Object Interconnections

Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers (Column 7)

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 appeared in the July/August 1996 issue of the SIGS C++ Report magazine.

1 Introduction

Developers of multi-threaded servers face many challenges. One important challenge is selecting a suitable concurrency model. There are several concurrency models to choose from including *thread-per-request*, *thread-pool*, and *thread-per-session*. Our last two columns discussed the thread-per-request and thread-pool concurrency models, respectively. We showed how each could be used to develop multi-threaded server programs for a distributed stock quote application.

This column discusses the thread-per-session model, in which each new session created for a client is assigned a thread that processes requests for that session. Following the format of our recent columns, this column will illustrate the thread-per-session model by developing new multi-threaded stock quote servers using C, C++ wrappers, and CORBA.

2 The Thread-per-Session Concurrency Model

Figure 1 illustrates the main components in the thread-per-session concurrency model. These components include a *main thread* and a set of *session threads*. The main thread receives new session initiation requests from clients. It creates a new session thread to handle each client. Session threads receive and service stock quote requests from the clients.

Under certain circumstances, thread-per-session performs better than the thread-per-request and thread-pool models. Its advantages are (1) it amortizes connection setup costs and (2) it supports efficient long-duration conversations with clients. On the other hand, thread-per-session is less useful if certain sessions receive considerably more requests than others since they will become a performance bottleneck. Moreover, if each client makes only one request per session, the performance of the thread-per-session model is roughly the same as the thread-per-request model.

Naturally, we strongly urge you to analyze, prototype, and measure the performance of various concurrency models before committing to a particular approach. As we examine the C, C++ wrapper, and CORBA solutions below, keep in mind the assumptions built into the thread-per-session model (such as the typical number or size of client requests, the interarrival time between requests, and request duration). Note how the strengths and weaknesses of the solution will change as the assumptions change.

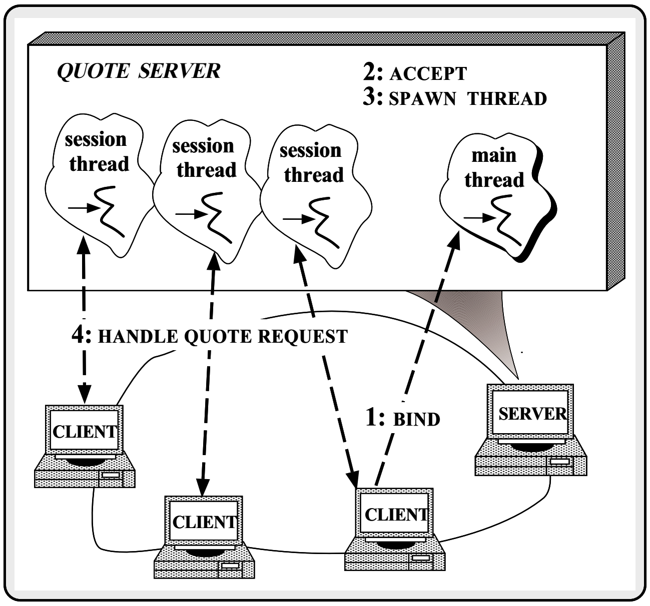

Figure 1: Thread-per-Session Architecture for the Stock Quote Server

3 The C Thread-per-Session Solution

3.1 Implementing Thread-per-Session in C

The following example shows a thread-per-session solution written using C, sockets, and Solaris threads [1].[1] As in

[1]Porting our implementation to POSIX pthreads or Win32 threads is straightforward.

previous columns, we use the following set of C utility functions:

```
/* WIN32 already defines this. */
#if defined (unix)
typedef int HANDLE;
#endif /* unix */

// Factory function that allocates a
// passive-mode listener socket.
HANDLE create_server_endpoint (u_short port);

// Receive stock quote requests from clients.
int recv_request (HANDLE h, Quote_Request *req);

// Return the quote to the client.
int send_response (HANDLE h, long value);

// Determine current stock price from the
// Quote Database.
long lookup_stock_price (Quote_Database *,
                         Quote_Request *);

// Calls recv_request(), lookup_stock_price(),
// and send_response().
int handle_quote (HANDLE h);
```

The implementations of these functions were first shown in the October 1995 issue of the C++ Report and were revised to become thread-safe in the February 1996 issue.

3.1.1 The main() Thread

The main thread runs an event loop that continuously accepts new connections from clients and spawns a thread to run each client session. Our server `main` is almost identical to the one we presented for the thread-per-request C solution in our February column:

```
const int DEFAULT_PORT = 12345;

int main (int argc, char *argv[])
{
  /* Port to listen for connections. */
  u_short port =
    argc > 1 ? atoi (argv[1]) : DEFAULT_PORT;

  /* Create a passive-mode listener endpoint. */
  HANDLE listener = create_server_endpoint (port);

  /* The event loop for the main server thread. */
  svc_run (listener);
  /* NOTREACHED */
}
```

The key difference is that unlike the thread-per-request model, we don't dynamically spawn a thread for each new client quote *request*. Instead, we spawn a thread for each client *session*, as shown in the following `svc_run` function (to emphasize the differences we've prefixed the changes with `/* !!!`):

```
void svc_run (HANDLE listener)
{
  /* Main event loop. */

  for (;;) {
    /* Wait to accept a new connection. */
    HANDLE handle = accept (listener, 0, 0);

    thr_create
       (0, /* Use default thread stack. */
        0, /* Use default thread stack size. */
        /* !!! Thread entry point. */
        &session_thread,
        (void *) handle, /* Entry point arg. */
        THR_DETACHED | THR_NEW_LWP, /* Flags. */
        0); /* Don't bother returning thread id. */
  }
  /* NOTREACHED */
}
```

