results. It does force clients to behave as servers, however, which can increase the complexity of certain applications, particularly "pure" clients.

The remainder of this column is organized as follows: Section 2 presents an example that illustrates the CORBA AMI callback programming model in more detail; Section 3 outlines the IDL compiler and ORB support necessary to implement the CORBA AMI callback model; Section 4 analyzes the results of systematically benchmarking the performance of the AMI callback implementation in TAO [6]; and Section 5 presents concluding remarks.

2 Programming the CORBA AMI Callback Model

In this section, we outline how the AMI callback model works from the perspective of a CORBA application developer. The steps required to program CORBA AMI callbacks are similar to developing any CORBA application, *i.e.*, OMG IDL interface(s) must be defined first and a client then must be written, as we describe below.

Step 1: define the IDL interface and generate the stubs: Throughout this column, we'll use our familiar `Quoter` IDL interface to illustrate how to use and implement the AMI Callback model:

```
module Stock
{
  interface Quoter {
    // Two-way operation to retrieve current
    // stock value.
    long get_quote (in string stock_name);
  };

  // ...
}
```

After IDL interfaces are defined, they are passed through an OMG IDL compiler, which generates a standard set of C++ stubs and skeletons. For each two-way operation in the IDL interface, the IDL compiler generates the synchronous and asynchronous invocation stubs that applications use to invoke operations. The skeletons generated by the IDL compiler are no different for AMI than for synchronous method invocations we've covered before [7], so we'll ignore them in this column. Stubs for asynchronous operations, however, are defined by having the same name as the synchronous operations, with a `sendc_` prefix prepended.

For example, an IDL compiler would generate the following synchronous and asynchronous stubs for our `Quoter` interface:

```
// Usual stub for synchronous invocations.
CORBA::Long Stock::Quoter::get_quote
  (const char *stock_name)
  { /* IDL compiler-generated stub code... */ }

// New stub for asynchronous invocations.
// (described below).
void Stock::Quoter::sendc_get_quote
  // ReplyHandler object reference
  (Stock::AMI_QuoterHandler_ptr,
   const char *stock_name)
  { /* IDL compiler-generated stub code... */ }
```

In addition to having a slightly different name, note how the asynchronous `sendc_get_quote` method has a different signature than the synchronous `get_quote` method. In particular, `sendc_get_quote` is passed an `AMI_QuoterHandler`, which is an object reference that determines where the reply from the server will be dispatched. Moreover, it doesn't have a return value, because the value of the stock will be passed back directly to the `get_quote` callback method on the automatically generated `AMI_QuoterHandler`, which is shown below:

```
class AMI_QuoterHandler :
  public Messaging::ReplyHandler {
    // Callback stub invoked by Client ORB
    // to dispatch the reply.
    virtual void get_quote (CORBA::Long stock_value)
    { /* IDL compiler-generated stub code... */ }
};
```

After the reply arrives from the server, the client ORB invokes the `get_quote` stub on the `AMI_QuoterHandler` callback object. This stub marshals the arguments and invokes the virtual `get_quote` method on the servant that implements the `AMI_QuoterHandler` object. For more information on the AMI callback mapping rules for OMG IDL to C++, please see [3].

Step 2: programming the client application: After the IDL compiler generates the synchronous and asynchronous stubs, programmers can develop a client that works much like other CORBA applications. First, the client must obtain an object reference to a target object and invoke an operation. Unlike a conventional synchronous two-way invocation, however, the client passes an object reference for a `ReplyHandler` object as a parameter when it invokes a two-way asynchronous operation. The client ORB keeps track of `ReplyHandler` objects for pending asynchronous invocations so that it can dispatch appropriate callback operations after servers reply.

The following code, excerpted from [3], illustrates how a C++ programmer would invoke the `get_quote` method using the AMI callback model. First, we'll define some global variables:

```
// NASDAQ abbreviations for ORB vendors.
static const char *stocks[] =
{
  "IONAY" // IONA Orbix
  "INPR"  // Inprise VisiBroker
  "IBM"   // IBM Component Broker
```

```
}
// Set the max number of ORB stocks.
static const int MAX_STOCKS = 3;

// Global reply count.
int replies_received = 0;
```

Next, we'll define our `ReplyHandler` servant implementation:

```
class My_Async_Stock_Handler
  : public POA_Stock::AMI_QuoterHandler
{
public:
  My_Async_Stock_Handler (const char *stockname)
    : stockname_ (CORBA::string_dup (stockname))
  {}

  // Callback servant method.
  virtual void get_quote (CORBA::Long value) {
    cout << stockname_ << " = " << value << endl;
  }

private:
 CORBA::String_var stockname_;
};
```

We store `stockname_` in each `QuoterHandler` servant because otherwise we can't differentiate callbacks that return from multiple invocations.

Finally, we define a function that issues asynchronous requests:

```
// Issue asynchronous requests.
void get_stock_quote (void)
{
  // ReplyHandler servants.
  My_Async_Stock_Handler *handlers[MAX_STOCKS];

  // ReplyHandler object references.
  Stock::AMI_QuoterHandler_var
    handler_refs[MAX_STOCKS];

  for (i = 0; i < MAX_STOCKS; i++) {
    // Initialize ReplyHandler servants
    handlers[i] =
       new My_Asynch_Stock_Handler (stocks[i]);

    // Initialize ReplyHandler object refs.
    handler_refs[i] = handlers[i]->_this ();
  }

  // Make asynchronous two-way calls using
  // the callback model.
  for (i = 0; i < MAX_STOCKS; i++)
    quoter_ref->sendc_get_quote
      (handler_refs[i],
       stocks[i]);
  }
  // ...
```

After making the asynchronous invocation, a client typically performs other tasks, such as checking for GUI events or invoking additional asynchronous methods. When the client is ready to receive replies from server(s), it enters the ORB event loop, using the standard `work_pending` and `perform_work` methods defined in the CORBA `ORB` interface, as follows:

```
  // Event loop to receive all replies as callbacks.
  while (reply_count > 0)
    if (orb->work_pending ())
      orb->perform_work ();
```

When a server responds, the client ORB receives the response and dispatches it to the appropriate C++ method on the `ReplyHandler` servant so the client can handle the reply. In other words, the ORB turns the response into a request on the client's corresponding `ReplyHandler` that was passed during the original invocation. Figure 3 illustrates how our client application uses the AMI Callback model. In the example above, the client implements the `ReplyHandler` object locally. A `ReplyHandler` also can be a remote object, in which case its servant will receive "third-party" replies for the requests invoked by our client.

