Object Interconnections

Comparing Alternative Server Programming Techniques (Column 4)

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

An earlier version of this column appeared in the October 1995 issue of the SIGS C++ Report magazine.

1 Introduction

This column examines and evaluates several techniques for developing client/server applications in order to illustrate key aspects of distributed programming. The application we're examining enables investment brokers to query the price of a stock from a distributed quote database. Our two previous columns outlined the distributed computing requirements of this application and examined several ways to implement the client-side functionality. Below, we compare several ways to program the server-side of this application.

The solutions we examine in this column range from using C, `select`, and the sockets network programming interface; to using C++ wrappers for `select` and sockets; to using a distributed object computing (DOC) solution based on the OMG's Common Object Request Broker Architecture (CORBA). Along the way, we'll examine various tradeoffs between extensibility, robustness, portability, and efficiency for each of the three solutions.

2 Server Programming

Developers who write the server-side of an application must address certain topics that client-side developers may be able to ignore. One such topic is *demultiplexing* of requests from multiple clients. For example, our stock quote server can be accessed simultaneously by multiple clients connected via communication protocols such as TCP/IP or IPX/SPX. Therefore, the server must be capable of receiving client requests over multiple connections without blocking indefinitely on any single connection.

A related topic that server programmers must address is *concurrency*. The two primary types of server concurrency strategies [1, 2] are distinguished as follows:

- *Iterative servers* – which handle each client request before servicing subsequent requests. While processing the current request, an iterative server typically queues new client requests. An iterative design is most suitable for short-duration services that exhibit relatively little variation in their execution time. Internet services like `echo` and `daytime` are commonly implemented as iterative servers.

- *Concurrent servers* – which handle multiple client requests simultaneously. Concurrent servers help improve responsiveness and reduce latency when the rate at which requests are processed is less than the rate at which requests arrive at the server. A concurrent server design may also increase throughput for I/O-bound and/or long-duration services that require a variable amount of time to execute. Internet services like `telnet` and `ftp` are commonly implemented as concurrent servers.

Concurrent servers generally require more sophisticated synchronization and scheduling strategies than iterative servers. For the example application in this column, we'll assume that each stock quote request in the server executes quickly. Therefore, we'll use a variant of the iterative server that meets our response and throughput requirements. Moreover, as shown below, our synchronization and scheduling strategies are simplified by using an iterative server.

We'll use the UNIX `select` event demultiplexing system call to provide a simple round-robin scheduler. The `select` call detects and reports the occurrence of one or more connection events or data events that occur simultaneously on multiple communication endpoints (*e.g.,* socket handles). The `select` call provides coarse-grained concurrency control that serializes event handling within a process or thread. This eliminates the need for more complicated threading, synchronization, or locking within our server.

In-depth coverage of sockets and `select` appears in [2]. In future columns we'll discuss how to extend our solutions to incorporate more sophisticated concurrency and demultiplexing strategies.

3 The Socket Server Solution

3.1 Socket/C Code

The following code illustrates how to program the server-side of the stock quote program using sockets, `select`, and C. The following two C structures define the schema for quote requests and quote responses:

```
#define MAXSTOCKNAMELEN 100

struct Quote_Request
{
  long len; /* Length of the request. */
  char name[MAXSTOCKNAMELEN]; /* Stock name. */
};

struct Quote_Response
{
  long value; /* Current value of the stock. */
  long errno; /* 0 if success, else errno value. */
};
```

These structures are exchanged between the client-side and server-side of the stock quote programs.

Next, we've written four C utility routines. These routines shield the rest of the application from dealing with the low-level socket interface. To save space, we've omitted most of the error handling code. Naturally, a robust production application would carefully check the return values of system calls, handle unexpected connection resets, and insure that messages don't overflow array bounds.

