Object Interconnections

Distributed Callbacks and Decoupled Communication in CORBA (Column 8)

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

Steve Vinoski
vinoski@ch.hp.com
Hewlett-Packard Company
Chelmsford, MA 01824

This column will appear in the October 1996 issue of the SIGS C++ Report magazine.

1 Introduction

We're changing gears in this column. Our recent columns have used a distributed stock quoting example to focus on different concurrency models for developing multithreaded server applications. In this column, we'll start looking at another aspect of distributed object computing systems: decoupling the relationship between "clients" and "servers."

Our examples to date have focused exclusively on request/response communication. In this approach, requests flow from client to server and responses flow back from server to client. In this column, we'll discuss *distributed callbacks* and extend our stock quoting example to show why they're useful. We'll also briefly discuss the OMG Events object service [1], which supports decoupled *peer-to-peer* relationships between consumers and suppliers. We intend to cover the OMG Events object service in more detail in future columns.

2 Revisiting the Distributed Stock Quoting Application

We're going to start off by revisiting the distributed stock quoting application to identify the limitations with our original client/server design. In particular, we'll see that our original design has serious problems that show up as our application requirements evolve. But first, we need to define our terminology more precisely.

2.1 "Client-Server" vs. "Peer-to-Peer"

The terms "client" and "server" are widely used in distributed computing circles, but what do they really mean? In distributed object computing systems these terms are useful only to describe the sender and receiver of a *single request*. That is, for a given request, the "client" is the entity making the request, while the "server" is the entity acting upon it. Moreover, the client of one request may well be the server for another request.

A more accurate way to view distributed object collaboration is to think of terms like client and server as *roles* that are played by various objects at various times. In practice, distributed objects change their client and server roles quite frequently. For instance, objects often participate in *peer-to-peer* relationships, rather than in strict client-server relationships.

2.2 The Drawbacks of Request/Response Systems

In our May 1995 column, we showed the following OMG IDL definition for a `Stock::Quoter` interface:

```
module Stock {
  // Requested stock does not exist.
  exception Invalid_Stock {};

  interface Quoter {
    // Returns the current stock value or
    // throw an Invalid_Stock exception.
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };
};
```

This interface provides a `get_quote` operation that allows a broker to query (*i.e., poll*) the stock quoter object. Since stock trading strategies are often triggered when a stock reaches a certain value, this design allows a broker to monitor a stock and to buy or sell it when a desired value is reached. Figure 1 illustrates the architecture of a distributed polling quoter.

The client application code for a polling broker might look as follows:

```
using namespace Stock;

// ...initialize ORB...

Quoter_var quoter =
// Use Naming or Trader service to
// get quoter object reference...

const CORBA::Long desired = TRADING_THRESHOLD;
CORBA::Long actual;

do {
  actual = quoter->get_quote ("ACME ORB Inc.");
} while (actual != desired);
```

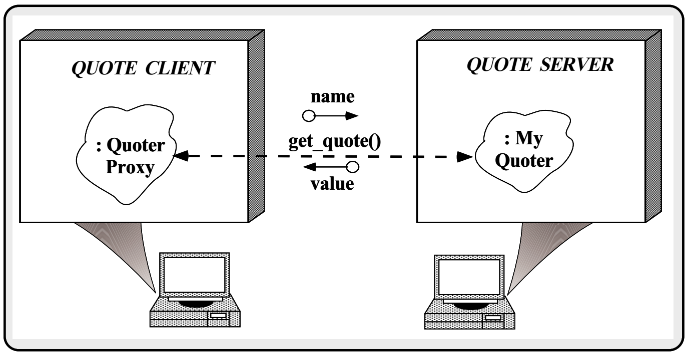

Figure 1: Architecture of a Distributed Polling Quoter

```
// Exercise buy or sell order.
```

While the polling approach could conceivably provide the desired results, it suffers from a number of drawbacks, including:

• **Server saturation:** This quoter application is capable of sending many requests to the server in a very short period of time. Depending on implementation issues (such as the hardware platform, operating system, or concurrency model), the server may become saturated and thus unable to fulfill new requests. For instance, an excessive amount of memory and CPU resources may be consumed if the server uses a thread-per-request concurrency model. This problem is worse for servers that take a long time to service requests. In this case, several polling clients can easily cause these servers to consume all their available resources while trying to keep pace with incoming requests.

