Object Interconnections

Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool (Column 6)

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

1 Introduction

Modern OS platforms like Windows NT, and OS/2 and many flavors of UNIX provide extensive library and system call support for multi-threaded applications. However, programming multi-threaded applications is hard and programming distributed multi-threaded applications is even harder. In particular, developers must address sources of *accidental* and *inherent* complexity:

• **Accidental complexity** of multi-threaded programming arises from limitations with programming tools and design techniques. For example, many debuggers can't handle threaded programs and can't step across host boundaries. Likewise, algorithmic design [1] makes it hard to reuse application components because it tightly couples the structure of a threaded application to the functions it performs.

• **Inherent complexity** of multi-threaded programming arises from challenges such as avoiding deadlock and livelock, eliminating race conditions for shared objects, and minimizing the overhead of context switch, synchronization, and data movement. An inherently complex aspect of programming multi-threaded distributed applications (particularly servers) involves selecting the appropriate concurrency model, which is the focus of this column.

Our previous column examined several ways to program multi-threaded stock quote servers using C, C++ wrappers, and two versions of CORBA (HP ORB Plus and MT Orbix). In that column, we focused on the *thread-per-request* concurrency model, where every incoming request causes a new thread to be spawned to process it. This column examines and evaluates another concurrency model: *thread pool*, which pre-spawns a fixed number of threads at start-up to service all incoming requests. We illustrate this model by developing new multi-threaded C, C++, and CORBA implementations of the stock quote server.

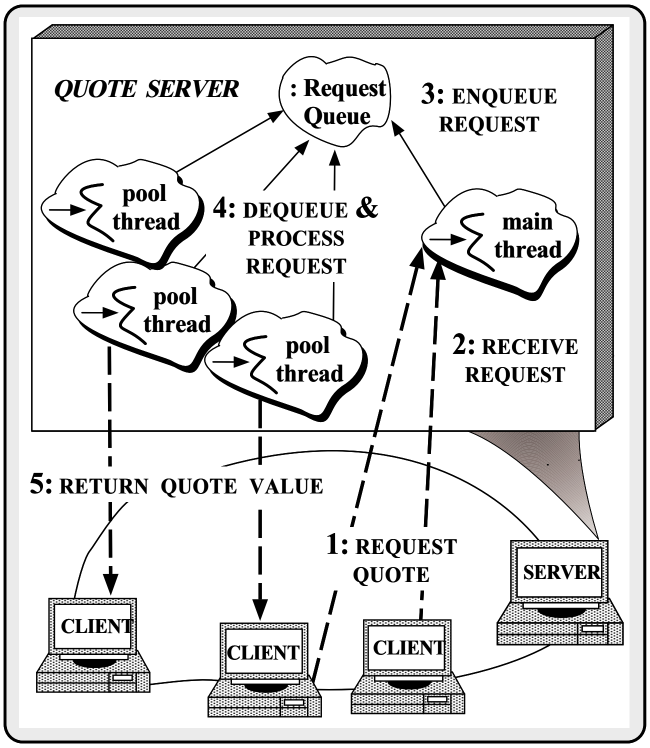

Figure 1: thread pool Architecture for the Stock Quote Server

2 The Thread Pool Concurrency Model

The thread pool concurrency model is a variation of the thread-per-request we examined last column. The main advantage of thread-per-request is its simplicity, which is why it's used in many multi-threaded ORBs (such as Orbix and HP ORB Plus). However, dynamically spawning a thread to handle each new request causes excessive resource utilization if the number of requests becomes very large and the OS resources required to support threads don't scale up efficiently.

The thread pool model avoids this overhead by pre-spawning a fixed number of threads at start-up to service all incoming requests. This strategy amortizes the cost of thread creation and bounds the use of OS resources. Client requests can execute concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests must be queued (or rejected) until a thread becomes available.

Figure 1 illustrates the main components in this concurrency model. These components include a *main thread*, a *request queue*, and a set of *pool threads*. The main thread receives new requests and inserts them into the tail of the request queue, while the pool threads remove requests from the head of the queue and service them. We'll explore the implementation and use of these components in this column using C, C++ wrappers, and CORBA, respectively.

3 The Multi-threaded C Thread Pool Solution

3.1 C Code

The following example shows a solution written using C, sockets, and Solaris threads [2][1] As in previous columns, we use a set of C utility functions to receive stock quote requests from clients (`recv_request`), look up 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 */

HANDLE create_server_endpoint (u_short port);
int recv_request (HANDLE, struct Quote_Request *);
int send_response (HANDLE, long stock_value);
int handle_quote (HANDLE);
```

These functions were first implemented 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

Our server `main` is similar to the one we presented for the multi-threaded C solution in our last column. The key difference is that we don't dynamically spawn a thread for each new client request. Instead, we create a thread-safe message queue, a pool of threads, and start an event loop in the main thread, as shown below:

```
const int DEFAULT_PORT = 12345;
const int DEFAULT_POOL_SIZE = 4;

int main (int argc, char *argv[])
{
  u_short port = /* Port to listen for connections. */
    argc > 1 ? atoi (argv[1]) : DEFAULT_PORT;
  int pool_size = /* Size of the thread pool. */
    argc > 2 ? atoi (argv[2]) : DEFAULT_POOL_SIZE;

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

  Handle_Queue handle_queue;

  /* Initialize the thread-safe message queue. */
  handle_queue_init (&handle_queue);

  /* Initialize the thread pool. */
  thread_pool_init (&handle_queue, pool_size);

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

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

The `svc_run` function runs the main thread's event loop, as follows:

