# Object Interconnections

Time-Independent Invocation and Interoperable Routing
(Column 17)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
60 Aberdeen Ave., Cambridge, MA 02138

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

## 1 Introduction

This column focuses on a new feature defined in the CORBA Messaging Specification [1] called *time-independent invocation* (TII), which adds store-and-forward features to CORBA. Prior to the Messaging spec, CORBA requests were sent by a client and handled immediately by the server, with the server returning any response as soon as it finished processing the request, as shown in Figure 1. As shown in

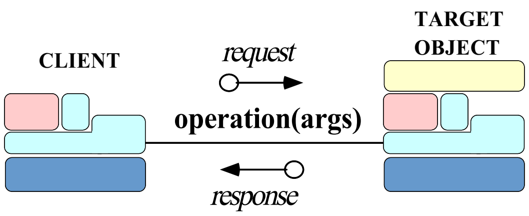


Figure 1: Synchronous CORBA Twoway Request/Response

Figure 2, in contrast, CORBA TII allows applications to **(1)** issue requests and then shut down or disconnect from the network. These requests can be **(2)** queued by any intermediate *client routers* along the way until they reach the *target router*. This router makes a synchronous call to the server ORB **(3)**, which then dispatches **(4)** the upcall on the target object and sends the response back **(5)** to the appropriate reply destination **(6)**. Later, the same application can reconnect or restart to receive the response. Likewise, an entirely different application can receive the response **(7)**.

This is the third column of our series covering the new CORBA Messaging Specification. The first column of the series outlined the capabilities of the new messaging features and described how they greatly improve the status quo for invoking nonsynchronous CORBA requests [2]. The second column showed examples of how to write applications that use CORBA *asynchronous method invocation* (AMI) features. AMI uses the static invocation interface (SII) to send requests asynchronously and receive responses either by polling via `Pollers` or through `ReplyHandler` callbacks [3], which we review briefly below. In this column,

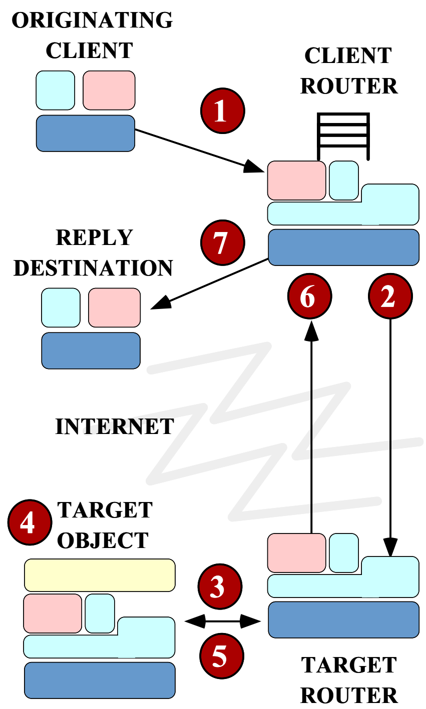


Figure 2: Store-and-Forward CORBA Twoway Request/Response via TII

we illustrate how TII essentially extends AMI with *persistent* `ReplyHandlers`, `Pollers`, and `Requests`.

### 1.1 AMI Review

CORBA Messaging supports the following two models of asynchronous method invocation:

**Polling model:** In this model, each asynchronous two-way invocation returns a `Poller` *valuetype*. The client can use the `Poller` methods to check the status of the request and to obtain the value of the reply from the server. If the server hasn't returned the reply yet, the client can elect to block awaiting its arrival. Alternatively, the client can return to the calling thread immediately and check on the `Poller` later when convenient. Figure 3 illustrates the polling model for asynchronous CORBA twoway operations.

