Object Interconnections

Comparing Alternative Programming Techniques for Multi-threaded Servers (Column 5)

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 February 1996 issue of the SIGS C++ Report magazine.

1 Introduction

This column examines and evaluates several techniques for developing multi-threaded servers. The server we're examining mediates access to a stock quote database. Desktop client applications operated by investment brokers interact with our server to query stock prices. As with our previous columns, we'll compare several ways to program multi-threaded quote servers using C, C++ wrappers, and CORBA.

1.1 Background

A *process* is a collection of resources that enable a program to execute. In modern operating systems like Windows NT, UNIX, and OS/2, process resources include virtual memory, handles to I/O devices, a run-time stack, and access control information. On earlier-generation operating systems (such as BSD UNIX and Windows), processes were single-threaded. However, many applications (particularly networking servers) are hard to develop using single-threaded processes. For example, the single-threaded, iterative stock quote server we presented in our last column cannot block for extended periods of time handling one client request since the quality of service for other clients would suffer. The following are several common ways to avoid blocking in single-threaded servers:

• **Reactive event dispatchers:** one approach is to develop an event dispatcher (such as the object-oriented Reactor framework described in [1]). Reactive dispatching is commonly used to manage multiple input devices in single-threaded user-interface frameworks. In these frameworks, the main event dispatcher detects an incoming event, demultiplexes the event to the appropriate handler object, and dispatches an application-specific callback method to handle the event.

The primary drawback with this approach is that long duration operations (such as transferring a large file or performing a complex database query) must be developed using non-blocking I/O and explicit finite state machines. This approach becomes unwieldy as the number of states increase. In addition, only non-blocking operations are used. This makes it hard to improve performance via techniques such as "I/O streaming" or schemes that exploit locality of reference in data and instruction caches on multi-processors.

• **Cooperative tasking:** another approach is to use a cooperative task library. A process can have multiple tasks, each containing a separate run-time stack, instruction pointer, and registers. Therefore, each task is a separate unit of execution, which executes within the context of a process. Cooperative tasking is non-preemptive, which means task context information will only be stored and retrieved at certain preemption points.[1] This enables the library to suspend a task's execution until another task resumes it. The multi-tasking mechanisms on Windows 3.1, Mac System 7 OS, and the original task library bundled with cfront are examples of cooperative tasking.

Cooperative task libraries can be hard to program correctly since developers must modify their programming style to avoid certain OS features (such as asynchronous signals). Another limitation with cooperative tasking is that the OS will block all tasks in a process whenever one task incurs a page fault. Likewise, the failure of a single task (*e.g.,* mistakenly spinning in an infinite loop) will hang the entire process.

• **Multi-processing:** another way to alleviate the complexity of single-threaded processes is to use coarse-grained multi-processing capabilities provided by system calls like `fork` on UNIX and `CreateProcess` on Windows NT. These calls create a separate child process that executes a task concurrently with its parent. Separate processes can collaborate directly (by using mechanisms such as shared memory and memory-mapped files) or indirectly (by using pipes or sockets).

However, the overhead and inflexibility of creating and using processes may be prohibitively expensive and overly complicated for many applications. For example, process creation overhead can be excessive for short-duration services (such as resolving the Ethernet number of an IP address,

[1] Preemption points commonly occur when a task acquires or releases a lock, invokes an I/O operation, or explicitly "yields".

retrieving a disk block from a network file server, or setting an attribute in an SNMP MIB). Moreover, it may not be possible to exert fine-grain control over the scheduling behavior and priority of processes. In addition, processes that share C++ objects in shared memory segments must make non-portable assumptions about the placement of virtual table pointers [2].

• **Preemptive multi-threading:** When used correctly, preemptive multi-threading provides a more elegant, and potentially more efficient, means to overcome the limitations with the other concurrent processing techniques described above. A thread is a single sequence of execution steps performed in the context of a process. In addition to its own instruction pointer, a thread contains other resources such as a run-time stack of function activation records, a set of general-purpose registers, and thread-specific data. A preemptive multi-threading operating system (such as Solaris 2.x [3] and Windows NT [4]) or library (such as the POSIX pthreads library [5] available with DCE) uses a clock-driven scheduler to ensure that each thread of control executes for a particular period of time. When a thread's time period has elapsed it is preempted to allow other threads to run.

Conventional operating systems (such as variants of UNIX, Windows NT, and OS/2) support the concurrent execution of multiple processes, each containing one or more threads. A process serves as the unit of protection and resource allocation within a separate hardware protected address space. A thread serves as the unit of execution that runs within a process address space that is shared with zero or more threads. The remainder of this column focuses on techniques for programming preemptive multi-threaded servers.

1.2 Multi-threaded Server Programming

Multi-threaded servers are designed to handle multiple client requests simultaneously. The following are common motivations for multi-threading a server:

• **Simplify program design:** by allowing multiple server tasks to proceed independently using conventional programming abstractions (such as synchronous CORBA remote method requests and replies);

• **Improve throughput performance:** by using the parallel processing capabilities of multi-processor hardware platforms and overlapping computation with communication;

• **Improve perceived response time:** for interactive client applications (such as user interfaces or network management tools) by associating separate threads with different server tasks so clients don't block for long.

There are a number of different models for designing concurrent servers. The following outlines several concurrency models programmers can choose from when multi-threading their servers:

• **Thread-per-request:** this model handles each request from a client in a separate thread of control. This model is useful for servers that handle long-duration requests (such as database queries) from multiple clients. It is less useful for short-duration requests due to the overhead of creating a new thread for each request. It can also consume a large number of OS resources if many clients make requests simultaneously.