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

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

    /* Enqueues the request for processing
       by a thread in the pool. */
    handle_queue_insert (handle_queue, handle);
  }
  /* NOTREACHED */
}
```

The main thread runs an event loop that continuously accepts new connections from clients and enqueues each connection in a `Handle_Queue`, which is a thread-safe queue of `HANDLEs`. Subsequently, a thread in the thread pool will remove the `HANDLE` from the queue, extract the client's stock quote request, look up the result, and return the result to the client.

The `Handle_Queue` plays several roles in this design. First, it decouples the main thread from the pool threads. This allows multiple pool threads to be active simultaneously and offloads the responsibility for maintaining the queue from kernel-space to user-space. Second, it enforces flow control between clients and the server. When there's no more room in the queue, the main thread blocks, which will "back-propagate" to the clients, preventing them from establishing new connections. New connection requests will not be accepted until pool threads have a chance to catch up and can unblock the main thread.

Each thread in the thread pool is spawned by the `thread_pool_init` function:

```
void
thread_pool_init (Handle_Queue *handle_queue,
                  int pool_size)
{
  int i;

  for (i = 0; i < pool_size; i++)
    /* Spawn off the thread pool. */
    thr_create
      (0, /* Use default thread stack. */
       0, /* Use default thread stack size. */
       &pool_thread, /* Entry point. */
       (void *) handle_queue, /* Entry point arg. */
       THR_DETACHED | THR_NEW_LWP, /* Flags. */
       0); /* Don't bother returning thread id. */
  }
}
```

3.1.2 The pool_thread() Function

Each newly created thread executes the following event loop in the `pool_thread` function:

```
void *pool_thread (void *arg)
{
  Handle_Queue *handle_queue =
    (Handle_Queue *) arg;

  /* The event loop for the each
     thread in the thread pool. */

  for (;;) {
    HANDLE handle;

    /* Get next available HANDLE. */
    handle_queue_remove (handle_queue, &handle);

    /* Return stock quote to client. */
    handle_quote (handle);

    /* Close handle to prevent leaks. */
    close (handle);
  }
  /* NOTREACHED */
  return 0;
}
```

When a pool thread becomes available, it will dequeue the next `handle` (corresponding to a client request), use it to look up the value of the stock quote, and return the quote to the client.

3.1.3 The Thread-Safe Handle Queue

Most of the complexity in the thread pool implementation resides in the thread-safe `Handle_Queue`. The main event loop thread uses this queue to exchange `HANDLEs` with the pool threads. We implement the queue as a C `struct` containing an array of `HANDLEs`, bookkeeping information, and synchronization variables:

```
#define MAX_HANDLES 100

/* Defines the message queue data. */
typedef struct Handle_Queue
{
  /* Buffer containing HANDLEs -- managed
     as a circular queue. */
  HANDLE queue_[MAX_HANDLES];

  /* Keep track of beginning and end of queue. */
  u_int head_, tail_;

  /* Upper bound on number of queued messages. */
  u_int max_count_;

  /* Count of messages currently queued. */
  u_int count_;

  /* Protect queue state from concurrent access. */
  mutex_t lock_;

  /* Block consumer threads until queue not empty. */
  cond_t notempty_;

  /* Block consumer threads until queue not full. */
  cond_t notfull_;

} Handle_Queue;
```

The `Handle_Queue` data structure is managed by the following C functions. The `handle_queue_init` function initializes internal queue state:

```
void handle_queue_init (Handle_Queue *handle_queue,
                        u_int max)
{
  handle_queue->max_count_ = max;
  handle_queue->count_ = 0;
  handle_queue->head_ = handle_queue->tail_ = 0;

  /* Initialize synchronization variables that
     are local to a single process. */
  mutex_init (&handle_queue->lock_,
              USYNC_THREAD, 0);
  cond_init (&handle_queue->notempty_,
             USYNC_THREAD, 0);
  cond_init (&handle_queue->notfull_,
             USYNC_THREAD, 0);
}
```

Three synchronization variables are used to implement the thread-safe `Handle_Queue`: two condition variables (`cond_t notempty_` and `notfull_`) and one mutex (`mutex_t lock_`). The condition variables enable threads to insert and remove `HANDLEs` to and from the queue concurrently. The mutex `lock_` is used by the condition variables to serialize access to the internal state of the queue, as shown in the `handle_queue_insert` function below:

```
void
handle_queue_insert (Handle_Queue *handle_queue,
                     HANDLE handle)
{
  /* Ensure mutual exclusion for queue state. */
  mutex_lock (&handle_queue->lock_);

  /* Wait until there's room in the queue. */
  while (handle_queue->count_
         == handle_queue->max_count_)
    cond_wait (&handle_queue->notfull_,
               &handle_queue->lock_);

  /* Code to insert handle into queue omitted... */

  /* Inform waiting threads that queue has a msg. */
  cond_signal (&handle_queue->notempty_);

  /* Release lock so other threads can proceed. */
  mutex_unlock (&handle_queue->lock_);
}
```

The `handle_queue_insert` function is called by the thread running the main event loop when it accepts a new request from a client. The client's `HANDLE` is inserted into the queue if there's room. Otherwise, the main event loop thread blocks until the `notfull_` condition is signaled. This condition is signaled when a pool thread dequeues a `HANDLE` from the queue via the following `handle_queue_remove` function:

```
void
handle_queue_remove (Handle_Queue *handle_queue,
                     HANDLE *first_handle)
{
  mutex_lock (&handle_queue->lock_);

  /* Wait while the queue is empty. */
  while (handle_queue->count_ == 0)
    cond_wait (&handle_queue->notempty_,
               &handle_queue->lock_);

