The Design and Use of the TAO Asynchronous Method Handling (AMH) Mechanism

---

Table of Contents

---

1. Motivation

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

This document describes the TAO asynchronous method handling (AMH) mechanism, which extends the concepts of AMI from clients to servers. AMH is useful for many types of applications, particularly middle-tier servers in multi-tiered distributed systems. In these types of systems, one or more middle-tier servers are interposed between a *source client* and a *sink server*. As shown in Figure 1, a source client's two-way request may visit multiple middle-tier servers before it reaches its sink server. The response then flows in reverse through these intermediary servers before arriving back at the source client.

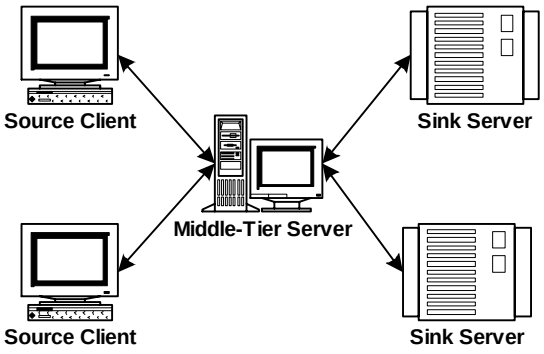

Figure 1: A Three-tier Client/Server Architecture

Without AMH capabilities, the general behaviour of a middle-tier server is shown in Figure 2, where a middle-tier server blocks waiting for a reply to return from a sink-server.

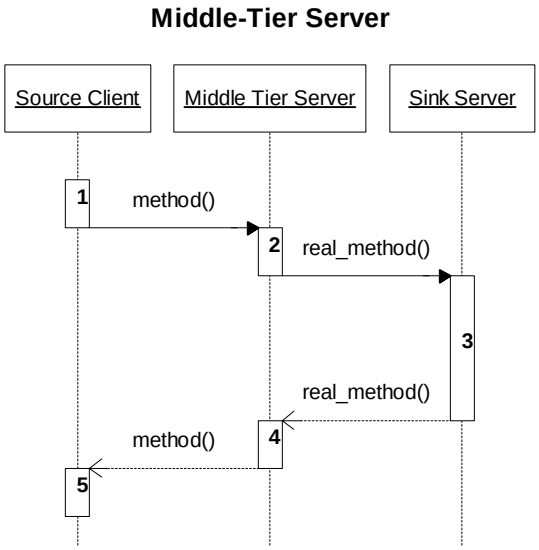

Figure 2: Middle-tier Server Interactions

As shown in the figure, the following steps typically occur in a middle-tier server:

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 and returns data
4. The middle-tier server returns the data to the source client.
5. The client processes the data.

A common way to improve the throughput of a middle-tier server is to multi-thread it using concurrency models, such as thread pool or thread-per-request. However, these concurrency models have the following limitations:

- 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 may degrade the throughput of busy middle-tier servers.
- In a thread-per-request model, each request creates a new thread, which may not scale when a high volume of requests spawns an excessive number of threads.

The TAO AMH mechanism can be used to mitigate the scalability limitations of conventional multi-threaded middle-tier servers. The benefits of AMH are similar to AMI, e.g.:

- 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 programs are generally harder to write and maintain than single-threaded programs. AMH can be used to design and code single-threaded servers that are more scalable than conventional multi-threaded servers.

2. High-level Design of AMH

At the highest level, AMH can be viewed as a continuation model, 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 stores enough information about the client request in a 'safe' place (e.g., on the heap) and returns control back to the ORB immediately. When the server is ready to send back the response, e.g., when a middle-tier server gets back the replies from a sink server, it

can access the stored information to send the response. Since server need not block waiting to send a response it can handle many requests concurrently, even within a single thread of control. The various steps in this process are explained below in the context of our middle-tier server example.

When a new request arrives at a server, the ORB creates a `ResponseHandler` and stores information about the request in it. Table 1 presents the various fields of information stored, explains their use, and shows how they can be accessed. [Mayur, can you please make it clear which of these methods are standard and which are TAO extensions? Please note this in the "Method to Access Feature" column!]