• **Thread-per-session:** this model is a variation of thread-per-request that amortizes the cost of spawning the thread across multiple requests. This model handles each client that connects with a server in a separate thread for the duration of the session. It is useful for servers that carry on long-duration conversations with multiple clients. It is not useful for clients that make only a single request since this is essentially a thread-per-request model.

• **Thread pool:** this model is another variation of thread-per-request that also amortizes thread creation costs by prespawning a pool of threads. It is useful for servers that want to bound the number of OS resources they consume. Client requests can be executed concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests must be queued until a thread becomes available.

• **Thread-per-object:** this model associates a thread for each logical object (*i.e.,* service) in the server. It is useful when programmers want to minimize the amount of rework required to multi-thread an existing server. It is less useful if certain objects receive considerably more requests than others since they will become a performance bottleneck.

In general, multi-threaded servers require more sophisticated synchronization strategies than single-threaded servers. To illustrate how to alleviate unnecessary complexity, we present and evaluate a number of strategies and tactics necessary to build robust and efficient thread-per-request servers. We first examine a simple solution using C and Solaris threads [3]. We then describe how using C++ wrappers for threads helps reduce the complexity and improves the portability and robustness of the C solution. Finally, we present a solution that illustrates the thread-per-request concurrency model implemented using two multi-threaded versions of CORBA (HP ORB Plus and MT-Orbix [6]). Our next column will show examples of the other concurrency models.

A word of caution: the multi-threading techniques we discuss in this column aren't standardized throughout the industry. Therefore, some of the code we show is not directly reusable across all OS platforms. However, the key concurrency techniques and patterns we illustrate *are* reusable across different platforms.

2 The Multi-threaded C Server Solution

2.1 Socket/C Code

The following code illustrates how to program the server-side of our stock quote program using sockets, Solaris threads, and C. Our previous column presented a set of utility routines written in C used below to receive stock quote requests from clients (`recv_request`), lookup quote information (`lookup_stock_price`), and return the quote to the client (`send_response`).

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

/* These implementations were in our last column. */
HANDLE create_server_endpoint (u_short port);
int recv_request (HANDLE h, struct Quote_Request *req);
int send_response (HANDLE h, long value);
int handle_quote (HANDLE);
```

2.1.1 Spawning Threads

The `main` function shown below uses these C utility routines to create a concurrent quote server. This server uses a thread-per-request concurrency model. The main program creates a "passive-mode" listener socket (which accepts connections from clients) and then calls `svc_run` to perform the main quote server's event loop:

```
int main(int argc, char *argv[])
{
  u_short port /* Port to listen for connections. */
      = argc > 1 ? atoi(argv[1]) : 10000;

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

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

The `svc_run` function waits in an event loop for connection requests to arrive from clients. When a request arrives, the Solaris `thr_create` function spawns off a new thread to perform the `handle_quote` query in parallel with other client requests:[2]

```
void svc_run (HANDLE listener)
{
  for (;;) {
    HANDLE handle = accept(listener, 0, 0);

    /* Spawn off separate thread for each client. */
    thr_create
       (0, /* Use default thread stack */
        0, /* Use default thread stack size */
        /* Thread entry point */
        (void *(*)(void *)) &handle_quote,
        (void *) handle, /* Entry point arg */
        THR_DETACHED | THR_NEW_LWP, /* Flags */
        0); /* Don't bother returning thread id */
  }
```

Each thread that runs the `handle_quote` function blocks awaiting the connected client to send a stock request on the socket `handle`. The `handle_quote` function can block since it runs in its own thread of control. Once this function receives a client request it processes this request, returns the appropriate stock quote, and exits the thread.

[2] `THR_DETACHED` and `THR_NEW_LWP` are flags to the Solaris `thr_create` function that tell it (1) the `handle_quote` function will exit silently when it's complete and (2) a new concurrent execution context should be created, respectively.

2.1.2 Synchronizing Threads

The `handle_quote` function looks up stock prices in the quote database via a global variable and an accessor function:

```
extern Quote_Database *quote_db;
long lookup_stock_price(Quote_Database*,
                        Quote_Request*);
```

The single-threaded implementation of `handle_quote` from our last column did not contain any locks to explicitly synchronize access to this database. Synchronization was unnecessary in the single-threaded implementation since only one client could access the database in the server at any time. With our new thread-per-request server implementation this assumption no longer holds. If the database maintains its own internal locks our existing code may still work. In this example we'll assume the database does not maintain internal locks. This is often the case when integrating multi-threading with legacy libraries (such as the UNIX `dbm` database libraries and X windows). Many of these libraries were developed before multi-threading became popular, so they don't use internal locks.

The code shown below explicitly serializes all lookups and updates to the database.[3] Serialization prevents race conditions that would otherwise occur when multiple threads accessed and updated data simultaneously. The following implementation of `handle_quote` illustrates a simple way to serialize database access:

```
/* Define a synchronization object that
   is initially in the "unlocked" state. */
rwlock_t lock;

int handle_quote(HANDLE h)
{
  struct Quote_Request req;
  long value;

  if (recv_request(h, &req) == 0)
      return 0;

  /* Block until read lock is available */
  rw_rdlock(&lock);

  /* lookup stock in database */
  value = lookup_stock_price(quote_db, &req);

  /* Must release lock or deadlock will result! */
  rw_unlock(&lock);

  return send_response(h, value);
}
```

The `rwlock_t` is a Solaris synchronization variable used to protect the integrity of a shared resource (like the quote database) that is accessed concurrently by multiple threads of control. A `rwlock_t` implements a "readers/writer" lock that serializes thread execution by defining a critical section where multiple threads can read the data concurrently, but only one thread at a time can write to the data. The implementation of `rwlock_t` ensures that acquiring and releasing a lock is an atomic operation.

[3] The solution we've shown does not show how stock prices are updated in the database. This is beyond the scope of this column and will be discussed in a future column.