3.1.2 The session_thread() Function

A session thread runs the the following function:

```
void *session_thread (void *arg)
{
  HANDLE handle = (HANDLE) arg;

  /* Process all stock quote requests from
     a client until it closes down. */

  while (handle_quote (handle) > 0)
    continue;

  /* Shutdown the handle to reclaim OS resources. */
  close (handle);

  /* Exit the thread. */
  thr_exit (0);
  /* NOTREACHED */
  return 0;
}
```

Each session thread runs for the duration of the client's conversation. The `session_thread` function repeatedly calls `handle_quote`. This function extracts stock quote requests from the database, looks up the results, and returns each result to the client. Since multiple session threads can access the quote database simultaneously, we'll reuse the thread-safe implementation of `handle_quote` from our February C++ Report column. This function returns 0 when a client closes down the session, at which point the session thread exits.

3.2 Evaluating the C Solution

The C solution presented above is very straightforward. The implementation is much simpler than the thread-pool solution we presented in our last column. In particular, there's no need to implement a thread-safe `HANDLE` queue because each session can block in its own thread. In addition, we can reuse most of the code from the thread-per-request C solution.

The thread-per-request C implementation from our February C++ Report column closed down the connection after every request. In contrast, our current C thread-per-session implementation keeps the connection open until the client explicitly closes it. This is beneficial if client applications make many requests to the same quote server.

Despite these advantages, the C solution suffers from the same problem described in previous columns, namely that it's written at a very low level. This makes it difficult to separate the problem of providing stock quotes from the problems associated with writing distributed applications. For instance, if we changed the format of stock quote requests and replies, we'd have to reimplement most of the utility code that we reused in this example.

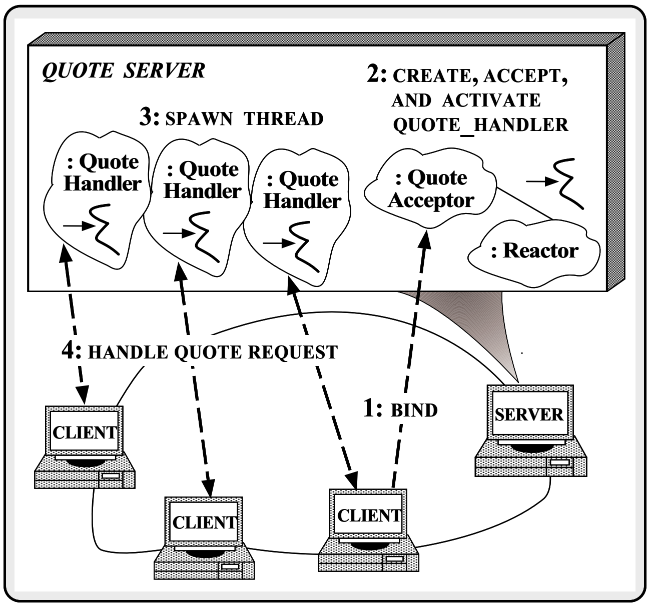

Figure 2: ACE C++ Architecture for the Thread-per-Session Stock Quote Server

4 The C++ Thread-per-Session Solution

4.1 Implementing Thread-per-Session in C++

This section illustrates a C++ thread-per-session implementation based on ACE [2]. The C++ solution is structured using the following three classes (shown in Figure 2):

• **Quote_Handler:** This class implements an *active object*[2] that interacts with clients by receiving quote requests, looking up quotes in the database, and returning responses.

• **Quote_Acceptor:** A factory that implements the strategy for accepting connections from clients, followed by creating and activating `Quote_Handlers`.

• **Reactor:** Encapsulates the `select` and `poll` event demultiplexing system calls with an extensible and portable callback-driven object-oriented interface. The `Reactor` dispatches the `handle_input` method of the `Quote_Acceptor` when connection events arrive from clients.

Variations of these components have been used in previous implementations of the quote server in our earlier C++ Report columns. Below, we illustrate how these components can be extended and reused to implement thread-per-session.

[2] An active object maintains its own thread of control, which allows it to block on I/O channels and process messages without directly impacting the quality of service of other active objects.

4.1.1 The Quote_Handler Class

The `Quote_Handler` class is responsible for processing client quote requests. Its implementation is very similar to the one used for the thread-per-request concurrency model:

```
// Reuse the C handle_quote() function.
extern "C" int handle_quote (HANDLE);

template <class STREAM> // IPC interface
class Quote_Handler
  : public Svc_Handler<STREAM>
    // This ACE base class defines "STREAM peer_;"
{
public:
  // !!! Assign a connected STREAM to this instance.
  Quote_Handler (STREAM &peer_stream) {
    Quote_Handler<STREAM>::peer_.set_handle
      (peer_stream.get_handle ());
  }

  // !!! This method is called by the Quote_Acceptor
  // to initialize a newly connected Quote_Handler,
  // which turns itself into an active object.
  virtual int open (void) {
    Thread::spawn
     // Static entry point into the thread.
     (&Quote_Handler<STREAM>::session_thread,
      (void *) this,              // Entry point arg.
     THR_DETACHED | THR_NEW_LWP); // Thread flags.
  }

  // !!! Static thread entry point method.
  static void *session_thread (void *args) {
    Quote_Handler<STREAM> *client =
      static_cast <Quote_Handler<STREAM> *> (args);

    // Extract out the client's socket HANDLE.
    HANDLE handle = client->peer_.get_handle ();

    // Process all stock quote requests from
    // a client until it closes down.  We
    // reuse our C handle_quote () function
    // we defined earlier.
    while (::handle_quote (handle) > 0)
      continue;

    // Shut down the STREAM to avoid memory
    // leaks and HANDLE leaks.
    client->close ();

    // Exit the thread.
    thr_exit (0);
    /* NOTREACHED */
  }

  // Close down the handler and release resources.
  void close (void) {
    // Close the connection to avoid HANDLE leaks.
    this->peer_.close ();

    // Commit suicide to avoid memory leaks...
    delete this;
  }

  // ...
};
```