Feature	Description	Method to Access Feature
Transport	Transport on which the request arrived. Used by ORB to send back the response.	`get_transport()`
RequestId	Uniquely identifies the request. Needed in the header of the reply message.	`get_request_id()`
POA Current	The `POACurrent` contains per-request information, such as the `ObjectId` of the target object. This interface can be used to implement stateless servers.	`get_POA_current()`

Table 1. Context Information Contained in a `ResponseHandler`

The `ResponseHandler` object and other 'in' parameters are passed to the servant. The servant can then either:

1. Start processing the request, which will wait for all the results from sink-servers to arrive, or
2. Store the `ResponseHandler` (e.g., in an STL container) and return control to the ORB immediately.

Assuming the middle-tier server stores the `ResponseHandler`, the middle-tier server can send the response back to the client when the reply arrives from the sink server,. It does this by extracting the correct `ResponseHandler` (e.g., from the STL container) and invoking the appropriate method on the `ResponseHandler`. This method is passed the 'out' and return arguments as parameters. When the ORB created the `ResponseHandler` originally, it contained enough information to send the response back to the client. The sequence of steps outlined above is illustrated in the interaction diagram in Figure 3.

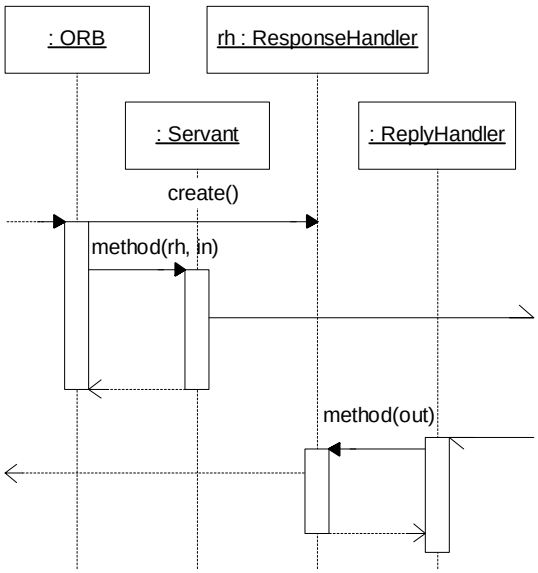

Figure 3: **Asynchronous Method Handling Model**

The AMH model introduces the following new concepts to CORBA server programmers:

1. Server programmers must generate special skeletons using an AMH-enabled IDL compiler and then derive their AMH servants from these AMH skeletons. As described in Section 3, our design ensures that the POA need not be aware of the type of the servant. AMH servants can therefore be registered in any POA, even those that contain normal servants, which provides maximum flexibility to developers.
2. Handling exceptions is slightly different from the standard synchronous case. To send an exception back to the client the AMH servants must call the corresponding exception-method on the `ResponseHandler`. Additional details are provided in Section 3.4

Through the rest of this document we use a simple IDL interface to illustrate the key AMH concepts. The IDL we use represents an interactive stock quoter that returns the latest price of a designated stock given its symbol, e.g., “MSFT”, “RHAT”, etc. The IDL interface is shown below:

```
module Stock
{
      exception Invalid_Stock_Symbol {};

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

Figure 4: The Initial Stock Quoter Interface

3. Detailed Design of TAO's AMH Capability

This section examines the design the TAO ORB uses to support AMH servers. We describe the components that are derived from the `Stock` module in Figure 4 to support AMH semantics. We describe each component using IDL to avoid tying the components to a particular programming language or ORB. Thus, while the IDL for these components do not actually exist in practice, it appears as if they were derived from this IDL. Using the nomenclature of the CORBA AMI, we call such IDL "implied-IDL," which refers to the fact that the TAO IDL compiler "logically" generates additional IDL based on the standard IDL declarations fed into it, and then compiles the original IDL and the implied IDL into C++ stubs and skeletons. The components described below are specific to the `Stock` interface but the descriptions, rules to generate them, and their interactions are applicable to any IDL.

3.1. Implied-IDL

Our implied-IDL mapping for the `Stock` module is shown in Figure 5.

```
module Stock
{
      local interface AMH_QuoterResponseHandler; // Forward decl.

