Jordan Alperin
David Dopson
Thomas Purtell

RMI for C++ Rapid Application Development of Distributed Event Driven Software

Abstract

In this paper we describe the implementation of a framework designed to be used for creating an event driven application with objects. The code to handle object marshalling over the network is automatically generated from the classes written by the application programmer. A demo peer-to-peer application was also developed to highlight the ease and elegance with which event driven software can be created.

Introduction

Remote Procedure Call (RPC) is a powerful design pattern that has been used to develop many different types of network applications. By thinking of the packet transmission and receipt as function calls it is possible to abstract the networking communications into a utility package and treat the entire application on both client and server as executing as one program. Servers, however, almost always maintain various elements of state. Since the server and client run in different address spaces it is not possible to refer to data on the server with pointers. Because of this the server usually maintains tables of id's for each type of structure that a user would refer to such as a file, or a user group, or another server's contact information. This tends to lead to code that is cluttered with references to these tables and id passing.

Remote Method Invocation (RMI), however, allows many of the actions performed by RPC servers to be abstracted into the utility package, leading to leaner, easier to read code. RMI also allows for the possibility of polymorphism which allows a client to the same operation on multiple objects references that each have different implementations.

Goals for this Implementation of RMI

- Enabling Rapid Application Development using a simple set of C++ classes and tools
- Singular interface for both local method calls on RMI objects and remote RMI invocations
- Transparent reference passing between clients
- RMI applications do not require locking
- Allow peer to peer RMI interaction when using our framework
- Also allow normal applications to act as RMI clients and make RMI calls without using our framework
- Robust guarantees that isolate network error handling from the application developer
- Efficient protocol with low RMI overhead
- RMI applications should have the same performance characteristics as the event driven implementations they replace

Entities

RMI Application – a set of RMI objects, servers, and pure clients

An application's functionality is implemented in a one or more typically single process pure clients, and a set of RMI objects which handle events called methods.

RMI Object – an object that lives on one server, but which can be referenced from other clients

An RMI object is implemented in three classes: interface, implementation, and RMI client stub. The application programmer is responsible for writing the implementation class which contains the all application code associated with the object. The interface class's header file is defined by the programmer and consists only of public virtual and static methods declarations. A tool is provided that parses interface headers in order to generate the RMI stub cpp file. The static methods of the interface are implemented in the stub file along with the RMI client stub class, which contains stubs for the virtual functions.

RMI Reference – an instance of an RMI client stub class. The reference can be passed between clients and used to access the associated RMI Object.

Method Call – the initiation of the method by the client

Method Invocation – the actual execution of the application method code

Method Call Return – the continuation of the client code

RMI Server – an event based server which coordinates access to its set of RMI objects. Running object methods can also act as clients.

The RMIServer class has an event loop which serves RMI objects. Only one server instance is allowed in a single process, and that server owns a set of RMI objects whose implementation objects are stored in it's process (the server can also have RMI references to other servers' objects exposed, thus allowing it to act as a proxy. Useful for NAT situations). Though an RMI server will only execute RMI object methods, since RMI objects can implement virtually anything, in practice it is possible and usually elegant to implement any other desired server capabilities as RMI objects.

Pure Client – a typically single threaded program that doesn't share objects

Since the execution of an RMI server is controlled by the event based RMIServer class, it is a poor place to take keyboard input or run a GUI. In order to interact with users on the local machine, an application has one or more pure clients.

RMI Client – the combination of a pure client and a local RMI server

An application's pure clients are typically single process, and thus do not serve objects. However, though they cannot share objects, it is possible to give every pure client an RMI server on the same machine. The client can store as much of its state on its server as it desires, and concurrent access to this state by other servers will be serialized by the RMIServer class. Thus the pure clients can be simple single threaded programs, yet still interact bidirectionally with RMI servers.

