Design and Performance of Asynchronous Method Handling for CORBA

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

This paper has been submitted to the Distributed Objects and Applications (DOA) conference, Irvine, CA, October/November, 2002.

Abstract

This paper describes the design and performance of a new asynchronous method handling (AMH) mechanism that allows CORBA servers to process client requests asynchronously. AMH decouples the association of an incoming request from the run-time stack that received the request, without incurring the context-switching, synchronization, and data movement overhead of conventional CORBA multi-threading models. A servant upcall can therefore return quickly, while the actual work performed by the servant can run asynchronously with respect to other client requests.

This paper provides two contributions to the study of asynchrony for CORBA servers. First, it describes the design and implementation of AMH in The ACE ORB (TAO), which is a widely-used, high-performance, open-source implementation of CORBA. The syntax and semantics of AMH are defined using the CORBA Interface Definition Language (IDL), the forces that guided the design of AMH are described, and the patterns and C++ idioms used to resolve these forces to implement AMH in TAO are presented. Second, we empirically compare a middle-tier server implemented using AMH against other CORBA server concurrency models, such as thread pool, thread-per-connection, and thread-per-request. The benchmarks show that AMH delivers better throughput and scalability for heavily loaded servers, though it lags a bit in performance for lightly loaded servers. Analysis and optimization techniques to improve the performance of AMH are then described.

1 Introduction

Problem → scalable servers. For many types of distributed applications, the CORBA asynchronous method invocation (AMI) mechanism can improve concurrency, scalability, and responsiveness significantly [1]. AMI allows clients to invoke multiple two-way requests without waiting synchronously for responses. The time normally spent waiting for replies can therefore be used to perform other useful work.

CORBA AMI is completely transparent to servers, *i.e.*, a server does not know whether a request it received, emanated from a synchronous or asynchronous method invocation. Therefore, while AMI improves throughput in client applications, it does not improve server applications, particularly *middle-tier servers* [2]. In these architectures, one or more intermediate servers are interposed between a source client and a sink server, as shown in Figure 1.

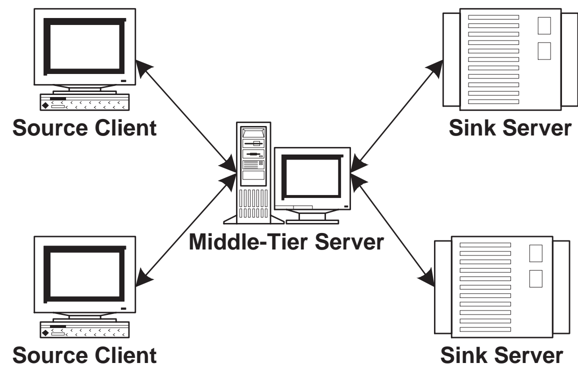

Figure 1: **A Three-tier Client/Server Architecture**

Middle-tier servers can be used for many types of systems, such as (1) a firewall gateway that validates requests from external clients before forwarding them to sink server or (2) a load-balancer [3] that distributes access to a group of database servers. In both cases, the middle-tier servers act as *intermediaries* that accept requests from a client and then pass the requests on to another server or external data source. When an intermediary receives a response, it sends its own response

*This work was supported in part by ATD, SAIC, and Siemens MED.

back to the source client.

The general behavior of a middle-tier server is summarized in Figure 2, where a middle-tier server blocks awaiting a reply to return from a sink server. The following steps typically occur in this type of server:

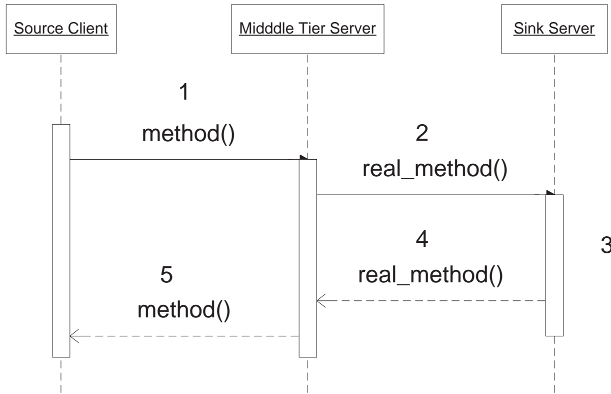

Figure 2: **Conventional CORBA Middle-tier Server Interactions**

1. The source client invokes an operation on the middle-tier server
2. The middle-tier server processes the request and invokes an operation on a sink server
3. The sink server processes the operation
4. The sink server returns the data to the middle-tier server and
5. The middle-tier server returns the data to the client.

Unlike ordinary source clients, middle-tier servers must communicate with multiple source clients and sink servers. They must therefore be highly scalable to avoid becoming a bottleneck. A common way to improve the throughput of a middle-tier server is to multi-thread it using various concurrency models [4], such as thread pool, thread-per-connection, or thread-per-request. In these models, threads can process new incoming client requests even while other threads are blocked waiting to receive a response from a sink server. Due to the cost of thread creation, context switching, synchronization, and data movement, however, it may not be scalable to have many threads in the server. In particular, each of the above concurrency models have the following limitations:

- **Thread pool**—The number of threads in the pool limits the throughput of the thread pool model. For example, if all threads are blocked waiting for replies from sink servers no new requests can be handled, which can degrade the throughput of busy middle-tier servers.
- **Thread-per-request**—If each request creates a new thread, this concurrency model may not scale when a high volume of requests spawns an excessive number of threads.
- **Thread-per-connection**—The server can also run out of threads in this model if a large number of clients connect to the server at once. Moreover, if a server is busy processing a client's request, that client can open a new connection to send a new request. If the server is slow in processing a client request, a single client may create a large number of connections and threads on the server, further slowing it down.

The overhead for threads motivates the need for another way to increase middle-tier server scalability. Unfortunately, these servers cannot leverage the benefits of AMI fully since AMI only provides asynchrony to clients. In a middle-tier server, therefore, outgoing requests can use AMI to return control from the ORB (Object Request Broker) quickly, but the request handler for incoming requests must remain active until a response can be returned to the source client. In particular, each request needs its own activation record, which effectively restricts a request/response pair to a single thread in standard CORBA.

Solution → Asynchronous method handling. Asynchronous method handling (AMH) is a technique that extends the capabilities of AMI from CORBA clients to CORBA servers. AMH is a CORBA-based variant of the *continuation model* [5], which allows a program's run-time system to transfer the control of a method closure from one part of the program to another. AMH is useful for many types of applications, particularly middle-tier servers in multi-tiered distributed systems.

Figure 3 illustrates the sequence of steps involved in handling a request by an AMH-enabled middle-tier server. Each of these steps is described below:

1. The source client invokes an operation on the middle-tier server
2. The middle-tier server uses AMH to store information about the client request in a heap-allocated object called `ResponseHandler` and returns control to the ORB immediately
3. The sink server processes the request
4. The sink server returns the data to the middle-tier server and
5. The middle-tier server fetches the corresponding `ResponseHandler` and uses it to send the reply data back to the client.