  /* Code to remove first_handle from
     queue omitted... */

  /* Inform waiting threads that queue isn't full. */
  cond_signal (&handle_queue->notfull_);
  mutex_unlock (&handle_queue->lock_);
}
```

The handle_queue_remove function is called by all the pool threads. This function removes the next available HANDLE from the queue, blocking if necessary until the queue is no longer empty. After it removes the next HANDLE it signals the notfull_ condition to inform the main event loop thread that there's more room in the queue.[2]

3.2 Evaluating the C Thread Pool Solution

Depending on the degree of host parallelism and client application behavior, the new thread pool solution can improve the performance of the original thread-per-request approach. In particular, it will bound the amount of thread resources used by the server. There are still a number of drawbacks, however:

• **Too much infrastructure upheaval:** The implementation of the thread pool concurrency model shown above is an extension of the thread-per-request server from our previous column. We were able to reuse the core stock quote routines (such as recv_request, send_response, and handle_quote). However, the surrounding software architecture required many changes. Some changes were relatively minor (such as pre-spawning a thread pool rather than a thread-per-request). Other changes required significant work (such as implementing the thread-safe Handle_Queue).

• **Lack of flexibility and reuse:** Despite all the effort spent on our thread-safe Handle_Queue, the current implementation is tightly coupled to the queueing of HANDLEs. Closer examination reveals that the synchronization patterns used in handle_queue_insert and handle_queue_remove can be factored out and reused for other types of thread-safe queue management. Unfortunately, it is hard to do this flexibly, efficiently, and robustly with the current solution because C lacks features like parameterized types and method inlining.

• **High queueing overhead:** Another problem with the thread pool solution shown above is that the it may incur a non-trivial amount of context switching and synchronization overhead due to implement the thread-safe message queue. One way to eliminate this overhead is to remove the explicit message queue and have each of the threads in the pool block in an accept call, as follows:

```
void *pool_thread (void *arg)
{
  HANDLE listener = (HANDLE *) arg;
  HANDLE handle;

  /* Each thread accepts connections
     and performs the client's request. */

  while ((handle = accept (listener)) != -1)
    /* Return stock quote to client. */
    handle_quote (handle);

    /* Close handle to prevent leaks. */
    close (h);
  }
  /* NOTREACHED */
}
```

[2]There are techniques for minimizing the number of calls to cond_signal, which can improve performance significantly by reducing context switching overhead. These techniques are beyond the scope of this column and are discussed in [2, 3].

The main program is similar to the one shown in Section 3.1.1, as shown below:

```
int main (int argc, char *argv[])
{
  /* ... */

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

  /* Initialize the thread pool. */

  for (i = 0; i < pool_size; i++)
    /* Spawn off the thread pool. */
    thr_create
       (0, /* Use default thread stack. */
        0, /* Use default thread stack size. */
        &pool_thread, /* Entry point. */
        (void *) listener, /* Entry point arg. */
        THR_DETACHED | THR_NEW_LWP, /* Flags. */
        0); /* Don't bother returning thread id. */
  }

  /* Block waiting for a notification to
     close down the server. */

  /* ... */

  /* Unblock the threads by closing
     down the listener. */
  close (listener);
}
```

The main difference between this main and the previous one is that we no longer need to use the thread-safe message queue since each thread in the pool blocks directly on the accept call.

There are factors that may make this new approach less desirable in some usecases, however:

- *Reprioritize request processing* – It may be desirable to handle incoming requests in a different order than they arrive. By separating request processing from passive connection establishment, the thread-safe queueing mechanism makes it possible to reorder the requests relative to some priority scheme.
- *Limits on OS socket accept queue* – Many implementations of sockets limit the number of connections that can be queued by the operating system. Typically, this number is fairly low (*e.g.,* 8 to 10). On highly active servers (such as many WWW sites), this low limit will prevent clients from accessing the server, even though there may be available resources to process the client requests. By queueing the requests in user-space, our original approach may be more scalable in many situations.
- *Lack of atomicity for* accept – Some operating systems (*e.g.,* kernels based on BSD UNIX) implement accept as a system call, so that calls to accept are atomic. Other operarting systems (*e.g.,* many kernels based on System V UNIX) implement it as a library

call, so that calls to `accept` are *not* atomic. If `accept` is not atomic then it's possible for threads to receive `EPROTO` errors from `accept`, which means "protocol error" [4]. One solution to this problem is to explicitly add mutexes around the `accept` call, but this locking can itself become a bottleneck.

- *Caching open connections* – Our alternative thread pool solution forces each thread to allocate a new connection since threads are always blocked in `accept`. As shown below, this may be inefficient in some situations.

Therefore, we'll continue to use the thread-safe message queue example throughout the remainder of this paper. Be aware, however, that there are other ways to implement the thread pool concurrency model. Some of these approaches may be better suited for your requirements in certain circumstances.

• **High connection management overhead:** All the thread pool and thread-per-request server implementations we've examined thus far have set up and torn down a connection for each client request. This approach works fine if clients only request a single stock quote at a time from any given server. When clients make a series of requests to the same server, however, the connection management overhead can become a bottleneck.

One way to fix this problem is to keep each connection open until the client explicitly closes it down. However, extending the C solution to implement this connection caching strategy is subtle and error-prone. Several "obvious" solutions will cause race conditions between the main thread and the pool threads. For example, the `select` event demultiplexing call can be added to the original `svc_run` event loop, as follows:

```
// Global variable shared by the svc_run()
// and pool_thread() methods.
static fd_set read_hs;

void svc_run (Handle_Queue *handle_queue,
              HANDLE listener)
{
  HANDLE maxhp1 = listener + 1;
  fd_set temp_hs;

  /* fd_sets maintain a set of HANDLEs that
     select () uses to wait for events. */
  FD_ZERO (&read_hs);
  FD_ZERO (&temp_hs);
  FD_SET (listener, &read_hs);