The first routine receives a stock quote request from a client:

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

int recv_request (HANDLE h,
                  struct Quote_Request *req)
{
  int r_bytes, n;
  int len = sizeof *req;

  /* Recv data from client, handle "short-reads". */
  for (r_bytes = 0; r_bytes < len; r_bytes += n) {
    n = recv (h, ((char *) req) + r_bytes,
              len - r_bytes, 0);
    if (n <= 0) return n;
  }
  /* Decode len to host byte order. */
  req->len = ntohl (req->len);
  return r_bytes;
}
```

The length field of a `Quote_Request` is represented as a binary number that the client's `send_request` encoded into network byte order. Therefore, the server's `recv_request` routine must decode the message length back into host byte order using `ntohl`. In addition, since we use the bytestream-oriented TCP protocol, the server code must explicitly loop to handle "short-reads" that occur due to buffer constraints in the OS and transport protocols.

The following `send_response` routine sends a stock quote from the server back to the client. It encodes the numeric value of the stock quote into network byte order before returning the value to the client, as follows:

```
int send_response (HANDLE h, long value)
{
  struct Quote_Response res;
  size_t w_bytes;
  size_t len = sizeof res;

  // Set error value if failure occurred.
  res.errno = value == -1 ? htonl (errno) : 0;
  res.value = htonl (value);

  /* Respond to client, handle "short-writes". */
  for (w_bytes = 0; w_bytes < len; w_bytes += n) {
    n = send (h, ((const char *) &res) + w_bytes,
              len - w_bytes, 0);
    if (n <= 0) return n;
  }
  return w_bytes;
}
```

As with `recv_request`, the server must explicitly handle short-writes by looping until all the bytes in the response are sent to the client.

The `handle_quote` routine uses the C functions shown above to receive the stock quote request from the client, look up the value of the stock in an online database, and return the value to the client, as follows:

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

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

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

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

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

The `handle_quote` function illustrates the synchronous, request/response style of communication between clients and the quote server.[1] Note that we only perform one request/response cycle for each client at a time since we've designed the `quote server` as an iterative server. This design ensures that a highly active client doesn't starve out other clients by sending multiple requests back-to-back.

The next routine creates a socket server endpoint that listens for connections from stock quote clients. The caller passes the port number to listen on as a parameter:

```
HANDLE create_server_endpoint (u_short port)
{
  struct sockaddr_in addr;
  HANDLE h;

  /* Create a local endpoint of communication. */
  h = socket (PF_INET, SOCK_STREAM, 0);

  /* Setup the address of the server. */
  memset ((void *) &addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port = htons (port);
  addr.sin_addr.s_addr = INADDR_ANY;
```

[1] In a future column we'll illustrate how to develop asynchronous "publish/subscribe" communication mechanisms that notify consumers automatically when stock values change.

```
  /* Bind server port. */
  bind (h, (struct sockaddr *) &addr, sizeof addr);

  /* Make server endpoint listen for connections. */
  listen (h, 5);
  return h;
}
```

The `main` function shown below uses the C utility routines defined above to create an iterative quote server. The `select` system call demultiplexes new connection events and data events from clients. Connection events are handled directly in the event loop, which adds the new `HANDLE` to the `fd_set` used by `select`. Data events are presumed to be quote requests, which trigger the `handle_quote` function to return the latest stock quote from the online database. Note that data events are demultiplexed using a round-robin scheduling policy that dispatches the `handle_quote` function in order of ascending `HANDLE` values.

```
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);
    HANDLE maxhp1 = listener + 1;

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

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

        /* Check for stock quote requests and
           dispatch the quote handler in
           round-robin order. */
        for (h = listener + 1; h < maxhp1; h++)
            if (FD_ISSET(h, &temp_hs))
                if (handle_quote(h) == 0) {
                    /* Client's shutdown. */
                    FD_CLR(h, &read_hs);
                    close(h);
                }

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

The main program iterates continuously accepting connections and returning quotes. Once a client establishes a connection with the server it remains connected until the client explicitly closes down the connection. This design amortizes the cost of establishing connections since clients can request multiple quote values without reconnecting. As long as there are sufficient OS resources (such as descriptors and memory) available, keeping multiple clients connected helps improve performance.

The `select`-based iterative server we've show above is actually "pseudo-concurrent" with respect to connection acceptance since multiple connections can be active at the same time. However, it is iterative with respect to request processing since only one `handle_quote` function is actively processing client requests at a time. Moreover, no single client can block other clients for more than one request/response cycle since the `select`-based iterative server uses round-robin demultiplexing and dispatching of client requests. Naturally, this works since the `handle_quote` function only processes with a single request at a time.[2]

There are other variants of iterative servers [3]. For example, one common variant looks like this:

```
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);

    for (;;) {
      HANDLE h = accept(listener, 0, 0);
      handle_quote(h);
      close(h);
    }
    /* NOTREACHED */
}
```

In this variant, the server is iterative with respect to *both* accepting connections and request processing. However, this may cause unacceptably high overhead for "conversation-oriented" applications since a new connection must be established for each request.