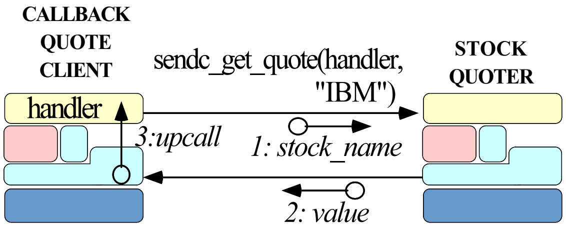


Figure 3: AMI Callback Client

3 Implementing the CORBA AMI Callback Model

Section 2 outlined how to program the AMI callback model from a CORBA application developer's perspective. This section describes how key AMI callback components can be implemented from an ORB developer's perspective. Generalizing from the example in Section 2, an ORB must implement the following functionality to support AMI callbacks:

1. Asynchronous stubs: For each two-way operation in the IDL interface, the ORB's IDL compiler should generate an asynchronous invocation stub that applications can use to issue asynchronous operations. High-quality IDL compilers should provide an option to suppress the generation of asynchronous stubs to reduce the footprint of applications that do not use them.

2. Manage pending invocations: The client ORB must keep track of `ReplyHandler` objects for all asynchronous invocations. If the `ReplyHandler` object reference points to an object that's collocated with the client, the client ORB stores the `ReplyHandler` associated with that invocation. Thus, when a reply arrives the ORB will dispatch the reply to the appropriate servant. If, however, the `ReplyHandler` object is remote, the client ORB's Object Adapter will not store any information about the `ReplyHandler`.

3. Explicit event loop methods: Implement the standard CORBA `work_pending` and `perform_work` operations that can be used to explicitly invoke the CORBA event loop in a client.

In Section 3.1, we explain the features required in an IDL compiler to generate the stubs necessary to support the AMI Callback model. Then, in Section 3.2, we discuss the various components an ORB should support to implement the AMI functionality outlined above.

3.1 IDL Compiler Support for CORBA AMI Callbacks

When AMI Callback model is enabled, an IDL file maps to an "implied-IDL" file. This "implied-IDL" will consist of the tt sendc_ method for each two-way method and the `ReplyHandler` interface for each interface found in the original IDL file.

The "implied-IDL" for the `Quoter` IDL will look as described below:

```
module Stock
{
  interface Quoter {
    // Original two-way operation.
    long get_quote (in string stock_name);

    // Implied asynch operation.
    void sendc_get_quote (in AMI_QuoterHandler hanler,
                          in string stock_name);
  };

  // ...

 // Implied type specific ReplyHandler.
 interface AMI_QuoterHandler :
   Messaging::ReplyHandler {
   // Callback for reply.
  void get_quote (in long result);
 };
}
```

An OMG IDL compiler that supports the AMI callback model should provide the functionality described below[1]. The IDL compiler may choose to generated the mapping code for the "implied-IDL" directly from the original IDL file instead of generating the "implied-IDL" file, thus avoiding the additional pass during the code generation.

3.1.1 Generate Stubs for Asynchronous Invocations

For each two-way operation found in the IDL interface, an IDL compiler generates a `sendc_` method, which client applications use to invoke methods asynchronously. The first argument of a `sendc_` method is a reference to the `ReplyHandler` object, followed by the `in` and `inout` arguments found in the signature of the original two-way IDL operation. The return type for the `sendc_` method is `void`, because the stub returns immediately without waiting for the server to reply.

In our `Quoter` application, for example, the IDL compiler generates the `sendc_get_quote` stub method in the client source file, as outlined below:

```
// Stub for asynchronous invocations.
void Stock::Quoter::sendc_get_quote
  // ReplyHandler object reference
  (Stock::AMI_QuoterHandler_ptr,
   const char *stock_name)
{
  // Setup connection.

  // Store ReplyHandler and
  // stub to handle reply (smart-stub)
  // in the ORB.

  // Marshal arguments.
  request_buffer << stock_name;

  // Send request buffer to server and return.
}
```

Figure 4 examines each of these steps in more detail.

3.1.2 Generate ReplyHandler Classes

For each interface in the IDL file, an IDL compiler generates an interface-specific class that derives from the standard `Messaging::ReplyHandler` base class. The client ORB uses this class to dispatch the reply to the servant that implements the `ReplyHandler` object. For example, the client stub header file generated by TAO's IDL compiler for the `Quoter` interface contains the following class and methods:

```
namespace Stock
{
  class AMI_QuoterHandler
     : public Messaging::ReplyHandler
  {
  public:
    // Reply handler smart-stub.
    static void get_quote_smart_stub
      (Input_CDR reply_buffer,
       AMI_QuoterHandler_ptr);

    // Callback stub invoked by Client ORB
    // to dispatch the reply.
    virtual void get_quote (CORBA::Long l);
  };
};
```

The `get_quote_smart_stub` and `get_quote` methods are stubs generated automatically by TAO's IDL compiler. We examine both of these methods below.

