Object Interconnections

Comparing Alternative Client-side Distributed Programming Techniques (Column 3)

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

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

This column appeared in the May 1995 issue of the SIGS C++ Report magazine.

1 Introduction

This month's column examines and evaluates various programming techniques for developing distributed applications. Our previous column outlined the requirements for a representative distributed application in the financial services domain. This application enables investment brokers to query the price of a stock, as well as to buy shares of stock. As shown in Figure 1, the quote service that maintains the current stock prices is physically remote from brokers, who work in geographically distributed sites. Therefore, our application must work efficiently, robustly, and securely across various wide area (WAN) and local area (LAN) networks.

We selected the stock trading application because the issues raised by analyzing, designing, and implementing it are representative of the issues that arise when developing many other types of distributed applications. Some issues we identified in our previous column were *platform heterogeneity, high system reliability and availability, flexibility of object location and selection, support for transactions, security, and deferred process activation, and the exchange of binary data between different computer architectures.*

After identifying many requirements and problems we must tackle to create a distributed stock trading application, it's time to look at some actual code. This month's column examines various distributed programming techniques used for the client-side of the stock trading application (our next column will explore server-side techniques). Below, we compare several solutions that range from using the socket network programming interface (which is defined using the C programming language), to using C++ wrappers for sockets, all the way up to using a distributed object computing (DOC) solution. The DOC solution is based upon the CORBA interface definition language (IDL) and a CORBA Object Request Broker (ORB). Each solution illustrates various tradeoffs between extensibility, robustness, portability, and efficiency.

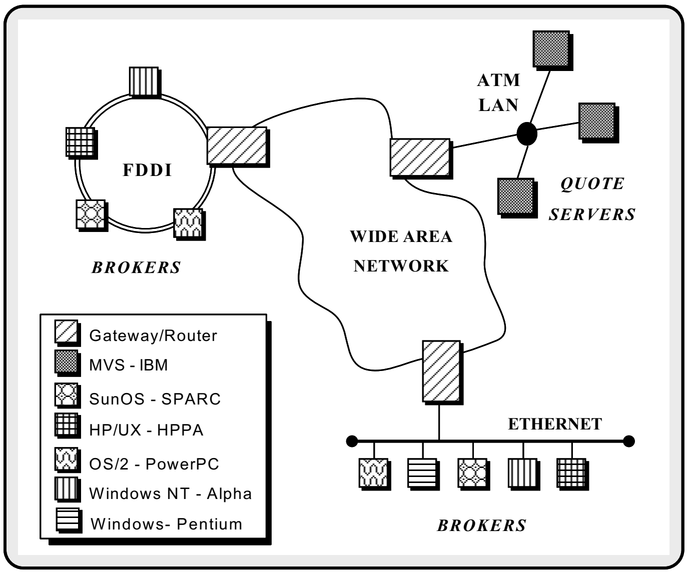

Figure 1: Distributed Architecture of Financial Services System

2 The Socket Client Solution

Distributed applications have traditionally been written using network programming interfaces such as sockets or TLI. Sockets were developed in BSD UNIX to interface with the TCP/IP protocol suite [1]. The "Transport Layer Interface" (TLI) is another network programming interface available on System V UNIX platforms. Our primary focus in this article is on sockets since it is widely available on many platforms, including most variants of UNIX, Windows, Windows NT, OS/2, Mac OS, etc.

From an application's perspective, a socket is an endpoint of communication that is bound to the address of a service. The service may reside on another process or on another computer in a network. A socket is accessed via an I/O descriptor, which is an unsigned integer handle that indexes into a table of open sockets maintained by the OS.

The standard socket interface is defined using C functions. It contains several dozen routines that perform tasks such as locating address information for network services, establishing and terminating connections, and sending and receiving data. In-depth coverage of sockets and TLI appears in [2].