Regardless of which variant is used, a key characteristic of an iterative server is that request processing is serialized at the event demultiplexing layer. Thus, no additional synchronization is necessary within the server application code.

3.2 Evaluating the Socket Solution

Programming with C, sockets, and `select` as shown above yields relatively efficient sequential programs. However, sockets and `select` are low-level interfaces. Our previous column described the many communication-related activities that must be performed by programs written at this level. Briefly, these activities include initializing the socket endpoints, establishing connections, marshalling and unmarshalling of stock quote requests and responses, sending and receiving messages, detecting and recovering from errors, and providing security.

In addition to these activities, the server must also perform demultiplexing and concurrency. Directly programming `select` to demultiplex events is particularly problematic [4]. The `select` call requires programmers to explicitly handle many low-level details involving bitmasks, descriptor counts, time-outs, and signals. In addition to being tedious and error-prone, `select` is not portable across OS platforms.

[2]Our subsequent column will illustrate how to remove this limitation.

Another drawback with the current structure of the quote server is that it hard-codes the application-specific service behavior directly into the program. This makes it hard to extend the current solution (*e.g.,* changing from an iterative to a concurrent server) without modifying existing source code. Likewise, it is hard to reuse any pieces of this solution in other servers that implement similar, but not identical, services.

4 The C++ Wrappers Solution

Using C++ wrappers is one way to simplify the complexity of programming network servers. C++ wrappers encapsulate lower-level network programming interfaces such as sockets and `select` with type-safe, object-oriented interfaces. The `IPC_SAP` [5], `Reactor` [4, 6], and `Acceptor` [7] C++ wrappers shown below are part of the ACE object-oriented network programming toolkit. `IPC_SAP` encapsulates sockets and TLI network programming interfaces; the `Reactor` encapsulates the `select` and `poll` event demultiplexing system calls; and the `Acceptor` combines `IPC_SAP` and the `Reactor` to implement a reusable strategy for establishing connections passively.[3]

4.1 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 three components in the quote server architecture:

- `Reactor` – defines a mechanism for registering, removing, and dispatching `Event_Handlers` (such as the `Quote_Acceptor` and `Quote_Handler` described below). The `Reactor` encapsulates the `select` and `poll` event demultiplexing system calls with an extensible and portable callback-driven object-oriented interface.
- `Quote_Acceptor` – a factory that implements the strategy for accepting connections from clients, followed by creating and activating `Quote_Handlers`.
- `Quote_Handler` – interacts with clients by receiving quote requests, looking up quotes in the database, and returning responses. `Quote_Handlers` can be implemented as either passive or active objects, depending on how they are configured.

Both the `Quote_Acceptor` and `Quote_Handler` inherit from the `Reactor`'s `Event_Handler` base class. This enables the `Reactor` to callback to their `handle_input` methods when connection events and data events arrive, respectively.