Each session thread executes the static member function `session_thread`. This function is almost identical to the C function of the same name defined in Section 3.1.2. In fact, the C++ version even calls the C `handle_quote` utility to perform the stock quote lookup. As usual, C++'s ability to integrate existing C code pays off by reducing effort.

When the client closes down, the `Quote_Handler` cleans up the connection. The only real difference between the C and C++ `session_thread` functions is that the C++

version must deallocate itself before the thread exits since the Quoter_Acceptor factory dynamically allocated the memory for the Quote_Handler.

4.1.2 The Quote_Acceptor Class

The Quote_Acceptor class is an implementation of the Acceptor pattern [3] that creates Quote_Handlers to process quote requests from clients. Its implementation is identical to the one shown in our previous column:

```
typedef Acceptor <
        Quote_Handler <SOCK_Stream>, // Quote service.
        SOCK_Acceptor> // Passive conn. mech.
        Quote_Acceptor;
```

When a client connects with the server, the Reactor invokes the handle_input method of the Quote_Acceptor automatically. This method initializes a client's Quote_Handler by performing the following three-step Quote_Acceptor strategy:

1. *Handler creation* – which dynamically creates a Quote_Handler.
2. *Handler connection acceptance* – which accepts the connection into the handler using the SOCK_Acceptor (this is a C++ wrapper for passive-mode sockets that creates connected SOCK_Streams [4]).
3. *Handler activation* – which invokes the Quote_Handler::open method. In the thread-per-session implementation, this open method spawns a new thread to handle client requests, as we showed in Section 4.1.1 above.

4.1.3 The main() Server Function

The C++ server main is responsible for initializing the Quote_Acceptor and running the main event loop, as follows:

```
// Default constants.
const int DEFAULT_PORT = 12345;

int main (int argc, char *argv[])
{
  u_short port =
    argc > 1 ? atoi (argv[1]) : DEFAULT_PORT;

  // Server address.
  INET_Addr server_addr (port);

  // Factory that produces connected Quote_Handlers.
  Quote_Acceptor acceptor (server_addr);

  // The event loop for the main server thread.
  svc_run (acceptor);
  /* NOTREACHED */
}
```

The svc_run function shown below is identical to the one used by the thread-pool:

```
void svc_run (Quote_Acceptor &acceptor)
{
  // Install Quote_Acceptor with Reactor.
  REACTOR::instance ()->register_handler (&acceptor);

  // Event loop that dispatches all events as
  // callbacks to appropriate Event_Handler subclass
  // (such as the Quote_Acceptor).

  for (;;)
    REACTOR::instance ()->handle_events ();
  /* NOTREACHED */
}
```

The main thread's event loop runs continuously within the REACTOR Singleton, which calls back to the Quote_Acceptor's handle_input method when connections arrive from clients. This method implements the Acceptor pattern strategy shown in Section 4.1.1 to create, accept, and activate a new Quote_Handler. Unlike our thread-pool implementation, however, all subsequent the stock quote request dispatching and processing takes place in the session threads.

4.2 Evaluating the C++ Solution

The thread-per-session C++ solution is an improvement over the thread-pool and thread-per-request C++ implementations in previous columns. Like the C version, the Quote_Handler thread keeps running until the client disconnects and doesn't need a Request_Queue since each session thread blocks independently. The following are some other advantages of our C++ solution.

• **Simpler connection management:** Our C++ implementation of thread-pool from our last column maintained a cache of client connections. However, our thread-per-session C++ connection management is much simpler than the thread-pool model. For instance, there's no need for complex reference counting to ensure that a Quote_Handler is not deleted until all Quote_Requests stored in the Request_Queue are removed.

• **More flexible design:** The C++ version of thread-per-session is more flexible than the C version largely because it is based on components in the ACE framework. The ACE components provide a generic software architecture consisting of a Reactor, Acceptor, and Svc_Handlers, as well as a set of standard default behavior (such as event demultiplexing and factories). In addition, the ACE components help to decouple the concurrency model of the Quote_Handler's from the rest of the quote server architecture. For instance, the decision to become an active object is localized within the open method of the Quote_Handler, rather than with the main svc_run method, as it is with the C implementation. This makes it possible to switch concurrency schemes very easily without affecting existing code. If you examine the C++ solutions in our recent columns, you'll see that they all have a common software architecture that can be customized easily to support different concurrency models.

As usual, the C++ solution is an improvement over the C version, but it still doesn't adequately automate some common tasks (such as marshaling and object activation) necessary to build distributed applications. Therefore, we'll take a look at a CORBA solution that does address more of these issues.