  /* Main event loop. */

  for (;;) {
    HANDLE handle;
    /* Demultiplex connection and data events */
    select (maxhp1, &temp_hs, 0, 0, 0);

    /* Check for stock quote requests and
       insert the handle in the queue. */
    for (handle = listener + 1;
         handle < maxhp1;
         handle++)
      if (FD_ISSET (handle, &temp_hs))
        handle_queue_insert (handle_queue, handle);

    /* Check for new connections. */
    if (FD_ISSET (listener, &temp_hs)) {
      handle = accept (listener, 0, 0);
      FD_SET (handle, &read_hs);
      if (maxhp1 <= handle)
        maxhp1 = handle + 1;
    }
    temp_hs = read_hs;
  }
  /* NOTREACHED */
}
```

In addition, the `pool_thread` function would have to change (to emphasize the differences we've prefixed the changes with `/* !!!`):

```
void *pool_thread (void *arg)
{
  Handle_Queue *handle_queue =
    (Handle_Queue *) arg;

  /* The event loop for each
     thread in the thread pool. */

  for (;;) {
    HANDLE handle;

    /* Get next available HANDLE. */
    handle_queue_remove (handle_queue, &handle);

    /* !!! Return stock quote to client.  A
       return of 0 means the client shut down. */
    if (handle_quote (handle) == 0) {
      /* !!! Clear the bit in read_hs (i.e., the
         fd_set) so the main event loop will ignore
         this handle until it's reconnected. */
      FD_CLR (handle, &read_hs);

      /* Close handle to prevent leaks. */
      close (handle);
    }
    /* NOTREACHED */
    return 0;
  }
}
```

Unfortunately, this code contains several subtle race conditions. For instance, more than one thread can access the `fd_set` global variable `read_hs` concurrently, which can confuse the `svc_run` method's demultiplexing strategy. Likewise, the main thread can insert the same `HANDLE` into the `Handle_Queue` multiple times. Therefore, multiple pool threads can read from the same `HANDLE` simultaneously, potentially causing inconsistent results.

Alleviating these problems will force us to rewrite portions of the server by adding new locks and modifying the existing `handle_quote` code. Rather than spending any more effort revising the C version, we'll incorporate these changes into the C++ solution in the next section.

4 The Multi-threaded C++ Wrappers Thread Pool Solution

4.1 C++ Wrapper Code

This section illustrates a C++ thread pool implementation based on ACE [5]. The C++ solution is structured using the following four classes (shown in Figure 2):

• **Quote_Handler:** This class interacts with clients by receiving quote requests, looking up quotes in the database, and returning responses.

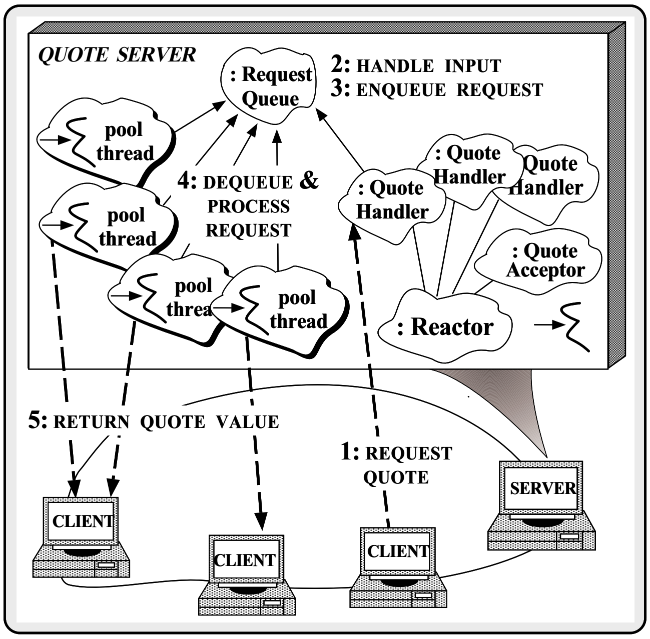

Figure 2: ACE C++ Architecture for the Thread Pool Stock Quote Server

• **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` methods of `Quote_Acceptor` and `Quote_Handler` when connection events and quote requests arrive from clients, respectively.

• **Request_Queue:** This thread-safe queue passes client requests from the main thread to the pool threads.

The C++ implementation of the thread pool model is considerably easier to develop than the C solution because we don't need to rewrite all the infrastructure code from scratch. For instance, variations of the `Quote_Handler`, `Quote_Acceptor`, and `Reactor` have been used in previous implementations of the quote server in the October 1995 and February 1996 C++ Report. Likewise, the `Request_Queue` can be implemented by using components available with C++ libraries like ACE and STL [6]. Below, we illustrate how these components are used to construct a multi-threaded quote server based on the C++ thread pool concurrency model.

4.1.1 The Thread-Safe C++ Request Queue

We'll start off by using several ACE and STL classes to create a thread-safe C++ queue that holds a tuple containing information necessary to process a client request. Since there is only one of these, we'll define it using the Singleton pattern [7]. Doing this is easy using the following components provided by STL and ACE:

```
// Forward declaration.
template <class PEER_STREAM>
class Quote_Handler;

// Use the STL ``pair'' component to create a
// tuple of objects to represent a client request.
typedef pair<Quote_Handler<SOCK_Stream> *,
             Quote_Request *>
        Quote_Tuple;

// An ACE thread-safe queue of Quote_Pairs.
typedef Message_Queue<Quote_Tuple> Quote_Queue;