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

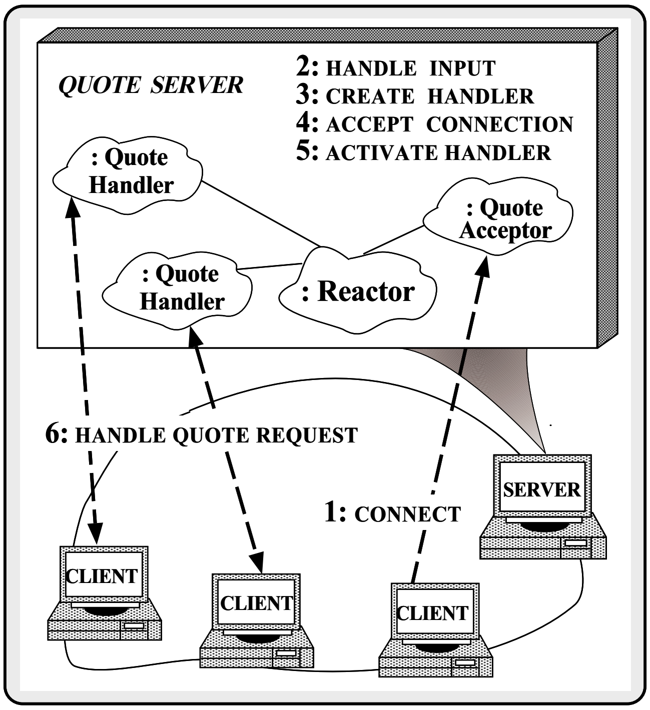

Figure 1: The C++ Wrapper Architecture for the Stock Quoter Server

We'll start by showing the `Quote_Handler`. This template class inherits from the reusable `Svc_Handler` base class in the ACE toolkit. A `Svc_Handler` defines a generic interface for a communication service that exchanges data with peers over network connections. For the stock quote application, `Svc_Handler` is instantiated with a communication interface that receives quote requests and returns quote values to clients. As shown below, it uses the `IPC_SAP` `SOCK_Stream` C++ wrapper for TCP stream sockets. `IPC_SAP` shields applications from low-level details of network programming interfaces like sockets or TLI.

```
template <class STREAM> // IPC interface
class Quote_Handler
  : public Svc_Handler<STREAM>
    // ACE base class defines "STREAM peer_;"
{
public:
  Quote_Handler (Quote_Database *db,
                 Reactor *r)
   : db_ (db), reactor_ (r) {}

  // This method is called by the Quote_Acceptor
  // to initialize a newly connected Quote_Handler,
  // which simply registers itself with the Reactor.
  virtual int open (void) {
    reactor_->register_handler (this, READ_MASK);
  }

  // This method is invoked as a callback by
  // the Reactor when data arrives from a client.
  virtual int handle_input (HANDLE) {
    handle_quote ();
    peer_.close();
  }

  virtual int handle_quote (void) {
```

```
    Quote_Request req;

    if (recv_request (req) <= 0)
      return -1;

    long value = db_->lookup_stock_price (req);

    return send_response (value);
  }

  virtual int recv_request (Quote_Request &req) {
    // recv_n handles "short-reads"
    int n = peer_.recv_n (&req, sizeof req);
    if (n > 0)
      /* Decode len to host byte order. */
      req.len (ntohl (req.len ()));
    return n;
  }

  virtual int send_response (long value) {
    // The constructor performs the error checking
    // and network byte-ordering conversions.
    Quote_Response res (value);

    // send_n handles "short-writes".
    return peer_.send_n (&res, sizeof res);
  }
private:
  Quote_Database *db_; // Database reference.
  Reactor *reactor_; // Event dispatcher.
};
```

The next class is the `Quote_Acceptor`. This inherits from the following reusable `Acceptor` connection factory in the ACE toolkit:

```
template <class SVC_HANDLER, // Service handler
          class PEER_ACCEPTOR> // Passive connection factory
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 creating and activating SVC_HANDLERs
  // to process data exchanged over the connections.

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

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

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

  // Pure virtual Factory method to make a svc handler.
  virtual SVC_HANDLER *make_svc_handler (void) = 0;

  // Returns the underlying passive-mode HANDLE.
  virtual HANDLE get_handle (void) {
    return peer_acceptor_.get_handle ();
  }

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 (`SOCK_Acceptor` or `TLI_Acceptor`) and (2) reactively perform the quote service (`Quote_Handler`): Note that using C++ classes and templates makes it efficient and convenient to conditionally choose between sockets and TLI, as shown below:

```
// Conditionally choose network programming interface.
#if defined (USE_SOCKETS)
typedef SOCK_Acceptor PEER_ACCEPTOR;
typedef SOCK_Stream PEER_STREAM;
#elif defined (USE_TLI)
typedef TLI_Acceptor PEER_ACCEPTOR;
typedef TLI_Stream PEER_STREAM;
#endif /* USE_SOCKET */

typedef Quote_Handler <PEER_STREAM> 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, PEER_ACCEPTOR>
{
public:
  typedef Acceptor <QUOTE_HANDLER, PEER_ACCEPTOR>
          inherited;

  Quote_Acceptor (const PEER_ACCEPTOR::ADDR &ad,
                  Quote_Database *db,
                  Reactor *r)
   : inherited (ad), db_ (db), reactor_ (r) {
     // Register acceptor with the reactor, which
     // calls the get_handle() method) to obtain
     // the passive-mode peer_acceptor_ HANDLE.
     reactor.register_handler (this, READ_MASK);
   }