5 The CORBA Thread-per-Session Solution

The thread-per-session concurrency model is supported by a number of CORBA implementations including MT-Orbix and ORBeline. This section illustrates how to program the client and server sides of our thread-per-session stock quote implementation using the concurrency features of MT-Orbix. We'll first examine the changes we had to make to the IDL `Stock` module and the client-side application and then explore the thread-per-session server implementation in detail.

5.1 IDL Changes

To accommodate the thread-per-session concurrency model, we modified the IDL `Stock` module as shown below:

```
// Define the interface for a stock quote server.
module Stock
{
  exception Invalid_Stock {};
  exception Invalid_Quoter {};

  interface Quoter {
    // Returns the current stock value or
    // throws an exception.
    long get_quote (in string stock_name)
      raises (Invalid_Stock, Invalid_Quoter);

    // Destroy a Quoter session and
    // release resources.
    void destroy ();
  }

  // Manage the lifecycle of a Quoter object.
  interface Quoter_Factory {
    // Returns a new Quoter selected by name
    // e.g., "Dow Jones," "Reuters,", etc.
    Quoter create_quoter (in string name)
      raises (Invalid_Quoter);
  };
};
```

The new IDL interface adds a `Quoter_Factory` that creates `Quoters`. There are several benefits of the Object-style approach vs. the RPC-style approach:

• **Customized quality of service:** Clients can use a `Quoter_Factory` to create different types of `Quoters` that support a range of functionality or performance characteristics tailored to their individual needs. For instance, the `Quoter_Factory` can return a new `Quoter` selected by a stock quoting service name such as "Dow Jones" or "Reuters." Likewise, the factory operation provided by the `Quoter_Factory` might take parameters that determine the implementation and behavior of the created `Quoter`. One such parameter might control how recent the stock value quotes handed out by the `Quoter` must be. In addition, on public access ATM networks that support variable bandwidth allocation, a `Quoter_Factory` might create `Quoter` objects whose quality of service depends on rates paid by clients.

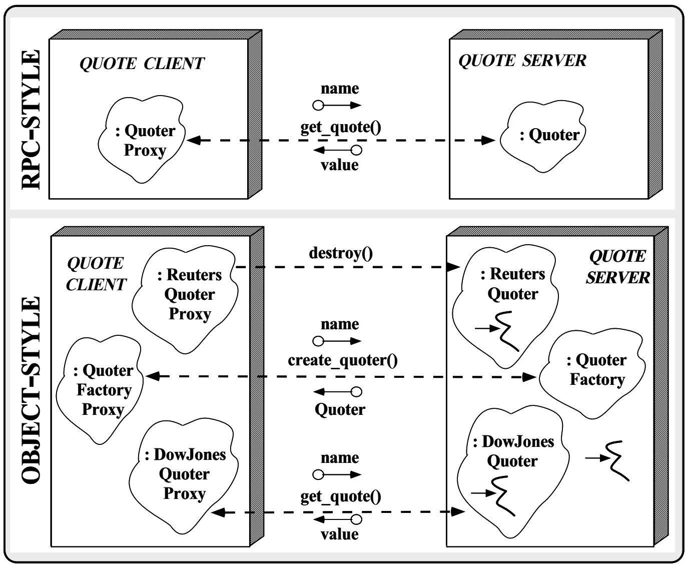

Figure 3: RPC-style vs. Object-style Communication

• **Efficient load balancing:** A `Quoter_Factory` can ensure new `Quoter` objects are created in particular locations to simplify administration or reduce overhead. Section 5.3 illustrates one approach, where the `Quoter_Factory` always creates `Quoter` objects within its own process. Directing all `Quoter` creation requests to only one `Quoter_Factory` can create a performance bottleneck, however. Therefore, if multiple host machines are available, several `Quoter_Factory` objects can be created, one on each machine. A *factory finder* service could be used to select the `Quoter_Factory` that can create `Quoter` objects on the host machine with the lightest load.

• **Flexible lifecycle control:** Our `Quoter_Factory` gives clients more flexibility to control the lifecycle of stock quoter implementations than our previous design. For instance, we've also added a `destroy` operation to the `Quoter` interface. This allows clients to release server resources (such as threads or client-specific state) when a session terminates. Without the `destroy` interface, the server must implement a more complex distributed reference counting schemes to determine when to release client resources.

5.2 Client Changes

In addition to modifying the server implementation, our use of a `Quoter_Factory` affects the way that clients interact with the server. The client-side approach in our May 1995 column used the following "RPC-style" interface to invoke remote operations:

```
// Create desired service name.
const char *name = "Quoter";
Name service_name;
service_name.length (1);
service_name[0].id = name;

// Initialize and locate the Quote service.
Object_var obj =
```

```
    bind_service (argc, argv, service_name);

  int result = -1;

  try {
    // Narrow to Quoter interface and away we go!
    Quoter_var q = Quoter::_narrow (obj);

    const char *stock_name = "ACME ORB Inc.";
    CORBA::Long value = q->get_quote (stock_name);
    cout << "value of " << stock_name
         << " = $"
         << value << endl;
    result = 0;
  } catch (CORBA::BAD_PARAM) {
    cerr << "_narrow() failed: "
         << service_name
         << " is not a Quoter!";
  } catch (Invalid_Stock &) {
    cerr << stock_name
         << " is not a valid stock name!\n";
  }
  return result;
  // Destructor of q releases object reference.
}
```

This code isn't much different from programming with DCE or Sun RPC. As shown in the top part of Figure 3, a client uses the `bind_service` utility function[3] to acquire a "binding handle" to the remote service. It then uses this handle to invoke a method call on that service. The primary difference between an RPC solution and our CORBA solution is that we encapsulate the binding handle within an *object reference*. This reference refers to an individual CORBA object instead of referring to an RPC server port.