Design

```
class OBJNAME_RMIref{
  ObjectID
  RMIConnection*
+ stubs for virtual functions
}

OBJNAME::dispatch(callID, char* argstring);
+stubs for static functions

class RMIServer{
  Map<ObjectID => local Object*>
  Map<local Object* => ObjectID>
  ObjectID getObjectID(RMIObject*);
  releaseObject(RMIObject*);
}

class RMIConnection{
  Socket socket;
  String hostName;
  int port;
  send(ObjectID, CallID, char* argumentFormat,
… call args …);
  getReply(char*  returnFormat,  void*
returnVariable);
}

writeArgs(char* argumentFormat, …);
readArgs(char* argumentFormat, …);
```

An RMI reference has to be able to communicate with the correct RMI server and process the client's half of the RMI protocol. In order to do this it keeps a reference to the connection to the server and the ID of the object on that server. In order to allow the object to be passed to other clients who do not have a preexisting connection to the server, the RMIConnection class maintains identifying information about the server which allows the connection to be reestablished (for TCP, host name and port number work well).

The primary state is in two hash tables which maps object id's to object pointers and visa-versa. These tables are used when receiving method calls, and when sending objects by reference in either a return value or an outgoing method call.

On the server side, the object server to which the client is connected receives the RMI message and extracts from it an object ID and a method ID. The server looks the object up in its hash table and calls that object's dispatch method which unmarshals the arguments and calls the appropriate implementation class method. The dispatch method then marshals the return value and sends it back across the connection from which the request came.

Marshalling

In the RMI reference class for each object there is an implementation of all the virtual methods declared in the interface. These methods are responsible for controlling the interaction with the server holding the actual object. To do this, it first calls the connection class's send method with a format string describing the arguments and the arguments

themselves. If the method does not have a return value then the stub is done. If there is a return value, then the stub calls the connection's getReply method which takes a format string describing the return value and a pointer to a temporary local variable which will be returned to the user. The connection class uses two function writeArgs, and readArgs to do the actual marshalling and then sends a message over its socket.

Object constructors can be treated as static methods, since a constructor is a method that does not have a source object and returns a reference. These methods' marshalling stub code is implemented as static member functions of the interface class. These methods were already declared in the interface header written by the application programmer, but their implementations are generated along with the reference class. Because of the constructors, the new operator is also implemented for the interface class so that an instance of the reference class is generated. The generated implementations work the same as those for non-static methods described above, except for one caveat.

Static methods have the problem that they do not have a source object from which their intended parent server can be determined. Therefore, if a static method is declared with the first argument as an RMIConnection object, it will use that connection object to perform the call. Otherwise it is assumed to be a local static call, and will invoke the method locally.

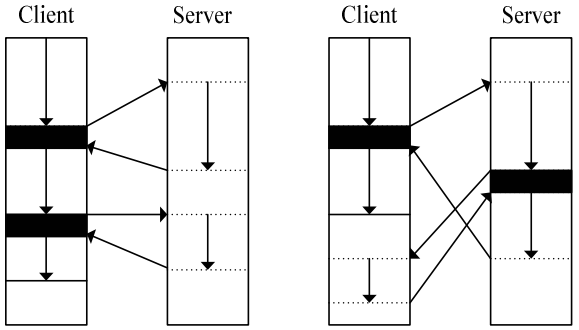
The marshalling implemented in printArgs and readArgs behaves similarly to printf. They proceed character by character and encodes through the format string encoding or decoding their arguments. The values that can be used in the format strings include the following.

- 'd' – a four byte value such as an integer.
- 's' – a string of characters passed as a char*.
- 'A?' – a C style array of type ?. the length of the array is passed as an additional argument.
- 'L?' – stl::list<?>. since the data format is the same, list types with different comparator operators are encoded the same. This additional knowledge is preserved in the argument type of the actual method.
- 'M??' – stl::map<?, ?> .
- 't' – a structure or object that can be serialized. This character requires passing both a pointer to the structure, and a pointer to a function capable of encoding the structure.
- 'r' – an RMIObject to be passed by reference.

Passing RMI Object References

To encode an RMIObject, first we check if it is a local object or a remote object. If it is reference to another server we encode its id and connection information. A local object can only be encoded normally if the code is running in a process which is an RMI server. The server is queried for the object's id which is encoded.