2.2 Evaluating the C Solution

Programming directly with C and Solaris threads as shown above yields a correct program. However, developing concurrent applications at this level of detail has several drawbacks:

• **Lack of portability:** The design and implementation of the multi-threaded version of the quote server differs considerably from the single-threaded version. It replaces the reactive `select`-driven event loop with a master dispatcher thread and a set of slave threads that perform database lookups. Unfortunately, the use of Solaris threads is not portable to other platforms (such as Windows NT, OS/2, or other versions of UNIX).

• **Lack of reusability:** The use of global variables (like the `lock` that protects the database from race conditions) introduces unnecessary dependencies between different parts of the code. These dependencies make it hard to reuse existing code [7].

• **Lack of robustness:** in a large program, instrumenting all the code with mutex locks can be tedious and error-prone. In particular, failing to release a mutex can lead to deadlock or resource failures.

The following section describes how we can use C++ wrappers to alleviate the problems described above.

3 The Multi-threaded C++ Wrappers Solution

Using C++ wrappers is one way to simplify the complexity of programming concurrent network servers. C++ wrappers encapsulate lower-level OS interfaces such as sockets, multi-threading, and synchronization with type-safe, object-oriented interfaces. The `IPC_SAP` [8], `Acceptor` [9], and `Thread` [10] C++ wrappers shown below are part of the ACE object-oriented network programming toolkit [11]. `IPC_SAP` encapsulates standard network programming interfaces (such as sockets and TLI); the `Acceptor` implements a reusable design pattern for passively[4] initializing network services, and the `Thread` wrapper encapsulates standard OS threading mechanisms (such as Solaris threads, POSIX pthreads, and Windows NT threads).

3.1 C++ Wrapper Code

This section illustrates how the use of C++ wrappers improves the reuse, portability, and extensibility of the quote server. Figure 1 depicts the following components in the quote server architecture:

[4]Communication software is typified by asymmetric connection behavior between clients and servers. In general, servers listen *passively* for clients to initiate connections *actively*.

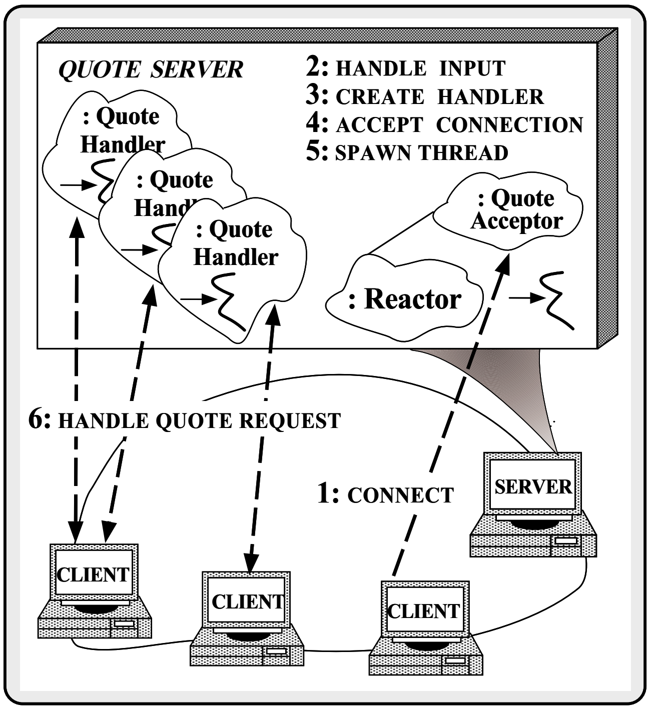

Figure 1: Thread-per-Request C++ Wrapper Architecture for the Stock Quote Server

Quote_Handler: this class interacts with clients by receiving stock quote requests, finding quote values in the database, and returning responses. Using the concurrency model in this example, each `Quote_Handler` runs in a separate thread of control.

Quote_Acceptor: this class is a factory [12] that implements the strategy for passively initializing a `Quote_Handler`. This involves assembling the resources necessary to create a new `Quote_Handler` object, accepting the connection into this object, and activating the `Quote_Handler` by calling its `open` method.

The architecture of the multi-threaded C++ solution is similar to the single-threaded C++ solution presented in our last column. The primary difference is that the `Quote_Handlers` and `Quote_Acceptor` all run in separate threads, rather than being driven by callbacks from the `Reactor` within a single thread. Figure 1 illustrates how the thread-per request concurrency model allows multiple requests from the same client to be processed simultaneously by different threads.

3.1.1 The Quote_Handler Class

We'll start by showing the `Quote_Handler`. This template class inherits from the reusable `Svc_Handler` base class in ACE. A `Svc_Handler` defines a generic interface for a communication service that exchanges data with peers over network connections:

```
template <class STREAM> // IPC interface
class Quote_Handler
  : public Svc_Handler<STREAM>
    // This ACE base class defines "STREAM peer_;"
{
public:
  Quote_Handler (Quote_Database &db,
                 RW_Mutex &lock)
    : db_ (db), lock_ (lock) {}

  // This method is called by the Quote_Acceptor
  // to initialize a newly connected Quote_Handler
  // (which spawns a new thread to handle client).
  virtual int open (void) {
    Thread::spawn
     // Thread entry point.
     (Quote_Handler<STREAM>::request_thread,
      (void *) this,             // Entry point arg.
     THR_DETACHED | THR_NEW_LWP); // Thread flags.
  }

  // Static thread entry point method.

  static void *request_thread (void *) {
    Quote_Handler<STREAM> *client =
      static_cast <Quote_Handler<STREAM> *> args;

    // Handle one request.
    int result = client->handle_quote ();

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

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

  // Handles the quote request/response.  This
  // can block since it runs in its own thread.
  virtual int handle_quote (void) {
    Quote_Request req;
    int value;

    if (recv_request (req) <= 0) return -1;
    else {
      // Constructor of m acquires lock_.
      Read_Guard<RW_Mutex> m (lock_);

      value = this_ptr->db_.lookup_stock_price (req);

      // Destructor of m releases lock_.
    }
    return send_response (value);
  }

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

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

private:
  Quote_Database &db_; // Reference to quote database.
  RW_Mutex &lock_; // Serialize access to database.
};
```