• **Network saturation:** Even if servers are very efficient at servicing requests, a polling approach can still allow applications to use excessive network bandwidth. Polling applications that flood the network with request messages and their responses may cause all services on the network to suffer due to increased response times and error rates. Moreover, if congestion becomes too severe, the entire network may become incapacitated.

• **Limited application utility:** A polling application that saturates its server and its network isn't good for much else. If it weren't polling, it might be able to perform other useful work (*e.g.,* refreshing the user-interface, providing adequate quality of service to other users or applications, etc). Performing the polling in a separate thread can help alleviate this problem, but not without increasing application resource consumption and programming complexity.

2.3 The Distributed Callback Solution

One way to avoid the problems associated with polling is to employ *distributed callbacks*. In terms of our example

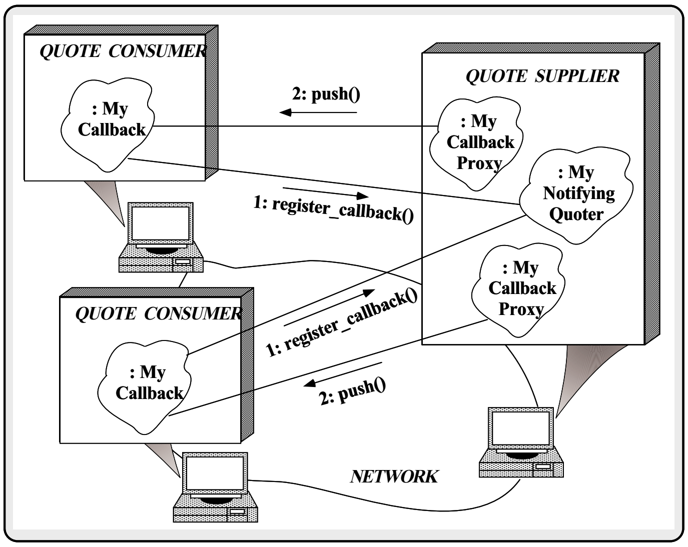

Figure 2: Architecture of a Distributed Callback Quoter

described above, a distributed callback involves a "role reversal," *i.e.,* the "server" calls back to the quoter "client" application. The Quoter object uses this callback to notify the quoter application that the stock it's interested in has reached the desired value. Figure 2 illustrates the participants and collaboration in the distributed callback solution.

2.3.1 Common Examples of Callbacks

CORBA distributed callbacks are similar to ordinary callbacks used in C++ programming. An example of a C++ callback is the function pointer passed to `set_new_handler`. If a new handler callback has been installed, any time `operator new` is unable to allocate memory it will invoke the callback function hoping that the application or runtime environment can somehow free up some memory. This approach is obviously simpler than having the application continually poll the status of the heap.

Another well-known example of a non-distributed callback is a function pointer registered to handle graphical user interface (GUI) events, such as the click of a mouse button in a window. When the button is clicked, the registered function is called to allow the application to react to the event. By using callbacks, the user interface framework can be decoupled from application-specific behavior performed in response to events, thereby increasing reuse and extensibility.

Distributed callbacks should not be confused with the "upcalls" made by the Object Adapter and the IDL skeletons, which dispatch requests to object implementations. These upcalls are mechanically similar to GUI callbacks, and in fact are typically implemented by ORB vendors as ordinary C++ callbacks. The major difference between distributed callbacks and object implementation upcalls is that they occur at different levels in the CORBA system. In fact, object implementation upcalls are a key component of a distributed callback or any other distributed request handling mechanism.

Note that the term "distributed callback" is a bit misleading. This is because an object can be located in the same address space as its caller. In this case, a quality ORB implementation will bypass remote messaging mechanisms and perform local function calls between a caller and a target object whenever possible. Thus, a callback in such a system is not always necessarily "distributed." In the current example, however, we'll assume that all callbacks occur between distributed objects.

2.3.2 Problems Solved by Distributed Callbacks

The following points explain how using distributed callbacks in CORBA can help address the problems with our original polling solution described above.