Smart-stubs: When the reply for an asynchronous invocation arrives, the client ORB must demarshal the arguments and demultiplex to the correct `ReplyHandler` callback, which

[1] In view of simplicity, we will avoid the discussion about the `Exception Delivery` [4] in this column.

then dispatches the reply to the servant method defined by the client application developer. For synchronous invocations, this dispatching is straightforward because demarshaling is performed by the stub that invoked the operation, which is blocked in the activation record waiting for the reply. For asynchronous invocations, however, the stub that invoked the operation goes out of scope after the request is sent when control returns to the client application. Thus, is does not block waiting for the server's reply.

To simplify the demultiplexing and dispatching of asynchronous replies, TAO's IDL compiler generates a `ReplyHandler` *smart-stub* static method for each two-way operation. These stubs are "smart" because they know how to demarshal the arguements and invoke the callback method on the reply handler object using the demarshaled arguments. In contrast, synchronous invocation stubs simply return control to the client when the demarshaling is complete.

When sending a request, the `sendc_` stub for asynchronous invocation passes a pointer to the `ReplyHandler` smart-stub method and a pointer to the `ReplyHandler` object to the client ORB. When the reply is available, the ORB invokes this smart-stub, passing in the reply buffer and the `ReplyHandler` object. For the `get_quote` method of the `Quoter` interface, TAO's IDL compiler generates `get_quote_smart_stub` method in the client stub source file, as shown below:

```
// Reply handler smart-stub.
void
Stock::AMI_QuoterHandler::get_quote_smart_stub
  (Input_CDR reply_buffer,
   AMI_QuoterHandler_ptr handler)
{
   // Result arguments.
   CORBA::Long l;

   // Demarshal results from reply_buffer using
   // CDR extraction operators.
   reply_buffer >> l;

   // Call reply handler callback method via its
   // stub.
   handler->get_quote (l);
}
```

Stubs for ReplyHandler callback methods: The stubs for the `ReplyHandler` callback methods dispatch asynchronous replies to servants that implement `ReplyHandler` objects. These stubs are invoked by the smart-stubs on behalf of the client ORB; they make *synchronous* invocations on the `ReplyHandler` object to dispatch the reply. The first argument in the callback method is the result of the asynchronous operation, followed by all the `out` and `inout` arguments of the original two-way operation defined in the IDL interface.

For the `Quoter` interface, TAO IDL generates the `get_quote` callback method shown near the beginning of Section 3.1.2.

3.1.3 Generate ReplyHandler Servant Skeletons

An OMG IDL compiler that supports CORBA's AMI callback model also generates servant skeletons for `ReplyHandler` classes. The `ReplyHandler` servant skeletons contain methods whose signatures define the result arguments, *i.e.*, the return value, followed by the `out` and `inout` arguments of the original two-way operation.

For each two-way operation in the IDL interface, a static `ReplyHandler` servant skeleton method is generated. This method demarshals the return value and any `inout` and `out` arguments. It then calls the `ReplyHandler` callback operation in the servant, which must be implemented by the client application developer. To ensure developers implement this callback operation, it is defined as a C++ pure virtual method.

For the `Quoter` interface, the TAO IDL generated `ReplyHandler` servant code in the client-side header file is defined as follows:

```
namespace POA_Stock
{
  class AMI_QuoterHandler
    : public POA_Messaging::ReplyHandler
  {
  public:
   // Pure virtual callback method (must be
   // overridden by client developer).
   virtual void get_quote
     (CORBA::Long l) = 0;

   // Servant skeleton.
   static void get_quote_skel
     (Input_CDR input_buffer);
  };
}
```

The implementation of the generated `get_quote_skel` servant skeleton extracts the AMI return value and `out`/`inout` parameters from the `Input_CDR` buffer and dispatches the upcall on the appropriate servant callback method. For example, the following code is generated by TAO's IDL compiler for the `Quoter` interface:

```
void
POA_Stock::AMI_QuoterHandler::get_quote_skel
  (Input_CDR cdr)
{
  // Demarshal the AMI ``return value.''
  CORBA::Long l;
  cdr >> l;

  // Invoke callback method on this servant.
  this->get_quote (l);
}
```

TAO's IDL compiler has been designed to be scalable and can be configured to support various optimization techniques [8]. The back end of TAO's IDL compiler uses several design patterns, such as Visitor, Abstract Factory, and Strategy [9], which makes it easier to enhance the compiler to generate AMI stubs.

3.2 ORB Architecture Support for AMI Callbacks

Below, we describe how an CORBA implementations can support the AMI callback model, focusing on the general collaboration between ORB components. Then, to focus the discussion, we examine specifically how TAO implements this feature.

3.2.1 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 4 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 asynchronous invocation consist of the following steps:

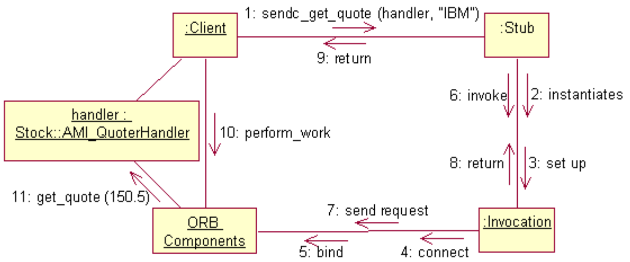


Figure 4: Interactions Between Client ORB Components for Asynchronous Invocation

- 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, *i.e.*, 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 to ensure that connections are established with the remote server **(3)** & **(4)**, store the `AMI_QuoterHandler` object in the ORB **(5)**, and send requests **(6)** & **(7)**.
- Once the request is sent, `Invocation` returns control to the `Stub` **(8)**, which itself returns control to the client **(9)**.
- When it is prepared to handle callbacks, the client application 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 `ReplyHandler` object that was passed in by the application when the AMI method was invoked originally **(11)**.