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

      // Exception Holder for raising AMH exceptions
      valuetype AMH_QuoterExceptionHolder :
            Messaging::ExceptionHolder
      {
            void raise_get_quote () raises (Invalid_Stock_Symbol);
      };

      // The AMH_Quoter ResponseHandler. Note that the
      // AMH_QuoterResponseHandler derives from a base
      // "ResponseHnadler" (RH), but that base RH is ORB-specific and
      // thus has no IDL associated with it, but we want to show
      // the 'class hierarchy', hence the commented base RH.
      local interface AMH_QuoterResponseHandler //: "ResponseHandler"
      {
            void get_quote (in long return_value);
            void get_quote_excep
                  (in AMH_QuoterExceptionHolder holder);
      };
};
```

Figure 5: Implied-IDL for the Stock Quoter Interface

The implied-IDL has three new generated interfaces:

- `AMH_QuoterResponseHandler`, which [Mayur, can you briefly outline what this interface does?]
- `AMH_Quoter` which [Mayur, can you briefly outline what this interface does?] and
- `AMH_QuoterExceptionHolder`, which [Mayur, can you briefly outline what this interface does?]

The rules that generate these interfaces and their interactions are described below.

3.2. The AMH Servant

The AMH servant implements the `AMH_Quoter` interface. This interface compares and contrasts with the original `Quoter` interface in the following ways:

- The AMH_Quoter interface contains all the operations specified in the original `Quoter` interface and with the same names
- Each AMH_Quoter operation has a different signature, however, which we refer to as *asynchronous operation*s.

.

Compared to the original IDL interface, the changes to the signature of an asynchronous operation are:

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

3.3 The ResponseHandler

All `ResponseHandlers` (e.g., the `AMH_QuoterResponseHandler`) can be viewed as deriving from a base `ResponseHandler` interface can collaborate with the ORB to send responses back to the client. The `ResponseHandler` interface is unique in that it is not defined explicitly by any IDL interface, though all implied-IDL `ResponseHandlers` derive from it implicitly. The base `ResponseHandler` interface contains certain ORB-specific state, such as 'connection' on which the request arrived and the service context of the request. The base `ResponseHandler` also interacts closely with the ORB to send responses to the client. [Mayur, in the following sentence it's not clear what the "it" is. Can you please clarify this?] Since it is implementation-specific for an ORB, it is not specified via IDL. In TAO, we implement this interface in the concrete class `TAO_ResponseHandler`.

All implied-IDL `Response Handlers` are `local`, i.e., they are always collocated in the address space of the server. Although they appear as regular CORBA objects to the server, they are not accessible outside of its address space. Another special characteristics of the `ResponseHandler` is that it can be invoked just once. After it has been invoked, invoking it again will raise an exception on the server, e.g., `BAD_INV_ORDER`, with an appropriate minor exception code.

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

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

The implied-IDL `ResponseHandler` object is passed as a parameter to every asynchronous operation of an AMH-enabled servant. This `ResponseHandler` object is used by a server to send a response back to the client, as explained in Section 2.

3.4 Exceptions

The `AMH_QuoterExceptionHolder valuetype` is used to return an exception to the client if an exception was raised in the server while handling a request. The implied-IDL `AMH_QuoterExceptionHolder` interface is related to the original `Quoter` interface as follows:

The benefits of using a `valuetype` to hold an exception are:

- The design is similar to the existing AMI implied-IDL C++ mapping
- It works with the standard C++ mapping, as well as with the alternative mapping for C++ compiler dialects that lack native exceptions
- It is illegal to pass exceptions as arguments in IDL.
- It decouples our AMH design from language dependencies, so it is not restricted to C++. This decision should make it easier to get AMH standardized by the OMG.