The C++ implementation of `handle_quote` ensures the `lock_` will be released regardless of whether the method throws an exception or returns normally. To ensure this behavior the following `Read_Guard` class is used:

```
template <class LOCK>
class Read_Guard
{
public:
  Read_Guard (LOCK &m): lock (m) {
    lock.acquire_read ();
  }
  ~Read_Guard (void) {
    lock.release ();
  }
private:
  LOCK &lock;
}
```

The `Read_Guard` class defines a block of code where a lock is acquired when the block is entered and released automatically when the block is exited. `Read_Guard` employs a C++ idiom (first described in [13]) that uses the constructor to acquire a resource automatically when an object of the class is created and uses the destructor to release the resource automatically when it goes out of scope. In this case, the resource is an `RW_Mutex`, which is a C++ wrapper for the Solaris `rwlock_t` readers/writer lock. Since the `LOCK` type of `Read_Guard` is parameterized, this class can be used with a family of synchronization wrappers that conform to the `acquire_read`/`release` interface.

In the stock quote server, the `Quote_Handler` template is instantiated with the `SOCK_Stream` wrapper for TCP stream sockets available in the `SOCK_SAP` class category from the ACE `IPC_SAP` class library:

```
typedef Quote_Handler <SOCK_Stream> QUOTE_HANDLER;
```

`SOCK_SAP` contains a set of C++ classes that shields applications from tedious and error-prone details of programming at the socket level [8].

3.1.2 The Quote_Acceptor Class

Next we'll show the `Quote_Acceptor` class. This class is a factory that implements the strategy for passively initializing a `Quote_Handler`. The `Quote_Acceptor` supplies concrete template arguments for the following implementation of the Acceptor pattern [9]:

```
template <class SVC_HANDLER, // Service handler
          class PEER_ACCEPTOR> // Passive conn. mech.
class Acceptor
{
public:
  // Initialize a passive-mode connection factory.
  Acceptor (const PEER_ACCEPTOR::ADDR &addr)
   : peer_acceptor_ (addr) {}

  // Implements the strategy to accept connections
  // from clients, and create and activate
  // SVC_HANDLERs to exchange data with peers.

  int handle_input (void) {
    // Create a new SVC_HANDLER.
    SVC_HANDLER *svc_handler = make_svc_handler ();

    // Accept connection into the SVC_HANDLER.
    peer_acceptor_.accept (*svc_handler);

    // Delegate control to the SVC_HANDLER.
    svc_handler->open ();
  }

  // Virtual Factory Method to make a SVC HANDLER.
  virtual SVC_HANDLER *make_svc_handler (void) = 0;

private:
```

```
  PEER_ACCEPTOR peer_acceptor_;
  // Factory that establishes connections passively.
};
```

The Quote_Acceptor subclass is defined by parameterizing the Acceptor template with concrete types that (1) accept connections (*e.g.,* SOCK_Acceptor or TLI_Acceptor) and (2) concurrently perform the quote service (Quote_Handler):

```
// Make a specialized version of the Acceptor
// factory to create QUOTE_HANDLERs that
// process quote requests from clients.
class Quote_Acceptor :
  public Acceptor <QUOTE_HANDLER, // Quote service.
                   SOCK_Acceptor> // Passive conn. mech.
{
public:
  typedef Acceptor <QUOTE_HANDLER, SOCK_Acceptor>
          inherited;

  Quote_Acceptor (const SOCK_Acceptor::ADDR &addr,
                  Quote_Database &db)
   : inherited (addr), db_ (db) {}

  // Factory method to create a service handler.
  // This method overrides the base class to
  // pass pointers to the Quote_Database and
  // the RW_Mutex lock.

  virtual QUOTE_HANDLER *make_svc_handler (void) {
    return new QUOTE_HANDLER (db_, lock_);
  }

private:
  Quote_Database &db_; // Reference to database
  RW_Mutex lock_; // Serialize access to database.
}
```

The main function uses the components defined above to implement the quote server:

```
int main (int argc, char *argv[])
{
  u_short port = argc > 1 ? atoi (argv[1]) : 10000;

  // Factory that produces Quote_Handlers.
  Quote_Acceptor acceptor (port, quote_db);

  // Single-threaded event loop that dispatches all
  // events in the Quote_Acceptor::handle_input()
  // method.

  for (;;)
    acceptor.handle_input ();

  /* NOTREACHED */
}
```

After the Quote_Acceptor factory has been created the application goes into an event loop. This loop runs continuously accepting client connections and creating Quote_Handlers. Each newly-created Quote_Handler spawns a separate thread in which it handles the client quote request and response.

3.2 Evaluating the C++ Wrappers Solution

Implementing the quote server with C++ wrappers is an improvement over the direct use of sockets, Solaris threads, and C for the following reasons:

• **Simplified programming and increased robustness:** Tedious and error prone low-level details of concurrent programming are encapsulated by Thread wrappers. For example, the Read_Guard idiom automatically acquires and releases mutex locks in critical sections.

• **Improved portability:** C++ wrappers shield applications from platform-specific details of multi-threaded programming interfaces. Encapsulating threads with C++ classes (rather than stand-alone C functions) improves application portability. For instance, the server no longer accesses Solaris thread functions directly. Therefore, the implementation shown above can be ported easily to other OS platforms without changing the Quote_Handler and Quote_Acceptor classes.