Section 3.2.3 revisits these steps in more detail after we've explained the components in TAO's ORB architecture.

3.2.2 The Design of TAO's AMI Callback Architecture

To make our discussion concrete, we now describe how the ORB architecture of TAO supports the AMI callback model. Below, we outline our resolutions to various problems encountered when migrating to TAO new AMI-enabled architecture.

Determining how to process asynchronous replies:

- **Context:** Early TAO implementations only supported the Synchronous Method Invocation (SMI) model. In SMI, the calling thread that makes a two-way invocation blocks waiting for the server's reply. Thus, the client ORB can use the calling thread to process the response.

For example, consider the *Leader/Followers* concurrency model [10] illustrated in Figure 5.

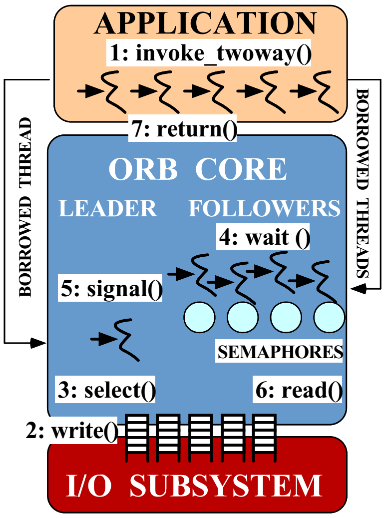


Figure 5: Synchronous Two-way Invocations using the Leader/Followers Concurrency Model

[2]The names of certain objects in this discussion are specific to TAO, though the general flow of control and behavior is generic to other ORB that implements AMI callbacks.

TAO uses this concurrency model to support multi-threaded client applications efficiently. In this concurrency model, the ORB borrows the application threads that are waiting for replies, to receive and process the replies, instead of having additional threads in the ORB to achieve that. One of the threads is chosen as the `leader` which blocks on the `select` operation. All the other threads block on semaphores. When the reply is available on any of the connections, the leader thread signals the semaphore and wakes up the correct thread that is waiting for the reply on that connection.

The following sequence of steps takes place in 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 others as the *followers*. The leader thread blocks on the `select` operation **(3)**, whereas 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 promotes the next follower to become the new leader and returns to process the reply. 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 model described above works well for SMI, it does not work for AMI. The problem stems from the fact that the calling stub goes out of scope as soon as the request is sent and the 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, the ORB must maintain certain state information, such as `ReplyHandler` object and `ReplyHandler` smart-stub, to complete the processing of server replies to asynchronous invocations.

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

- **Solution → Strategizing the reply dispatching mechanisms:** The problem of processing asynchronous replies can be solved by *strategizing* the reply processing and dispatching mechanisms used for synchronous and asynchronous invocations. Figure 6 illustrates the components in TAO's `Reply Dispatcher` hierarchy. An `Synchronous Reply Dispatcher` is created during a synchronous invocation on the local stack activation record by an `Invocation` object. When the reply is received, the reply buffer (*i.e.*, TAO's `Reply CDR` 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

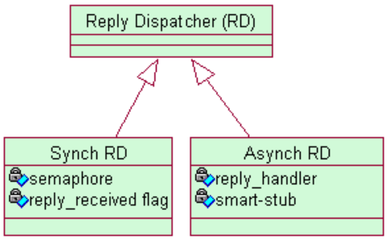


Figure 6: Reply Dispatching Strategy

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 asynchronous invocation, an `Asynchronous Reply Dispatcher` is create on the heap by an `Invocation` object. This object is created on the heap because the scope of the activation record where the `Invocation` object is created is exited before the reply is received. The asynchronous invocation stub, *i.e.*, the `sendc_*` operation, stores the `ReplyHandler` object given by the application in the `Asynchronous Reply Dispatcher` object. It also stores the pointer to the appropriate smart-stub method in this object, as well.

A Leader/Followers implementation using TAO's `Reply Dispatcher` architecture is illustrated in Figure 7. In this

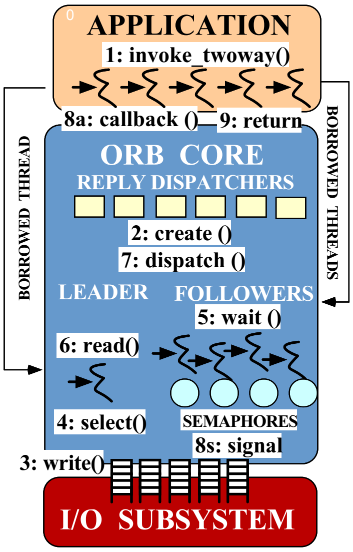


Figure 7: TAO's AMI-enabled Leader/Followers Implementation

architecture, when the application threads make the two-way invocations **(1)**, a `Reply Dispatcher` object is created for each invocation **(2)** and the request is sent **(3)**. The leader then blocks on the `select` call **(4)** and the followers 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)**. In the case of synchronous invocation, the `Synchronous Reply Dispatcher` signals **(8s)** the thread waiting for that reply and completes the invocation **(9)**. In the case of asynchronous invocation, however, the `Asynchronous Reply Dispatcher` object invokes the callback method in the `ReplyHandler` object **(8a)**.

Minimizing connection utilization:

- **Context:** Early implementations of TAO just supported a *non-multiplexed* connection model. Thus, a connection could not be used for another two-way request until the reply for the previous request was received. This non-multiplexed connection model is illustrated in Figure 8, where five threads

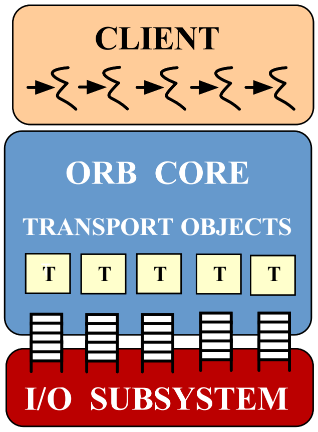


Figure 8: One Outstanding Request Per-Connection

make two-way invocations to the same server, which creates five connections. A new `Transport` object[3] is created for each connection.