3.5 Design Tradeoffs

The design of AMH (Section 3.1-3.4) has many alternatives. In this section we discuss some of the important alternatives that we considered and why we chose one approach versus another.

- **AMH_QuoterResponseHandler implied-IDL:** The `ResponseHandler` class can be defined in several ways:
  1. It could either be a new class, i.e., a class that is not part of any existing class hierarchy or
  2. It could be derived from the existing AMI `ReplyHandler` class. The advantage of this design is that its interface is compatible with the AMI class and it involves fewer changes to the IDL compiler. The disadvantage, however, is that the interface may be confusing since the functionality of the `get_quote()` method differs radically in the two classes. In the `ReplyHandler`, it is an `upcall,' whereas in the `ResponseHandler` it is a 'downcall'.

  We therefore decided to adopt the first approach. It is straightforward, however, to have the `RequestHandler` and `ReplyHandler` work together via the Adapter pattern [GoF].

- **Activation Granularity:** There are two ways to activate an AMH object:
  1. Per-POA—In this approach, all AMH enabled servants would be registered in a separate POA that was set up explicitly with a Policy to handle AMH.
  2. Per-object—In this approach, both AMH-enabled and non-AMH servants could exist in the same POA.

  We chose the per-object approach since we wanted to maximize application developers flexibility. Moreover, it is straightforward for developers to designate some POA as an "AMH –POA" and register all AMH servants in it. With this design, the behavior of the servants (asynchronous or not) is transparent to the POA, which avoids changes to existing POA implementations.

4. The TAO AMH Implementation

This section describes how we implemented the AMH design described above in TAO. Since TAO is a C++ ORB, we explain the implementation with respect to a C++ mapping of the IDL and implied-IDL. In the remainder of this section, we describe the C++ implementation of the `ResponseHandler` classes, the AMH skeleton class that supports the AMH_Quoter

implied-IDL, and show how all these classes interact to provide asynchronous support to CORBA servers.

4.1. The AMH_Quoter Skeleton

The TAO IDL compiler generates stub and skeleton classes for each IDL interface (even implied-IDL). Client applications use stub classes to narrow server references and to marshal and demarshal method arguments. The skeleton classes are primarily responsible for the demarshaling method arguments, calling the servant-method with those arguments, and marshaling the return/out parameters. The `AMH_Quoter` skeleton differs from the normal skeleton class in the following ways:

- Since the *asynchronous method* is always a void return type (Refer Section 3.2), the skeleton does not marshal any return or out parameters. Instead, these are now marshaled by the `ResponseHandler`. Figure-6 shows the *asynchronous method* for `AMH_Quoter`.
- The `AMH_Quoter` interface is a server-specific interface that is not visible to clients. Instead, it masquerades as a `Quoter` interface to clients i.e., the client invokes method calls as if the server object were a normal `Quoter` object. On the server, however, an `AMH_Quoter` servant handles the method invocations. This design is achieved as follows:
  - In the `_this()` method of `AMH_Quoter`, we return a `Quoter` object instead of an `AMH_Quoter` object, which means that the `AMH_Quoter` object registers itself as a `Quoter` object on the server. The `_this()` method of the `AMH_Quoter` is shown in Figure-6.
  - The `Quoter` reference can then be narrowed by the client and used to invoke operation calls as usual, while the server handles the requests via the `AMH_Quoter`, which has registered itself as a `Quoter` servant. The onus is now on server to demarshal the arguments that have been marshaled via the original `Quoter` interface.

  We embed the logic for handling this conversion in the `AMH_Quoter` skeleton class. Since the rules described in Section 3.2 for converting between a normal interface and an AMH interface are standard, the IDL compiler can generate this logic.

The `AMH_Quoter` interface is only applicable on the server, thus there is no need to generate stub classes for it and only skeleton-specific classes are generated.