As discussed in Section 5.1, using an RPC-style interface to implement thread-per-session is less flexible than using the interface provided by the `Quoter_Factory`. As illustrated in the bottom part of Figure 3, the `Quoter_Factory` interface allows clients to talk to objects via object references that serve as proxies to individual sessions. In this "object-style" interface, sessions can use utilize a different quoter service (such as Dow Jones, Reuters, etc.) and can each run in their own thread of control.

Adding a `Quoter_Factory` results in the the following changes to the client (as before, we've marked them with `// !!!` to emphasize the difference):

```
  // !!! Create desired service name.
  const char *name = "Quoter_Factory";
  Name service_name;
  service_name.length (1);
  service_name[0].id = name;

  // !!! Initialize and locate Quoter_Factory
  // service.
  Object_var obj =
    bind_service (argc, argv, service_name);

  int result = -1;

  try {
    // !!! Narrow to Quoter_Factory
    // interface and away we go!
    Quoter_Factory_var qf =
      Quoter_Factory::_narrow (obj);

    // !!! Select name of desired quoter.
    const char *quoter_name = "My Quoter";

    // !!! Ask factory to produce a new Quoter.
    Quoter_var q = qf->create_quoter (quoter_name);

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

    CORBA::Long value = q->get_quote (stock_name);
    cout << "value of " << stock_name
         << " = $"
         << value << endl;

    // !!! Explicitly destroy the Quoter.
    q->destroy ();

    // !!! Destructor of q releases object reference.
    result = 0;
  } catch (CORBA::BAD_PARAM) {
    cerr << "_narrow() failed: "
         << service_name
         << " is not a Quoter_Factory!";
  } catch (Invalid_Stock &) {
    cerr << stock_name
         << " is not a valid stock name!\n";
  }
  return result;
  // !!! Destructor of qf releases object reference.
}
```

[3] The `bind_service` function hides the details of initializing the ORB and locating object references with the CORBA Naming service [5].

Incidentally, the notion of customized object creation is supported by the OMG Lifecycle Service Specification [5]. It specifies a `GenericFactory` interface intended to allow clients to create a wide variety of objects. One of the arguments to the `GenericFactory::create_object` operation is a sequence of named `any` values. Different `GenericFactory` implementations can use these values to help decide what object to create, or might pass the values on to the newly-created object, or might even do both. Although we've omitted the COSS Lifecycle services in our example to save space, future columns will address this topic in depth.

5.3 Implementing Thread-per-Session in MT-Orbix

The MT-Orbix implementation of thread-per-session is a classic example of the Active Object pattern [6]. Each active object is responsible for servicing a different client session. Our thread-per-session implementation uses the thread-safe `Message_Queue` class defined in the thread-pool implementation from our previous column. Rather than maintaining one queue of incoming requests per server, however, each session has its own thread and its own queue. MT-Orbix needs this queue to provide its multiple concurrency models in a relatively uniform way.

There are several differences from the thread-per-request and thread-pool implementations shown in our previous two columns. First, the thread-per-session concurrency model does not pre-spawn any threads in advance. In addition, we've added a `destroy` operation, which helps manage the lifecycle of client sessions.

5.3.1 The My_Quoter Class

The `My_Quoter` class shown below implements the bulk of the thread-per-session stock quoter behavior using MT-Orbix and ACE components:

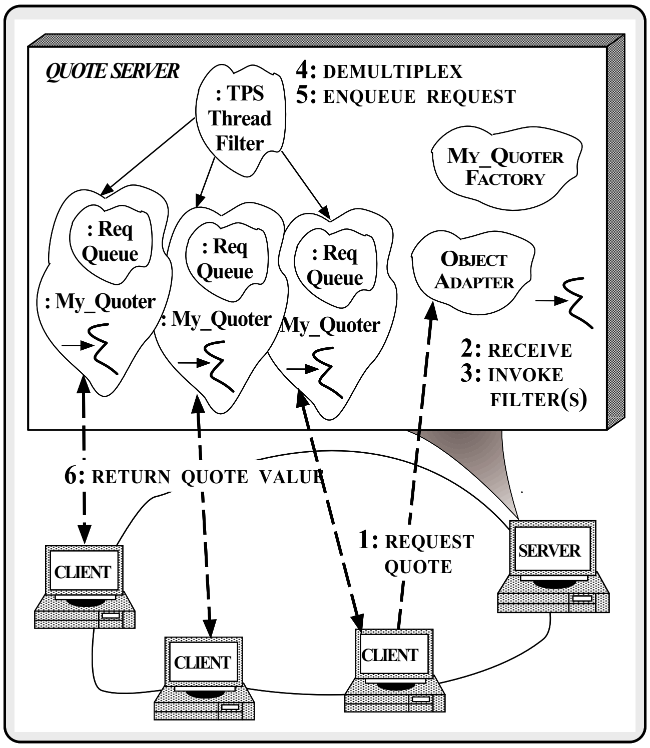

Figure 4: MT-Orbix Architecture for the Thread-per-Session Stock Quote Server

```
class My_Quoter
{
public:
  // Constructor
  My_Quoter (const char *name);

  // A thread executes this per-active object.
  static void *session_thread (void *);

  // Returns the current stock value (this is
  // the same implementation as the thread-pool).
  virtual long get_quote (const char *stock_name,
                          CORBA::Environment &);

  // Thread filter uses this method to queue
  // the Request to the thread than handles the
  // client session.
  virtual void insert_at_tail (CORBA::Request *req)
  {
    // Insert Request into queue, blocking if full.
    msg_queue_.insert (req);
  }

  // Destruction operation
  virtual void destroy (CORBA::Environment &) {
    // Insert a NULL pointer, which notifies
    // the session thread to shutdown.
    msg_queue_.insert (NULL);
  }

protected:
  // Queue of pending requests handled by our thread.
  CORBA::Request *remove_head (void) {
    CORBA::Request *req;

    // Called by the session thread to dequeue
    // the next message from its client.  Will block
    // if queue is empty.
    msg_queue_.dequeue (req);
    return req;
  }

  // ACE thread-safe message queue containing
  // CORBA Request pointers for this session.
  Message_Queue<CORBA::Request *> msg_queue_;

  // ...
};

DEF_TIE_Quoter (My_Quoter)
```

As in our previous column, the `My_Quoter` class is connected into the Orbix Object Adapter by using the Orbix "TIE" approach instead of inheriting from a skeleton class generated from the `Quoter` IDL. In particular, note that the `My_Quoter` class doesn't inherit from any base class. Instead, it use the Orbix "TIE" approach to associate the CORBA interfaces with our implementation. The "TIE" approach is used for both our `My_Quoter` class and for our `Quoter_Factory` implementation, shown below.

5.3.2 The My_Quoter_Factory Class

Factory objects provide construction operations that can take different numbers and types of arguments. The `My_Quoter_Factory` class is a CORBA "constructor" that creates a suitable `Quoter` implementation in response to a client request, as follows:

```
class My_Quoter_Factory
{
  Quoter_ptr create_quoter (const char *name,
                            CORBA::Environment &env) {
    Quoter_ptr quoter;

    // Perform Factory Method selection of
    // the subclass of Quoter.

    if (strcmp (name, "Dow Jones") == 0)
      quoter = new TIE_Quoter (Dow_Jones_Quoter)
                      (new Dow_Jones_Quoter (name);
    else if (strcmp (name, "Reuters") == 0)
      quoter = new TIE_Quoter (Reuters_Quoter)
                      (new Reuters_Quoter (name);
    else if (strcmp (name, "My Quoter") == 0)
      // Dynamically allocate a new My_Quoter object.
      quoter = new TIE_Quoter (My_Quoter)
                        (new My_Quoter (name);
    else {
      // Raise exception.
      env.exception (new Stock::Invalid_Quoter);
      return;
    }

    // Increment reference count.
    quoter->_duplicate ();

    // Attach a new thread to the Quoter object.
    Thread::spawn (&My_Quoter::session_thread,
                   // Get the Quoter instance.
                   DEREF (quoter),
                   THR_DETACHED | THR_NEW_LWP);
    return quoter;
  }
};

DEF_TIE_Quoter_Factory (My_Quoter_Factory)
```

The `create_quoter` operation is a Factory Method [7] that's called by the Object Adapter when a client initiates a session. It uses the `name` of the quoter service passed in by the client to help select an appropriate `Quoter` implementation.

Our client in Section 5.2 specified the My_Quoter implementation. Therefore, the factory will create a new My_Quoter, duplicate its object reference, spawn a thread for the new client session, and return the object reference of the newly-created Quoter object.

Note how the Orbix-specific DEREF macro is used to access the actual implementation object of My_Quoter. This implementation object is encapsulated within the "TIE" that associates the automatically generated IDL skeleton with the My_Quoter implementation.

5.3.3 The session_thread Method

The session_thread method shown below is a static C++ member function used as the entry point into the thread maintained for each client session:

```
void *My_Quoter::session_thread (void *arg)
{
  My_Quoter *quoter = static_cast<My_Quoter *> (arg);

  // Loop forever, receiving new Requests,
  // and dispatching them....
  for (;;) {
    CORBA::Request *request = quoter->remove_head ();

    if (request != NULL)
      // This call will perform the upcall,
      // send the reply (if any) and
      // delete the Request for us...
      CORBA::Orbix.continueThreadDispatch (*request);
    else {
      // A NULL pointer signifies that the client
      // has shutdown via the destroy() operation.
      CORBA::release (quoter);
    }
  }

  return 0;
}
```

Note how similar this event loop is to the pool_thread method in our previous column. The primary difference is that in the thread-pool implementation, there were a fixed number of threads running the same event loop (*i.e.,* one for each thread in the pool). In contrast, there is a separate thread running the event loop shown in session_thread for each active client.

One advantage of the thread-per-session model is that the same connection can be maintained as long as the association between the client and its Quoter object is maintained. In contrast, the thread-pool model doesn't necessarily maintain this association (though we implemented it both ways in our previous column).

An interesting part of the session_thread function is its handling of the destroy operation. As shown above, the implementation of destroy puts a NULL Request pointer onto the object's message queue. When session_thread removes a NULL pointer from its queue, it calls CORBA::release to release the object reference and destroy the CORBA object. Some ramifications of this approach are discussed below in Section 5.4.