- **Problem:** Non-multiplexed connection architectures are well-suited for hard real-time applications that possess highly deterministic QoS requirements [10]. A non-multiplexed connection model is inefficient for CORBA AMI, however, because applications can issue thousands of asynchronous requests before waiting for the replies. Thus, a non-multiplexed connection architecture would use a corresponding 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 are accessing a connection, the access should be synchronized so that requests are sent one-by-one and not intermingled.

3. Applications should be able to configure multiplexed and non-multiplexed connection behavior statically and dynamically to accommodate various use-cases.

- **Solution → Strategize the transport multiplexing mechanisms:** To overcome the scalability limitations of a non-multiplexed connection architecture, we extended TAO to optionally support multiplexed connections for 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, connection multiplexing yields better use of connections and other limited OS resources [10].

To implement this design in TAO, we applied the `Strategy` pattern [9] 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 9.

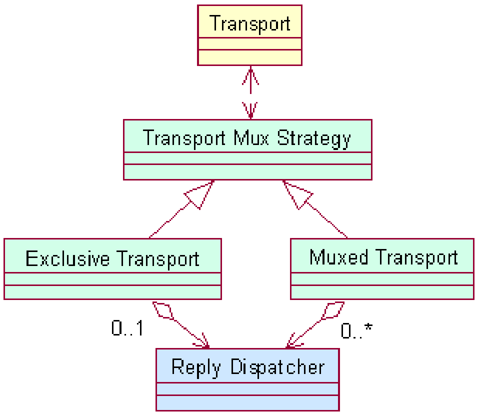


Figure 9: Transport Mux Strategy

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 more than one request is not possible at the same time. 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 9, the `Transport Mux Strategy` base class provides a common interface for these two different implementations. TAO

[3] TAO's `Transport` object provides a uniform interface to the TAO's `pluggable protocols framework` [11], which 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, such as the `Reactor`, `Acceptor`, and `Connector`, provided by ACE [12].

uses the `Service Configurator` pattern [13] so that applications can select between these two strategies and 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. During that time, if a thread tries to send another request, it either recycles a cached connection or creates a new connection. After the request is sent, the `Transport` object is marked as "idle" and cached so that it can be reused for sending subsequent requests.

Scalability of reply wait mechanisms:

• **Context:** Quality ORB implementations should support "nested upcalls," which is the ability to process incoming requests while waiting for replies. This support can be implemented using `select` to wait for both the reply and any incoming requests. However, this approach adds unnecessary overhead to "pure" clients that do not receive any requests at all. Therefore, TAO provides the following three strategies to wait for replies to allow developers to select a mechanism that's most appropriate for their applications:

1. *Wait-on-Read* – In this strategy, the calling thread blocks on `read` to receive the reply. This is a very efficient strategy for the "pure" clients that do not have to receive upcalls while waiting for replies.

2. *Wait-on-Reactor:* `Reactor` [14] is a framework implemented in ACE [12] to provide event demultiplexing and event handler dispatching. In this strategy, single-threaded `Reactor` is used to dispatch the events such as reply arrivals and upcalls.

   This strategy efficiently supports single-threaded client applications. In this approach, the waiting thread runs 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 [10].

3. *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 loop at the same time. This strategy synchronizes access to the Reactor using the Leader/Followers pattern [10]. 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 the complete reply and dispatches the reply. If the reply is for an asynchronous request, the reply gets dispatched to the callback method in the reply handler object. 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 some other thread, however, it signals this thread so that it can wakeup from the semaphore and return to its stub to process the reply.

• **Problem:** Pre-AMI-enabled versions of TAO implemented the three reply wait strategies described above within `Connection Handlers` in TAO's pluggable protocols framework, as shown in Figure 10. However, every

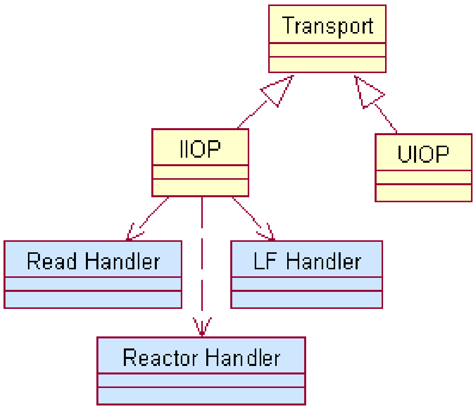


Figure 10: Earlier Implementation of Wait Mechanisms

`Transport` mechanism, such as IIOP and UNIX-domain sockets (UIOP), in TAO's pluggable protocols framework [11] reimplemented three `Connection Handler` implementations to support all the reply wait strategies in its `Transport` implementation. Not surprisingly, this approach did not scale up when TAO incorporated additional transport mechanisms, such as VME, Fibrechannel, or TP4. The original design also complicated the integration of AMI callback model, because changes to the reply wait mechanisms had to be done 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 the ORB, therefore, we moved the reply wait mechanisms from the `Connection Handlers` to the new `Wait Strategy` and decoupled it from the

Transport and the underlying Connection Handler objects. The new Wait Strategy architecture is illustrated in the UML diagram in Figure 11.