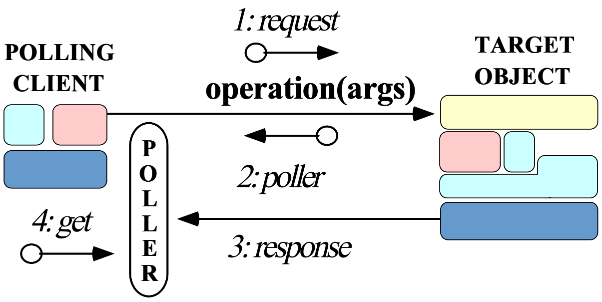


Figure 3: Polling Model for CORBA Asynchronous Twoway Operations

**Callback model:** In this model, the client passes an object reference for a `ReplyHandler` object as a parameter when it invokes a two-way asynchronous operation on a server. When the server responds, the client ORB turns the response into a request on the client's `ReplyHandler`. Figure 4 illustrates the callback model for asynchronous CORBA twoway operations.

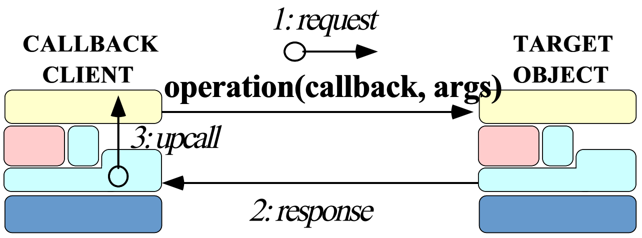


Figure 4: Callback Model for CORBA Asynchronous Twoway Operations

In general, the callback model is more efficient than the polling model because the client need not poll for results. However, it forces clients to behave as servers, which increases the complexity of certain applications, particularly "pure" clients.

### 1.2 Motivation for Time-Independent Invocation (TII)

To motivate the new CORBA TII feature, imagine that you're sitting on a plane late at night returning home from a business trip. You're tired, but you've been waiting all day to look into a hot stock tip you received earlier from a friend. You start up your CORBA-based stock application on your laptop, and start reviewing information that you downloaded over the phone just before boarding the plane.

After investigating the tip, you decide your friend was right, and you issue a buy order. Your stock application informs you that your buy request has been queued and that it will be issued the next time you connect your laptop to a network. Satisfied, you put away your laptop and drift off to sleep.

Because CORBA has traditionally been associated only with synchronous RPC-style requests, you might think this scenario is far-fetched or even impossible. It isn't. TII allows you to develop applications that rely on store-and-forward messaging, just like the stock buying example we describe above.

## 2 Overview of TII Basics

TII is essentially a specialization of CORBA AMI. The difference between the two lies in the length of request and reply lifetimes. With regular AMI, the client request lives only until it reaches the target object. Likewise, the reply lives until the client application obtains it, either by polling or through a callback. With TII, however, requests and replies can be stored persistently in *routers* until their targets become available.

In general, TII is useful for applications that require guaranteed delivery of requests to target objects that may not be connected to a network at the time a message is sent. Email is a good analogy to understand the CORBA TII model. When you send an email message, the email program on your laptop might queue it until you plug into the network. At that point, your laptop mailer sends the message to a local Simple Mail Transfer Protocol (SMTP) server. In turn, this server routes your message to the next SMTP server, which stores and forwards it to other SMTP servers until your message reaches its intended recipient.

Just as an SMTP server accepts email messages and stores them persistently until it can forward them to the next hop, a TII router accepts CORBA requests and replies and stores them persistently until it can forward them to the next router. Because CORBA routers can store TII requests and replies persistently, they can outlive the clients and servers that send and receive them. Therefore, TII is useful for applications running on "occasionally-connected" clients, such as the laptop computer described in our stock buying application example above. By using TII, requests and replies can be delivered to their targets when network connections, routing agents, and QoS properties permit.

## 3 Illustrating TII Features with the Stock Buying Application

To illustrate TII features in more detail, let's examine the workings, step-by-step, of the stock buying application example described in Section 1.2. Below, we'll explain how (1) the request was initially created, (2) the first router was identified, (3) the request was sent, (4) the request is invoked by the target router, and (5) the reply is handled.