// An ACE Singleton that accesses the Quote_Queue.
typedef Singleton<Quote_Queue, Mutex> Request_Queue;
```

The STL `pair` class is a template that stores two values. We use `pair` to create a tuple containing pointers to a `Quote_Handler` and a `Quote_Request`. This tuple contains the information necessary to process client requests efficiently and correctly in the thread pool model.

The ACE `Message_Queue` is a flexible, type-safe C++ wrapper that uses templates to generalize the type of data that can be stored in the C `Handle_Queue` implementation from Section 3:

```
template <class TYPE, size_t MAX_SIZE = 100U>
class Message_Queue
{
public:
  int insert (const TYPE &);
  int remove (TYPE &);
  // ...

private:

  // Buffer of TYPE, managed as a queue.
  TYPE queue_[MAX_SIZE];

  // ...
```

The ACE `Singleton` class is an adapter that turns ordinary classes into Singletons [7], as follows:

```
template <class TYPE, class LOCK = Mutex>
class Singleton
{
public:
  static TYPE *instance (void) {
    // Perform the Double-Checked Locking
    // pattern to ensure proper initialization.
    if (instance_ == 0) {
      Guard<LOCK> lock (lock_);
      if (instance_ == 0)
        instance_ = new TYPE;
    }
    return instance_;
  }

protected:
  // Singleton instance of TYPE.
  static TYPE *instance_;

  // Lock to ensure serialization.
  static LOCK lock_;
};
```

The ACE `Singleton` adapter avoids subtle race conditions by using the Double-Checked Locking pattern [8]. This pattern allows atomic initialization, regardless of thread initialization order, and eliminates subsequent locking overhead).

Using the ACE `Singleton` wrapper in conjunction with the ACE `Message_Queue` and STL `pair`, the thread pool server can insert and remove `Quote_Handler` objects as follows:

```
Quote_Tuple qt (quote_handler, quote_request);
// ...
Request_Queue::instance ()->insert (qt);

// ...
Request_Queue::instance ()->remove (qt);
```

The first time that `insert` or `remove` is called, the `Singleton::instance` method dynamically allocates and initializes the thread-safe `Request_Queue`. The Singleton pattern also minimizes the need for global objects, which is important in C++ since the order of initialization of global objects in C++ programs is not well-defined. Therefore, we'll use the same approach for the `Quote_Database` and the `Reactor`:

```
// Singleton for looking up quote values.
typedef Singleton<Quote_Database> QUOTE_DB;

// Singleton event demultiplexing and dispatching.
typedef Singleton<Reactor> REACTOR;
```

4.1.2 The Quote_Handler Class

The `Quote_Handler` class is responsible for processing client quote requests. Its implementation differs considerably from the one used for the thread-per-request concurrency model in the February C++ Report.

```
template <class STREAM> // IPC interface
class Quote_Handler
  : public Svc_Handler<STREAM>
    // This ACE base class defines "STREAM peer_;"
{
public:
  // !!! This method is called by the Quote_Acceptor
  // to initialize a newly connected Quote_Handler,
  // which registers with the Reactor Singleton.
  virtual int open (void) {
    REACTOR::instance ()->register_handler
       (this, READ_MASK);
  }

  // !!! This method is called by the Reactor when
  // a quote request arrives.  It inserts the request
  // and the Quote_Handler into the thread-safe queue.
  virtual int handle_input (void) {
    Quote_Request *request = new Quote_Request;
    if (recv_request (*request) <= 0)
      return -1; // Destroy handler...
    else {
      Quote_Tuple qt (request, this)

      // Insert tuple into queue, blocking if full.
      Request_Queue::instance ()->insert (qt);
    }
  }

  // !!! Static method that runs in the thread,
  // dequeueing next available Quote_Request.
  static void *pool_thread (void *) {
    for (;;) {
      Quote_Tuple qt;

      // Get next request from queue.  This
      // call blocks if queue is empty.
      Request_Queue::instance ()->remove (qt);

      // typeid (qt->first) == Quote_Handler *
      // typeid (qt->second) == Quote_Request *
      if (qt->first->handle_quote
           (qt->second) == 0)
        // Client shut down, so close down too.
        qt->first->close ();
      delete qt->second;
    }
    /* NOTREACHED */
  }

  // !!! Complete the processing of a request.
  int handle_quote (Quote_Request *req) {
    int value;
    {
      // Constructor of m acquires lock.
      Read_Guard<RW_Mutex> m (lock_);

      // Lookup stock price via Singleton.
      value = QUOTE_DB::instance ()->
                  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 ();

    // Reference counting omitted...

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

private:
  // Ensure mutual exclusion to QUOTE_DB.
  RW_Mutex lock_;
};
```

Each thread in the pool executes the static `pool_thread` function. This function runs an event loop that continuously removes `Quote_Tuples` from the queue. The `first` field in this tuple is the `Quote_Handler` associated with the client and the `second` field is a client `Quote_Request`. The `pool_thread` uses the `first` field to invoke the `handle_quote` method, which lookups the value of the desired stock and returns it to the client.

When the client closes down, the `Quote_Handler` cleans up the connection. Even though the client has already closed the connection, note that the `close` function must perform reference counting on its target `Quote_Handler` object (to save space, we've omitted this code). If this reference counting were not performed, the `close` function could prematurely delete the `Quote_Handler`. This could cause the `pool_thread` function to invoke `handle_quote` on a dangling `first` pointer, which in turn would probably cause the server to crash.

Note that both `handle_input` and `pool_thread` can block since each manipulates the global thread-safe queue. The `handle_input` method will block if the queue is full, whereas the `pool_thread` function will block if the queue is empty.

4.1.3 The Quote_Acceptor Class

The `Quote_Acceptor` class is an implementation of the Acceptor pattern [9] that creates `Quote_Handlers` to process quote requests from clients. Its implementation is similar to the one shown in our previous column:

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

The Quote_Acceptor's strategy for initializing a Quote_Handler is driven by upcalls from the Reactor. Whenever a new client connects with the server, the Quote_Acceptor's handle_input method dynamically creates a Quote_Handler, accepts the connection into the handler, and automatically calls the Quote_Handler::open method. In the thread pool implementation, this open method registers itself with the Reactor, as we showed in Section 4.1.2 above.

4.1.4 The main() Server Function

The server main is responsible for creating a thread pool and the Quote_Acceptor, as follows:

```
// !!! Default constants.
const int DEFAULT_PORT = 12345;
const int DEFAULT_POOL_SIZE = 4;

int main (int argc, char *argv[])
{
  u_short port =
    argc > 1 ? atoi (argv[1]) : DEFAULT_PORT;
  int pool_size = // !!! Size of the thread pool.
    argc > 2 ? atoi (argv[2]) : DEFAULT_POOL_SIZE;

  // !!! Create a pool of threads to
  // handle quote requests from clients.
  Thread::spawn_n
    (pool_size,
     Quote_Handler<SOCK_Stream>::pool_thread,
     (void *) 0,
     THR_DETACHED | THR_NEW_LWP);

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

  svc_run (acceptor);

  /* NOTREACHED */
  return 0;
}
```

First, the ACE method spawn_n [3] is called to create a pool of n threads. Each thread executes the Quote_Handler::pool_thread function. Next, a Quote_Acceptor object is created. This object is used to accept connections from clients and create Quote_Handler objects to service them. Finally, the following svc_run function is called to run the main thread's event loop:

```
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 or Quote_Handlers).

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

The main thread's event loop runs continuously, handling events like client connections and quote requests. The server's event handling is driven by callbacks from the REACTOR Singleton to the Quote_Acceptor and Quote_Handler objects. Since this server uses the thread pool model, requests can be handled concurrently by any available thread.

4.2 Evaluating the C++ Thread Pool Solution

The C++ implementation solves the drawbacks with the C version shown in Section 3.2 as follows.

• **Less infrastructure upheaval:** Compared to the changes between our C program in our last column and the C program shown in this column, the changes between the respective C++ programs are much fewer and more localized. In addition to creating a thread-safe Request_Queue Singleton, the primary changes to our C++ thread pool implementation are in the Quote_Handler class and in our server main routine.

In our last column, our Quote_Handler::open function spawned a thread to handle each incoming request. Here, open has been changed to register the new Quote_Handler with the Reactor. Then, when client requests arrive, the Quote_Handler's handle_input method will queue both the request and the handler until a thread from the pool becomes available to service it. The only other change required was to make main create the thread-safe queue, the thread pool, and the Reactor before entering into its event loop.

• **Greater flexibility and reuse:** Fewer changes were required in the C++ version than in the C version due to the encapsulation of connection handling, queueing, and request servicing within C++ classes.

• **Minimal connection management overhead:** The C++ solution keeps each client connection open until the client closes it down. In addition, by using the thread-safe Request_Queue and the Quote_Tuple, we can avoid the subtle race conditions that plagued the earlier C version.

Obviously, the C++ solution is not without its drawbacks. For instance, we've omitted the code that performs reference counting to ensure that a Quote_Handler is not deleted until all of the Quote_Requests stored in the Request_Queue are removed. In addition, the programmer must either be able to buy or build a thread-safe queue class. Developing such a class is not trivial, especially when portability among different threads packages, OS platforms, and C++ compilers is required. The Standard Template Library (STL) is of no help here since the draft C++ standard does not require its queue class to be thread-safe. Fortunately, we are able to leverage the ACE components to simplify our implementation. ACE has been ported to most versions of UNIX, as well as the Microsoft Win32 platform.

5 The Multi-threaded CORBA Thread Pool Solution

This section illustrates how to implement the thread pool concurrency model with MT-Orbix. The solution we describe below uses the same general design as our C++ implementation above. It also uses many of the same components (such as the ACE `Singleton` and `Message_Queue` classes).

5.1 Implementing Thread Pools in MT-Orbix

The `My_Quoter` implementation class shown below is almost identical to the one we used in our previous column to implement the thread-per-request model. The main difference is the use of object composition to associate the `My_Quoter` implementation class with the `Quoter` IDL interface. We'll discuss this below, but first, here's the complete implementation:

```
class My_Quoter // Note the absence of inheritance!
{
public:
  // Constructor
  My_Quoter (const char *name);

  // Returns the current stock value.
  virtual CORBA::Long get_quote
    (const char *stock_name,
     CORBA::Environment &env)
  {
    CORBA::Long value;
    {
      // Constructor of m acquires lock.
      Read_Guard<RW_Mutex> m (lock_);

      value = QUOTE_DB::instance ()->
                        lookup_stock_price (stock_name);
      // Destructor of m releases lock.
    }
    if (value == -1)
      // Raise exception.
      env.exception (new Stock::Invalid_Stock);
    return value;
  }

protected:
  // Serialize access to database.
  RW_Mutex lock_;
};
```

As before, it's necessary to protect access to the quote database with a readers/writer lock since multiple requests can be processed simultaneously by threads in the pool.