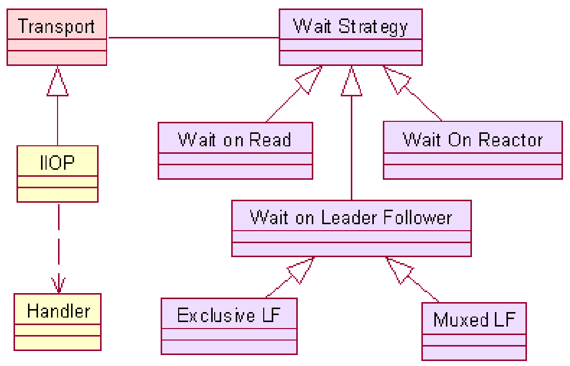


Figure 11: TAO's Enhanced Wait Strategy Implementation

In TAO's enhanced architecture, each Transport implements only one Connection Handler. Due to the patterns-based OO design used in TAO, this modification just required changes in the Transport implementation and the Connection Handler implementation and did not affect other ORB components.

In addition to the refactoring of the wait strategies, a variation on the Leader/Followers implementation has been integrated onto the Wait-on-Leader/Followers strategy. This is because the Leader/Followers implementation mentioned earlier was based on the assumption that the connection is exclusive for one request at a time. Therefore the state variables such as the Semaphores etc were kept in the Transport and the Connection Handler objects, which are per-connection objects. This implementation works fine in the Exclusive Transport case. It is not suitable for the Muxed Transport mechanism, however, because there will be multiple threads waiting simultaneously for replies on a single connection.

To address this problem, therefore, we enhanced the Leader/Followers model described above to create a variation called Muxed-Wait-on-Leader/Followers strategy class. As shown in Figure 11, the Muxed-Wait-on-Leader/Followers model uses TAO's thread specific storage implementation to keep the per-ORB-per-thread condition variable, which is created only once and also on demand by a factory method in the ORB Core. The ORB Core class provides a common place to keep the global ORB resources.

The wait strategies have been implemented using the Strategy pattern and the Service Configurator pattern is used to configure TAO's wait strategy dynamically.

Interaction between the stub and the various ORB components:

$bullet$**Context:** Now that we have discussed the various components in the ORB which the client stub can make use to achieve tasks such as setting up the connection, creating the Reply Dispatchers, sending the request, keeping track of the Reply Dispatchers and smart-stubs, waiting for replies, processing replies and delivering replies. The stub can either directly invoke methods on the various ORB components to achieve the above or it can go through helper classes which can be implemented as part of the ORB. The helper classes can interact with the various components in the ORB on behalf of the stub and execute all the above functionalities.

$bullet$**Probelm:** If the stubs are implemented to directly interact with the internal ORB components, the code size of the stub increases. This will lead to increase in the footprint of the stub files, since stubs are generated by IDL compiler for each method in the IDL interface.

$bullet$**Forces:** Therefore, stubs should make use of helper classes which will factor out as much code as possible from the stubs into the ORB core. The helper classes should efficiently support both synchronous and the asynchronous invocations.

$bullet$**Optimized invocation helper facades:** The helper classes Synchronous Invocation and Asynchronous Invocation provide the stubs with facades that encapsulate the details of the various features implemented internally to the ORB.

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 Asynchronous Invocation class is similar, but it returns control to the stub as soon as it sends the request.

As we discussed earlier in the Reply Dispatching Strategy, the Synchronous Invocation object creates the Synchronous Reply Dispatcher on the local stack activation record whereas the Asynchronous Invocation object creates the Asynchronous Reply Dispatcher on the heap.

As illustrated in Figure 12, TAO's synchronous and asynchronous variants inherit from a common Invocation class, which provides a uniform interface to other components in the ORB. Both classes delegate the tasks described above to the other components in the ORB that we discussed earlier.

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

Now that we've described (1) the code generated by TAO's IDL compiler and (2) the components in its ORB architecture

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

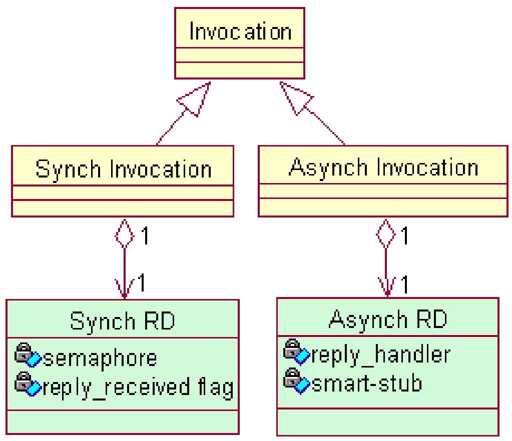


Figure 12: Invocation Interface

that process synchronous and asynchronous requests, we can present the overall AMI-enabled ORB architecture of TAO, as shown in the UML diagram in Figure 13. Moreover, we can re-

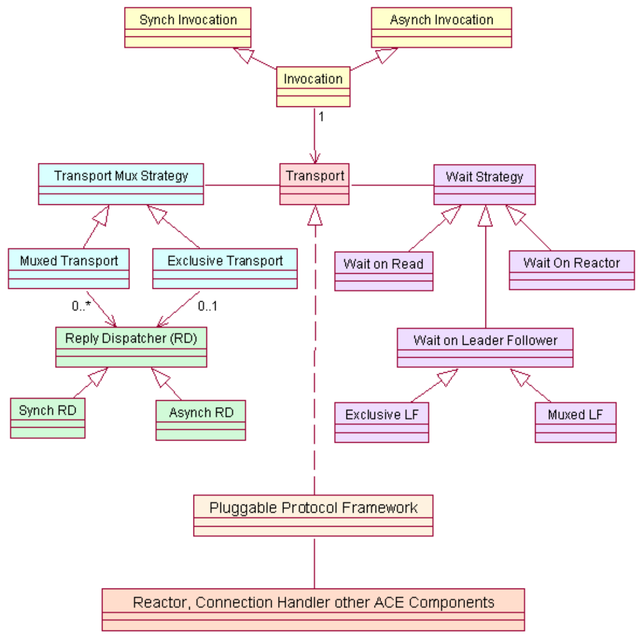


Figure 13: AMI-enabled TAO ORB Architecture

examine the sequence of steps that occur when an application issues a synchronous or asynchronous invocation. Figure 14 illustrates an interaction diagram that shows the sequence of steps in TAO, each of which is described below.