• **Increased reusability and extensibility of components:** The Quote_Acceptor and Quote_Handler components are not as tightly coupled as the C version shown in Section 2.1. This makes it easier to extend the C++ solution to include new services, as well as to enhance existing services. For example, to modify or extend the functionality of the quote server (*e.g.,* to add stock trading functionality), only the implementation of the Quote_Handler class must change. Likewise, the readers/writer lock that protects the Quote_Database is no longer a global variable. It's now localized within the scope of Quote_Handler and Quote_Acceptor.

Note that the use of C++ features like templates and inlining ensures that the improvements described above do not penalize performance.

The C++ wrapper solution is a significant improvement over the C solution for the reasons we mentioned above. However, it still has all the drawbacks we've discussed in previous columns such as not addressing higher-level communication topics like *object location, object activation, complex marshalling and demarshalling, security, availability and fault tolerance, transactions, and object migration and copying.* A distributed object computing (DOC) framework like CORBA or Network OLE is designed to address these issues. DOC frameworks allow application developers to focus on solving their domain problems, rather than worrying about network programming details. In the following section we describe several ways to implement thread-per-request servers using CORBA.

4 The Multi-threaded CORBA Solution

The CORBA 2.0 specification [14] does not prescribe a concurrency model. Therefore, a CORBA-conformant ORB need not provide multi-threading capabilities. However, commercially-available ORBs are increasingly providing support for multi-threading. The following section outlines several concurrency mechanisms available in two such ORBs: HP ORB Plus and MT-Orbix.

As in previous columns, the server-side CORBA implementation of our stock quote example is based on the following OMG-IDL specification:

```
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);
  };
};
```

In the following section we'll illustrate how a server programmer can implement multi-threaded versions of this OMG-IDL interface using HP ORB Plus and MT-Orbix.

4.1 Overview of Multi-threaded ORBs

A multithreaded ORB enables many simultaneous requests to be serviced by one or more CORBA object implementations. In addition, objects need not be concerned with the duration of each request. Without multiple threads, each request must execute quickly so that incoming requests aren't starved. Likewise, server applications must somehow ensure that long-duration requests do not block other requests from being serviced in a timely manner.

To understand multi-threaded CORBA implementations, let's first review how a conventional reactive implementation of CORBA is structured. Here's the main event loop for a typical single-threaded Orbix server:

```
int main (void)
{
  // ...
  // Create an object implementation.
  My_Quoter my_quoter(db);

  // Listen for requests and dispatch object methods.
  CORBA::Orbix.impl_is_ready("My_Quoter");
  // ...
}
```

Likewise, the main event loop for an equivalent single-threaded HP ORB Plus server might appear as follows:

```
int main (void)
{
  // ...
  // Create an object implementation.
  My_Quoter my_quoter(db);

  // Generate an object reference for quoter object.
  Stock::Quoter_var qvar = quoter._this();

  // Listen for requests and dispatch object methods.
  // (hpsoa stands for "HP Simplified Object Adapter").
  hpsoa->run();
  // ...
}
```

The `impl_is_ready` and `run` methods are public interfaces to the CORBA event loop. In a single-threaded ORB, these methods typically use the Reactor pattern [1] to wait for CORBA method requests to arrive from multiple clients. The processing of these requests within the server is driven by upcalls dispatched by `impl_is_ready` or `run`. These upcalls invoke the `get_quote` method of the `My_Quoter` object implementation supplied by the server programmer. The upcall borrows the thread of control from the ORB to execute `get_quote`. This makes serialization trivial since there can only be one upcall in progress at a time in a single-threaded ORB.

In a multi-threaded ORB like MT-Orbix or HP ORB Plus, however, there can be multiple threads of control executing CORBA upcalls concurrently. Although the `impl_is_ready` and `run` interfaces don't change, the internal behavior of the respective ORBs do change. In particular, they must perform additional locking of ORB internal data structures to prevent race conditions from corrupting the private state of their implementation.

4.1.1 Implementing Thread-per-Request in HP ORB Plus

An implementation of the `My_Quoter` class for HP ORB Plus is shown below:

```
// Implementation class for IDL interface.

class My_Quoter
    // Inherits from an automatically-generated
    // CORBA skeleton class.
  : virtual public HPSOA_Stock::Quoter
{
public:
  My_Quoter (Quote_Database &db): db_ (db) {}

  // Callback invoked by the CORBA skeleton.
  virtual long get_quote (const char *stock_name,
                          CORBA::Environment &ev) {
    // assume no exceptions
    ev.clear();
    long value;
    {
      // Constructor of m acquires lock_.
      MSD_Lock m(lock_);

      value = db_.lookup_stock_price (stock_name);
      // Destructor of m releases the lock_.
    }
    if (value == -1)
      ev.exception(new Stock::Invalid_Stock);
    return value;
  }

private:
  Quote_Database &db_; // Reference to quote database.
  MSD_Mutex lock_; // Serialize access to database.
};
```

`My_Quoter` is our object implementation class. It inherits from the `HPSOA_Stock::Quoter` skeleton class. This class is generated automatically from the original IDL `Quoter` specification. The `Quoter` interface supports a single operation: `get_quote`. Our implementation of `get_quote` relies on an external database object that maintains the current stock price. If the lookup of the desired stock price is successful the value of the stock is returned to the caller. If the stock is not found, the database `lookup_stock_price` function returns a value of -1. This value triggers our implementation to return a `Stock::Invalid_Stock` exception.

