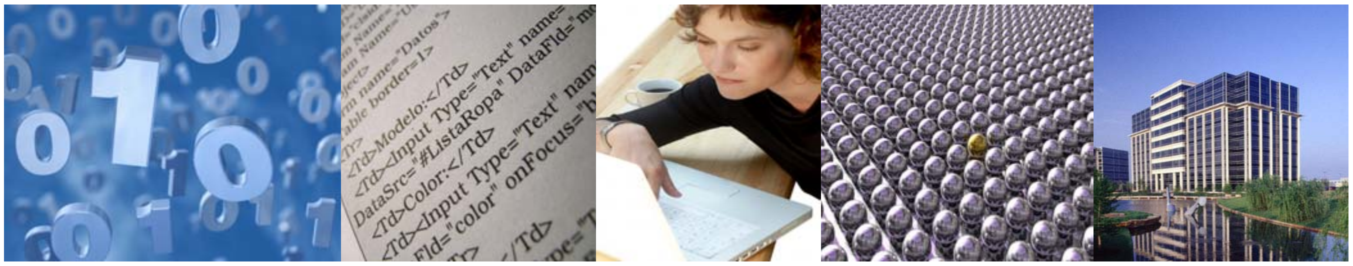
Software for Software Developers™

Platform Independent Client/Server Communication with Full C++ Types

By Dr. Zorabi Honargohar

Recursion Software, Inc.
November 29, 2004



Platform Independent Client/Server Communication with Full C++ Types

By Dr. Zorabi Honargohar

Introduction

The ability to provide services is a necessity of our time rather than a plain desire. In doing so, quite naturally we would like to choose the most suitable language that has stood the test of time, namely C++. Indeed, C++ is the most expressive language for creating sophisticated services quite efficiently. However, exposing those services have been frustrating and expensive.

Exposure of services is carried out by an exchange of data, usually between a client and a server. The richness of data types of C++ has been the impeding factor in arriving at a feasible solution for exchanging objects in a client/server session. The CORBA solution is very intrusive and too costly to be practical. XML derivatives, such as the SOAP protocol, apart from being too slow, distort C++ code thereby causing huge maintenance due to their limited data representation capabilities.

Ideally, a solution should not require any change to the original class types. That way, the services can be written entirely in C++ without any consideration to their transport. Then, the solution should allow the exposure of the services in full C++ data types and features like templates and multiple-inheritance, etc. The solution must be independent of platform to be of any use. Thus, a C++ program on Windows should be able to exchange complex data with a C++ program on Linux, in fast binary format.

In this article we present an optimal solution for exposing C++ objects. The solution is optimal for the following reasons.

- Reduces data exchange to plain streaming via familiar operators >> and <<
- Supports all C++ type mechanisms such as templates, multiple-inheritance
- Requires no change to the classes to be exposed
- Is type safe and the transport is optimally in binary
- Supports all platforms.

First we will demonstrate how the facilities of C++ Toolkits mechanize the binary transport of C++ objects utilizing the Network Data Representation standard. Then, we illustrate a simple technique to expose a particular service using C++ Toolkits mechanization introduced in this paper. Finally, we discuss the desirable distinction between the protocol for data representation, and the high-level protocol for negotiating services.

Streaming a class

Consider an already existing class somewhere in our code, such as the following class with three members.

```
class argumentType {
       int intMember;
       double dblMember;
       string stgMember;
public:
       argumentType(void);
       void set(intMember, dblMember, stgMember);
       int getInt(void);
       double getDouble(void);
       string getString(void);
};
```

In a new header file, we include the header file in which the above definition resides, and then add the following lines (similarly repeated for each class).

```
void os_write( os_bstream&, argumentType& );
void os_read( os_bstream&, argumentType& );

OS_CLASS( argumentType )
OS_STREAM_OPERATORS( argumentType )
```

Then, in a related source file, we write the following lines.

```
OS_STREAMABLE( argumentType* )

void os_write( os_bstream& stream, argumentType& object )
{
stream << object.getInt() << object.getDouble() << object.getString();
}

void os_read( os_bstream& stream, argumentType& object )
{
       int i;
       double d;
       string s;
       stream >> i >> d >> s;
       object.setInt(i);
       object.setDouble(d);
       object.setString(s);
}
```

The streaming of the class is complete. Next we illustrate its usage.

Transporting a streamed class

In the previous section we illustrated a systematic pattern for streaming any C++ class. In this section, we show how easy it is to send an instance of a class from a client to a server on different platforms. The independence from platform is furnished by C++ Toolkits.

On the client side, first we create a socket on a selected port. Then, we associate the socket with a stream and simply output the object to the stream. Note that the calls starting with `os_` are from the C++ Toolkits library.

```
argumentType argument;
int port = 6000;

os_tcp_socket socket;
socket.connect_to( os_socket_address( port ) );
```

```
os_bstream stream( socket );
stream << argument;
```

The code on the server side is just as simple and straightforward. The last line retrieves the class from the stream, as sent to us by the client. Note the familiar IO pattern of C++ stream libraries.

```
argumentType argument;

os_tcp_connection_server server( os_socket_address( port ) );
os_tcp_socket socket;
server.accept( socket );

os_bstream stream( socket );
stream >> argument;
```

Exposing a Service

It is important to mention at this point that the binary transfer of primitive types is carried out following the Network Data Representation (NDR) standard. C++ Toolkits extends the application of NDR to C++ objects. Furthermore, C++ Toolkits supports all C++ features, including templates, multiple-inheritance and virtual bases.

We have seen how C++ Toolkits reduces the transfer of objects to plain IO. Indeed, the sequence of statements for performing the reduction (streaming) is a repetitive pattern that can be automated.

Let us consider the application of streaming objects to exposing a service. Suppose a Server provides a service named `Record`, which expects two C++ objects of types `Person` and `Pet.` We define a new C++ type as follows.

```
class RecordType {
       Person person;
       Pet pet;
public:
       void Record(void);
};
```

On the Server side we will have the implementation for the service `Record.` The Client only needs the above definition. Note that the members `person` and `pet` of the class are playing the role of arguments to the call for service.

All that is needed now, is to stream the class `RecordType`. The Client will simply output an instance of `RecordType` to the stream, and the Server will perform the service upon receiving the instance from the stream.

Turning this technique into a service protocol is only a matter of coming to an agreement. Indeed, the only thing left to agree upon is simply the order in which communication between a Client and Server should proceed. Below is an outline of a sample scenario.

Client:	“Makes a connection to the Server”
Server:	100 ready
Client:	150 requesting Record
Server:	101 proceed
Client:	151 “binary stream as discussed earlier”
Server:	102 service completed
Client:	155 All Done

Conclusion

C++ is the most expressive and efficient language for providing sophisticated services. C++ Toolkits provides powerful mechanisms for streaming C++ objects without any restrictions. Furthermore, C++ Toolkits uses the NDR standard for low-level streaming.

Any high-level protocol for controlling the communication with regard to exposing services can be applied to the streaming mechanism of C++ Toolkits. The service negotiation protocol is a higher-level protocol and can be separated from the streaming protocol.

An enterprise can simply deploy the C++ Toolkits solution to immediately expose its services to its various departments. There is no need to convert or modify existing C++ code. The mechanism provided by C++ Toolkits can expose the services without any change of existing code, and with support for all C++ features.

About the author

Dr. Zorabi Honargohar is a senior software engineer at Recursion Software, Inc. He may be contacted by email at engineer@recursionsw.com.

Recursion Software, Inc.
2591 North Dallas Parkway
Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com