Since the middle-tier server need not block waiting to receive a response, it can handle many requests concurrently using a single thread of control.

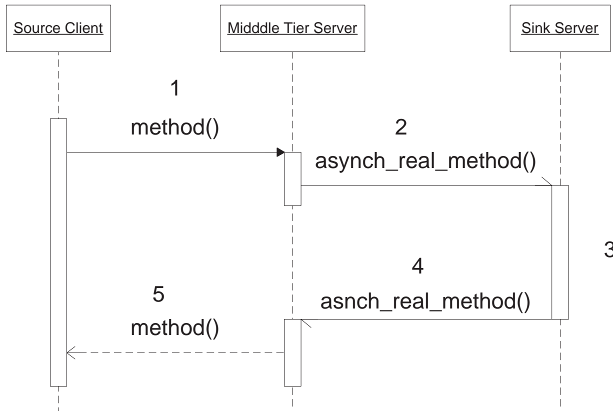

Figure 3: **AMH CORBA Middle-tier Server Interactions**

Adding AMH to a middle-tier server that also uses the AMI callback model yields a server that can take full advantage of asynchrony. Such an AMI/AMH server has many of the same properties of a message-oriented middleware (MOM) [6] system, which is well suited for middle-tier servers. Much as AMI allows a client application to provide a reply handler object to an ORB, AMH allows an ORB to provide a response handler object to a server application's servant. The server application can either return a response via this response handler during the servant upcall or at some point later during the server's execution, obviating the need to use threads to achieve concurrency.

Paper outline. The remainder of this paper is organized as follows: Section 2 specifies the AMH capabilities using the CORBA interface definition language (IDL); Section 3 describes the challenges faceed while designing and implementing AMH and how we resolved them using patterns and C++ idioms; Section 4 illustrates with an example how AMH can be applied and the inner workings of AMH; Section 5 examines the results of empirically comparing TAO's AMH implementation with other common server concurrency models, such as thread-per-connection and thread pool; Section 6 outlines future enhancements and research directions related to AMH; Section 7 compares our work on AMH with related research; and Section 8 presents concluding remarks.

2 The Asynchronous Method Handling Specification

This section describes the interface and semantics of AMH. We specify AMH using OMG IDL so that it can be implemented in CORBA-compliant ORBs and mapped to many programming languages. Since AMH is not yet part of the Object Management Group (OMG) CORBA standard [7], an IDL specification provides a common schema for describing AMH.

As with the CORBA AMI specification, we define the semantics of AMH using "implied-IDL." Implied-IDL refers to additional IDL that the IDL compiler "logically" generates based on the standard IDL declarations it parses.[1] The original IDL and the implied IDL are then compiled into stubs and skeletons.

To focus our discussion, we use the following IDL interface for all our examples:

```
module Stock {
  exception Invalid_Stock_Symbol {};

  interface Quoter {
    long get_quote (in string stock_name)
      raises (Invalid_Stock_Symbol);
  };
};
```

The AMH implied-IDL for the above `Stock::Quoter` interface is:

```
module Stock
{
  // Forward declarations.
  local interface AMH_QuoterResponseHandler;
  valuetype AMH_QuoterExceptionHolder;

  // The AMH skeleton.
  local interface AMH_Quoter {
    // The AMH operation.
    void get_quote
      (in AMH_QuoterResponseHandler handler,
       in string stock_name);
  };

  // The AMH ResponseHolder.
  local interface AMH_QuoterResponseHandler {
    // Operation to send asynchronous reply
    void get_quote (in long return_value);

    // Operation to asynchronous exception.
    void get_quote_excep
      (in AMH_QuoterExceptionHolder holder);
  };

  // Exception Holder for raising AMH exceptions.
  valuetype AMH_QuoterExceptionHolder {
    void raise_get_quote ()
         raises (Invalid_Stock_Symbol);
  };
};
```

We next describe the rules by which the additional interfaces are generated.[2] The implied-IDL for AMH has three new generated interfaces that we describe below.

The AMH Skeleton. The AMH skeleton (`AMH_Quoter`) is the AMH version of a normal skeleton. A skeleton demarshals method arguments, invokes an upcall to the designated

[1] In the case of TAO, this is triggered by specifying the `-GH` option to TAO's IDL compiler.

[2] Though we use the `Quoter` interface to illustrate the generation of implied-IDL, the rules themselves can be applied to any IDL interface.

servant method with these arguments, and marshals any return/out parameters. All operations in an AMH skeleton can return control back to the ORB immediately, *i.e.*, they need not block while the server processes the invocation.

The AMH skeleton interface contains all the operations specified in the original interface. However, the signatures of the AMH skeleton operations (which we refer to as *asynchronous operations*) are different from the original operations in the following ways:

- The `in` and `inout` parameters in each operation of the original IDL interface are mapped to `in` parameters for each asynchronous operation.
- An extra `in` parameter of type `ResponseHandler` (described below) is passed as the first argument.
- The asynchronous operation has a `void` "return" type.
- The `out` and return arguments are omitted from the asynchronous operation.

When used in conjunction with some straightforward programming idioms shown in Section 4.1, these changes enable an asynchronous operation to return immediately. In contrast, a non-AMH operation must wait for the processing of an upcall to complete before it can return control back to the ORB.

The AMH ResponseHandler. The `ResponseHandler` (`AMH_QuoterResponseHandler`) object stores the relevant client request information. All operations in the original interface are present in the `ResponseHandler` interface. Invoking a `ResponseHandler` operation sends a reply (`out/inout/return` values) to the client .

The implied-IDL `AMH_QuoterResponseHandler` interface is related to the `Quoter` interface as follows:

- The `out`, `inout`, or return values for an operation in the original interface are mapped to `in` parameters in the corresponding method of the `ResponseHandler` interface.
- The `in` parameters for an operation in the original interface are omitted in the corresponding operation of the `ResponseHandler`.
- All `ResponseHandler` operations have a void "return" type.

All implied-IDL `ResponseHandlers` are *local interfaces*, *i.e.*, they are always collocated in the address space of the server. Although they appear as regular CORBA objects to server programmers, they cannot be passed or accessed outside of the server's address space. Another special characteristic of a `ResponseHandler` is that it can be invoked only once. Invoking it more than once raises the `BAD_INV_ORDER` system exception in the servant.

Any ORB-specific state that is needed to send a reply to the client is stored in a base `ResponseHandler` class. This state includes the connection on which the request arrived and the service context of the request. All derived `ResponseHandlers` (*e.g.*, the `AMH_QuoterResponseHandler`) can be viewed as deriving from the base `ResponseHandler`. The base `ResponseHandler` is not specified by any implied-IDL; in TAO we implement this interface in the concrete class `TAO_AMH_ResponseHandler`.

The AMH `ResponseHandler` also contains an *exception operation* (`get_quote_excep()`) for every operation (`get_quote()`) in the original IDL interface. The exception operation coordinates with the `ExceptionHolder` (see next paragraph) and the stored state to return the exception to the client.