  // Factory method to create a service handler.
  // This method overrides the base class to
  // pass in the database and Reactor pointers.
  virtual QUOTE_HANDLER *make_svc_handler (void) {
    return new QUOTE_HANDLER (db_, reactor_);
  }

private:
  Quote_Database *db_;
  Reactor *reactor_;
}
```

A more dynamically extensible method of selecting between sockets or TLI can be achieved via inheritance and dynamic binding by using the Abstract Factory or Factory Method patterns described in [8]. An advantage of using parameterized types, however, is that they improve run-time efficiency. For example, parameterized types avoid the overhead of virtual method dispatching and allow compilers to inline frequently accessed methods. The downside, of course, is that template parameters are locked in at compile time, templates can be slower to link, and they usually require more space.

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;

  // Event demultiplexer.
  Reactor reactor;

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

  // Single-threaded event loop that dispatches
  // all events as callbacks to the appropriate
  // Event_Handler subclass object (such as
```

```
  // the Quote_Acceptor or Quote_Handlers).

  for (;;)
    reactor.handle_events ();

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

After the Quote_Acceptor factory has been registered with the Reactor the application goes into an event loop. This loop runs continuously handling client connections, quote requests, and quote responses, all of which are driven by callbacks from the Reactor. Since this application runs as an iterative server in a single thread there is no need for additional locking mechanisms. The Reactor implicitly serializes Event_Handlers at the event dispatching level.

4.2 Evaluating the C++ Wrappers Solution

Using C++ wrappers to implement the quote server is an improvement over the use of sockets, select, and C for the following reasons:

- *Simplify programming* – low-level details of programming sockets (such as initialization, addressing, and handling short-writes and short-reads) can be performed automatically by the IPC_SAP wrappers. Moreover, we eliminate several common programming errors by not using select directly [4].
- *Improve portability* – by shielding applications from platform-specific network programming interfaces. Wrapping sockets with C++ classes (rather than stand-alone C functions) makes it easy to switch wholesale between different network programming interfaces simply by changing the parameterized types to the Acceptor template. Moreover, the code is more portable since the server no longer accesses select directly. For example, the Reactor can be implemented with other event demultiplexing system calls (such as SVR4 UNIX poll, WIN32 WaitForMultipleObjects, or even separate threads) [9].
- *Increase reusability and extensibility* – the Reactor, Quote_Acceptor, and Quote_Handler components are not as tightly coupled as the version in Section 3.1. Therefore, it is 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 adding stock trading functionality), only the implementation of the Quote_Handler class must change.

In addition, C++ features like templates and inlining ensure that these improvements do not penalize performance.

However, even though the C++ wrapper solution is a distinct improvement over the C solution it still has the same drawbacks as the C++ wrapper client solution we presented in our last column: *too much of the code required for the application is not directly related to the stock market*. Moreover, the use of C++ wrappers does not address higher-level communication topics such as object location, object activation, complex marshaling and demarshaling, security, availability and fault tolerance, transactions, and object migration and copying (most of these topics are beyond the scope of this article). To address these issues requires a more sophisticated distributed computing infrastructure. In the following section, we describe and evaluate such a solution based upon CORBA.

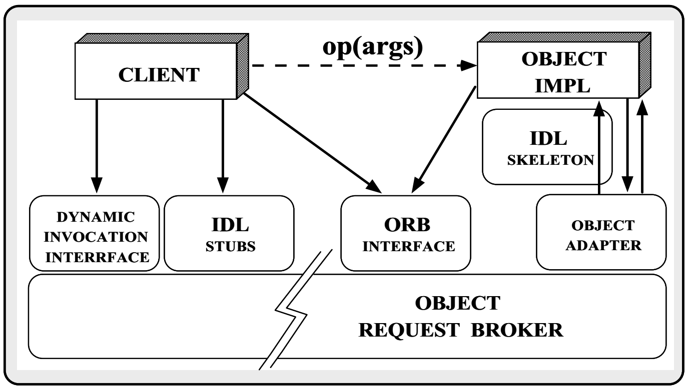

Figure 2: Key Components in the CORBA Architecture

5 The CORBA Solution

Before describing the CORBA-based stock quoter implementation we'll take a look at the key components in the CORBA architecture. In a CORBA environment, a number of components collaborate to allow a client to invoke an operation op with arguments args on an object implementation. Figure 2 illustrates the primary components in the CORBA architecture. These components are described below:

- *Object Implementation* – defines operations that implement an OMG-IDL interface. We implement our examples using C++. However, object implementations can be written in other languages such as C, Smalltalk, Ada95, Eiffel, etc.
- *Client* – this is the program entity that invokes an operation on an object implementation. Ideally, accessing the services of a remote object should be as simple as calling a method on that object, *i.e.,* obj->op(args). The remaining components in Figure 2 support this behavior.
- *Object Request Broker (ORB)* – when a client invokes an operation the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.
- *ORB Interface* – an ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface

provides various helper functions such as converting object references to strings and back, and creating argument lists for requests made through the dynamic invocation interface described below.