**1. Initial request:** When you click the button on your stock application's GUI to commit your buy order, your application's ORB first attempts to issue your request directly to the target using normal AMI semantics. If your laptop were hooked into a network and connectivity to the target stock broker object could be established, the ORB would send the request, receive the reply, and the twoway operation would be complete. In our use-case, however, the ORB's attempt to deliver the request fails because your laptop is not plugged into a network.

It's important to note that most of the actual work involved in sending the request via the TII is done by the ORBs and intermediate CORBA routers. For instance, the fact that the ORB fails to deliver the request might not be revealed to the application, *e.g.*, the ORB might catch the failure and decide to send the request to a router instead. Thus, all you need to worry about is writing the client and server code, as usual.

**2. Router identification:** Because the attempt to deliver the AMI request failed, the client ORB looks for a router to handle the request instead. Exactly how the ORB finds a router is implementation-specific. For example, the ORB might look in the target object's Interoperable Object Reference (IOR) for routing information, or it might just use its own local CORBA router. A CORBA router implements standard CORBA Messaging IDL interfaces. Therefore, it's identified by a normal CORBA object reference.

**3. Sending the request:** Once the client ORB identifies the initial router, it must send the request to it. How it sends the request depends on whether the client application uses the callback or polling AMI model, both of which we examine below.

- **Callbacks:** If the client application uses a callback `ReplyHandler`, the client ORB uses the `Router` interface of the `MessageRouting module`, shown below:

```
module MessageRouting
{
  struct RequestInfo { /* described below */ };
  typedef sequence<RequestInfo> RequestInfoSeq;

  interface Router
  {
    // Send one request to a server.
    void send_request (in RequestInfo req);

    // Send multiple requests to a server.
    void send_multiple_requests
           (in RequestInfoSeq req_seq);
  };
};
```

The `RequestInfo struct` contains enough information for a router to either locate the request target object or to find the next router it should pass the request to. Therefore, the client ORB can supply a list of routers, the object can supply a list of routers in its IOR, or a router can use its own list.

```
module MessageRouter
{
  interface Router; // Defined above
  typedef sequence<Router> RouterList;

  struct RequestInfo
  {
    // List of routers visited thus far.
    RouterList visited;

    // List of routers remaining to visit.
    RouterList to_visit;

    // Object we're invoking our request upon.
    Object target;

    // Index of IOR profile being used to send
    // this request.
    unsigned short profile_index;

    // Location where the reply should return to.
    ReplyDestination reply_destination;

    // Quality of service parameters.
    Messaging::PolicyValueSeq selected_qos;

    // Contents of the message.
    RequestMessage payload;
  };
```

A `RequestInfo` contains the `payload` field, which is the contents of the `RequestMessage` itself. In addition, it contains the target object reference, along with the series of routers that have been visited and that remain to be visited in the `visited` and `to_visit` fields respectively. The `selected_qos` field contains quality of service (QoS) parameters that can be used by routers to decide how to store and forward this request relative to other requests.

The `reply_destination` field stores the destination of the reply, which is of type `ReplyDestination`:

```
module MessageRouting
{
  enum ReplyDisposition {
    TYPED, UNTYPED
  };

  struct ReplyDestination {
    ReplyDisposition handler_type;
    Messaging::ReplyHandler handler;
    sequence<string> typed_excep_holder_repids;
  };
};
```

To use an application-specific `ReplyHandler`, the *client ORB* – not the client application, but the ORB on which it runs – sets the `handler_type` field of the `ReplyDestination` to the `TYPED` enum value. In this case, the `handler` field contains the object reference for the client application's `ReplyHandler` that will receive the AMI reply callback. Note that `UNTYPED` handlers are used between routers because a router can't possibly implement all typed operations, so it must use untyped handlers.

To implement TII using the callback model, the client ORB invokes the `send_request` method, passing in a `RequestInfo` structure.