The AMH exception holder. The `ExceptionHolder` (`AMH_QuoterExceptionHolder`) object is used to store a user- or system-defined exception. It is also used by the exception operation (Section 2) to send an exception to the client. The AMH Exception Holder is generated according to the following rules:

- For every operation (`get_quote()`) in the original interface, a corresponding *raise operation* (`raise_get_quote()`) is present in the exception holder.
- The signature of the `raises` clause of the raise operation matches the `raises` clause of the corresponding original operation exactly.

3 Resolving AMH Design Challenges

TAO's AMH implementation was designed to resolve the following challenges:

- Providing complete client transparency
- Ensuring AMH servants have the same semantics as non-AMH servants
- Minimizing memory footprint and
- Simplifying maintainability and evolution of AMH functionality.

This section describes how we designed AMH to resolve these challenges.

3.1 Challenge 1: Providing Complete Client Transparency

Problem. AMH is purely a server mechanism. It should therefore be completely transparent to clients, which must not require changes to interact with an AMH server.

Solution. The AMH skeleton (`AMH_Quoter`) interface is a server-specific interface that is not visible to clients. The AMH skeleton is transparent to the client because it masquerades as the original server (`Quoter`) interface and receives and handles all the client operation invocations. We use the `Quoter`

interface as a concrete example below to demonstrate how the masquerade works.

In the `_this()` method of `AMH_Quoter`, we return a `Quoter` object reference instead of an `AMH_Quoter` object reference, which implies that the `AMH_Quoter` servant has registered itself to implement the `Quoter` interface. The `_this()` code is shown below:

```
class AMH_Quoter
  : public virtual PortableServer::ServantBase
{
public:
  Stock::Quoter *_this ();
};
```

When the object reference exported by this `AMH_Quoter` servant is narrowed by the client, the resulting reference is a `Quoter` and not an `AMH_Quoter`. Whenever the client invokes an operation on the `Quoter`, the operation parameters are marshaled using the `Quoter` stub.

On the server, the `AMH_Quoter` skeleton demarshals the arguments it is passed by the ORB. The logic to demarshal the arguments for an `AMH_Quoter` that have been marshaled using the `Quoter` interface are generated automatically by TAO's IDL compiler since the rules to generate the `AMH_Quoter` implied-IDL from the original Quoter IDL are specified rigorously (Section 2). The client is therefore oblivious to whether its request is handled synchronously or asynchronously. Moreover, the `is_a()` method of the `AMH_Quoter` is also changed to return `true` when the object reference being tested is a `Quoter`, thereby making the `AMH_Quoter` completely transparent to the client.

3.2 Challenge 2: Ensuring AMH Servants have the Same Semantics as non-AMH Servants

Problem. Server application developers should be able to program AMH servants largely like they do non-AMH servants. For example, the semantics of registering AMH servants with the POA (Portable Object Adapter) and the interaction with the ORB should be the same as with non-AMH servants. Having the same semantics simplifies ease of use and yields faster adoption of the AMH mechanism.

Solution. If the application programming interface between the AMH servant to the ORB and POA is kept the same as a non-AMH servant, the semantics of use of AMH servants is the same as non-AMH servants. AMH servants differ from non-AMH servants in the skeleton classes they derive from. Thus, if all changes required for asynchrony are restricted to AMH skeletons, then we can provide the required guarantee of semantics.

In TAO, the `ResponseHandler` is created by the AMH skeleton (Section 4.2.1). In turn, the `ResponseHandler` duplicates certain ORB data structures that it needs (Section 4.2.2). Thus, by taking on the responsibility for creating the `ResponseHandler`, the AMH skeleton has made asynchrony transparent to the ORB and the POA, from an application developer perspective.

This design provides maximum flexibility, while requiring minimal changes to existing servant code. For example, AMH servants can be registered in any POA, even along with normal servants. No new POA-Policies need to be defined to make a servant asynchronous. AMH servants can be created and used just as any other servant. Also, the ORB transparency makes AMH orthogonal to the threading model in the server; AMH servants can be used in any multi-threaded server, *e.g.*, thread-per-connection or thread pool, supported by TAO.

3.3 Challenge 3: Minimizing Memory Footprint

Problem. Memory footprint is crucial to many types of embedded systems and increases in memory footprint can reduce the benefits of using an ORB for memory-constrained embedded systems. Thus, clients residing in embedded systems that need to contact an AMH server must not pay any significant memory penalty.

Solution. We followed the following three guidelines to minimize memory footprint:

- Confine all changes required in the ORB for the AMH mechanism to a server-specific library that pure clients do not link.
- Restrict all IDL generated code for implementing AMH to skeleton files.
- If IDL generated code is needed in the stub files, make the classes abstract in the stub files and define their implementation by deriving form these abstract classes in the skeleton files.

All ORB-related AMH code is subsetted into the `PortableServer` library. Since this is a server library, pure clients never link with it. In addition, since we ensured that AMH servers are completely transparent to clients, pure clients need no additional functionality(and hence no additional code or memory) to contact AMH servers.

Clients compiled with stub files that have been generated without the AMH option (`'-GH'` flag) have no AMH-related code in their stub files. If for some reason clients must be linked with stub files that have been generated with the AMH option, they still pay minimal memory overhead. The `ResponseHandler` (`AMH_QuoterResponseHandler`) is the only class that needs to be present in the stub files (Section 3.4, Figure 4). Since this class is declared *abstract* in the stub header file, it requires no definition.

3.4 Challenge 4. Simplifying Maintainability and Evolution of AMH Functionality

Problem. AMH is not (yet) a CORBA standard so any changes that occur in the standard must be easy to incorporate. Moreover, the AMH functionality in TAO must be easy to extend and maintain.

Solution. The various AMH components (both the IDL generated classes and the ORB files) have been designed so as to to cleanly separate and encapsulate functionality. The base `ResponseHandler` class (`AMH_QuoterResponseHandler`) is declared abstract in the stub header file.[3] We implement the `ResponseHandler` functionality in the skeleton files (`TAO_AMH_QuoterResponseHandler`) by deriving from the base `ResponseHandler` class. This cleanly separates the `ResponseHandler` interface from it's implementation. An added benefit is that clients do not have to link in any server code even though a `ResponseHandler` is purely a server functionality.

We have taken this separation of concerns a step further and encapsulated all ORB-Core related functionality of a `ResponseHandler` in a TAO specific class called `TAO_ResponseHandler`. All ORB-specific data structures manipulation and interaction is localized to this class. Thus, the final implementation of a `ResponseHandler` (`TAO_AMH_QuoterResponseHandler`) only implements the marshaling of the return, `out`, and `inout` values. Localizing all ORB-interaction in a ORB-specific file also ensures that the IDL compiler generated code for a `ResponseHandler` is minimal, thereby reducing the code in the IDL compiler itself. The inheritance hierarchy of the `ResponseHandler` is shown in Figure 4.