```
class  AMH_Quoter : public virtual PortableServer::ServantBase
{
public:

  Stock::Quoter *_this ();
  virtual void get_quote
     (Stock::AMH_QuoterResponseHandler_ptr response_handler,
      const char *stock_name) raises CORBA::SystemException = 0;
};
```

Figure 6: The AMH_Quoter Skeleton Class

4.2 The ResponseHandler Classes

As explained in Section 2, a `ResponseHandler` is an object containing certain information that a server uses to send a response back to a client. In a normal upcall, this information is available on the activation –record. For AMH, however, this information must be stored explicitly elsewhere. In TAO we store this information on the heap by creating a new `ResponseHandler` object. This object can now be accessed anytime in the lifetime of the server to send the response.

Three `ResponseHandler`-related classes are associated with every AMH-enabled interface. As shown in Figure 7, the IDL compiler generates two of them and the third is part

of the TAO ORB Core implementation. We explain below why three classes are needed to implement one `ResponseHandler` and the relationship between the three classes.

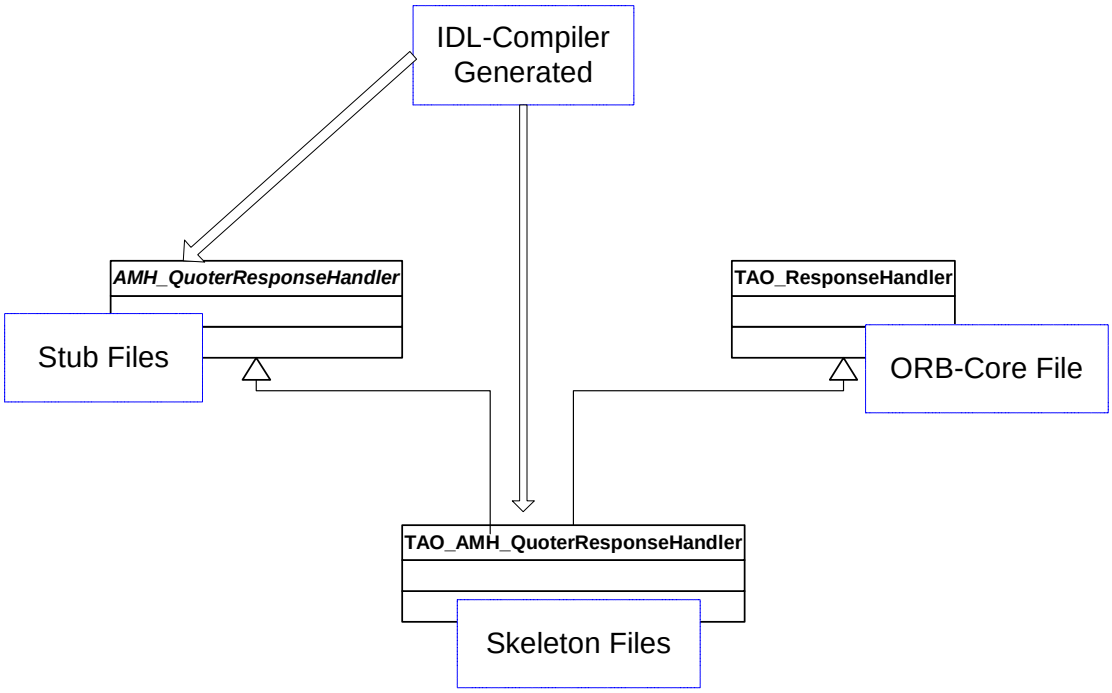

Figure 7: `ResponseHandler` Class Relationships

The `TAO_ResponseHandler` class shown in Figure 8 provides common base functionality to all `ResponseHandlers`, such as initializing and sending the marshaled output stream over the correct connection.

```
class TAO_ResponseHandler

{
    void init_response (void);
    void send_response (void);
}
```

Figure 8: The `TAO_ResponseHandler` Class

The AMH-skeleton expects an `AMH_*ResponseHandler` shown in Figure 6, so this class must be generated into the stub files. The `AMH_*ResponseHandler` class is on used by the server, however, i.e., it references server-specific ORB information. Defining the class in the stub file would therefore link in server-side files, which is unnecessary for pure clients. To avoid this additional overhead, we declare and `AMH_*ResponseHandler` as an abstract class. In the skeleton file, we then extend the base classes as a suitable `TAO_*ResponseHandler` class.

```
namespace Stock
{
  class AMH_QuoterResponseHandler
  {
  public:
    virtual void get_quote (CORBA::Long return_value) = 0;
  };
};
```

Figure 9: The `AMH_QuoterResponseHandler` Class

The `TAO_AMH_*ResponseHandler` class shown in Figure 10 is the implementation class for the abstract `AMH_*ResponseHandler` class and contains code that does the marshaling of 'out' and return parameters. It uses the base `TAO_ResponseHandler` class functionality to send a marshaled response. It therefore derives from both the stub `AMH_*ResponseHandler` class and the base `TAO_ResponseHandler` class.