5.1.1 Associating the IDL Interface with an Implementation

If you've been following our columns carefully, you'll notice that the Orbix implementation of the `My_Quoter` class in the May 1995 C++ Report inherited from a skeleton called `QuoterBOAImpl`. This class was automatically generated by the Orbix IDL compiler, *i.e.*:

```
class My_Quoter
    // Inherits from an automatically-generated
    // CORBA skeleton class.
  : virtual public Stock::QuoterBOAImpl
```

In contrast, our current implementation of `My_Quoter` does *not* inherit from any generated skeleton. Instead, it uses an alternative provided by Orbix called the "TIE" approach, which is based on object composition rather than inheritance:

```
class My_Quoter // Note lack of inheritance!
{
  // ...
};
```

We use the Orbix "TIE" approach to associate the CORBA interfaces with our implementation as follows:

```
DEF_TIE_Quoter (My_Quoter)
```

The TIE approach is an example of an "object form" of the Adapter pattern [7], whereas the inheritance approach we used last column uses the "class form" of the pattern. The object form of the Adapter uses *delegation* to "tie" the interface of the `My_Quoter` object implementation class to the interface expected by the `Quoter` skeleton generated by MT-Orbix. When a request is received, the Orbix Object Adapter upcalls the TIE object. In turn, this object dispatches the call to the `My_Quoter` object that is associated with the TIE object.

The TIE approach is mentioned in the C++ Language Mapping chapters of the CORBA 2.0 specification [10]. Not surprisingly, the idea for putting it there originally came from IONA Technologies, the makers of Orbix. Conforming ORB implementations are not required to support either the TIE approach or the inheritance approach, however.[3]

5.1.2 The C++ Thread-Safe Request Queue

The `Request_Queue` used by the CORBA implementation is reused almost wholesale from the C++ implementation shown in Section 4.1.1:

```
// An ACE Singleton that accesses an ACE
// thread-safe queue of CORBA Request pointers.
typedef Singleton<Message_Queue<CORBA::Request *>,
                  Mutex>
        Request_Queue;
```

The primary difference is that we parameterize it with a `CORBA::Request` pointer, rather than a `Quote_Tuple`. The reason for this is that MT-Orbix performs the low-level demultiplexing, so we don't have to do it ourselves.

5.1.3 Thread Filters

Orbix implements a non-standard CORBA extension called "thread filters." Each incoming CORBA request is passed through a chain of filters before being dispatched to its target object implementation. To dispatch an incoming CORBA request to a waiting thread, a subclass of `ThreadFilter` must be defined to override the `inRequestPreMarshal` method. By using a `ThreadFilter`, the MT Orbix ORB and Object Adapter are unaffected by the choice of concurrency model selected by a CORBA server.

[3] The lack of a clear specification of whether CORBA C++ server skeletons use inheritance or delegation is another indication of the CORBA server-side portability problems we have described in previous columns.

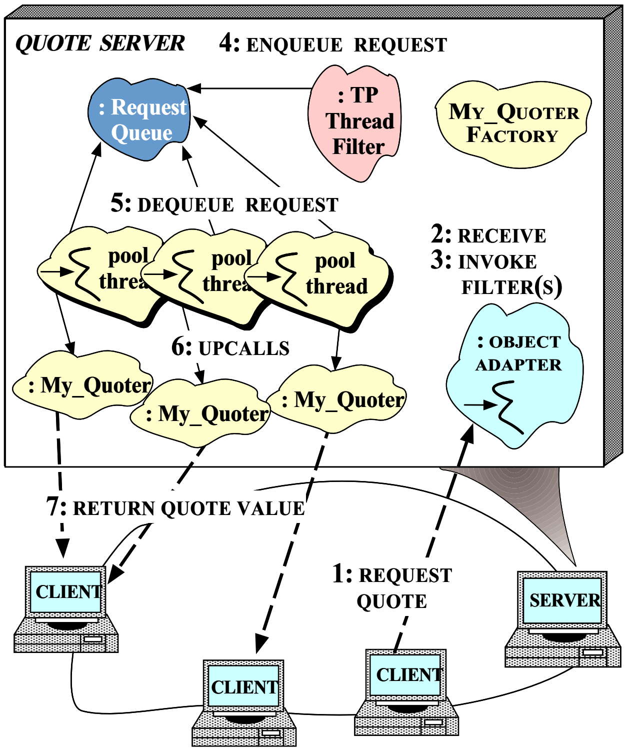

Figure 3: MT Orbix Architecture for the Thread Pool Stock Quote Server

The following class defines a server-specific thread filter that handles incoming requests in accordance with the Thread Pool concurrency model:

```
class TP_Thread_Filter : public CORBA::ThreadFilter
{
public:
  // Intercept request insert at end of msg_que.
  virtual int inRequestPreMarshal (CORBA::Request &,
                                   CORBA::Environment &);

  // A pool thread uses this as its entry point,
  // so this must be a static method.
  static void *pool_thread (void *);
};
```

Orbix calls `inRequestPreMarshal` method before the incoming request is processed. In the Thread Pool model, requests are inserted in FIFO order at the end of a thread-safe `Message_Queue` as they arrive, as follows:

```
TP_Thread_Filter::inRequestPreMarshal
  (CORBA::Request &req,
   CORBA::Environment&)
{
  // Will block if queue is full...
  Request_Queue::instance ()->insert (&req);

  // We'll dispatch the request later.
  return -1;
}
```

Note that this method must return the magic number -1 to indicate to the Orbix Object Adapter that it has dealt with the request. This value informs the Object Adapter that it need not perform the operation dispatch itself, nor should it return the result to the client. These operations will be performed by one of the threads in the thread pool, as shown in Figure 3.