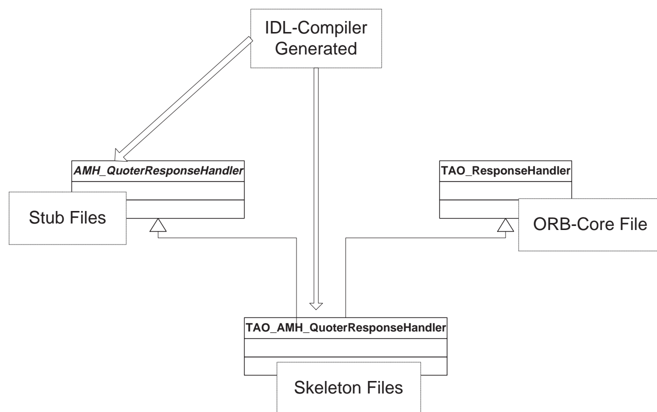

Figure 4: **The** `ResponseHandler` **Class Hierarchy**

[3]The `ResponseHandler` implied-IDL is a `local` interface, so it must be declared in the stub files.

All the differences between an asynchronous servant and a synchronous servant are present only in the AMH skeleton. Likewise, all ORB-Core related functionality of a `ResponseHandler` is encapsulated in `TAO_ResponseHandler`. This design ensures changes to AMH functionality are restricted to a known and minimal set of classes, without effecting the rest of the ORB-core code. Moreover, the IDL compiler-generated code is separated from the ORB-implementation of AMH functionality, which makes it easy to change ORB functionality without affecting the IDL compiler code that generates AMH stubs and skeletons. Thus, IDL compiler code, ORB code for asynchronous servants, and all other ORB code is cleanly separated so that each can evolve separately without adversely affecting the other.

4 Applying AMH In Practice

This section first illustrates how to program TAO's AMH implementation and then shows how it works internally. We also illustrate how to send exceptions via AMH.

4.1 An Example AMH Server Application

We illustrate how AMH may be used in a middle-server that is based upon the `Stock::Quoter` interface from Section 2.[4] This server accepts client requests for stock quotes and forwards them to sink server(s). The sink server processes the stock-quote request and then sends the reply to the middle-server, which forwards it to the client. The middle-server operates as follows. It obtains the reference to the sink server(s) when it starts up and instantiates the servant with it. For simplicity, we only use one instance of a sink server in this example.

```
int main (int argc, char *argv[])
{
  // Initialize the ORB and POA as usual...

  // Get a reference to the sink server.
  CORBA::Object_var obj =
    orb->string_to_object (argv[1]);
  Stock::Quoter_var sink_server =
         Stock::Quoter::_narrow (obj.in ());

  // Create an AMH enabled servant
  Stock_AMHQuoter_i *amh_servant =
         new Stock_AMHQuoter_i (sink_server);

  // Register servant as usual with POA
  Stock::Quoter_var quoter = amh_servant->this ();

  // Export quoter reference for clients to use...
}
```

The AMH servant stores the reference to the sink server (`target_quoter_`) and uses it in the `get_quote()`

[4]A complete example of an AMH middle-tier server is available in the TAO open-source release in `$TAO_ROOT/examples/AMH`.

method to send the request to the sink server. The servant implementation for the asynchronous-operation looks like this:

```
void Stock_AMHQuoter_i::get_quote
       (Stock::AMH_QuoterResponseHandler_ptr rh,
        const char *stock_name)
{
  // Create AMI Callback object.
  Quoter_Callback *callback =
    new Quoter_Callback (rh);

  // Activate callback.
  AMI_Quoter_var callback = callback->_this ();

  // Invoke the AMI request.
  target_quoter_->sendc_get_quote (callback,
                                   stock_name);
}
```

The `get_quote()` method returns almost immediately since `sendc_get_quote` is asynchronous. The middle-server is now ready to accept another client request.

The AMI `Quoter_Callback` class stores the `ResponseHandler (rh_)` internally and uses it to send the reply to the client as follows:

```
void Quoter_Callback::get_quote (CORBA::Long retval)
{
  rh_->get_quote (retval);
}
```

The AMH servant stores the `ResponseHandler` into the AMI callback object so that when the reply from the sink server arrives, the AMI callback object can send the reply to the client.

4.2 AMH in Action

Now that we've shown how to program an AMH server, we can describe its internal behavior. The remainder of this section describes how AMH handles a request asynchronously, stores enough state from a request so that the reply can be sent later, and sends exceptions asynchronously.

4.2.1 Asynchronous Request Handling

The stub and skeleton classes generated for each IDL interface by an OMG IDL compiler are used as follows:

- Client applications use stub classes to narrow server references and to marshal/demarshal method arguments (when operations are invoked on the narrowed reference).
- Server applications implement servant classes that derive from skeleton classes and implement the functionality for the interface.

Figure 5 shows how the various components (servant/skeleton/poa/orb) interact to handle an asynchronous request:

1. The client request is received by the server ORB and is dispatched to the POA as usual.

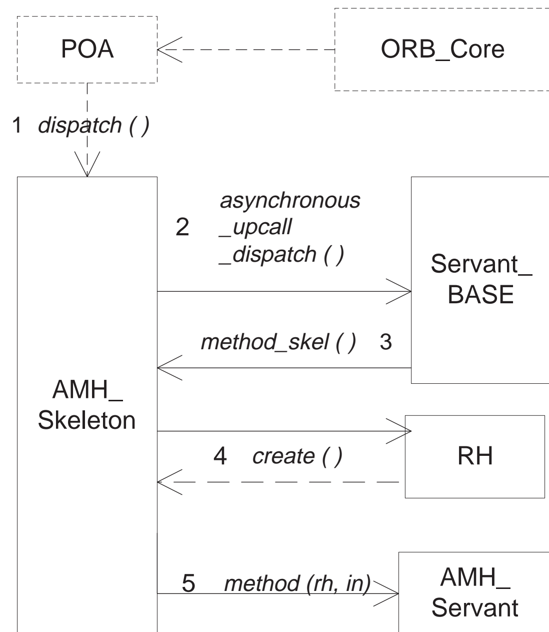

Figure 5: **TAO's AMH Implementation**

2. The POA locates the servant that handles the request and dispatches the request to it. Until this point, the path taken by the client request is identical to the path of a synchronous request.
3. The servant that is dispatched by the POA is an asynchronous servant (henceforth referred to as *AMH servant*), since it derives from an asynchronous skeleton (henceforth referred to as an *AMH skeleton*). Section 3.1 describes how the AMH servant registers with the POA to handle asynchronous requests.
4. The `Servant_Base` calls the appropriate method of the derived AMH skeleton.
5. The AMH skeleton method first demarshals the `in` and `inout` parameters and then creates the `ResponseHandler` for the request.
6. The `ResponseHandler` along with the `in` parameters are then passed to the AMH servant that processes the request.