```
namespace POA_Stock
{
  class TAO_AMH_QuoterResponseHandler :
            public Stock::AMH_QuoterResponseHandler,
            public TAO_ResponseHandler
  {
  public:
    TAO_AMH_QuoterResponseHandler (TAO_ServerRequest &server_request,
                                   void *_tao_object_reference,
                                   void *_tao_servant_upcall);
     void get_quote (CORBA::Long return_value);
     virtual ~TAO_AMH_QuoterResponseHandler (void);
  };
};
```

Figure 10: The TAO_AMH_QuoterResponseHandler Class

The AMH-skeleton expects an `AMH_*ResponseHandler` shown in Figure 6, but is instead passed a `TAO_AMH_*ResponseHandler` object. It doesn't know or care as long as there is an implementation that handles the `ResponseHandler` functionality.

This design not only allows pure-clients to not link in server code, but also provides a clean separation between the interface and the implementation of the `Response Handler` functionality. Moreover, by shifting all common code between the `TAO_AMH_*Response Handlers` into a base `TAO_ResponseHandler` class, we reduce the amount of code that the IDL compiler generates, which reduces the overall code-bloat, as well.

5. Example AMH Server Applications

This section presents two examples that show how applications can exploit AMH to improve their throughput and concurrency without requiring an excessive number of threads. AMH servers must derive AMH servants from AMH skeletons and implement the required functionality in the servants. AMH servers will then register these servants with the POA as normal servants and export their object references. Since client applications require no changes, the asynchrony of servers is transparent to them.

5.1 An AMH Sink Server

In this example, the server's main thread enqueues client quote requests in a global queue. Since the server's servants are AMH-enabled, they simply queue the requests without processing them. A separate worker thread is responsible for dequeueing the requests, processing them, and sending the responses to the clients as shown in Figure-11.

The queue used by the server can be defined as follows:

```
typedef Std::pair <Stock::QuoterResponseHandler_ptr, char *>
        Value_Pair;

Typedef ACE_Message_Queue_Ex <Value_Pair, ACE_MT_SYNCH>
        REQUEST_QUEUE;
```

The servant implementation would use this queue as follows:

```
class Stock_AMH_Quoter_i : public virtual POA_Stock::AMH_Quoter
{
public:
  Stock_AMH_Quoter_i (REQUEST_QUEUE *queue): queue_ (queue) {}

  void Stock_AMH_Quoter_i::get_quote
     (Stock::AMH_QuoterResponseHandler_ptr rh,
      const char *stock_name)
  {
    // Store the ResponseHandler and stock_name value-pair
    // in a global queue.
    Value_Pair vp (rh, stock_name);
    queue_->enqueue_tail (vp);
    // Return control back to ORB.
  }

private:
  REQUEST_QUEUE *queue_; // Pointer to the request queue.
};
```

The main server can now be defined as follows:

```
int main (int argc, char *argv[])
{
    // Initialise the ORB
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);


    // get access to the Root POA
    CORBA::Object_var root_poa =
      orb->resolve_initial_references ("RootPOA");

    // initialise it
    PortableServer::POA_var poa =
      PortableServer::POA::_narrow (root_poa.in ());

    // access it's poa's manager
    PortableServer::POAManager_var poa_manager =
      poa->the_POAManager ();

    // and finally activate the poa via it's manager
    poa_manager->activate ();

    REQUEST_QUEUE queue;

    // Create the AMH Servant
    Stock_AMH_Quoter_i *quoter_servant =
      new Stock_AMH_Quoter_i (&queue);

    // Register the AMH servant with the rootPOA
    Stock::Quoter_var Quoter = quoter_servant->_this ();

    ACE_Thread_Manager::instance ()->spawn (process_queue, &queue);

    // Start accepting client requests.
    orb->run ();

    return 0;
}
```

The worker thread defines the following method to handle the client requests:

```
void *process_queue (void *arg)
{
  REQUEST_QUEUE *queue = static_cast<REQUEST_QUEUE *> (arg);

  for (Value_Pair vp;;)
  {
    queue->dequeue_head (vp);
    Stock::AMH_QuoterResponseHandler_var handler = vp.first;

    CORBA::Long value = get_stock_value (vp.second);