• **Saturation problems:** Since the quoter application isn't flooding the server or the network with requests any more, the saturation problems no longer occur. However, depending upon how the detection of the stock price changes is implemented in the server, a distributed callback solution may still cause the server to consume a large amount of computing resources. For instance, it may need to filter data arriving from a real-time quote feed to determine which events to forward to clients that subscribe to the data.

• **Limited application utility:** Since the quoter application is no longer polling, and is instead waiting for the server to notify it, it can be used to work on other problems while it waits. The use of distributed callbacks also alleviates the need for multiple threads and the added complexity associated with multi-threaded programming.

Note that the definition of "distributed callback" provided here shows the terminology problem we described in Section 2.1: for a distributed callback, the server *sends* the request (and is thus a *client*), whereas the quoter application *receives* the request (and is thus a *server*). As we'll see below, the use of distributed callbacks results in peer-to-peer relationships – the "client" and "server" are peers because each is an object that both sends and receives requests. Therefore, we'll use different terminology from now on. A "supplier" is an entity that produces events, while a "consumer" is one that receives event notifications and data. This model, where one or more applications register to receive data as it is generated, is often referred to as the "publish/subscribe" model. The publish/subscribe model has its roots in patterns like Observer [2] and Model/View/Controller [3].

3 Using Distributed Callbacks in the Quoter Application

This section illustrates how to use distributed callbacks to implement a more flexible, and potentially more efficient, stock quoter application.

3.1 Defining the IDL Interface

As we've shown in our last few columns, CORBA applications must have an object reference before a request can be issued. Therefore, two resources are needed before a CORBA distributed callback can be made: a callback object and its object reference. One way to obtain these resources is to pass the distributed callback object reference as a parameter to another object, which registers the distributed callback. This behavior is shown by the `register_callback` invocations in Figure 2.

The example below extends the IDL declarations shown above with a distributed callback interface and registration operation:

```
module Stock {
  // Requested stock does not exist.
  exception Invalid_Stock {};

  // Distributed callback information.
  module Callback {
    struct Info {
      string stock_name;
      long value;
    };

    // Distributed callback interface
    // (invoked by the Supplier).
    interface Handler {
      void push (in Info data);
    };
  };

  // This is the same as in our earlier columns.
  interface Quoter {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };

  interface Notifying_Quoter {
    // Register a distributed callback
    // handler that is invoked when the
    // given stock reaches the desired
    // threshold value.
    void register_callback
           (in string stock_name,
            in long threshold_value,
            in Callback::Handler handler)
      raises (Invalid_Stock);

    // Remove the handler.
    void unregister_callback
          (in Callback::Handler handler);
  };
};
```

Several key changes have been made to our original IDL interface:

• A new module named `Callback` has been nested inside the `Stock` module. We used nesting so that both the `Info` struct and the `Handler` interface could be properly scoped and grouped together, without having to include the string "Callback" in the name of each type (*e.g.*, `Callback_Info`).

• Within the `Callback` module, the `Info` struct and the `Handler` distributed callback interface have been defined. The `Info` struct is used to inform the callback object of the name and value of the stock in question. This information is necessary to allow one callback object to be registered for multiple stock callbacks. The `Handler` interface defines the type expected for object references registered as callbacks.

- A new interface, `Notifying_Quoter`, has been added. It supplies two operations: `register_callback` allows callback handler object references to be added to the supplier, while `unregister_callback` is used to remove callback handlers. Note that `Notifying_Quoter` could derive from our original `Quoter` interface. However, doing this means that all `Notifying_Quoter` objects must support both polling and callbacks. This feature is something that most clients are unlikely to require, so we omitted it.

3.2 Defining the Consumer's Callback Behavior

To register a distributed callback object, a consumer passes a `Callback::Handler` object reference to the `Notifying_Quoter::register_callback` operation, which is implemented by the supplier. The following class defines the callback object implementation provided by the consumer:

```
// Implemented by the consumer.

class My_CallBack
  : public HPSOA_Stock::Callback::Handler
    // This base class is explained below.
{
public:
  // ...

  // Handle callback from quoter supplier.

  void push (const Stock::Callback::Info& info)
  {
    // Class BuyOrder is defined elsewhere.
    Buy_Order buy (info.stock, 1000);

    // Issue a "buy" order.
    buy.issue ();

    // ...
  }
};
```

The `push` method of `My_Callback` is invoked by the supplier when the threshold specified by the consumer is reached. In response to this notification, the consumer creates an order to buy 1,000 shares of the stock.