- The `Client` object calls the `Stub` to make an invocation. In the case of asynchronous request, it passes the reference to the `ReplyHandler` object **(1)**.

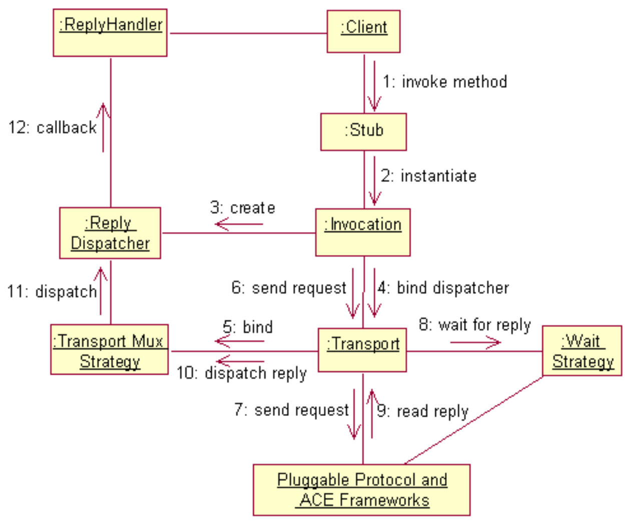


Figure 14: Sequence of Steps in TAO's SMI & AMI Invocations

- The stubs generated by TAO's IDL compiler are different for the synchronous and the asynchronous invocations. The synchronous and the asynchronous stubs instantiate the corresponding `Invocation` objects **(2)**.
- The `Invocation` object creates 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 `Transport` object which in turn uses TAO's pluggable protocols framework and ACE [12] to send the request (**6** & **7**).
- In the AMI model, the stub returns control to the application at this point. Later, the `Client` object can wait for the replies. 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 the dispatch method on it **(11)**.
- If a `Synchronous Reply Dispatcher` is being

used, it simply stores the reply buffer and sets the state variables within the object to indicate that the reply has been received and returns. Conversely, the `Asynchronous Reply Dispatcher` invokes the smart-stub stored in the object passing in the `ReplyHandler` and the reply buffer and dispatches the reply **(12)**.

4 Benchmark Results

As we discussed in Section 1, good implementations of AMI can help improve the scalability of CORBA applications by minimizing the number of client threads required to perform two-way invocations. It's also important that ORBs support both synchronous and asynchronous invocations efficiently. In particular, the support for asynchronous invocations should not increase the ORB run-time overhead for synchronous invocations.

In this section, we show how TAO's AMI implementation helps to build scalable applications, minimizing the number of client threads. We prove the efficiency of the implementation by comparing both the latency and the throughput of synchronous and asynchronous two-way invocations in TAO. The experiments were executed on a quad-CPU Sun UltraSPARC 60, running Solaris 7 with 1 GB of RAM, using the SunPro C++ v.4.2 compiler. The client and server communicate using the loopback interface, to avoid the jitter introduced by the network.

The same server is used in all the experiment, but we vary the client applications as follows:

Simple SMI client: In this case, the client is single threaded. It makes two-way synchronous requests and measures the latency for each request and the throughput.

Simple AMI client: The AMI client performs the same task, but of course by using the asynchronous callback interfaces. After sending each request it blocks on the ORB event loop to collect the reply for that request measures the latency for each request that the total time required to make all the requests to obtain the throughput.

MT SMI client: This is similar to the simple SMI client, but there are multiple concurrent threads executing the same task.

High-throughput AMI client: This client has one thread to send asynchronous requests and another thread dedicated to receive the replies. To measure the latency the request is time stamped by the sending thread.

The results of our benchmarks are presented below.

Figure 17 compares the throughputs of the MT SMI client and High-performance AMI client. The MT SMI has `t` threads and issues `n` requests. In the High-performance AMI client, the sender thread issues `tn` asynchronous requests, essentially performing the same task that the MT SMI application does.

In this experiment, the MT SMI client has 10 threads and each thread invokes a synchronous two-way invocation 1000 times. The High-performance AMI client issues 10,000 asynchronous invocations in one thread and handles replies in some ... ughput becau... ontext switch... hieves high t... replies async...