5.3.4 The TPS_Thread_Filter Class

Now we need a way to bring all the pieces together. In MT-Orbix, this is accomplished via a ThreadFilter. As we've shown in previous columns, Orbix allows applications to interpose C++ "filter" objects into the request dispatch path. Filters can perform a number of tasks such as intercepting, modifying, or examining each request sent to and from the system.[4] To dispatch an incoming CORBA request to its intended session thread, we've created a subclass of the Orbix ThreadFilter class that overrides the inRequestPreMarshal method as follows:

```
class TPS_Thread_Filter : public ThreadFilter
{
  int inRequestPreMarshal (CORBA::Request &req,
                           CORBA::Environment &env) {
    // Get the target of the request.
    CORBA::Object_ptr obj = req.target ();

    // Ensure it's a Quoter object (it could
    // be a Quoter_Factory).
    Quoter_ptr quoter =
      Quoter::_narrow (obj, env);

    if (env)
      // Must be the Quoter_Factory..
      // continue the work in the main thread by
      // telling Orbix to dispatch as normal.
      return 1;

    // Get the My_Quoter object.
    if (My_Quoter *my_quoter =
        dynamic_cast<My_Quoter *> DEREF (quoter)) {
      // Pass the request to the per-session thread.
      my_quoter->insert_at_tail (&req);
    else if (/* Check for Dow Jones */)
      // ...
    else if (/* Check for Reuters */)
      // ...
    else {
      // Not supported, suppress further
      // dispatching and raise an exception.
      env.exception (new Stock::Invalid_Quoter);
      return 0;
    }

    // If success, tell Orbix we'll dispatch
    // the request later...
    return -1;
  }
}
```

Our TPS_Thread_Filter acts only on incoming quote requests. When this filter is invoked, Orbix has already demultiplexed the incoming CORBA request to the implementation object identified by the CORBA::Request::target method. For each request, our filter first tries to obtain a reference to target object representing the client's session. We attempt to narrow this to a Quoter object reference. This example illustrates how to check for _narrow failures using Environment variables rather than C++ exceptions. If the narrow fails (*i.e.,* if env is "true") it means the request is targeted to another object (specifically, a Quoter_Factory). In this case, a 1 is returned to tell Orbix to continue dispatching the request normally in the main thread. This causes Orbix to invoke

[4] Orbix filters are an implementation of Shapiro's *Stub-Scion Pair* (SSP) Chains; see [8] for more details.

the `create_quoter` upcall on the `My_Quoter_Factory` implementation.

If the narrow succeeds, we use C++ RTTI to determine the actual type of the `quoter` object. If it's a `My_Quoter`, the request is inserted at the end of the per-session queue for the target `Quoter` active object. The selected active object will subsequently remove the request from its queue and perform the appropriate session processing, as shown in Figure 4.

We've omitted the code for the Dow Jones and Reuter's implementations, which would be similar to `My_Quoter`. Note that if `quoter` doesn't match any of the alternatives we'll raise an `Invalid_Quoter` exception and return 0, which tells Orbix not to continue dispatching the operation. Otherwise, if we find a match -1 is returned, which tells Orbix not to continue dispatching the request since it will be handled in the specified session thread.

5.3.5 The main() Function

The main server program implements the thread-per-session concurrency model as follows:

```
int main (int argc, char *argv[])
{
  // Initialize the factory implementation.
  My_Quoter_Factory_var quoter_factory =
    new TIE_My_Quoter_Factory (My_Quoter_Factory)
                         (new My_Quoter_Factory);

  // Wait for work to do in the main thread
  // (which is also the thread that shepherds
  // CORBA requests through TPS_Thread_Filter).
  try {
     CORBA::Orbix.impl_is_ready ("Quoter_Factory");
  } catch (...) {
     // ...
  }

  return 0;
}
```

When the `Quote` server first starts up, it creates a `My_Quoter_Factory` object to service client session initiation requests. Then, the main server thread calls `Orbix.impl_is_ready` to notify Orbix that the `Quoter_Factory` implementation is ready to service requests. The main thread is responsible for shepherding CORBA requests through the filter chain to the `TPS_Thread_Filter`. The filter then demultiplexes the requests to the appropriate `session_thread` active object, which runs them to completion.

5.4 Evaluating the MT-Orbix Solution

As we've seen in previous columns, the effort required to transform the CORBA solution from the original thread-per-request server to the thread-per-session concurrency model was relatively minor, even though we also changed from using RPC-style communication to Object-style. This change added a `Quoter_Factory`, which supports the creation of customized `Quoter` objects. By having the server export `Quoters` created by `Quoter_Factories`, servers can transparently create different custom alternatives and pass them back to clients.

There are some drawbacks to implementing the thread-per-session concurrency model with CORBA, however. Some of these drawbacks are related to MT-Orbix, whereas others are more subtle issues related to programming with CORBA.

- **Performance:** One potential drawback to the MT-Orbix solution is its use of the `Message_Queue` to buffer CORBA requests to session threads. This is a consequence of the use of thread filters in MT-Orbix. Thread filters are a very powerful way of decoupling the concurrency model used by the server from the Object Adapter and the ORB itself, which enables MT-Orbix to support multiple concurrency model in a convenient, uniform manner. By using the Orbix thread filter, the CORBA solution required only a few changes to the thread-per-request code.

However, the MT-Orbix thread filter architecture can cause additional overhead due to the extra context switching and synchronization necessary to queue requests on the thread-safe `Message_Queue`. Other ORBs that support the thread-per-session model, such as ORBeline, don't have this particular restriction, though they typically don't support as many concurrency models either. Our future columns will address other topics related to the performance of alternative multi-threaded ORB designs.