Interfacing the AMH Skeleton to the ORB-Core: All skeletons (whether synchronous or asynchronous) derive from a base skeleton class called the `Servant_Base` class. In the case of a synchronous request, this base class dispatches the method in the derived skeleton class and sends the marshaled `out`, `inout`, and return parameters back to the client. To implement AMH, we added a new method in the `Servant_Base` class called `asynchronous_upcall_dispatch()`, which upcalls the method in the derived AMH

skeleton class but does not implement the functionality of sending the return values to the client. This design enables the asynchronous-upcall to return immediately without waiting for the upcall to process the request and marshal the `out`, `inout`, and return parameters. By restricting all changes to within the skeleton classes we ensure that asynchronous servants are transparent to both the ORB and the POA. Section 3.2 describes the advantages of this degree of transparency in TAO's AMH design.

4.2.2 Asynchronous Reply

All `ResponseHandler` classes derive from a base class `ResponseHandler` called `TAO_AMH_ResponseHandler`, as described in Section 2. When a server finishes processing an upcall and is ready to send a reply to the client, it invokes the `ResponseHandler` with the appropriate `out`, `inout`, and return parameters. Figure- 6 shows how the `ResponseHandler` sends the reply back to the client.

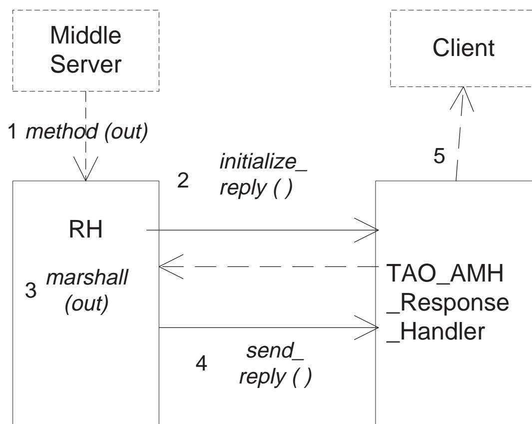

Figure 6: **TAO's Asynchronous Return Implementation**

Each of these steps is described below:

1. The middle-tier server invokes the `ResponseHandler` (RH) with the appropriate parameters
2. The derived RH invokes a method on its base RH to initialize the ORB parameters needed to send a reply
3. The derived RH marshals the `out`, `inout`, and return parameters
4. The derived RH then invokes the base class's `send_reply` method and
5. This method sends the marshaled parameters to the client.

Interfacing the ResponseHandler to the ORB-Core: When the derived RH is first constructed, various ORB parameters are stored in the heap memory by the RH constructor These parameters are subsequently used when the marshaled parameters are sent to the client. The derived RH functionality is limited to only marshaling the parameters, *i.e.*, all ORB-specific functionality is present in the base `TAO_AMH_ResponseHandler` class. The advantages of this design are described in Section 3.3.

4.3 Asynchronous Exceptions

To send exceptions asynchronously, the exception operation (Section 2) of the `ResponseHandler` is used. As usual, the exception to be sent is constructed by the server application developer:

```
Stock::Invalid_Stock_Symbol stock_exception;
```

This exception is now "wrapped" by an `ExceptionHolder` (Section 2):

```
Stock::AMH_QuoterExceptionHolder holder
  (&stock_exception);
```

Invoking the exception operation of the `ResponseHandler` with the `ExceptionHolder` results in the exception being sent to the client, as follows:

```
// POA_Stock::TAO_AMH_QuoterResponseHandler rh;
rh.get_quote_excep (&holder);
```

Internally, the `ResponseHandler` invokes the raise operation of the `ExceptionHolder`. This operation raises the exception that has been stored when the `ExceptionHolder` was constructed. The `ResponseHandler` catches the raised exception and after narrowing it to the correct type, uses the state saved in the `ResponseHandler` to marshals the exception to the client. The restriction of the raise operation having the same raises clause as the original operation ensures that only the exceptions declared for that operation can be sent to the client.

5 An Empirical Comparison of ORB Concurrency Models

A key goal motivating our work on AMH was to improve the scalability of CORBA servers. To determine whether our design and implementation of AMH in TAO achieved this goal, we benchmarked the throughput of AMH against other common CORBA server models: thread-per-connection, thread pool and single-threaded reactive model. We defined throughput as the total number of replies that the middle-tier server sends to the client divided by the total time the middle-tier server takes to send those replies. The benchmark involved a middle-tier server run using various types of concurrency models. This section describes the benchmark testbed and the results of our experiments.

5.1 Overview of Benchmarking Testbed

Hardware and systems software. The benchmark was performed using three separate machines all interconnected by a 10 Mbps LAN. Figure 7 shows the configuration of the

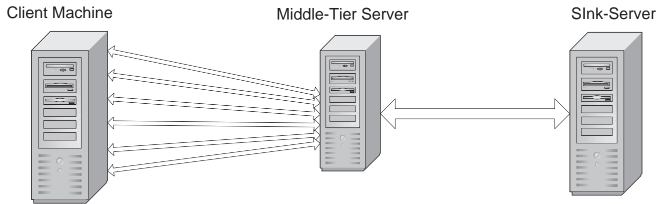

Figure 7: **Benchmarks Testbed**

testbed, which is outlined below:

- The *source client machine* was a dual Intel XEON CPU machine with 1GB of RAM and each CPU running at 1,700MHz.
- The *middle-tier server machine* was an Intel single CPU 733 MHz machine with 512KB of RAM.
- The *sink-sever machine* was a dual Intel Pentium III CPU machine with 512 KB of RAM and each CPU running at 733 MHz.

This configuration of the middle-tier server being the most resource constrained machine was chosen so that the middle-tier server machine could be pushed to its maximum capacity and any bottleneck would only exist on the middle-server machine. The client machine ran Red-Hat Linux 7.2 (Kernel 2.4.7), the middle-tier ran Debian Linux (Kernel 2.2.18) and the sink server ran Debian Linux (Kernel 2.4.16). The client, middle-tier server and sink server were all compiled and linked statically with TAO release 1.2.2. Complete source-code for the benchmark tests are available in the TAO open-source release in `$TAO_ROOT/performance-tests/AMH_Middle_Server`.

Throughput benchmark test description. The client machine spawns a specified number of clients that then flood the middle-tier server with requests. Each client sends the next request when it receives a reply to the current request. The middle-tier server forwards these requests to a sink server. The sink server can be configured to delay sending response to the middle-tier server by a designated number of milliseconds. The sink server sends the response to the middle-tier server after the predefined delay and the middle-server then returns that response to the client.

The source client requests consisted of invoking an operation on the middle-tier server with a timestamp and waiting for a return value. The value returned to the client was the same timestamp that it had originally send to the middle-tier server. The middle-tier server knew how many clients would be connected to it before a test-run began. Thus, when a test started the middle-tier server started recording the throughput only after all clients had connected to it. Throughput measurement starts when all clients connect to the middle-tier server and stops when the first client disconnects. The *throughput* is thereby computed as the total number of replies sent to the client during this time-window, divided by the number of seconds that make up the time-window.

Middle-tier server models. We used the following five concurrency models of middle-tier servers in our experiments:

- Thread-per-connection (TPC)
- Single-Threaded Reactive (TPR-ST)
- Thread Pool with two threads (TPR-2)
- Single-threaded AMH (AMH-ST) and
- Multithreaded AMH with two threads (AMH-2).