3.3 The Consumer Main Function

After defining the consumer's callback behavior, we'll write our consumer application `main`. This function gets the callback object reference and registers it with the `Notifying_Quoter` on the supplier.

```
// Consumer application.

int main (int argc, char *argv[])
{
  using namespace Stock;

  CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv, 0);
  CORBA::HPSOA_var hpsoa =
    orb->HPSOA_init (argc, argv, CORBA::HPSOAid);

  const char *stock_name = "ACME ORB Inc.";

  // Create a new implementation object.
  My_Callback *cb = new My_Callback;

  // Get the callback object reference.
  Callback::Handler_var handler = cb->_this ();

  // Obtain a Notifying_Quoter object reference,
  // e.g., from the Naming or Trader service
  // (which is not shown).
  Notifying_Quoter_var quoter = // ...
  CORBA::Long threshold = TRADING_THRESHOLD;

  // Register callback with the supplier.
  quoter->register_callback (stock_name,
                             threshold, handler);

  // Now instruct the object adapter to
  // wait for callbacks from the supplier.
  hpsoa->run ();
  /* NOTREACHED */
}
```

In this example, we're using the *HP Simplified Object Adapter* supplied by the HP ORB Plus product to obtain an object reference for our callback object. As we've described in our previous columns, CORBA 2.0 currently lacks a portable Object Adapter interface. Therefore, in this column we'll use the object adapter supplied by the ORB vendor we're using. When the OMG completes its portability enhancement work (which is currently in progress), we'll update our examples to use the new portable object adapter mapping.

When the value of ACME ORB stock reaches the "`TRADING_THRESHOLD`" (*e.g.,* $103.00), the supplier (in the same process as the Quoter object, perhaps even the Quoter itself) will call back to the registered handler object like this:

```
// Implemented by the supplier.

Stock::Callback::Handler_ptr handler;
Callback::Info info;

// Assign name and value.
info.stock_name =
  CORBA::string_dup ("ACME ORB Inc.");
info.threshold_value = TRADING_THRESHOLD;

// Query the real-time quote feed database
// to see if the threshold is reached.
// (see detailed implementation below).
if (reached_threshold (info.stock_name,
                       info.threshold_value))
{
  // Disseminate info to the consumer(s).
  handler->push (info);

  // ...
}
```

Upon receiving the distributed callback from the supplier (the quoter service), the consumer (the stock quoter application) can examine the members of the `Callback::Info` `struct` passed to it. The consumer application can use this `struct` in various ways. For example, it can determine the name of the stock and its current value, and use this information to issue a buy or sell order for that particular stock.

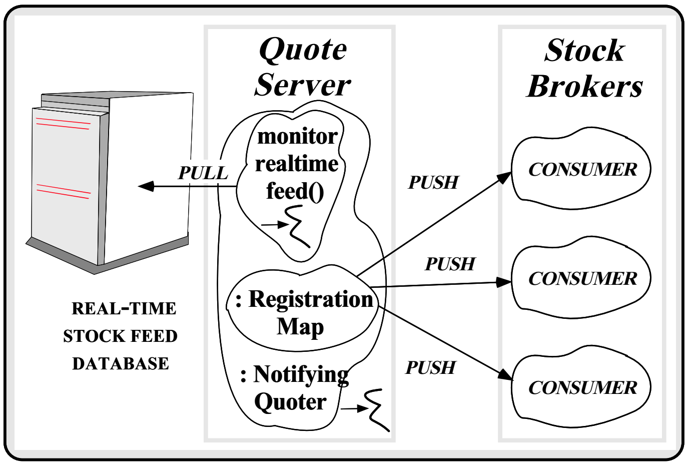

Figure 3: Internal Architecture of the Quote Supplier

If the consumer is no longer interested in receiving callbacks, it can unregister the callback handler using the `unregister_callback` operation:

```
quoter->unregister_callback (handler);
```

4 Implementing the Distributed Callback Supplier

This section examines the implementation of the supplier-side of the quoter service.