- **Polling:** If the application uses a `Poller` instead of the `ReplyHandler`, the client ORB uses the `PersistentRequestRouter` interface of the router to invoke the `create_persistent_request` operation:

```
module MessageRouting
{
  // ... RequestInfo and Router
  // declarations from above.

  // Forward declaration.
```

```
  interface PersistentRequest;

  interface PersistentRequestRouter
  {
    PersistentRequest
    create_persistent_request
      (in unsigned short profile_index,
       in RouterList to_visit,
       in Object target,
       in CORBA::PolicyList current_qos,
       in RequestMessage payload);
    };
};
```

Note how most fields included in the RequestInfo are included
as parameters in create_persistent_request. The PersistentRequest object reference returned by the router provides operations and attributes that allow the client ORB to determine whether a reply is available for the request:

```
module MessageRouting
{
  // ...

  interface PersistentRequest
  {
    readonly attribute boolean
      reply_available;

    GIOP::ReplyStatus get_reply
      (in boolean blocking,
       in unsigned long timeout,
       out MessageBody reply_body)
        raises (ReplyNotAvailable);

    attribute Messaging::ReplyHandler
      associated_handler;
  };
};
```

The PersistentRequest object reference that the router returns from create_persistent_request is implemented by the router itself. The client ORB uses this object reference to (1) check for reply availability, via the reply_available attribute and (2) obtain the reply itself, via the get_reply operation.

The MessageBody is the marshaled contents of the request itself. The associated_handler attribute is nil for a polling client. However, it can be used if a client switches from the polling model to the callback model by associating a ReplyHandler with its Poller. A client can make this switch by setting the associated_handler attribute of its Poller valuetype.

If the client uses the polling model, the router establishes itself as the reply target by implementing the UntypedReplyHandler interface:

```
module MessageRouting
{
  interface UntypedReplyHandler
    : Messaging::ReplyHandler {
    void reply
      (in string operation_name,
       in GIOP::ReplyStatus reply_type,
       in MessageBody reply_body);
    };
};
```

To implement TII using the polling model, the client ORB obtains a PersistentRequest object reference via create_persistent_request and then uses it to poll for the reply.

**4. Invoking the target router:** The request passes through one or more routers until it reaches a router that can invoke the operation on the target object. This router is called the *target router*. The target router acts as a "remote client proxy" that dispatches the request on behalf of the client application and subsequently receives the reply.

Recall from our previous column that AMI, and thus TII, does not affect the target object implementation. This means that the target router invokes the request on the target object *synchronously*. Thus, the target carries out the request and returns a reply just as it does for any normal synchronous request.

**5. Reply handling:** When the target object replies, the reply returns to the target router. The target router turns the reply into a request, invoking either another router's send_request operation, or targeting the original reply destination. The handler_type disposition of the reply destination indicates whether the reply handler is typed or untyped. In turn, this indicates whether the original application used the callback or polling reply model, allowing the reply to be delivered properly.

The reason a router would invoke another router rather than invoking the target directly is that it might not have direct connectivity to the target. For example, the router in the client enterprise might route the request through a firewall, onto the Internet, and onward to the router in the server enterprise. The server-side router, which lives in the same network as the target object, then routes the request to the target.

# 4 Concluding Remarks

This concludes the penultimate column in our series on the CORBA Messaging specification. In this column, we motivated the need for the new time-independent invocation (TII) feature in CORBA. The TII is a specialization of AMI that supports "store-and-forward" semantics. Time-independent requests may actually outlive the requesting client process, meaning that the response may be processed by a completely different application than the originating client.

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

# References

[1] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05yes ed., May 1998.

[2] D. C. Schmidt and S. Vinoski, “Introduction to CORBA Messaging,” *C++ Report*, vol. 10, November/December 1998.

[3] D. C. Schmidt and S. Vinoski, “Programming Asynchronous Method Invocations with CORBA Messaging,” *C++ Report*, vol. 11, February 1999.