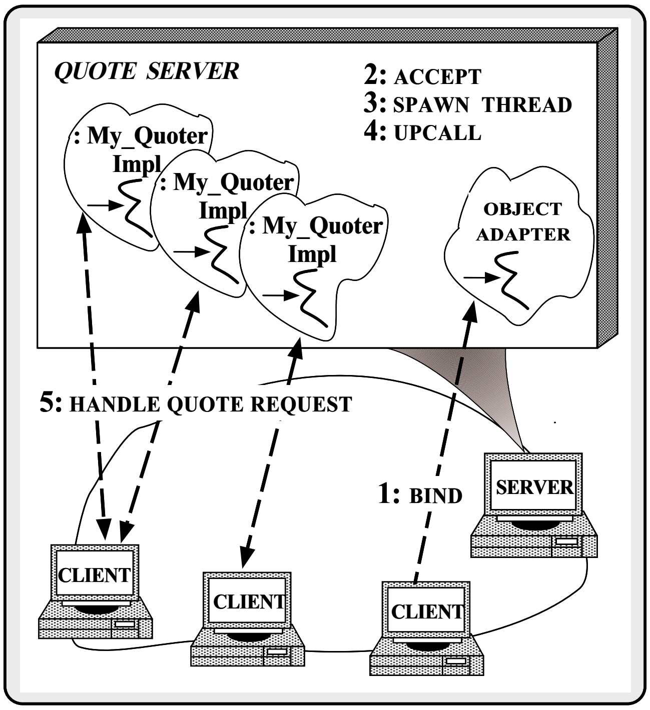

Figure 2: HP ORB Plus Architecture for the Thread-per-Request Stock Quote Server

Since this code is multi-threaded, the object implementation must acquire a lock before accessing the state of the `db_` object, just like the C and C++ solutions presented earlier. The code uses the *MSD Threads Abstraction Library*[5] provided by HP ORB Plus. The `MSD_Lock` class is similar to the `Read_Guard` shown in Section 3.1. By using the `MSD_Lock`, we ensure that the lock is released, even if an exception is thrown.

By default, the ORB Plus *HP Simplified Object Adapter* (HPSOA) provides a thread-per-request concurrency model. It spawns a separate thread for each incoming request if a server is configured to use multi-threading. Therefore, applications need not explicitly create threads unless they require a greater degree of control than that provided by the HPSOA. Note, however, that the `main` function need not change at all, regardless of whether a single-threaded or multi-threaded configuration is used.

Figure 2 illustrates the HP ORB Plus architecture for the thread-per-request stock quote server. The HP Simplified Object Adapter (`HPSOA`) and the HP ORB are responsible for spawning a new thread for each incoming request and dispatching the `My_Quoter::get_quote` implementation to execute each stock request in a separate thread.

[5] *MSD* stands for "Measurement Systems Department," the HP laboratory where the threads abstraction library was developed.

4.1.2 Implementing Thread-per-Request in MT-Orbix

The `My_Quoter` implementation class shown below illustrates how the thread-per-request concurrency model can be implemented in MT-Orbix:

```
// Implementation class for IDL interface.

class My_Quoter
    // Inherits from an automatically-generated
    // CORBA skeleton class.
  : virtual public Stock::QuoterBOAImpl
{
public:
  My_Quoter (Quote_Database &db): db_ (db) {}

  // Callback invoked by the CORBA skeleton.
  virtual long get_quote (const char *stock_name,
                          CORBA::Environment &ev) {
    // Constructor of m acquires lock.
    Read_Guard<RW_Mutex> m (lock_);

    long value =
      db_.lookup_stock_price (stock_name);

    if (value == -1)
      ev.exception(new Stock::Invalid_Stock);
    return value;

    // Destructor of m releases lock.
  }

private:
  Quote_Database &db_; // Reference to quote database.
  RW_Mutex lock_; // Serialize access to database.
};
```

This version of the `My_Quoter` object implementation class is similar to the one shown for HP ORB Plus in Section 4.1.1. One minor difference is that the `My_Quoter` class inherits from a different skeleton class: `Stock::QuoterBOAImpl`. This class is generated automatically from the original IDL `Quoter` specification, just like the HP ORB Plus `HPSOA_Stock::Quoter` base class,

The main program for the MT-Orbix quote server is identical to the one single-threaded version shown in Section 4.1. The only extra C++ code we have to write is called a `ThreadFilter`. Each incoming CORBA request is passed through the chain of filters before being dispatched to its target object implementation.

Filters are an MT-Orbix extension to CORBA that implement the "Chain of Responsibility" pattern [12]. Orbix uses this pattern to decouple (1) the demultiplexing of CORBA requests (*e.g.,* generated by the client-side `get_quote` proxy) to their associated target object (*e.g.,* `my_quoter`) from (2) the eventual dispatching of the upcall method implementation (*e.g.,* `My_Quoter::get_quote`). This decoupling enables applications to transparently extend the behavior of Orbix without modifying the ORB itself.

Figure 3 illustrates the role of the `ThreadFilter` in the MT Orbix architecture for the thread-per-request stock quote server. Note how the `TPR_ThreadFilter` is responsible for spawning a thread that dispatches the `get_quote` upcall. Thus, the ORB and MT Orbix Object Adapter are unaffected by this threading architecture.