- *OMG-IDL stubs and skeletons* – OMG-IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. The OMG-IDL → programming language transformation is automated. Therefore, the potential for inconsistencies between client stubs and server skeletons is greatly reduced.
- *Dynamic Invocation Interface (DII)* – allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests) the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.
- *Object Adapter* – assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles, (*e.g.,* OODB object adapters, library object adapters for non-remote (same-process) objects, etc).

Below, we outline how an ORB supports diverse and flexible object implementations via *object adapters* and *object activation*. We'll cover the remainder of the components mentioned above in future columns.

5.1 Object Adapters

A fundamental goal of CORBA is to support implementation diversity. In particular, the CORBA model allows for diversity of programming languages, OS platforms, transport protocols, and networks. This enables CORBA to encompass a wide-spectrum of environments and requirements.

To support implementation diversity, an ORB should be able to interact with various types and styles of object implementations. It is hard to achieve this goal by allowing object implementations to interact directly with the ORB, however. This approach would require the ORB to provide a very "fat" interface and implementation. For example, an ORB that directly supported objects written in C, C++, and Smalltalk could become very complicated. It would need to provide separate foundations for each language or would need to utilize a least-common-denominator binary object model that made programming in some of the languages unnatural.

By having object implementations plug into *object adapters* (OAs) instead of plugging directly into the ORB, bloated ORBs can be avoided. Object adapters can be specialized to support certain object implementation styles. For example, one object adapter could be developed specifically

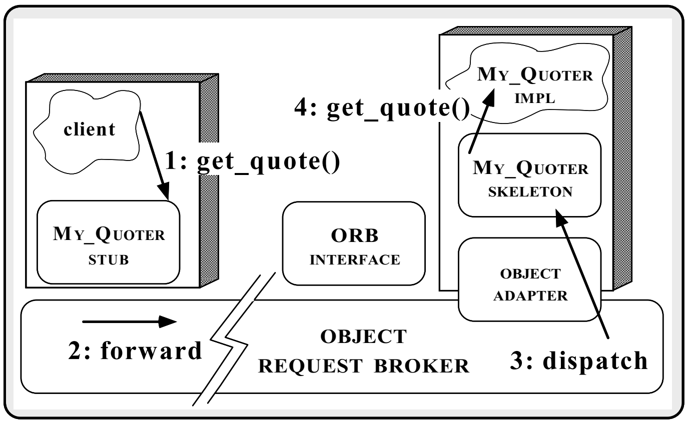

Figure 3: CORBA Request Flow Through the ORB

to support C++ objects. Another object adapter might be designed for OO database objects. Still another object adapter could be created to optimize access to objects located in the same process address space as the client.

Conceptually, object adapters fit between the ORB and the object implementation (as shown in Figure 2). They assist the ORB with delivering requests to the object and with activating the object. By specializing object adapters, ORBs can remain lightweight, while still supporting different types and styles of objects. Likewise, object implementors can choose the object adapter that best suits their development environment and application requirements. Therefore, they incur overhead only for what they use. As mentioned above, the alternative is to cram the ORB full of code to support different object implementation styles. This is undesirable since it leads to bloated and potentially inefficient implementations.

Currently, CORBA specifies only one object adapter: the *Basic Object Adapter* (BOA). According to the specification, the BOA is intended to provide reasonable support for a wide spectrum of object implementations. These range from one or more objects per program to *server-per-method* objects, where each method provided by the object is implemented by a different program. Our stock quoter object implementation below is written in a generic fashion – the actual object implementations and object adapter interfaces in your particular ORB may vary.

5.2 Object Activation

When a client sends a request to an object, the ORB first delivers the request to the object adapter that the object's implementation was registered with. How an ORB locates both the object and the correct object adapter and delivers the request to it depends on the ORB implementation. Moreover, the interface between the ORB and the object adapter is implementation-dependent and is not specified by CORBA.

If the object implementation is not currently "active" the ORB and object adapter activate it before the request is delivered to the object. As mentioned above, CORBA requires that object activation be transparent to the client making the request. A CORBA-conformant BOA must support four different activation styles:

- *Shared server* – Multiple objects are activated into a single server process.
- *Unshared server* – Each object is activated into its own server process.
- *Persistent server* – The server process is activated by something other than the BOA (*e.g.,* a system boot-up script) but still registers with the BOA once it's ready to receive requests.
- *Server-per-method* – Each operation of the object's interface is implemented in a separate server process.

In practice, BOAs provided by commercially-available ORBs do not always support all four activation modes. We'll discuss issues related to the BOA specification in Section 5.4.

Our example server described below is an unshared server since it only supports a single object implementation. Once the object implementation is activated, the object adapter delivers the request to the object's *skeleton*. Skeletons are the server-side analog of client-side stubs[4] generated by an OMG-IDL compiler. The skeleton selected by the BOA performs the callback to the implementation of the object's method and returns any results to the client. Figure 3 illustrates the request flow from client through ORB to the object implementation for the stock quoter application presented below.

5.3 CORBA Code

The server-side CORBA implementation of our stock quote example is based on the following OMG-IDL specification:

```
// OMG-IDL modules are used to avoid polluting
// the application namespace.

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 this section we'll illustrate how a server programmer might implement this OMG-IDL interface and make the object available to client applications.

Our last column illustrated how client programmers obtain and use *object references* supporting the `Quoter` interface to determine the current value of a particular `stock_name`. Object references are opaque, immutable "handles" that uniquely identify objects. A client application must somehow obtain an object reference to an object implementation before it can invoke that object's operations. An object implementation is typically assigned an object reference when it registers with its object adapter.

[4]Stubs are also commonly referred to as "proxies" or "surrogates."

ORBs supporting C++ object implementations typically provide a compiler that automatically generates server-side skeleton C++ classes from IDL specifications (*e.g.,* the Quoter interface). Programmers then integrate their implementation code with this skeleton using inheritance or object composition. The `My_Quoter` implementation class shown below is an example of inheritance-based skeleton integration:

```
// 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)
    throw (Stock::Invalid_Stock) {
    long value =
      db_->lookup_stock_price (stock_name);
    if (value == -1)
      throw Stock::Invalid_Stock();
    return value;
  }

private:
  // Keep a pointer to a quote database.
  Quote_Database *db_;
};
```

`My_Quoter` is our object implementation class. It inherits from the `Stock::QuoterBOAImpl` 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. Since we are single-threaded we don't need to acquire any locks to access object state like `db_`.

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 throw a `Stock::Invalid_Stock` exception.

The implementation of `get_quote` shown above uses C++ exception handling (EH). However, EH is still not implemented by all C++ compilers. Thus, many commercial ORBs currently use special status parameters of type `CORBA::Environment` to convey exception information. An alternative implementation of `My_Quoter::get_quote` could be written as follows using a `CORBA::Environment` parameter:

```
long
My_Quoter::get_quote (const char *stock_name,
                      CORBA::Environment &ev)
{
  long value =
    db_->lookup_stock_price (stock_name);
  if (value == -1)
```

```
    ev.exception (new Stock::Invalid_Stock);
  return value;
}
```

This code first attempts to look up the stock price. If that fails it sets the exception field in the `CORBA::Environment` to a `Stock::Invalid_Stock` exception. A client can also use `CORBA::Environment` parameters instead of C++ EH. In this case the client is obligated to check the `Environment` parameter after the call returns before attempting to use any values of `out` and `inout` parameters or the return value. These values may be meaningless if an exception is raised.

If the client and object are in different address spaces, they don't need to use the same exception handling mechanism. For example, a client on one machine using C++ EH can access an object on another machine that was built to use `CORBA::Environment` parameters. The ORB will make sure they interoperate correctly and transparently.