4.1 Supplier Architecture

Our stock quote supplier has three primary components (shown in Figure 3):

1. *Registration map* – This map associates each callback object reference with a stock name and threshold value. To simplify programming, we use STL multimaps to implement this association.
2. *Real-time stock feed database* – We assume that the stock feed provides a continuous stream of stock names and their associated values are stored in a database that can be read by our implementation of the `Notifying_Quoter` interface.
3. *Monitor for real-time stock feed* – This monitor provides up-to-date stock information to the stock quote supplier. The supplier uses this information to determine when callbacks should be issued to consumer (s).

Our `Notifying_Quoter` implementation is an "active object" [4]. It spawns a thread in its constructor to monitor the real-time stock feed. This means that our registration and unregistration operations must occur in their own threads. Therefore, we must make sure that updates to the registration map do not occur while the monitoring method accesses the map. To do this, we'll use mutex locks, which ensure the integrity of our registration map.

HP ORB Plus uses a C++ threads portability class library called `MSD` to achieve independence from platform-specific threads APIs. This library is also available to applications that use the ORB. Our `Notifying_Quoter` implementation therefore makes use of the `MSD_Thread`, `MSD_Mutex`, and `MSD_Lock` classes provided by the `MSD` library to spawn threads and provide mutual exclusion.

To actually implement a `Notifying_Quoter`, we first define a C++ class. ORBs generally allow the implementor to select how to integrate their code with the skeleton generated by the IDL compiler. As we've shown in previous columns, the two main choices are variants of the Adapter pattern [2]:

- *Class Adapter* – which derives the implementation class from a skeleton class generated by an OMG IDL compiler;
- *Object Adapter* – which makes a stand-alone class where an interface (the Adapter) delegates to an object it holds (the implementation).

In this column, we'll use the inheritance method provided by HP ORB Plus. Therefore, we derive our class from the abstract `HPSOA_Stock::Notifying_Quoter` skeleton class, as follows:

```
class My_Callback_Quoter
  : public HPSOA_Stock::Notifying_Quoter
{
public:
  My_Callback_Quoter (void) {
    // Spawn a thread to monitor the feed.
    thread_ = new MSD_Thread (start_thread, this);
  }
  ~My_Callback_Quoter (void) {
    delete thread_;
  }

  // Register a distributed callback
  // handler to callback when threshold
  // is reached.
  virtual void register_callback
          (const char *stock,
           CORBA::Long threshold_value,
           Stock::Callback::Handler_ptr handler)
       throw (CORBA::SystemException,
             Stock::InvalidStock);

  // Remove the handler when consumer is
  // no longer interested in receiving callbacks.
  virtual void unregister_callback
 (Stock::Callback::Handler_ptr handler);

private:
  // Perform the work of monitoring the real-time
  // quote feed.
  void monitor_realtime_feed (void);

  // Determines if stock has reached its threshold.
  int reached_threshold (char *, long);

  // Static member function passed to
  // the MSD_Thread constructor.
  static void *start_monitor (void *p) {
    My_Callback_Quoter *q =
       static_cast<My_Callback_Quoter*> (p);
    q->monitor_realtime_feed ();
    return 0;
```

```
  }

  // Maps stock names to callbacks.
  Callback_Map cb_map_;

  // Pointer to monitoring thread.
  MSD_Thread *thread_;

  // Mutex for the Callback_Map.
  MSD_Mutex mutex_;

  // ...
};
```

Our Notifying_Quoter implementation requires some means to store the association between the stocks it monitors and CORBA object references it maintains to distributed callbacks. We use the STL multimap type Callback_Map for this purpose, as follows:

```
// Use STL pair and multimap containers.
typedef pair<Stock::Callback::Handler_ptr,
             CORBA::Long> Callback_Value;
typedef multimap<string, Callback_Value>
        Callback_Map;
```

A multimap is necessary because it allows duplicate keys. We need this feature because multiple callbacks from multiple consumers could be registered for the same stock name. For example, two separate quoter applications could each register a callback for the stock of "ACME ORBs, Inc." In addition, we'll have to provide our own locking using the mutex mechanisms in Hewlett-Packard's ORB Plus product since the STL implementation we use isn't thread-safe.