To enable a new thread to be spawned for an incoming request, a subclass of `ThreadFilter` must be defined to override the `inRequestPreMarshal` method, as follows:

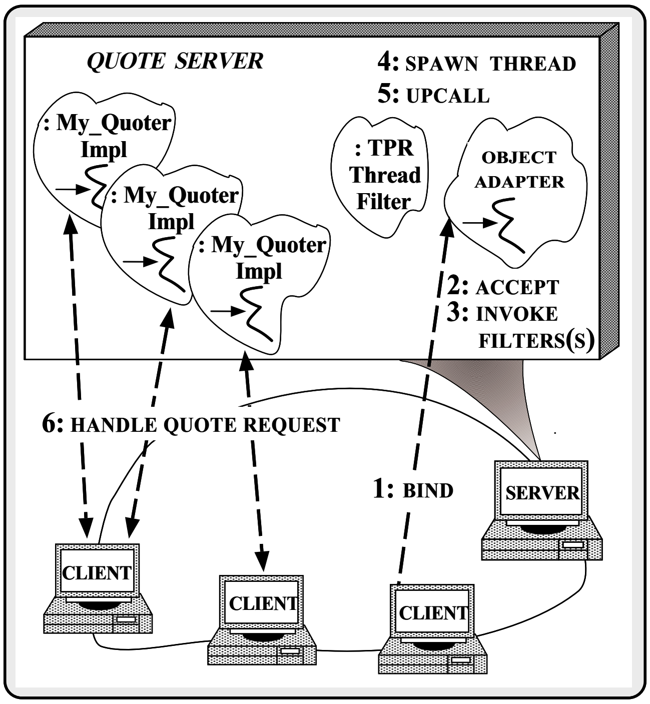

Figure 3: MT Orbix Architecture for the Thread-per-Request Stock Quote Server

```
// Create a filter that spawns a "thread-per-request"
// to dispatch object implementation upcalls.
class TPR_ThreadFilter : public CORBA::ThreadFilter
{
  // Intercept request and spawn thread
  virtual int inRequestPreMarshal (CORBA::Request &,
                                   CORBA::Environment&);
  // ...
};
```

Orbix will call `inRequestPreMarshal` method before the incoming request is processed. This method must return -1 to indicate that it has spawned a thread to deal with the request. Threads can be spawned according to whatever concurrency model is appropriate for the application. In this example, we're using the thread-per-request model. Therefore, the implementation of `inRequestPreMarshal` could be written as follows:

```
// Implementation of inRequestPreMarshal

int
TPR_ThreadFilter::inRequestPreMarshal
  (CORBA::Request &req, // Incoming CORBA Request.
   CORBA::Environment&)
{
  Thread::spawn
    (continueDispatching, // Entry point.
     (void *) &req, // Entry point arg.
     THR_DETACHED | THR_NEW_LWP); // Thread flags.

  // Tell Orbix we will dispatch request later.
  return -1;
}
```

The `continueDispatching` function is the entry point where the new thread begins executing:

```
void *continueDispatching (void *vp)
{
  CORBA::Request *req = (CORBA::Request *) vp;
  CORBA::Orbix.continueThreadDispatch (*req);
  return 0;
}
```

The MT-Orbix method `continueThreadDispatch` will continue processing the request until it sends a reply to the client. At this point, the thread will exit.

Our quote server must explicitly create an instance of `TPR_ThreadFilter` to get it installed into the filter chain:

```
TPR_ThreadFilter tpr_filter;
```

The constructor of this object automatically inserts the filter at the end of the filter chain.

4.2 Evaluating the CORBA Solutions

The multi-threaded CORBA solutions presented above are similar to the C++ wrapper solution shown in Section 3. All the solutions contain a master thread that is responsible for accepting connections from clients. The master thread then explicitly or implicitly spawns slave threads to execute client requests concurrently. All the solutions require application code to explicitly lock any data structures (such as the quote database) shared with other threads. This design provides developers with greater control over application concurrency, at the expense of additional programming effort.

As we pointed out in our previous column, server-side portability is currently a problem area for CORBA. In particular, the MT-Orbix and HP ORB Plus examples illustrate the following trouble spots:

- **Non-standard Object Adapter Mappings:** In the MT-Orbix code, the skeleton base class for the `My_Quoter` object implementation class is named `Stock::QuoterBOAImpl`, while in the HP ORB Plus code the skeleton base class is named `HPSOA_Stock::Quoter`. This is because Orbix provides an implementation of the CORBA *Basic Object Adapter* (BOA), while HP ORB Plus provides another object adapter called the *HP Simplified Object Adapter* (HPSOA). The differences in object adapters between these products is due to portability problems with the BOA that are currently being addressed by the OMG ORB Task Force.

- **Non-Standard Concurrency Models:** Another difference between the MT-Orbix and HP ORB Plus Quoter implementations involves threading. Both HP ORB Plus and MT-Orbix allow the choice of whether an application is multi-threaded or single-threaded to occur at link time.

In HP ORB Plus, linking an application against the *null threads library* makes it single-threaded, while linking against a *multiple threads library* makes it multi-threaded. If a server application is linked against the HP ORB Plus multiple threads library, each request is handled using the thread-per-request concurrency model. MT-Orbix also support this model, but contains hooks that allow programmers to implement other concurrency models. The choice of concurrency model in MT-Orbix depends on the type of `ThreadFilter` configured into the filter chain.

Despite their differences, the HP ORB Plus and MT-Orbix examples show that programming concurrent CORBA applications is straightforward.

• **Non-Standard Synchronization Mechanisms:** HP ORB Plus provides the MSD Threads Abstraction Library for threads portability. The MSD library implements a common interface for the null threads and multiple threads libraries. In addition, this library shields applications from incompatibilities across HP-UX, Solaris, and Windows NT thread libraries. MT-Orbix applications, in contrast, use whatever threads package is provided by the underlying platform. This flexibility can be both an advantage and a disadvantage. Although it may require programmers to develop or reuse C++ threads wrappers, it allows servers to take advantage of more efficient mechanisms provided by an OS or threads package. For instance, the MT-Orbix implementation in Section 4.1.2 uses readers/writer locks. Often, these can be more efficient than the regular mutexes provided by the HP ORB Plus MSD library shown in Section 4.1.1.

Since CORBA makes no mention of threads, it remains to be seen whether the OMG ORB Task Force will take it upon itself to address this issue as a CORBA portability problem. Clearly, in the short term we could use techniques like the Adapter pattern [12] and reusable C++ toolkits like ACE to make our object implementations relatively portable across different OS platforms and different ORBs.