The main program for our quote server initializes the ORB and the BOA, defines an instance of a `My_Quoter`, and tells the BOA it is ready to receive requests by calling `CORBA::BOA::impl_is_ready`, as follows:

```
// Include standard BOA definitions.
#include <corba/orb.hh>

// Pointer to online stock quote database.
extern Quote_Database *quote_db;

int main (int argc, char *argv[])
{
  // Initialize the ORB and the BOA.
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);
  CORBA::BOA_var boa = orb->boa_init (argc, argv, 0);

  // Create an object implementation.
  My_Quoter quoter (quote_db);

  // Single-threaded event loop that handles CORBA
  // requests by making callbacks to the user-supplied
  // object implementation of My_Quoter.
  boa->impl_is_ready ();
  /* NOTREACHED */
  return 0;
}
```

After the executable is produced by compiling and linking this code it must be registered with the ORB. This is typically done by using a separate ORB-specific administrative program. Normally such programs let the ORB know how to start up the server program (*i.e.,* which activation mode to use and the pathname to the executable image) when a request arrives for the object. They might also create and register an *object reference* for the object. As illustrated in our last column, and as mentioned above, clients use object references to access object implementations.

5.4 Evaluating the CORBA Solution

The CORBA solution illustrated above is similar to the C++ wrappers solution shown in Section 4. For instance, both approaches use a callback-driven event-loop structure. However, the amount of effort required to maintain, extend, and port the CORBA version of the stock quoter application should be less than the C sockets and C++ wrappers versions. This reduction in effort occurs since CORBA raises the level of abstraction at which our solution is developed. For example, the ORB handles more of the lower-level communication-related tasks. These tasks include automated stub and skeleton generation, marshalling and demarshalling, object location, object activation, and remote method invocation and retransmission. This allows the server-side of the CORBA solution to focus primarily on application-related issues of looking up stock quotes in a database.

The benefits of CORBA become more evident when we extend the quote server to support concurrency. In particular, the effort required to transform the CORBA solution from the existing iterative server to a concurrent server is minimal. The exact details will vary depending on the ORB implementation and the desired concurrency strategy (*e.g.,* thread-per-object, thread-per-request, etc.). However, most multi-threaded versions of CORBA (such as MT Orbix [10]) require only a few extra lines of code. In contrast, transforming the C or C++ versions to concurrent servers will require more work. A forthcoming column will illustrate the different strategies required to multi-thread each version.

Our previous column also described the primary drawbacks to using CORBA. Briefly, these drawbacks include the high learning curve for developing and managing distributed objects effectively, performance limitations [11], as well as the lack of portability and security. One particularly problematic drawback for servers is that the BOA is not specified very thoroughly by the CORBA 2.0 specification [12].

The BOA specification is probably the weakest area of CORBA 2.0. For example, the body of the `My_Quoter::get_quote` method in Section 5.3 is mostly portable. However, the name of the automatically-generated skeleton base class and the implementation of `main` remain very ORB-specific. Our implementation assumed that the constructor of the `Stock::QuoterBOAImpl` base skeleton class registered the object with the BOA. Other ORBs might require an explicit object registration call. These differences between ORBs exist because registration of objects with the BOA is not specified at all by CORBA 2.0.

The OMG ORB Task Force is well aware of this problem and has issued a Request For Proposals (RFP) asking for ways to solve it. Until it's solved (probably mid-to-late 1996), the portability of CORBA object implementations between ORBs will remain problematic.

6 Concluding Remarks

In this column, we examined several different programming techniques for developing the server-side of a distributed stock quote application. Our examples illustrated how the CORBA-based distributed object computing (DOC) solution simplifies programming and improves extensibility. It achieves these benefits by relying on an ORB infrastructure that supports communication between distributed objects.

A major objective of CORBA is to let application developers focus primarily on application requirements, without devoting as much effort to the underlying communication infrastructure. As applications become more sophisticated and complex DOC frameworks like CORBA become essential to produce correct, portable, and maintainable distributed systems.

CORBA is one of several technologies that are emerging to support DOC. In future articles, we will discuss other OO toolkits and environments (such as OODCE and OLE/COM) and compare them with CORBA in the same manner that we compared sockets and C++ wrappers to CORBA. In addition, we will compare the various distributed object solutions with more conventional distributed programming toolkits (such as Sun RPC and OSF DCE).

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 Ron Resnick and Barry Keepence for comments on this column.

References

[1] D. C. Schmidt, "A Domain Analysis of Network Daemon Design Dimensions," *C++ Report*, vol. 6, March/April 1994.

[2] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[3] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol III: Client – Server Programming and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[4] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

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

[6] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[7] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

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

[9] D. C. Schmidt and P. Stephenson, "Using Design Patterns to Evolve System Software from UNIX to Windows NT," *C++ Report*, vol. 7, March/April 1995.

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

[11] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

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