4.2 Registering Callbacks

The register_callback method can be written as follows:

```
void
My_Callback_Quoter::register_callback (
    const char *stock,
    CORBA::Long threshold,
    Stock::Callback::Handler_ptr handler)
{
  using namespace Stock::Callback;

  // Create Callback_Value structure to hold
  // threshold and Handler object reference.
  Callback_Value val;
  val.first = Handler::_duplicate (handler);
  val.second = threshold;

  // Create item type for multimap, consisting
  // of the stock name and the Callback_Value.
  pair<const string, Callback_Value> item;
  item.first = string (stock);
  item.second = val;

  // Lock the callback map and insert the
  // new registration item. (The destructor
  // of the lock object unlocks the mutex).
  MSD_Lock lock (mutex_);
  cb_map_.insert (item);
}
```

Most of the register_callback method is straightforward. When a consumer registers a Callback::Handler we store a "pair of pairs" in the callback map. The monitor_realtime_feed method shown below illustrates how this is used.

One tricky aspect of our design involves the use of Handler::_duplicate. This is required because the object reference passed to register_callback is only valid for this method invocation. Once this invocation of the register_callback method returns, the object reference might be released. For example, if register_callback is upcalled from a skeleton, the skeleton will probably release the object reference before responding to the remote caller. Therefore, the _duplicate call ensures that the object reference we insert into cb_map_ does not become a dangling object reference.

4.3 Monitoring the Quote Feed

The following is a simple implementation of a method that monitors the realtime quote feed:

```
// Supplier application.

void
My_Callback_Quoter::monitor_realtime_feed ()
{
  using namespace Stock;

  for (;;) {
    // Sleep for a period to avoid ``busy waiting.''
    sleep (POLL_PERIOD);

    // Lock the registration table and
    // iterate through it (destructor
    // releases the mutex).
    MSD_Lock lock (mutex_);

    Callback_Map::iterator iter;

    for (iter = cb_map_.begin ();
         iter != cb_map_.end ();
         iter++) {
      // Access Callback_Value structure.
      const string& stock = (*iter).first;
      Callback_Value& cbv = (*iter).second;
      CORBA::Long threshold = cbv.second;

      // Query the real-time quote feed database
      // to see if the threshold is reached.
      if (reached_threshold (stock, threshold)) {
        // Create Info structure to push
        // to the callback consumer.
        Callback::Info info;
        info.stock_name =
          CORBA::string_dup (stock.c_str ());
        info.threshold_value = threshold;

        // Retrieve Handler from Callback_Value.
        Callback::Handler_ptr handler = cbv.first;

        // Disseminate info the consumer (s).
        handler->push (info);

        // Release the handler object reference
         // and destroy the callback info.
        CORBA::release (handler);
        cb_map_.erase (iter);
      }
    }
  }
  /* NOTREACHED */
}
```

As shown in Section 4.3, the monitor_realtime_feed method is spawned by the My_Callback_Quoter constructor. This method run continuously in its own thread. The main portion of the method iterates over all the entries in the callback registration map. As explained above, the callback map must be locked to prevent simultaneous access by this method and the register_handler and unregister_handler methods.

For each entry, the stock name and the threshold value are passed to the reached_threshold method. This method queries the real-time quote feed database to see if that stock's value matches the desired threshold. If a match occurs, a Callback::Info struct is created and initialized with the stock name and value, the callback handler object reference is retrieved from the callback map, and the Info struct is "pushed" to the callback handler. Finally, the registration for the callback that was just pushed by the supplier is destroyed.

The decision to destroy the callback handler after each successful push is designed to prevent suppliers from flooding consumers with multiple callbacks if the stock data remains at the designated threshold. This implies that the quoter application consumer must re-register the My_Callback handler if it wants to receive subsequent notifications for the stock. There are obviously other protocols that we could have used here, as well. The "best" approach will most likely be revealed through prototyping and benchmarking experiments.