We briefly describe the characteristics of each of these configurations below.

• **Thread-Per-Connection (TPC).** In this server, a new server thread is spawned for each new client. Each middle-server thread is exclusive for a client and only handles requests/responses specific to that client connection. After a thread is assigned to a client it cannot be used to process requests for another client. The maximum number of threads that can be spawned is limited only by the number of threads that the OS can spawn.

• **Single-Threaded Reactive (TPR-ST).** In this server, a single thread handles all requests and replies. Since the server is single-threaded, TAO can be configured so that all synchronizers are removed, which increases the throughput of the server. This single thread is dispatched reactively by the ORB so it can handle many different client connections, client requests, and sink server responses.

• **Thread Pool Reactive (TPR-2).** In this server, a predefined number of threads are spawned when the server starts up. In our benchmarks the TPR-2 server spawned two threads. Since the threads are managed by a reactive ORB, they are not exclusive to any client and thus can be reused to handle requests for multiple clients. For example, if a thread is blocked waiting for a response to arrive from a sink server, the same thread can be used again to process a new client request. In the TPC model, this was not possible since a thread waiting for response from the sink server could not be used to service a new client request from a different client.

• **Single-Threaded AMH (AMH-ST).** In this server, the `ResponseHandler` and timestamp for each client request is passed to an AMI servant. The AMI servant stores this value-pair in a hash map and invokes an asynchronous call on the sink server. The AMH servant returns immediately to handle more client requests. When a response from the sink server shows up, the AMI-servant extracts the appropriate `ResponseHandler` from the hash map using the timestamp (since the sink server returns back the same timestamp)

and invokes the method on the `ResponseHandler` with the timestamp. The `ResponseHandler` sends the timestamp value to the client. In brief, the ST-AMH middle-tier server stores `ResponseHandlers` and timestamps in a hash map, extracts the appropriate `ResponseHandler` when a sink server response arrives, and invokes the `ResponseHandler` method. Since this server is also single-threaded, all thread-locks in the ORB are removed at compile time.

• **Multi-Threaded AMH (AMH-2).** In this server, we have two threads controlled by a reactive-ORB that do the same work as the ST-AMH server. Except for the spawning of two threads (and hence synchronizers being present in the ORB), this test is the same as the AMH-ST middle-tier server.

5.2 Client Scalability Test

Overview. In this test, the number of clients are steadily increased. Each client makes 1,000 twoway requests to the middle-tier server. The sink server is configured to delay the response by 40 milliseconds. In an ideal middle-tier server we could expect the throughput to increase with the number of clients until the middle-tier server reaches its maximum capacity. Thereafter, the throughput should remain constant, even with further increase in the number of clients.

Empirical results. Figure 8 shows the result of running the benchmark with different middle-tier servers. The test results for each concurrency model are described below.

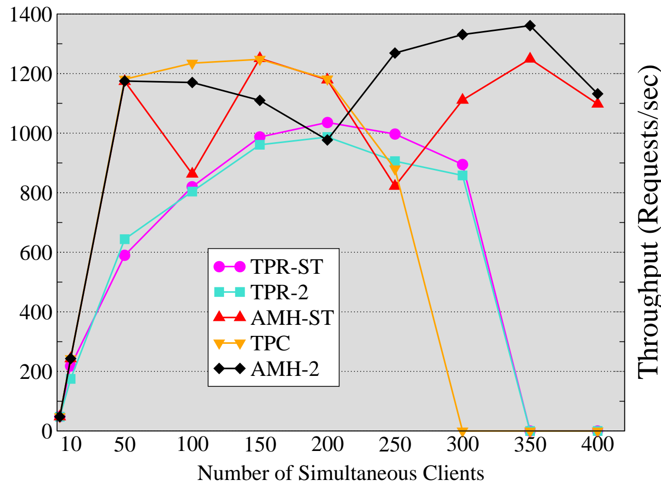

Figure 8: **Throughput of Middle-tier Servers with Increasing Clients**

• **Thread-Per-Connection (TPC).** In our middle-tier configuration, the middle-tier server could only spawn 250 threads before the context switching overhead surpassed any useful work done. The highest throughput is at 150 clients, after which there is a steady decrease in throughput. This degradation seems to stem from the excessive time the server is spending context-switching between the various threads.

• **Thread Pool Reactive (TPR-2.** This server has the worst throughput among all the servers, though it is a bit more scalable than TPC, up to 50 clients more. In TPR-2, two threads can handle many simultaneous client requests. Since a new activation record is created for each new client request, however, the threads cannot handle the responses sent by the sink server in any order. Instead, they can unwind the stack only if the current response corresponds to the top of the activation record; otherwise, the thread cannot process the response. This constraint not only decreases the throughput of the server, but when the stack of the thread grows beyond the 2 MB limit on our machine, the server crashes due to unprocessed replies when more client requests arrive. This behavior occurs when more than 300 simultaneous clients are connected to the middle-tier server.

• **Single-Threaded Reactive (TPR-ST).** The performance of this server closely follows the TRP-2 server. Although there are no synchronizers in this server, the gain in performance by the removal of the locks is offset by having to make a single thread handle all the work, *i.e.*, client request processing, invoking requests on the sink server, sink server-reply processing, and sending replies to clients.

• **Single-Threaded AMH (AMH-ST).** The AMH-ST server scales well, handling up to 400 simultaneous clients. We did not test the server with more than 400 clients since at that point the client-machine and the sink server machine were starting to bias the results because the load on them was quite high. With faster and much more powerful client and sink server machines, AMH-ST server would likely have been able to handle even more clients. The throughput of the AMH-ST server, however, tends to dip sharply at certain points (100 and 250 clients). We are investigating what is triggering this errant behavior.

• **Multi-Threaded AMH (AMH-2).** The AMH-2 server performs better than the AMH-ST server under conditions of heavy load (upwards of 250 clients) but with light or moderate load, the AMH-ST server sometimes performs better (150 and 200 clients).

Results synopsis. AMH middle-servers deliver higher throughput than conventional ORB threading models, such as thread-per-connection and thread pool, under conditions of heavy load. Under conditions of light or medium load, the AMH servers are at par or only slightly below par the conventional CORBA servers. Only at one data-point (100 clients) is one of the conventional servers (thread-per-connection) better than either of the AMH servers. For some cases (more than 300 clients), AMH based servers are the only option since only they can cope with that kind of load.