    // send back response to client
    handler.get_quote (value);
  }
  return 0;
}
```

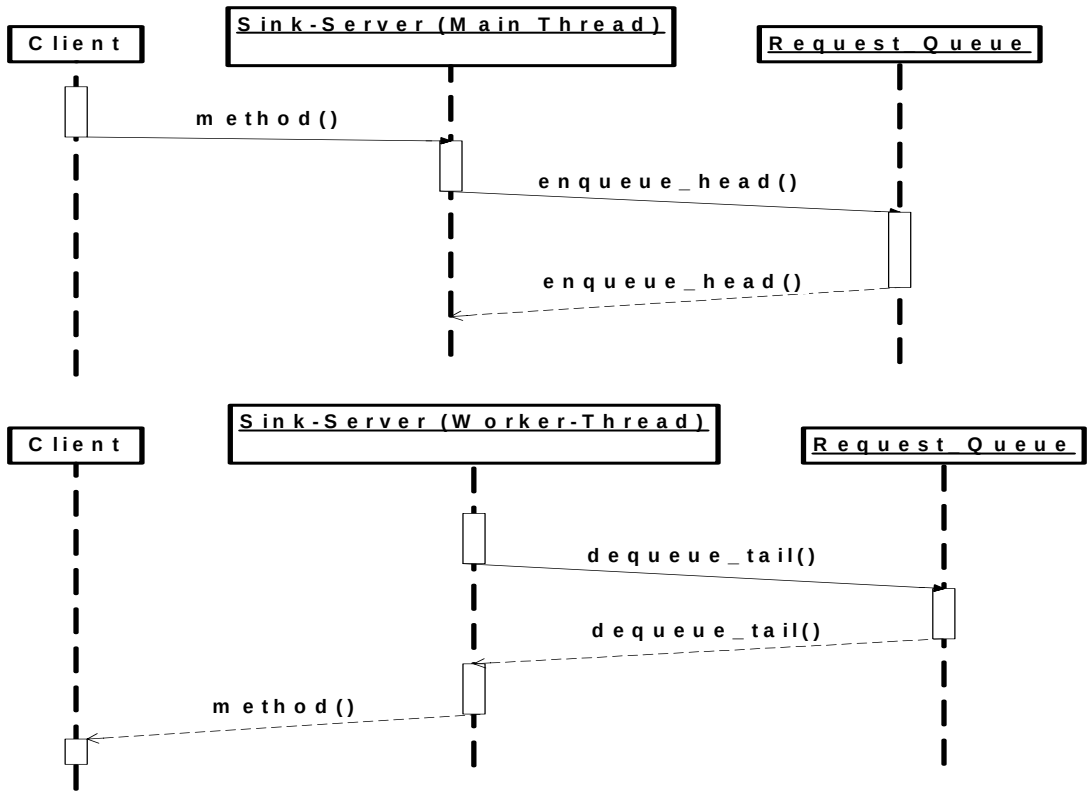

Figure 11: Steps of an asynchronous sink-server.

5.2 An AMH Middle-tier Server

In this example, the middle-tier server shown in Figure 1 uses AMH to handle a large volume of client requests efficiently. [Mayur, the following sentence doesn't make sense as written. Can you please fix this?!] A possible application of use for the middle-tier server is a gateway server sitting on a firewall that verifies and forwards stock-quote requests to an end-server. The end-server processes the request and sends the latest stock price to the middle-tier server.

The middle-tier server then sends the reply back to the client. This example is more sophisticated than the one in Section 5.1 since it uses both AMI and AMH to maximize potential concurrency.

The middle-tier server operates as follows. It obtains the reference to the sink-server when it starts up and instantiates the servant with it:

```
int main( int argc, char* argv[] )
{
    // Initialise the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "Mid");

    // Destringify argv[1]
    CORBA::Object_var obj = orb->string_to_object (argv[1]);

    // Narrow
    Stock::Quoter_var sink_server = Stock::Quoter::_narrow (obj.in
());

    // get access to the Root POA
    CORBA::Object_var root_poa =
          orb->resolve_initial_references( "RootPOA" );

    // initialise it
    PortableServer::POA_var poa =
          PortableServer::POA::_narrow( root_poa.in() );

    // access it's poa's manager
    PortableServer::POAManager_var poa_manager =
          poa->the_POAManager ();

    // and finally activate the poa via it's manager
    poa_manager->activate ();