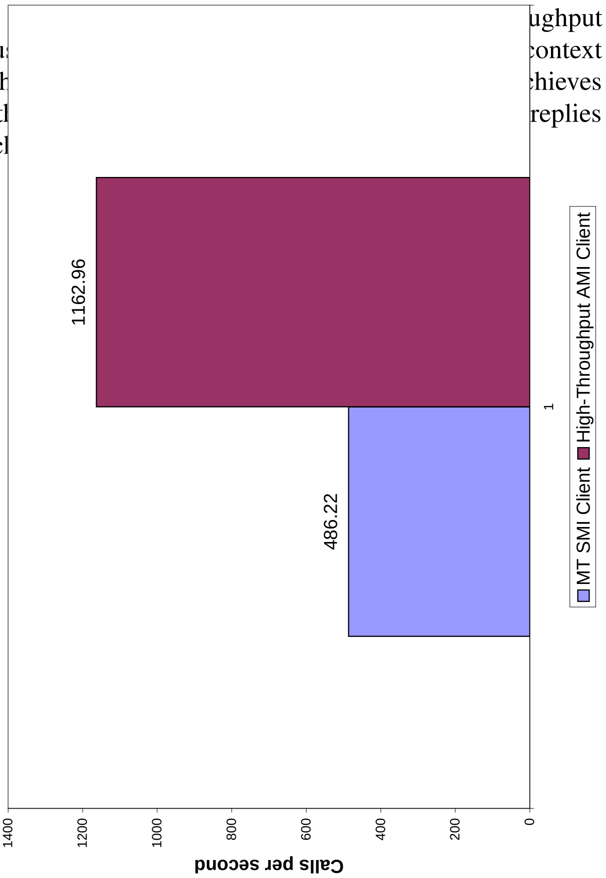
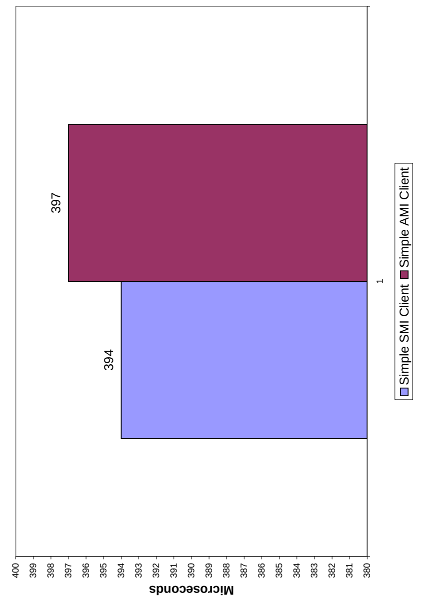


Figure 15: Blackbox Throughput for Synchronous vs. Asynchronous Method Invocations

This supports our contention that AMI can be used to build more scalable clients than traditional methods such as multi-threading; as we will show next this is achieved without considerable overhead over SMI for simple clients.

Figure 16 shows the latency for the synchronous and the asynchronous invocations. It shows that the ORB introduces *very* little overhead for the asynchronous invocations. Part of the latency overhead is due to memory allocations for the ReplyDispatcher objects, in the future we plan to use specialized memory pools to minimize this source of overhead. We also expect that further white box analysis will allow us to identify and eliminate other sources of overhead.

Figure 17 shows the throughput for the synchronous and



Figu[...]hronous vs. Asyn-
chro[...]

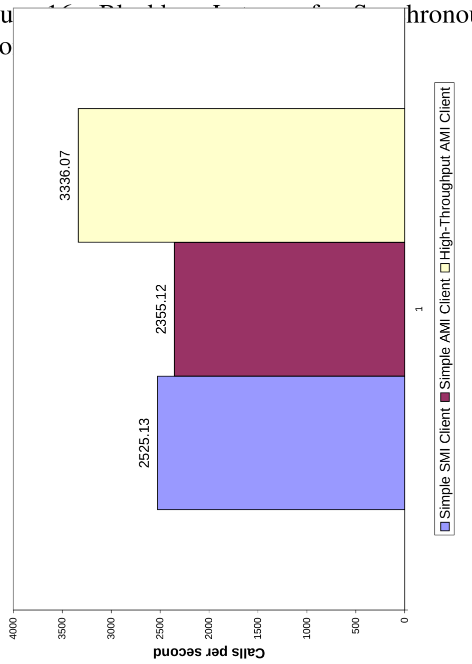


Figure 17: Blackbox Throughput for Synchronous vs. Asynchronous Method Invocations

the asynchronous invocations. As shown in the figure, there is a only a small decrease in the throughput for the simple AMI client. However, the high-performance AMI client, which has dedicated threads to issue the requests and to handle the replies, achieves higher throughput compared to the other clients, since the calling thread does not spend time on waiting for replies.

5 Concluding Remarks

The CORBA AMI model is an important feature that has been integrated into CORBA via the CORBA Messaging specification. A key aspect of AMI is that operations can be invoked asynchronously using the *static invocation interface* (SII), thereby eliminating much of the complexity inherent in the *dynamic invocation interface* (DII)'s deferred synchronous model.

This column explains how IDL compilers and ORBs can be structured to support the CORBA AMI callback model efficiently and scalably. The ORB should implement the synchronous and asynchronous reply handling as transparent as possible to the other components in the ORB. This makes the ORB components to be more flexible and scalable. Optimizations such as connection multiplexing should be supported in the ORB to efficiently support AMI clients. To avoid the footprint in the IDL generated stubs, ORB core implementation should factor out as much code as possible out of the stubs. The IDL compiler and ORB enhancements to support AMI should be very carefully made so that they do not add overhead to synchronous method invocations. The implementation should be guided by benchmarks and profiling on the newer enhancements. The existing optimizations in the ORB should be preserved, while allowing flexibility to configure the ORB based on the application needs.

We also showed how familiar design patterns can 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 [9] to support both scalable connection multiplexing strategies, while still allowing configurations that ensure the determinism required for hard real-time applications. Likewise, applications can configure these method invocation strategies using the Service Configurator pattern [9], which makes the TAO framework highly configurable and flexible.

As always, if you have any questions about the material we covered in this column or in previous ones, please email us at `object_connect@cs.wustl.edu`.

References

[1] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, October 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] 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.

[7] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology," *C++ Report*, vol. 9, November/December 1997.

[8] A. Gokhale, D. C. Schmidt, C. O'Ryan, and A. Arulanthu, "The Design and Performance of a CORBA IDL Compiler Optimized for Embedded Systems," in *Submitted to the LCTES workshop at PLDI '99*, (Atlanta, GA), IEEE, May 1999.

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

[10] 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*, To appear 1999.

[11] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Proceedings of the IFIP 6^{th} International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.

[12] 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.

[13] P. Jain and D. C. Schmidt, "Dynamically Configuring Communication Services with the Service Configurator Pattern," *C++ Report*, vol. 9, June 1997.

[14] 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.