• **Non-C++ Exception Handling:** Both the HP ORB Plus and MT-Orbix implementations currently use `Environment` parameters to convey exception information rather than C++ exceptions. Both ORBs will support C++ exceptions in the near future. When exception handling is supported the `get_quote` implementation would look like this:

```
virtual long get_quote (const char *stock_name,
                        CORBA::Environment& ev) {
  // ...
  // Constructor of m acquires lock_.
  Read_Guard<RW_Mutex> m (lock_);

  if (value == -1)
    throw Stock::Invalid_Stock();
  // ...
```

Coding defensively with idioms like the `Read_Guard` is essential to protect programs from hazardous side-effects of C++ exceptions [15].

5 Evaluating the Thread-per-Request Concurrency Model

All the servers shown above were designed using a thread-per-request concurrency model. This is a fairly straightforward model to design and implement. However, it is probably the wrong concurrency model for the task of retrieving stock quotes. There are two primary problems:

• **Thread creation overhead:** The time required to lookup a stock quote may be low relative to the time required to create the thread. In addition, even if the thread ran for a longer amount of time, the performance of the thread-per-request may not scale. For example, it may lead to unacceptably high resource utilization if there are hundreds or thousands of simultaneously active clients.

• **Connection creation overhead:** The thread-per-request model sets up a new connection for each request. Therefore, the overhead of establishing the connection is not amortized if clients send multiple requests to the server. The single-threaded solution we showed in our previous column kept the connection open until it was explicitly shut down by the client. Although our new solution might not affect how the client was programmed, the difference in connection strategies would likely show up in performance measurements.

The actual performance of a particular concurrency model depends to a large extent on the following factors:

• **The types of requests received from clients:** *e.g.,* short vs. long duration;

• **How threads are implemented:** *e.g.,* in the OS kernel, in a user-space library, or some combination of both;

• **Operating system and networking overhead:** *e.g.,* how much other overhead results from setting up and tearing down connections repeatedly;

• **Higher-level system configuration factors:** such as whether replication and/or dynamic load balancing are used, also ultimately affect performance.

We'll discuss these performance issues in future columns.

Another drawback with our solution is that the `handle_quote` function above serializes access to the `Quote_Database` at a very coarse-grained level, *i.e.,* at the database level. The scope of the mutex ensures that the whole database is locked. This is fine if most operations are lookups and a readers/writer lock is used. However, it may lead to performance bottlenecks if stock prices are frequently updated, or if regular mutexes must be used. A more efficient solution would push the locking into the database itself, where record or table locking could be performed.

One important conclusion from this evaluation is the importance of distinguishing between concurrency *tactics* (such as threading and synchronization mechanisms provided by an OS threads library) and concurrency *strategies* (such as thread-per-request, thread-per-session, thread-per-object, etc.). Threading libraries provide low-level mechanisms for creating different concurrency models. However, developers are ultimately responsible for knowing how to use these mechanisms successfully. Design patterns are a particularly effective way to help application developers master subtle differences between different strategies and firmly understand the applicability and consequences of different concurrency models. We'll explicitly cover key patterns for concurrent distributed object computing in future articles.

6 Concluding Remarks

In this column, we examined several different programming techniques for developing multi-threaded servers for a distributed stock quote application. Our examples illustrated how object-oriented techniques, C++, and higher-level abstractions help to simplify programming and improve extensibility.

Programming distributed applications without multiple threads is hard, especially for server applications. Without multi-threading capabilities, the server developer must either ensure that requests can be handled so quickly that new requests aren't "starved" or they must use heavyweight mechanisms like `fork` or `CreateProcess` to create a new process to service each request. In practice, though, most non-trivial requests can't be serviced quickly enough to avoid starving clients. Likewise, creating entire new processes to service requests is time consuming, requires too many system resources, and can be hard to program.

With multiple threads, each request can be serviced in its own thread, independent of other requests. This way, clients aren't starved by waiting for their requests to be serviced. Likewise, system resources are conserved since creating a thread is often less expensive than creating a whole new process.

In general, multithreaded systems can be difficult to develop due to subtle synchronization issues. Moreover, not all platforms provide good support for threads or thread-aware debuggers yet. Often, however, the benefits of threads outweigh the disadvantages. When used properly, multithreaded programming can enable simpler designs and implementations than single-threaded programming. Much of this simplicity derives from the fact that scheduling issues are handled by the threads package, not the application.

The benefits of CORBA and C++ become more evident when we extend the quote server to support different concurrency models. In particular, the effort required to transform the CORBA solution from the existing thread-per-request server to other forms of concurrency models is minimal. The exact details will vary depending on the ORB implementation and the desired concurrency strategy. However, multi-threaded versions of CORBA typically require only a few extra lines of code. Our next column will illustrate how to implement the other concurrency models (such as thread-per-session and thread-pool). Future columns will address other topics related to multi-threaded ORBs, such as the performance impacts of using different concurrency models. 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 Andy Gokhale, Prashant Jain, and Ron Resnick for comments on this column.

References

[1] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[2] A. Banerji and D. L. Cohn, "Shared Objects and Vtbl Placement for C++ ," *Journal of C Language and Translation*, vol. 6, pp. 44–60, Sept. 1994.

[3] 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.

[4] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[5] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

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

[7] R. C. Martin, "The Open-Closed Principle," *C++ Report*, vol. 8, Jan. 1996.

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

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

[10] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

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

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

[13] Bjarne Stroustrup, *The C++ Programming Language,* 2^{nd} *Edition*. Addison-Wesley, 1991.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[15] H. Mueller, "Patterns for Handling Exception Handling Successfully," *C++ Report*, vol. 8, Jan. 1996.