Figure 9 compares one server each from each camp (AMH-2 from the AMH camp and TPC from the traditional servers

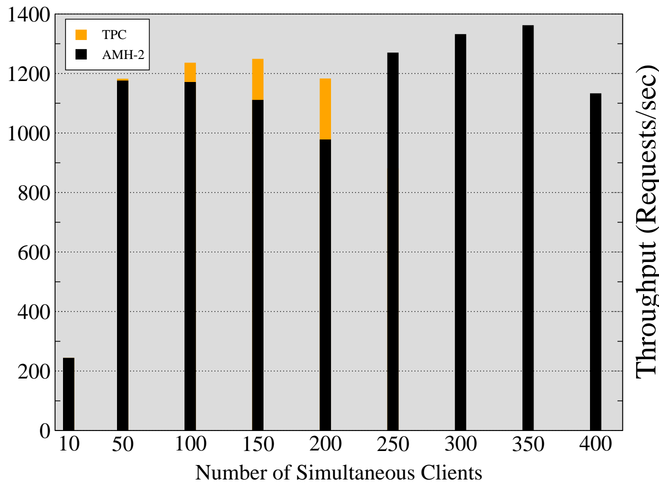

Figure 9: **Comparison Between TPC and AMH wrt Throughput Scalability**

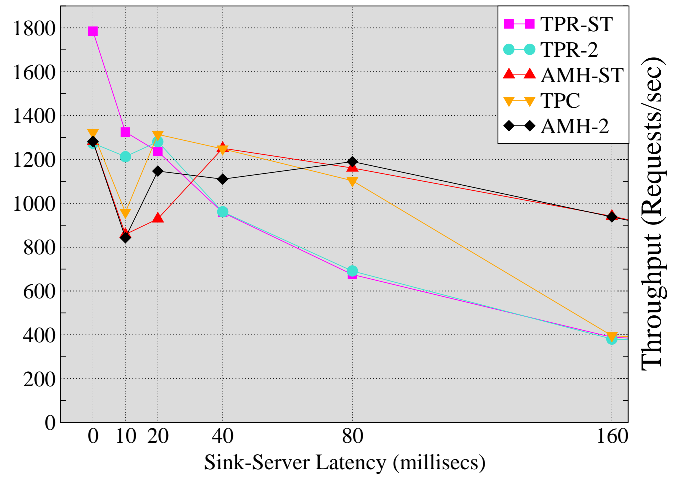

Figure 10: **Throughput of Middle-tier Servers with Increasing Sink Server Latency**

camp). As shown in the figure, AMH-2 lags in performance as compared to TPC in the medium-range of load (from 100 to 200 clients). When the load is heavy, however, the AMH-2 server performs far better. It is also illustrative to see that if it were possible to switch dynamically between middle-server implementations the throughput can be almost constant, across all types of loads, *i.e.*, light, medium or heavy.

5.3 Latency Scalability Test

Overview. In this test, the number of clients is held constant at 150 clients. Each client makes 1,000 requests to the middle-tier server. The sink server delay is then varied by an increasing amount of time.

Empirical results. Figure 10 shows the result of this benchmark. Even though Figure 10 shows a sink server latency to be 0 milliseconds, the delay experienced by the middle-tier server is not 0 due to network latency. In our benchmark, this network latency was ∼0.3 milliseconds. We can safely ignore this latency since for the rest of the data points the sink server latency is large (>10 milliseconds) compared to the network latency. The test results for each concurrency model are described below.

• **Single-Threaded Reactive (TPR-ST).** This server shows the highest throughput when the latency is minimum, delivering up to 500 requests/sec more than any of the other servers at 0 sink server delay. Since this server is single-threaded, there are no locks in the ORB, which helps it to achieve a high throughput. As the sink server latency increases, however, the throughput degrades rapidly in a linear fashion.

• **Thread Pool Reactive (TPR-2).** The throughput of the thread pool reactive middle-server (TPR-2) is almost constant until ∼20 milli-second sink server delay. It also reaches it maximum throughput at this point with 1,281 requests/sec. After this point, there is a rapid and linear decline in throughput.

• **Thread-Per-Connection (TPC).** This middle-server model does much better, showing an almost consistent throughput around 1,000 requests/sec until the delay from the sink server hits 80 milliseconds. At this point, the throughput suddenly drops. When the delay is high enough (in this case 80 milliseconds), all threads in the middle-tier server are blocked waiting for the sink server replies to arrive. When the replies arrive the middle-tier server suddenly becomes busy in a burst, spending considerable time context switching between the many threads that have suddenly become active to handle the pending client requests and the pending sink server responses.

• **AMH servers (AMH-ST and AMH-2).** The AMH servers behave quite similarly with respect to each other. Initially, when the latency is low they lag behind in throughput compared to the other servers (TPC and TPR-2). This behavior may occur since the AMH servers allocate `ResponseHandlers` on the heap, which is costlier than the stack allocation used by the other servers. As the time to allocate on the heap becomes insignificant compared to the blocking time, however, the AMH servers clearly show their ability to scale with respect to sink server delay. At 160 milliseconds delay from the sink server, the AMH servers are very close at achieving the maximum throughput (1,000 requests/sec) at that delay, achieving a throughput of 941 requests/sec and 938 requests/sec for AMH-ST and AMH-2, respectively. The other

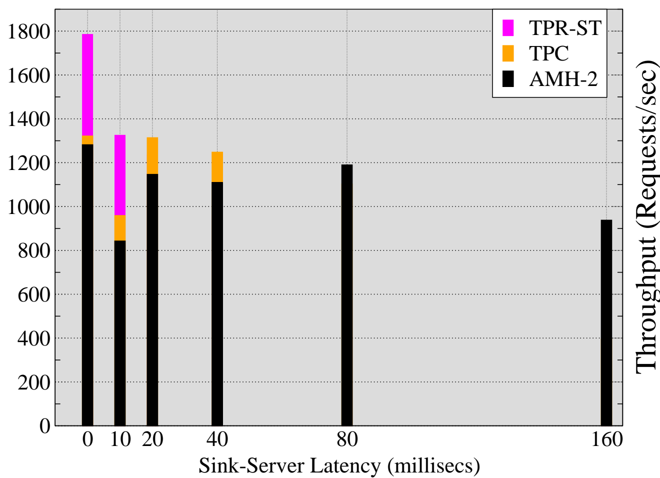

Figure 11: **Sink Server Latency Comparison Between TPC, TPR, and AMH**

servers are half that throughput, achieving 381 requests/sec and 396 requests/sec for TPR-2 and TPC, respectively.

Result synopsis. AMH is a clear winner when the blocking time is quite large. Conversely, if it is known that the blocking time will be low and the load on the server will be constantly high, thread pool or thread-per-connection may be a better choice. Figure 11 compares the TPR-ST, TPC and AMH-2 concurrency models. As seen in Figure 11, when the sink server latency is small (0-10 milli secs), the single-threaded reactive server (TPR-ST) delivers maximum throughput; when the latency is between 20-40 milliseconds, the thread-per-connection server works best; and for latencies higher than 40 milliseconds, the AMH servers are the most scalable.

6 Future Enhancements and Research Directions

This section outlines future enhancements and research directions related to our work on AMH.

Future enhancements. While we have tried to provide maximum flexibility to application developers who want to use AMH, the following situations could arise where developers would like even more fine grained control:

- The current way to generate an AMH servant is to compile the whole IDL interface with the `-GH` flag. There could be interfaces in which some operations would not block and the AMH mechanism may be a overhead in those operations while AMH is needed in other mechanisms. Currently AMH is applicable only at the interface-level and not at the operation-level. Support for operation-level control and specification for AMH is one enhancement we are considering.
- Currently, one `ResponseHandler` is created for each client request. Application developers have no control over this creation policy of `ResponseHandlers`. Future enhancement may include policies to control the creation of `ResponseHandlers`, *e.g.*, from a memory pool, or reassignment to an existing `ResponseHandler`.