    // create an AMH enabled Servant
    Stock_AMH_Quoter_i *quoter_servant =
            new Stock_AMH_Quoter_i (sink_server);

    // and register it with the POA
    Stock::Quoter_var quoter = quoter_servant->_this();
    //    Stock::Quoter_var quoter = Stock::Quoter::_narrow( obj.in()
);

    // export middle-tier server stringified reference for the client
    CORBA::String_var str = orb->object_to_string( quoter.in() );
    cout<<str<<endl;

    // start accepting client requests
    orb->run();
  }
  return 1;
}
```

The servant stores the reference to the sink-server (`target_quoter`) and uses it in the `get_quote()` method to send the request to the sink-server. The servant implementation code for `get_quote()` method looks like this:

```
void Stock_AMH_Quoter_i::get_quote
   (Stock::AMH_QuoterResponseHandler_ptr rh,
    const char *stock_name)
```

```
{
  // We want to send AMI request.
  // 1. Create the AMI callback object.
  Quoter_Callback *callback = new Quoter_Callback (rh);

  // 2. Activate the callback with the default POA.
  AMI_Quoter_var callback = callbback->_this ();

  // 3. Make the AMI request.
  target_quoter_->sendc_get_quote (callback, stock_name);
}
```

This method returns almost immediately since **sendc_get_quote** is asynchronous. The middle-tier server is now ready to accept another client request.

The AMI Quoter_Callback class is implemented as follows:

```
class Quoter_Callback : public POA_AMI_QuoterReplyHandler
{
public:
 // Save AMH response handler to send the response later.
 Quoter_Callback (Stock::AMH_QuoterResponseHandler_ptr rh)
    : rh_ (Stock::AMH_QuoterResponseHandler::_duplicate (rh)) {}

 // Callback operation, invoked by ORB to send response to client
 // when sink server reply returns.
 void get_quote (CORBA::Long retval)
 {
   rh_->get_quote (retval);
 }

private:
  Stock::AMH_QuoterResponseHandler_var rh_;
};
```

In summary, the AMH-servant stores the ResponseHandler into the AMI callback object so that when the reply for the end-server arrives, the AMI callback object can send the reply to the client, as shown in Figure 3. This example illustrates how AMI and AMH can work together to maximize concurrency. [Mayur, that isn't exactly true if you change the type of **Stock_AMH_Quoter_i**. Can you please make this point clear?!]

6. Implementation Considerations

While adding AMH capabilites to TAO, we addressed issues regarding its current implementation and the assumptions on which TAO was designed, built, and optimized. This section explores some of those assumptions, their validity with the introduction of AMH, and the potential impact of AMH on the current implementation. [Mayur, in the discussion below you kept saying "object" where I think you mean "request." Please check this carefully!]

- Many of TAO's optimizations 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 requests's lifetime since when the thread exits the activation record, the request is destroyed implicitly. With AMH, however, different threads can access information created when a request first arrives (Table 1 describes this information). AMH therefore stores this information on the heap. The implications of this design decision are:
    - The lifetime of (C++) requests that was originally bound by the lifetime of the activation record is now possibly unbound. All interactions between the POA,

skeleton, and servant must therefore be analyzed carefully. Reference counting of the requests could be a potential solution.

- 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 a potential drop in performance and higher jitter. Moreover, memory leaks can be disastrous for long-running servers. Thus, AMH code in the ORB Core must not only be free of memory leaks, it must also handle insufficient memory errors gracefully, e.g., informing the server and client of the problem if possible.

Some CORBA specifications implicitly assume a single activation record. An example is the Portable Interceptors specification. We are in the process of determining how TAO's Portable Interceptors implementation may break, in which scenario (in request path or reply path), and how these problems can be avoided.

- TAO supports many concurrency models, such as reactive, thread-per-connection, and several thread pools variants. We are analyzing and testing to see how AMH works with all these concurrency models. Due to the way TAO is built with patterns such as Factory and Strategy, AMH should work seamlessly with many concurrency models.
- Currently, AMH is possible only on a per-object basis. It is foreseeable that in some interfaces, certain operations may be long-running and hence need to be handled asynchronously, whereas other operations are short enough to be handled synchronously. Thus, we need to devise intelligent implementations that do not unnecessarily penalize or complicate synchronous operations in an AMH servant.