4.4 Unregistering Callbacks

In addition to having successful "pushes" automatically remove handlers from the supplier, consumers can also unilaterally decide to remove callback handlers by invoking the supplier's unregister_callback method. In this case, the implementation of the unregister_callback must remove the given callback object reference from the cb_map_ multimap. This is accomplished using STL iterators, as follows:

```
using namespace Stock::Callback;

void
My_Callback_Quoter::unregister_callback
 (Stock::Callback::Handler_ptr handler)
{
  // Destructor releases lock.
  MSD_Lock lock (mutex_);
  Callback_Map::iterator iter;

  for (iter = cb_map_.begin ();
       iter != cb_map_.end ();
       iter++) {
    // Access Callback_Value structure.
    Callback_Value& cbv = (*iter).second;

    // Retrieve Handler from Callback_Value.
    Handler_ptr h = cbv.first;

    // Check handler equivalence.
    if (h->_is_equivalent (handler)) {
      CORBA::release (h);
      cb_map_.erase (iter);
      break;
    }
  }
}
```

This method simply iterates over the cb_map_ multimap, comparing each stored callback handler object reference to the one passed in as an argument. When a match is found, the object reference stored in the multimap is released, and then the entry is erased, which frees up the dynamically allocated memory.

5 Evaluation of the Distributed Callback Solution

One benefit of using CORBA is that the distributed callback architecture enables the consumer application to avoid polling the stock quote supplier. Instead, the consumer is notified when quotes reach their threshold, which may result in a more efficient overall solution. However, this approach has its own set of problems:

- **Object reference ambiguity:** For example, although our unregister_handler implementation appears to be straightforward and portable, it contains a subtle problem. The means by which the object reference passed to unregister_handler is compared to those stored in cb_map_ (the _is_equivalent function) may not actually work the way we need it to!

The _is_equivalent function is available for all object references because it is part of the CORBA::Object interface, which is the base interface for all OMG IDL interfaces. According to the CORBA 2.0 specification, _is_equivalent returns true if its object reference argument and the target object reference both refer to the same object. Unfortunately, the converse is not true: if _is_equivalent returns false, it may just mean that the *ORB was unable to determine* whether or not the two object references refer to the same object. In fact, a conforming ORB could implement the _is_equivalent operation to just always return false. In CORBA, there is no guaranteed way to determine object reference "equality" from the object references alone.

The main reason _is_equivalent has these somewhat odd semantics is to allow greater freedom for ORB implementors. The OMG members in favor of these semantics were concerned that stronger guarantees for _is_equivalent would be difficult to implement in some CORBA environments. For more information, see [5] and [6], which discuss the pros and cons of this issue, respectively.

Since we can't count on _is_equivalent in our implementation of the unregister_handler operation, our implementation must be redesigned. One way to solve the handler identification problem is to change register_handler to return a token that can later be passed to unregister_handler. Depending upon the quality of service offered by our Notifying_Quoter implementation, this token could be an integer counter, a Universal Unique Identifier (UUID), or even another object reference. Each of these token types provide different tradeoffs in terms of scalability, robustness, and performance.

• **Supplier-side polling:** With the addition of the distributed callback capability, our stock quoter went from simply having to look up stock values from an external source to having to continually monitor those stock values that its consumers have subscribed for. Achieving this required nontrivial changes in our stock quoter implementation. For instance, in our implementation shown above, the data arrives via a real-time quote feed. In this case, we'll have a number of design choices to avoid swamping the consumers, such as:

- We could use an active database that uses triggering and filtering on the supplier's side;
- We could spawn one or more threads on the supplier-side (as we did above) to poll the quote feed continuously looking for threshold matches.

• **Callback persistence:** A robust implementation of the distributed callback service must be able to store certain information persistently. This information might include the information in the STL multimap (such as the distributed callback object references, their associated callback object references, and their threshold data). Adding persistence to our supplier will require some major modifications from our original polling quoter. Because the previous implementation of our stock quoter merely had to look stock values up from an external source, it had no persistent storage requirements (other than the quote database itself).

• **Notification scalability and delivery timeliness:** The stock quoter must be able to deliver notifications quickly after it detects that a stock value it monitors has reached the desired value. This may be hard if thousands of distributed callback objects have registered with it. Performance may suffer if a Supplier blocks while notifying a Consumer that is slow to accept the `push` when the Supplier has other consumers to serve. Moreover, what "quick delivery" means to each distributed callback object can differ greatly, depending on its quality of service requirements.