Research directions. Many of TAO's existing ORB optimizations [8] are based on the assumption that a single activation record handles a request. Associating a request with an activation record has certain advantages, *e.g.*, it is faster than allocating on the heap and it is easy to analyze the request's lifetime. When the thread exits the activation record, the request is destroyed implicitly. With AMH, however, this basic assumption is violated. For example, the Portable Interceptors specification relies solely on this assumption. We are in the process of determining the scenarios in which AMH can cause problems and how these problems can be avoided.

Allocating requests on the heap raises many dynamic memory management issues, such as heap fragmentation, jitter induced by heap-allocation algorithms, and obtaining and releasing locks during memory management operations. Since AMH is on the critical-path, this overhead could lead to higher jitter, which is undesirable in real-time systems. We are in the process of analyzing and empirically benchmarking if AMH introduces jitter and ways to reduce it.

Many services offered by TAO, such as Load-Balancing and Event-Service, are used in middle-tier servers. Incorporating AMH into these services would improve their efficiency. Future work would include designing and incorporating AMH into these services.

7 Related Work

Distributed systems modeled as entities, such as Actors [9], sending asynchronous messages to each other have been studied theoretically, as well as implemented commercially as Message Oriented Middleware (MOM) (with less stringent semantics [6, 10]). Having an explicit asynchronous model offers more flexibility and dynamicity in the system since individual messages can be 'acted-upon'/transformed or rerouted easily [11]. However, this flexibility often yields lower performance, *e.g.*, These types of systems have been known to perform around two orders of magnitude worse than synchronous systems in which initial connection setup takes time, but after the connection is established communication between the client and the server is fast [12].

CORBA has an explicitly synchronous model based upon the RPC mechanism, with even oneway calls having an explicit void return type [13]. Throughput in a synchronous system, though, may degrade dramatically when system resources, such as threads, are wasted while waiting for a long I/O or external events. This degradation becomes easily apparent when a client is blocked waiting for a response from a server that is taking a long time to send a reply. The AMI specification in CORBA was designed specifically to overcome this problem for clients. The same problem exists for servers, however, as described in Section 1. Different server concurrency models have been proposed and implemented [14, 15, 16] to better utilize system resources for various types of client requests [17]. AMH is specifically aimed at addressing the root of this problem: *explicitly dissociating the request reception from request processing*. An initial inspiration for this approach was the implementation of *continuations* in the MACH kernel, which resulted in significant performance improvements in that OS kernel [5]. Other examples of similar work include Futures [18] and Promises [19], which are language mechanisms that decouple method invocation from method return values passed back to the caller when a method finishes executing.

Other distributed object computing middleware, such as DCOM [20, 21], also support asynchronous invocations on the client and the server side. Java RMI [22, 23] does not provide asynchronous functionality on either the client or the server, resulting in performance degradation when servers take a long time to respond to calls. The Java Messaging Service (JMS) [10] tries to alleviate this problem by providing a mechanism for receiving sending and messages asynchronously.

8 Concluding Remarks

The CORBA asynchronous method invocation (AMI) mechanism can significantly improve the scalability and responsiveness of many types of client applications. A similar mechanism for servers to handle method calls asynchronously has not been available until now. Asynchronous server support is useful in many situations, such as building scalable middle-tier servers, improving the scalability of servers that are constrained to be single-threaded, or allowing servers to perform multiple requests in parallel to backend-servers for a single client request.

This paper defines a specification for an asynchronous method handling mechanism (AMH) and describes how it has been designed and implemented in The ACE ORB (TAO). AMH can be used to mitigate the scalability limitations of conventional multi-threaded middle-tier servers. Empirical benchmarks show how asynchronous servers scale better than other server concurrency models, such as thread-per-connection and thread pool, in terms of the number of concurrent clients and long running server upcalls. The TAO AMH feature is particularly useful for highly loaded middle-tier servers that must contact multiple back-end servers to service a client request, *e.g.*, that of a firewall server.

In general, the benefits of AMH are:

- Middle-tier servers can process new incoming requests without having to wait for responses from sink servers, which improves the scalability of middle-tier servers without needing to spawn a large number of threads.
- AMH allows servers to handle requests in an order other than the order in which they were received, even when using a single-threaded reactive concurrency model.
- Multi-threaded servers are generally harder to write and maintain than single-threaded servers. AMH can be used to develop single-threaded servers that are more scalable than conventional multi-threaded servers.

An open-source version of AMH can be downloaded in the latest TAO release from `deuce.doc.wustl.edu/Download.html`.

References

[1] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[2] W. W. Eckerson, "Three Tier Client/Server Architecture: Achieving Scalability, Performance and Efficiency in Client Server Applications," *Open Information Systems*, vol. 10, January 1995.

[3] O. Othman, C. O'Ryan, and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, vol. 2, Mar. 2001.

[4] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[5] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," in *Proceedings of the* 13th *Symposium on Operating System Principles*, (Pacific Grove, CA), pp. 122–136, ACM, Oct. 1991.

[6] IBM, "MQSeries Family." www-4.ibm.com/software/ts/mqseries/, 1999.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 ed., Dec. 2001.

[8] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *IEEE Concurrency Magazine*, vol. 8, no. 1, 2000.

[9] G. Agha, *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[10] SUN, "Java Messaging Service Specification." http://jcp.org/aboutJava/communityprocess/maintenance/JMS/, 2002.

[11] N. Venkatasubramaniam, M. Deshpande, S. Mohapatra, S. Gutierrez-Nolasco, and J. Wickramasuriya, "Design and Implementation of a Composable Reflective Middleware Framework," in *International Conference on Distributed Computer Systems (ICDCS-21)*, (Phoenix, Arizona), IEEE, April 2001.

[12] K. P. Birman, "Building secure and reliable network applications," in *WWCA*, pp. 15–28, 1997.

[13] D. C. Schmidt and S. Vinoski, "Standard C++ and the OMG C++ Mapping," *C/C++ Users Journal*, Jan. 2001.

[14] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[15] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request," *C++ Report*, vol. 8, February 1996.

[16] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool," *C++ Report*, vol. 8, April 1996.

[17] J. Hu, I. Pyarali, and D. C. Schmidt, "The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks," *Parallel and Distributed Computing Practices Journal, special issue on Distributed Object-Oriented Systems*, vol. 3, Mar. 2000.

[18] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.

[19] 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.

[20] Microsoft Corporation, *Distributed Component Object Model Protocol (DCOM)*, 1.0 ed., Jan. 1998.

[21] D. Box, *Essential COM*. Addison-Wesley, Reading, Massachusetts, 1997.

[22] SUN, "Java Remote Method Invocation (RMI) Specification." java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html, 2002.

[23] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.