- **Violating thread-per-session semantics:** Our MT-Orbix solution assumes the client that creates the `Quoter` object is the only one who uses the object and is the one who destroys it. However, when using CORBA, a common practice is to have the object reference obtained from a factory be made available to multiple applications. For example, an application may invoke a factory operation to create a COSS Event Channel and advertise it in the COSS Naming Service [5]. Other applications can then obtain the event channel's object reference from the Naming Service and attach themselves to it as producers or consumers of events.

The solution we showed above will not be a thread-per-session model if the client passes off the `My_Quoter` object reference obtained from the factory to other applications. In this case, multiple clients can have their requests serviced by the same thread. If this occurs, the solution becomes a *thread-per-object* solution. As its name implies, the thread-per-object approach causes all requests for a specific object to be handled on a single thread dedicated only to that object. Both thread-per-session and thread-per-object allow only one of the object's operations to be active at any time.[5] In contrast, the thread-per-request and thread-pool models allow several of an object's operations to be invoked simultaneously on multiple threads.

In this column, we've used thread-per-object to implement thread-per-session by following a convention that assumes only a single client uses each `Quoter`'s object reference. To really implement thread-per-session in CORBA, the ORB

[5] Incidentally, this is the concurrency model supported by Network OLE, where it is called the "apartment" model of threading.

would have to maintain a separate thread for each client connection. Our stock quoter application can't do it because MT-Orbix does not expose the association between the client's connection and the request is not available to our thread filter. This isn't necessarily a drawback, however. If the ORB were to allow access to such information, it might prevent itself from implementing intelligent connection management (*e.g.*, reusing connections in a least-recently-used fashion to avoid running out of file descriptors).

• **Managing Object References:** The `create_factory` method in `My_Quoter_Factory` in Section 5.3.2 contained a call to increment the dynamically allocated `Quoter`'s object reference count before returning it from the function. Forgetting to duplicate object references before passing them as operation results is a very common mistake with beginning CORBA programmers. The OMG C++ Mapping Specification requires that the client of an operation returning an object reference to assume ownership of that object reference and `release` it when it has finished using it. However, this gets a little tricky when the client and object are located on two different machines. In that case, the ORB must marshal the object reference into a form suitable for network transmission in order to return it to the client.

To maintain local/remote transparency, the server-side ORB must call `release` after marshaling the object reference and sending it back to the client. Likewise, the client-side ORB must receive the returned object reference and unmarshal it into a object reference variable that can later be passed to `release` by the client. If the object's method does not first `_duplicate` the object reference before returning it, the newly-created object will be destroyed when the server-side ORB calls `release`, thus leaving the client with a "dangling object reference," which refers to an object that has been destroyed.

Much of this discussion is specific to Orbix, due to the fact that Orbix skeleton classes derive from `CORBA::Object`. In other ORBs, such as HP ORB Plus, skeletons are kept separate from the `CORBA::Object` inheritance hierarchy. Therefore, calling `release` on object references does not result in the destruction of the C++ object that implements the CORBA object being referenced. In fact, Orbix provides a CORBA extension called `propagateTIEdelete` that enables or disables propagation of delete calls on TIEs through to the implementation object.

It's hard to remember when to duplicate and release object references and when to delete data received as the result of an operation. That's why the OMG IDL C++ mapping provides the `_var` data types. These are similar in function to the ANSI C++ `auto_ptr` type since they are destroyed they automatically free the resources they manage. Storing an object reference returned from an operation (such as the factory `create_quoter` operation) into a stack-allocated `Quoter_var` relieves us from having to call `CORBA::release` on that object reference when we're finished with it. As our CORBA C++ examples from the past few columns have shown, using `_var` types can significantly ease resource management issues associated with CORBA programming.

• **Lack of portability for concurrent servers:** The client-side interfaces we showed in Section 5.2 use standard CORBA features and are implemented using the standard OMG C++ language mapping. This is in contrast to the concurrent CORBA server, which suffers from several portability problems in the current CORBA specification. These problems include (1) lack of a suitable Basic Object Adapter, (2) lack of a well-specified means to map generated IDL skeletons with IDL interface class implementations, and (3) lack of a portable concurrency model. Our previous columns explored these issues and their potential resolutions in more detail.

6 Concluding Remarks

In this column, we examined thread-per-session concurrency model and illustrated how to use it to develop multi-threaded servers for a distributed stock quote application. These examples illustrated how object-oriented techniques, C++, and higher-level abstractions help to simplify programming and improve extensibility. Our goal is to help you navigate through the design space of alternative concurrency models.

Using object-oriented design techniques and C++ programming features can help to abstract from low-level details in order to make different models easier to use. As we showed in this column, useful abstractions for the thread-per-session concurrency model include *thread filters*, *request queues*, *reactive dispatchers*, *acceptors*, *handlers*, and *session threads*.

As always, if there are any topics that you'd like us to cover, please send us email at `object_connect@ch.hp.com`.

References

[1] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[2] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] D. C. Schmidt, "Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns," *C++ Report*, vol. 7, November/December 1995.

[4] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

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

[6] 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.

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

[8] M. Shapiro, “Flexible Bindings for Fine-Grain, Distributed Objects,” Tech. Rep. Rapport de recherche INRIA 2007, INRIA, Aug. 1993.