There are several ways to handle these problems such as using oneway calls, timeouts, or additional threads to implement some form of asychronous "future." Another way to improve the scalability of distributed callback notifications is to utilize an ORB that supports reliable multicast semantics. The CORBA 2.0 specification deals mainly with point-to-point communication and offers no standard support for reliable multicast. However, there are ORB implementations (such as Electra [7] or Orbix+ISIS [8]) that extend CORBA to provide reliable multicast and fault-tolerant group communication.

• **Potential for deadlock:** Since `unregister_callback` is a twoway CORBA call, our distributed callback design can deadlock if the Supplier tries to `push` to a Consumer that simultaneously tries to unregister. This will almost certain happen if the ORB doesn't support "nested dispatching" of twoway calls. Nested dispatching enables an ORB to perform upcalls from incoming requests even while it is "blocked" on the request portion of a twoway request/response invocation. There are various ways to implement nested dispatching, such as spawning off a separate thread to handle each incoming request or using a non-blocking event loop within the ORB. If an ORB doesn't support nesting dispatching, it may be necessary to restructure the `Handler::push`, `Notifying_Quoter::register_callback`, and `Notifying_Quoter::unregister_callback` to use oneway semantics. Oneway calls can be problematic, however, since application developers then become responsible for ensuring end-to-end reliability.

Given the severity of these problems, it appears that our quest to avoid polling our stock quoters has revealed even more design challenges! Unfortunately, that's the way things often evolve when developing and deploying practical distributed computing systems. Such problems are very common, and often don't manifest themselves until late in a project lifecycle. This is yet one more reason why it's so important to build prototypes and conduct experiments using realistic use-cases and distributed environments [9], before adopting a particular communication architecture wholesale.

Luckily, the flexibility and higher levels of abstraction in CORBA help to alleviate unnecessary complexity and coupling. As we've seen repeatedly, CORBA features like "separating interface from implementation" and "making it possible to reconfigure the objects flexibly" allow us to defer some (but not all) of these decisions until we understand the system better.

The problems with our notifying stock quoter described above can be eased somewhat if we separate concerns even further. In particular, our quoter implementation has enough to worry about as it monitors and reports changing stock values. Therefore, we should avoid also making it responsible for delivery of multiple notifications and maintaining a persistent table of callbacks.

One way to relieve some of the burden we've placed on the stock quoter is to utilize the OMG Events Service for notification delivery. The Events Service is part of the OMG Common Object Services Specification (COSS) Volume 1 [1]. Its purpose is to provide delivery of event data from event suppliers to event consumers without requiring the suppliers and consumers to know about each other. An implementation of the Events Service acts as a "mediator" that provides for decoupled communications between objects. Our next column will focus on the COSS Events Service in detail.

6 Concluding Remarks

In this column, we've stepped away from the strict request/response model we've used for all of our examples to date, and described one way to decouple communication between suppliers and consumers. We've examined a complete design and implementation for distributed callbacks using CORBA.

Unlike our last few columns, this column focuses only on a solution based on the Object Management Architecture (OMA) and CORBA. The main reason for this is that even if we just showed the most important portions of the C and C++ code required for a solution, it would require far too much space. This is a consequence of all the C/C++ bookkeeping information that must be maintained for each callback, which makes the code too complex to explain succinctly. The fact that CORBA applications can pass object references helps hide all the bookkeeping information and allows us to present a fairly complete solution, without filling up the entire issue of C++ Report!

As always, if there are any topics that you'd like us to cover, please send us email at `object_connect@ch.hp.com`. Thanks to John Isner for his comments on the deadlock aspects of the distributed callback architecture.

References

[1] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1995.

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

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[4] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), (Reading, MA), Addison-Wesley, 1996.

[5] M. L. Powell, *Objects, References, Identifiers, and Equality White Paper*. SunSoft, Inc., OMG Document 93-07-05 ed., July 1993.

[6] W. Harrison, *The Importance of Using Object References as Identifiers of Objects: Comparison of CORBA Object*. IBM, OMG Document 94-06-12 ed., June 1994.

[7] S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," in *Proceedings of the Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

[8] C. Horn, "The Orbix Architecture," tech. rep., IONA Technologies, August 1993.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the 2^{nd} Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.