2.1 Socket/C Code

The following code illustrates the relevant steps required to program the client-side of the stock quote program using sockets and C. We first create two C structures that define the schema for the quote request and quote response, respectively:

```
#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 error value. */
};
```

Next, we've written a number of C utility routines. These routines shield the rest of the application from dealing with the low-level socket API. The first routine actively establishes a connection with a stock quote server at a port number passed as a parameter to the routine:

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

HANDLE connect_quote_server (const char server[],
                             u_short port)
{
  struct sockaddr_in addr;
  struct hostent *hp;
  HANDLE sd;

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

  /* Determine IP address of the server */
  hp = gethostbyname (server);

  /* Setup the address of server. */
  memset ((void *) &addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port = htons (port);
  memcpy (&addr.sin_addr, hp->h_addr, hp->h_length);

  /* Establish connection with remote server. */
  connect (sd,
           (struct sockaddr *) &addr, sizeof addr);
  return sd;
}
```

Even though we've omitted most of the error handling code, the routine shown above illustrates the many subtle details required to program at the socket level.

The next routine sends a stock quote request to the server:

```
void send_request (HANDLE sd,
                   const char stock_name[])
{
  struct Quote_Request req;
  size_t w_bytes;
  size_t packet_len;
  int    n;

  /* Determine the packet length. */
  packet_len = strlen (stock_name);
  if (packet_len > MAXSTOCKNAMELEN)
    packet_len = MAXSTOCKNAMELEN;
  strncpy (req.name, stock_name, packet_len);
  /* Convert to network byte order. */
  packet_len += sizeof req.len;
  req.len = htonl (packet_len);

  /* Send data to server, handling "short-writes". */
  for (w_bytes = 0; w_bytes < packet_len; w_bytes += n)
    n = send (sd, ((const char *) &req) + w_bytes,
              packet_len - w_bytes, 0);
}
```

Since the length field is represented as a binary number the `send_request` routine must convert the message length into network byte order. The example uses stream sockets, which are created via the `SOCK STREAM` socket type directive. This choice requires the application code to handle "short-writes" that may occur due to buffer constraints in the OS and transport protocols.[1] To handle short-writes, the code loops until all the bytes in the request are sent to the server.

The following `recv_response` routine receives a stock quote response from the server. If the server couldn't perform the request properly it passes back an `errno` value > 0 to indicate the problem. Otherwise, the function convents the numeric value of the stock quote into host byte order and returns the value to the caller.

```
int recv_response (HANDLE sd, long *value)
{
  struct Quote_Response res;

  recv (sd, (char*) &res, sizeof res, 0);
  /* Convert to host byte order */

  /* Check for failure. */
  errno = ntohl (res.errno);
  if (errno > 0)
    return -1;
  else { /* Success! */
    *value = ntohl (res.value);
    return 0;
  }
}
```

The `print_quote` routine shown below uses the C utility routines defined above to establish a connection with the server. The host and port addresses are passed by the caller. After establishing a connection, the routine requests the server to return the current value of the designated `stock_name`.

```
void print_quote (const char server[],
                  u_short port,
                  const char stock_name[])
{
  HANDLE sd;
  long value;

  sd = connect_quote_server (server, port);
  send_request (sd, stock_name);
  if (recv_response (sd, &value) != -1)
    display ("value of %s stock = $%ld\n",
             stock_name, value);
}
```

[1] Sequence packet sockets (`SOCK_SEQPACKET`) could be used to preserve message boundaries, but this type of socket is not available on many operating systems.

This routine would typically be compiled, linked into an executable program, and called as follows:

```
print_quote ("quotes.nyse.com", 5150, "ACME ORBs");
/* Might print: "value of ACME ORBs stock = $12" */
```

2.2 Evaluating the Socket Solution

Sockets are a relatively low-level interface. As illustrated in the code above, programmers must explicitly perform the following tedious and potentially error-prone activities:

• **Determining the addressing information for a service:** The service addressing information in the example above would be inflexible if the user must enter the IP address and port number explicitly. Our socket code provides a glimmer of flexibility by using the `gethostbyname` utility routine, which converts a server name into its IP number. A more flexible scheme would automate service location by using some type of name service or location broker.

• **Initializing the socket endpoint and connecting to the server:** As shown in the `connect_quote_server` routine, socket programming requires a non-trivial amount of detail to establish a connection with a service. Moreover, minor mistakes (such as forgetting to initialize a socket address structure to zero) will prevent the application from working correctly.

• **Marshaling and unmarshaling messages:** The current example exchanges relatively simple data structures. Even so, the solution we show above will not work correctly if compilers on the client and server hosts align fields in structures differently. It also won't work if `sizeof (long)` is a different value on the client and the server. In general, developing more complicated applications using sockets requires significant programmer effort to marshal and unmarshal complex messages that contain arrays, nested structures, or floating point numbers. In addition, developers must ensure that clients and servers don't get out of sync as changes are made.

• **Sending and receiving messages:** The code required to send and receive messages using sockets is subtle and surprisingly complex. The programmer must explicitly detect and handle many error conditions (such as short-writes), as well as frame and transfer record-oriented messages correctly over bytestream protocols such as TCP/IP.

• **Error detection and error recovery:** Another problem with sockets is that they make it hard to detect accidental type errors at compile-time. Socket descriptors are "weakly-typed," *i.e.,* a descriptor associated with a connection-oriented socket is not syntactically different from a descriptor associated with a connectionless socket. Weakly-typed interfaces increase the potential for subtle run-time errors since a compiler cannot detect using the wrong descriptor in the wrong circumstances. To save space, we omitted much of the error handling code that would normally exist. In a production system, a large percentage of the code would be dedicated to providing robust error detection and error recovery at run-time.

• **Portability:** Another limitation with the solution shown above is that it hard-codes a dependency on sockets into the source code. Porting this code to a platform without sockets (such as early versions of System V UNIX) will require major changes to the source.

• **Secure communications:** A real-life stock trading service that did not provide secure communications would not be very useful, for obvious reasons. Adding security to the sockets code would exceed the capabilities of most programmers due to the expertise and effort required to get it right.

3 The C++ Wrappers Client Solution

Using C++ wrappers (which encapsulate lower-level network programming interfaces such as sockets or TLI within a type-safe, object-oriented interface) is one way to simplify the complexity of programming distributed applications. The C++ wrappers shown below are part of the `IPC SAP` interprocess communication class library described in [3]. `IPC SAP` encapsulates both sockets and TLI with C++ class categories.

3.1 C++ Wrapper Code

Rewriting the `print_quote` routine using C++ templates simplifies and generalizes the low-level C code in the `connect_quote_server` routine, as shown below:

```
template <class CONNECTOR, class STREAM, class ADDR>
void print_quote (const char server[],
                  u_short port,
                  const char stock_name[])
{
  // Data transfer object.
  STREAM peer_stream;

  // Create the address of the server.
  ADDR addr (port, server);

  // Establish a connection with the server.
  CONNECTOR con (peer_stream, addr);

  long value;

  send_request (peer_stream, stock_name);
  if (recv_response (peer_stream, &value) != -1)
    display ("value of %s stock = $%ld\n",
             stock_name, value);
}
```

The template parameters in this routine may be instantiated with the `IPC SAP` C++ wrappers for sockets as follows:

```
print_quote<SOCK_Connector, SOCK_Stream, INET_Addr>
           ("quotes.nyse.com", 5150, "ACME ORBs");
```

`SOCK Connector` shields application developers from the low-level details of establishing a connection. It is a factory

[4] that connects to the server located at the INET Addr address and produces a SOCK Stream object when the connection completes. The SOCK Stream object performs the message exchange for the stock query transaction and handles short-writes automatically. The INET Addr class shields developers from the tedious and error-prone details of socket addressing shown in Section 2.1.

The template routine may be parameterized by different types of IPC classes. Thus, we solve the portability problem with the socket solution discussed in Section 2.1. For instance, only the following minimal changes are necessary to port our application from an OS platform that lacks sockets, but that has TLI:

```
print_quote<TLI_Connector, TLI_Stream, INET_Addr>
           ("quotes.nyse.com", 5150, "ACME ORBs");
```

Note that we simply replaced the SOCK* C++ class wrappers with TLI* C++ wrappers that encapsulate the TLI network programming interface. The IPC SAP wrappers for sockets and TLI offers a conformant interface. Template parameterization is a useful technique that increases the flexibility and portability of the code. Moreover, parameterization does not degrade application performance since template instantiation is performed at compile-time. In contrast, the alternative technique for extensibility using inheritance and dynamic binding exacts a run-time performance penalty in C++ due to virtual function table lookup overhead.

The send_request and recv_request routines also may be simplified by using C++ wrappers that handle short-writes automatically, as illustrated in the send_request template routine below:

```
template <class STREAM>
void send_request (STREAM &peer_stream,
                   const char stock_name[])
{
  // Constructor does the dirty work...
  Quote_Request req (stock_name);

  // send_n() handles the "short-writes"
  peer_stream.send_n (&req, req.length ());
}
```

3.2 Evaluating the C++ Wrappers Solution

The IPC SAP C++ wrappers is an improvement over the use of sockets and C for several reasons. First, they help to automate and simplify certain aspects of using sockets (such as initialization, addressing, and handling short-writes). Second, they improve portability by shielding applications from platform-specific network programming interfaces. Wrapping sockets with C++ classes (rather than stand-alone C functions) makes it convenient to switch wholesale between different IPC mechanisms by using parameterized types. In addition, by combining inline functions and templates, the C++ wrappers do not introduce any measurable overhead compared with programming with socket directly.

However, C++ wrappers and sockets both suffer from the same costly drawback: *too much of the code required for the application has nothing at all to do with the stock market.* Moreover, unless you already have a C++ wrapper library like IPC SAP, developing an OO communication infrastructure to support the stock quote application is prohibitively expensive. For one thing, stock market domain experts may not know anything at all about sockets programming. Developing an OO infrastructure either requires them to divert their attention to learning about sockets and C++ wrappers, or requires the hiring of people familiar with low-level network programming. Each solution would typically delay the deployment of the application and increase its overall development and maintenance cost.

Even if the stock market domain experts learned to program at the socket or C++ wrapper level, it is inevitable that the requirements for the system would eventually change. For example, it might become necessary to combine the stock quote system with a similar, yet separately developed, system for mutual funds. These changes may require modifications to the request/response message schema. In this case, both the original solution and the C++ wrappers solution would require extensive modifications. Moreover, if the communication infrastructure of the stock quote system and the mutual funds system were each custom-developed, interoperability between the two would very likely prove impossible. Therefore, one or both of the systems would have to be rewritten extensively before they could be integrated.

In general, a more practical approach may be to utilize a distributed object computing (DOC) infrastructure built specifically to support distributed applications. In the following section, we motivate, describe, and evaluate such a DOC solution based upon CORBA. In subsequent columns, we'll examine solutions based on other DOC tools and environments (such as OODCE [5] and OLE/COM [6]).

4 The CORBA Client Solution

4.1 Overview of CORBA

As described in [7], an Object Request Broker (ORB) is a system that supports distributed object computing in accordance with the OMG CORBA specification (currently CORBA 1.2 [8], though major pieces of CORBA 2.0 have already been completed). CORBA delegates much of the tedious and error-prone complexity associated with developing distributed applications to its reusable infrastructure. Application developers are then freed to focus their knowledge of the domain upon the problem at hand.

To invoke a service using CORBA, an application only needs to hold a reference to a target object. The ORB is responsible for automating other common communication infrastructure activities. These activities include locating a suitable target object, activating it if necessary, delivering the request to it, and returning any response back to the caller. Parameters passed as part of the request or response are automatically and transparently marshaled by the ORB. This marshaling process ensures correct interworking between applications and objects residing on different computer architectures.

CORBA object interfaces are described using an Interface Definition Language (IDL). CORBA IDL resembles C++ in many ways, though it is much simpler. In particular, it is not a full-fledged programming language. Instead, it is a declarative language that programmers use to define object interfaces, operations, and parameter types.

An IDL compiler automatically translates CORBA IDL into client-side "stubs" and server-side "skeletons" that are written in a full-fledged application development programming language (such as C++, C, Smalltalk, or Modula 3). These stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. Since the IDL $\rightarrow$ programming language transformation is automated, the potential for inconsistencies between client stubs and server skeletons is reduced significantly.

4.2 CORBA Code

The following is a CORBA IDL specification for the stock quote system:

```
module Stock {
  exception Invalid_Stock {};

  interface Quoter {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };
};
```

The `Quoter` interface supports a single operation, `get_quote`. Its parameter is specified as an `in` parameter, which means that it is passed from the client to the server. IDL also permits `inout` parameters that are passed from the client to the server and back to the client, and `out` parameters that originate at the server and are passed back to the client. When given the name of a stock as an input parameter, `get_quote` either returns its value as a `long` or throws an `Invalid_Stock` exception. Both `Quoter` and `Invalid_Stock` are scoped within the `Stock` module to avoid polluting the application namespace.

A CORBA client using the standard OMG Naming service to locate and invoke an operation on a `Quoter` object might look as follows[2]:

```
// Introduce components into application namespace.
using namespace CORBA;
using namespace CosNaming;
using namespace Stock;

// Forward declaration.
Object_ptr bind_service (int argc, char *argv[],
                         const Name &service_name);

int main (int argc, char *argv[])
{
  // Create desired service name
  const char *name = "Quoter";
  Name service_name;
  service_name.length(1);
  service_name[0].id = name;

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

  int result = 1;

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

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

    long value = q->get_quote (stock_name);
    cout << "value of " << stock_name
         << " = $"
         << value << endl;
    result = 0; // Success!
  } catch (CORBA::BAD_PARAM) {
    cerr << "_narrow() failed: "
         << service_name
         << " is not a Quoter!";
  } catch (Invalid_Stock &) {
    cerr << stock_name
         << " is not a valid stock name!\n";
  }
  return result;
}
```

[2]Note the use of the `using namespace` construct in this code to introduce scoped names into the application namespace; those unfamiliar with this construct should consult Stroustrup's "Design and Evolution of C++" (Addison-Wesley, 1994, ISBN 0-201-54330-3) for more details.

This application binds to the stock quote service, asks it for the value of ACME ORBs, Inc. stock, and prints out the value if everything works correctly. Several steps are required to accomplish this task. First, a `CosNaming::Name` structure representing the name of the desired service must be created. A `CosNaming::Name` is a `sequence` (which are essentially dynamically-sized arrays) of `CosNaming::NameComponents`, each of which is a `struct` containing two strings members, `id` and `kind`. In our application, we're only using the `id` member, which we set to the string "Quoter," the name of our service. Before the object reference returned from our utility routine `bind_service` can be used as a `Quoter`, it must be narrowed to the `Quoter` interface.

Narrowing to a derived interface is similar to using the C++ `dynamic_cast<T>` operator to downcast from a pointer to a base class to a pointer to a derived class. In the OMG C++ language mapping, the result of a successful narrow operation is a new object reference that is statically typed to match the requested derived interface. ORBs that support C++ exception handling will throw a `CORBA::BAD_PARAM` exception if the actual type of the object reference being narrowed isn't consistent with the requested type.

If the `_narrow` succeeds, the `get_quote` operation is called inside a `try` block. If an exception occurs at this point, the `catch` block for the `Invalid_Stock` exception prints a suitable error message and exits. If no exception is thrown, the value returned is displayed on the standard output as the current value of the stock of ACME ORBs, Inc.

To simplify our application, the details of initializing the ORB and looking up object references in the Naming service have been hidden inside the `bind_service` utility function. Object references are opaque, immutable "handles" that uniquely identify objects, and all object implementations must have one before they can be accessed by client applications. Here's how the client might be implemented:

```
Object_ptr bind_service (int argc, char *argv[],
                         const Name &service_name)
{
  // Get reference to name service.
  ORB_var orb = ORB_init (argc, argv, 0);
  Object_var obj =
    orb->resolve_initial_references ("NameService");
  NamingContext_var name_context =
    NamingContext::_narrow (obj);