Figure 3 illustrates the role of the `TP_Thread_Filter` in the MT Orbix architecture for the Thread Pool stock quote server. Our quote server must explicitly create an instance of `TP_Thread_Filter` to get it installed into the Orbix filter chain:

```
TP_Thread_Filter tp_filter;
```

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

The `pool_thread` static method serves as the entry point for each thread in the thread pool, as shown below:

```
void *TP_Thread_Filter::pool_thread (void *)
{
  // Loop forever, dequeueing new Requests,
  // and dispatching them....

  for (;;) {
    CORBA::Request *req;

    // Called by pool threads to dequeue
    // the next available message.  Will block
    // if queue is empty.
    Request_Queue::instance ()->remove (req);

    // This call will perform the upcall,
    // send the reply (if any) and
    // delete the Request for us...
    CORBA::Orbix.continueThreadDispatch (*req);
  }

  return 0;
}
```

All threads wait for requests to arrive on the head of the message queue stored in our `TP_Thread_Filter`. The MT-Orbix method `continueThreadDispatch` will continue processing the request until it sends a reply to the client. At this point, the thread will loop back to retrieve the next CORBA request. If there is no request available the thread will block until a new request arrives on the message queue. Likewise, if all the threads are busy, the queue will continue growing until it reaches its high-water mark, at which point the thread running the `inRequestPreMarshal` method will block. This relatively crude form of flow control was also used in the C and C++ implementations shown earlier. Naturally, robust servers should be programmed more carefully to detect and handle queue overflow conditions.

The main server program implements the Thread Pool concurrency model by spawning off `pool_size` number of threads, as follows:

```
const int DEFAULT_POOL_SIZE = 4;

int main (int argc, char *argv[])
{
  // Initialize the factory implementation.
  My_Quoter_var my_quoter =
    new TIE_My_Quoter (My_Quoter) (new My_Quoter);

  int pool_size = argc == 1 ? DEFAULT_POOL_SIZE
                            : atoi (argv[1]);

  // Create a pool of threads to handle
  // quote requests from clients.
```

```
  Thread::spawn_n (pool_size,
                   Thread_Filter::pool_thread,
                   (void *) 0,
                   THR_DETACHED | THR_NEW_LWP);

  // Wait for work to do in the main thread
  // (which is also the thread that shepherds
  // CORBA requests through TP_Thread_Filter).
  TRY {
     CORBA::Orbix.impl_is_ready ("Quoter",
                                 IT_X);
  } CATCHANY {
     cerr << IT_X << endl;
  } ENDTRY

  return 0;
}
```

When the `Quote` server first starts up, it creates a `My_Quoter` object to service client quote requests. It then creates a pool of threads to service incoming requests using the ACE `spawn_n` method. Finally, the main server thread calls `Orbix.impl_is_ready` to notify Orbix that the `Quoter` implementation is ready to service requests. The main thread is responsible for sheparding CORBA requests through the filter chain to the `TP_Thread_Filter`.

Finally, the object we initially created is implicitly destroyed by the destructor of the `My_Quoter_var`. The OMG C++ Mapping provides for each IDL interface a "`_var`" class that can manage object references ("`_ptr`" types) of that interface type. If we didn't use a `My_Quoter_var` type here, our code would have to manually duplicate and release the object as required. By using a `My_Quoter_var`, we let the smart pointer perform the resource management.

5.2 Evaluating the MT-Orbix Thread Pool Solution

The following benefits arise from using MT-Orbix to implement the thread pool concurrency model:

• **Almost no infrastructure upheaval:** The implementation of the MT-Orbix thread pool concurrency model shown above is almost identical to the thread-per-request server from our previous column. The primary changes we added were cosmetic (such as using Singletons rather than global variables and using the object composition to "tie" the `Quoter` skeleton with the `My_Quoter` implementation rather than using inheritance). The ability to quickly and easily modify applications in this manner allows them to be rapidly tuned and redeployed when necessary.

• **Increased flexibility and reuse:** The flexibility and reuse of the MT-Orbix solution is similar to the ACE C++ solution. The main difference is that MT-Orbix is responsible for most of the low-level demultiplexing and concurrency control that we had to implement by hand in our C++ solution. In particular, MT-Orbix hides all its internal synchronization mechanisms from the server programmer. Thus, we are only responsible for locking server-level objects (such as the `Request_Queue`).

• **Optimized connection management overhead:** MT-Orbix can perform certain optimizations (such as caching connections in a thread-safe manner) without requiring any programmer intervention. It also separates the concerns of application development from those involving the choice of suitable transports and protocols for the application. In other words, using an ORB allows an application to be developed independently of the underlying communication transports and protocols.

The primary drawback, of course, is that the mechanisms used by MT-Orbix are not standardized across the industry. In general, all the multi-threading techniques we discuss in this column aren't standardized yet, and in particular the `TP_Thread_Filter` approach shown above is proprietary to Orbix. The fact that the CORBA solution shown here is not portable is yet another indication of the server-side portability problems with CORBA that we've discussed in previous columns.

Despite these issues, it is important to note that the concurrency models, patterns, and techniques we discussed in this article *are* reusable. Our goal is to help you navigate through the space of design alternatives. We hope that you'll be able to apply them to your projects, regardless of whether you program in CORBA, DCE, Network OLE, ACE, or any other distributed computing toolkit.

6 Concluding Remarks

In this column, we examined the thread pool concurrency model and illustrated how to use it to develop multi-threaded servers for a distributed stock quote application. This example illustrated how object-oriented techniques, C++, CORBA, and higher-level abstractions like the Singleton pattern help to simplify programming and improve extensibility.

Our next column will explore yet another concurrency model: *thread-per-session*. This model is supported by a number of CORBA implementations including MT-Orbix and ORBeline. Having a choice of concurrency models can help developers meet the performance, functionality, and maintenance requirements of their applications. The key to success, of course, lies in thoroughly understanding the tradeoffs between different 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 Prashant Jain, Tim Harrison, Ron Resnick, and Esmond Pitt for comments on this column.

References

[1] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[2] 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.

[3] 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.

[4] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[5] 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.

[6] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

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

[8] D. C. Schmidt and T. Harrison, "Double-Checked Locking – An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

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

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