  // Find object reference in the name service.
  return name_context->resolve (service_name);
}
```

To obtain an object reference to the Naming service, `bind_service` must first obtain a reference to the ORB. It accomplishes this by calling `ORB_init`. This is a standard routine defined in the CORBA namespace – it returns an ORB object reference. Using this object reference, the application then invokes `resolve_initial_references`. This routine acts as a miniature name service provided by the ORB for certain well-known object references. From this it obtains an object reference to the name service. The `resolve_initial_references` call returns an object reference of type `CORBA::Object` (the base interface of all IDL interfaces). Therefore, the return value must be narrowed to the more derived interface type, `CosNaming::NamingContext`, before any of the Naming operations can be invoked on it.

Once the `NamingContext` object reference has been obtained, the `service_name` argument is passed to the `resolve` operation on the `NamingContext`. Assuming the name is resolved successfully, the `resolve` operation returns to the caller an object reference of type `CORBA::Object` for the `Quoter` service.

4.3 Evaluating the CORBA Solution

The example above illustrates how the client-side code deals directly with the application-related issues of obtaining stock quotes, rather than with the low-level communication-related issues. Therefore, the amount of effort required to extend and port this application will be reduced. This should not come as a surprise, since the CORBA solution significantly raises the level of abstraction at which the solution is developed.

Of course, the CORBA solution is not perfect. The use of CORBA has several potential drawbacks that are important to understand and evaluate carefully before committing to use it on a commercial project:

• **Learning Curve:** The level of abstraction at which our CORBA solution is developed is much higher than that of the socket-based solution. However, CORBA does not totally relieve the stock market domain expert of being able to program DOC software in C++. CORBA introduces a range of new concepts (such as object references, proxies, and object adapters), components and tools (such as interface definition languages, IDL compilers, and object-request brokers), and features (such as exception handling and interface inheritance). Depending on developer experience, it may take a fair amount of time to ramp-up to using CORBA productively. As with any other software tool, the cost of learning the new technology must be amortized over time and/or successive projects.

• **Interoperability and Portability:** Interoperability between different ORBs has traditionally been a major problem with CORBA. This problem was solved recently when the OMG approved an Interoperability protocol [9]. However, few if any ORBs actually implement the Interoperability protocol at this time. Therefore, interoperability will remain a real problem for CORBA-based applications in the near future.

Likewise, portability of applications from ORB to ORB will be limited until conformance becomes more commonplace. The OMG just recently approved the IDL-to-C++ language mapping, the Naming service, and the ORB Initialization service (*e.g.,* `ORB_init` and `resolve_initial_references`). However, like the standard Interoperability protocol mentioned above, at this time few if any commercially-available ORBs actually provide services conforming to these standards.

• **Security:** Any ORB hoping to serve as the distributed computing infrastructure for a real stock trading system must address the need for security within the system. Unfortunately, few if any of the CORBA ORBs available today address this issue, since the OMG has not yet standardized on a Security Service. However, given that the OMG Object Services Task Force is currently evaluating several proposals for such a service, a standard OMG security specification should be available by late 1995 or early 1996.

• **Performance:** The performance of the stock quote application may not be as good as that of the socket-based or C++ wrapper-based applications. In particular, the ORB is not tuned specifically to this application. In addition, we are accessing the Naming service, which probably requires one or more remote invocations of its own.

For large-scale distributed software systems, the small loss in micro-level efficiency is often more than made up for by the increased extensibility, robustness, maintainability, and macro-level efficiency. In particular, a well designed CORBA implementation may actually improve performance by recognizing the context in which a service is accessed and automatically applying certain optimizations.

For example, if an ORB determines that a requestor and a target object are co-located in the same address space, it may eliminate marshaling and IPC and simply invoke the target object's method directly. Likewise, if the ORB determines that the requestor and target object are located on the same host machine, it may suppress marshaling and pass parameters via shared memory rather than using a message-passing IPC mechanism. Finally, even if the requestor and target object are on different machines, an ORB may optimize marshaling if it recognizes that the requestor and target host support the same byte ordering.

All the optimizations listed above may be performed automatically without requiring a developer to modify the application. In general, manually programming this degree of flexibility using sockets, C, or C++ would be too time-consuming to justify the development effort.

5 Coping with Changing Requirements

Designing software that is resilient to change is a constant challenge for developers of large-scale systems. A primary motivation for DOC is to simplify the development of flexible and extensible software. Software with these two qualities adapts more easily to inevitable changes in requirements and environments during the lifetime of applications in large distributed systems.

A major benefit of using CORBA rather than sockets or C++ wrappers is revealed when application requirements change. For example, imagine that after deploying the first version of the stock quote application, the customer requests certain requirement changes described below.

5.1 Adding New Features

New features are inevitably added to successful software applications. For instance, end-users of the stock quote application might request additional query operations, as well as the ability to place a trade (*i.e.*, to automatically buy shares of stock) along with determining the current value.

Many new features will modify the request and response formats. For example, additional information may be returned in a query, such as the percentage that the stock has risen or fallen in value since the start of the day and the volume of trading that has taken place (*i.e.*, number of shares traded).

In a DOC framework that provides an interface definition language (such as CORBA or DCE), making these changes is straightforward. For example, changing the information provided by the service simply adds additional parameters to an operation's signature, as follows:

```
interface Quoter {
  long get_quote (in string stock_name,
                  out double percent_change,
                  out long trading_volume)
    raises (Invalid_Stock);
};
```

In contrast, adding new parameters to the original socket or C++ wrapper solution requires many tedious changes to be performed manually. For example, the `struct` defining the request format must change, necessitating a rewrite of the marshaling code. This modification may introduce inconsistencies into the source code that cause run-time failures. In addition, handling the marshaling and unmarshaling of the floating point `percent_change` parameter can be tricky.

Format changes (such as the adding parameters to methods) typically require recompiling both client and server software in many ORB development environments. Often, this is undesirable since tracking down all the deployed binaries may be hard. In addition, it may not be possible to take the system down for an upgrade. Therefore, a less obtrusive method for managing changes involves creating new interfaces. For example, rather than adding parameters as shown above, a `get_stats` operation could simply be added to a new derived interface:

```
interface Stat_Quoter
          : Quoter // a Stat_Quoter IS-A Quoter
{
  void get_stats (in string stock_name,
                  out double percent_change,
                  out long trading_volume)
    raises (Invalid_Stock);
};
```

CORBA's support for interface inheritance enables it to satisfy the "open/closed" principle of OO library design [10]. By using inheritance, existing clients may continue using the old interface (*i.e.,* existing library components are "closed," which ensures backwards compatibility). Conversely, clients requiring the new features and services use the new one (*i.e.,* the library components are "open" to future extensions).

As an example of adding a trading interface, we could define a new CORBA IDL interface called `Trader` to the `Stock` module:

```
interface Trader {
  void buy  (in string name,
             inout long num_shares,
             in long max_value)
    raises (Invalid_Stock);

  void sell (in string name,
             inout long num_shares,
             in long min_value)
    raises (Invalid_Stock);
};
```

The `Trader` interface provides two methods, `buy` and `sell`, that are used to trade shares of stock with other brokers.

By using CORBA IDL's support for multiple inheritance, an interface describing a full service broker might then be defined in the `Stock` module as follows:

```
interface Broker : Stat_Quoter, Trader {};
```

The `Broker` interface now supports all the operations of both the `Stat Quoter`, `Quoter`, and `Trader` interfaces.

Adding this functionality to either the C or C++ socket solution would probably require extensive changes to all existing code to incorporate the new features. For example, it would be necessary to define several new request and response message formats. In turn, these changes would require modifying and recompiling the client and server applications.

The the OMG-IDL `interface` subclassing solution we've shown above allows old client to deal with new servers. However, this solution by itself is inadequate for handling large-scale, enterprise-wide versioning. There are two limitations with the approach we show:

1. *Allowing new clients to interoperate with old servers –* It a new client has a proxy for a `Broker` class it won't be able to interoperate correctly with a server that only

implements the Stats_Quoter. An ORB will be able to detect this problem, however, and refuse to give out an object reference when the new client attempts to bind with an object of the Broker interface.

2. *Managing the application configuration* – the inheritance-based solution is rather tightly coupled to OMG-IDL, and requires too much manual intervention on the part of system administrators. A comprehensive set of tools and conventions is crucial to maintain the integrity and consistency of components in a large-scale distributed system.

5.2 Improving Existing Features

In addition to adding new features, let's consider changes occurring after extensive day-to-day usage and performance benchmarking of the trading application. Based on experience and end-user feedback, the following changes to existing features might be proposed:

• **Server location independence:** The socket and C++ wrapper code shown in Section 2.1 and 3.1 "hard-codes" the server name and port number of the service into the application. However, the application can be much more flexible if it delays binding the name to the service until run-time.

Run-time binding to the service can be accomplished in CORBA by a client locating the object reference of the service using a *Naming service* or a *Trader service*:

- A Naming service manages a hierarchy consisting of pairs of names and object references. The desired object reference can be found if its name is known. An example of this type of name service is the CosNaming name service used in the CORBA example shown above.
- A Trader service can locate a suitable object given a set of attributes for the object, such as supported interface(s), average load and response times, or permissions and privileges.

Run-time binding allows the application to locate and utilize the server with the lightest load, or the closest server in order to minimize network transmission delays. In addition, once the Naming or Trading services are developed, debugged, and deployed they can be reused by subsequent distributed applications.

• **Bulk requests:** rather than sending each quote request individually, it may be much more efficient to send an entire sequence of requests and receive a sequence of responses in order to minimize network traffic.

In CORBA, this type of optimization may be expressed succinctly using CORBA IDL sequences. While retaining backwards compatibility, we can extend our Quoter interface to incorporate this change using CORBA IDL inheritance and IDL sequences as follows:

```
interface Bulk_Quoter
  : Stat_Quoter // A Bulk_Quoter IS-A Stat_Quoter
{
  typedef sequence<string> Names;
  struct Stock_Info {
    string          name;
    long            value;
    double          change;
    long            volume;
  };
  typedef sequence<Stock_Info> Info;

  exception No_Such_Stock {
    Names stock; // List of invalid stock names
  };

  void bulk_quote (in Names stock_names,
                   out Info stock_info)
    raises (No_Such_Stock);
};
```

Notice how CORBA exceptions may contain user-defined fields that provide additional information about the causes of a failure. For example, in the Bulk Quoter class the No_Such_Stock exception contains a sequence of strings indicating which stock names were invalid.

6 Concluding Remarks

In this column, we examined several different techniques for developing the client-side of a distributed stock trading application. In general, the example illustrates how the CORBA-based DOC solution improves extensibility and robustness by relying on an ORB infrastructure built to support communication between distributed objects without unduly compromising efficiency. Relying on an ORB in this manner is not unlike relying on a good general-purpose library (such as as the C++ Standard Templates Library [11]) for non-distributed C++ applications. The ORB allows the application developer to focus mainly on the application and not worry nearly as much about the infrastructure required to support it.

Note that CORBA is only one of several key 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 to CORBA. Before we do that, though, we need to discuss various aspects of the server-side of our financial services application, which we will tackle in our next column. The server-side implements the various methods defined in the CORBA IDL interfaces.

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 of BNR for comments on improving this column.

References

[1] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

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

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

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

[5] J. Dilley, “OODCE: A C++ Framework for the OSF Distributed Computing Environment,” in *Proceedings of the Winter Usenix Conference*, USENIX Association, January 1995.

[6] Microsoft Press, Redmond, WA, *Object Linking and Embedding Version 2 (OLE2) Programmer's Reference, Volumes 1 and 2*, 1993.

[7] S. Vinoski, “Distributed Object Computing with CORBA,” *C++ Report*, vol. 5, July/August 1993.

[8] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[9] Object Management Group, *Universal Networked Objects*, TC Document 95-3-xx ed., Mar. 1995.

[10] B. Meyer, *Object Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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