For objects on a pure client, there is a mechanism to pass by reference, but the resulting reference is only valid for the duration of the call. A single-threaded system has many advantages on the client side. Most importantly it allows developers to avoid the hassle of dealing with object locking and debugging the failures of multithreaded code. This constraint allows the client to maintain a simple unthreaded structure while still giving the server temporary access to objects in the client process. The id passed to the server is a temporary id, starting at 1 for the first object passed and counting up. The connection information is for host 'localhost,' and port -1. The unmarshalling code recognizes this and creates a temporary connection object with the same socket as the one which the client is connected on. When the call finishes, this connection is destroyed to prevent a call from being attempted later. While the client waits for a return value from the server it listens for RMI messages being passed on the same connection.



A simple set of RMIs

A nested bidirectional set of RMIs

Garbage Collection.

Reference counting under a system such as this is difficult, and adds no functionality other than reduced memory consumption over time, so it is currently unimplemented. Responsibility, like in regular C++, currently rests on the object creator to manually destroy the objects they created.

It is possible to increment and reference counts as clients are passed references, but the burden of releasing references is on the client. In the situation that the client disconnects, there is ambiguity about the state of its references. If references were also tracked on a per client basis then all of the client's references could be freed after a timeout of an hour or so. However, if the client reconnects then it becomes necessary to find their old connection information and retrieve their old reference counts.

An alternative solution would be to do garbage collection based on access time. A last access time field would be kept with every object on the server. A background task would then write objects older than 2 hours to disk. The maximum size of the disk backup could be enforced by deleting the oldest sections first.

Code Generation

In order to facilitate RMI development, we implemented a code generation package that automatically generates marshalling code stubs. The tool tokenizes the header files of the interface classes and extracts all of the definition information. This information is then used to generate the marshalling stubs and dispatch method by following a simple template. The interface class definition should inherit from the RMIObject base class and have no data members except the ones inherited. Every method in the class definition should be public, and will have stubs generated for it and be exposed as the object's interface. All public methods including constructors, and operators are legal for export.

The same tool can also be used to create serializable classes. In order to be passed over the network, non RMI classes must be serialized. Only classes which only contain serializable members and RMI object references can be exported. The basic C data types are serializable, but pointers to them are not. The one exception is that char* is assumed to be a null terminated string. For each serializable class a function is generated that can control the marshalling of an instance of that type. In practice, the restrictions on serialization are not a very big problem, considering that all that is required to convert a C++ class into an RMIObject is defining an interface class. Thus classes that are primarily structures for holding data can be exported for serialization as arguments for functions.

For greater makefile flexibility, the tool can be run either with command line options, or scripted through an export file.

The file format used for the RMI remote and base class generation tool is extremely simple. It can contain only two kinds of entries. The first type of entry designates an RMI object's interface header and tells which class implements it. The second entry type exports a class for serialization.

object (HEADER.H : ImplementationClassName,)
export (HEADER.H : CLASS)

The file can contain any number of these entries in any order. The entire configuration file is processed by the tool and then it begins extracting all the type information from the header files. If it is necessary to follow the header directives in the header file to gather further type information, it will do that. Every parameter to an RMI call must either be a primitive type, an RMI Object, or an object which is serializable. The C++ header and source generation tool will generate and display any relevant error message if inappropriate object types are used as parameters to a method which is exported.

A class can provide an explicit serializing method if it so desires. The tool will flag a class

that has a specific serialize method implemented as serializable and a corresponding deserialize method. If an RMI Object is marked as serializable then the marshalling function will be chosen by parameter type. If pointer type of type X is used as an argument in an exported method, then it will be passed by reference. If type X is passed as a C++ reference then the serialize version will be used.

Polymorphism

Sometimes a client program will have base class references to an instance of a subclass; however, because the implemented methods are all virtual functions, the correct sub class's method is automatically called after object lookup.

Type casting to a subclass is generally an unsafe process in C++. The newer versions of gcc support a dynamic_cast<TYPE> operator which checks runtime type information. Implementing the necessary code to allow dynamic casting using the C++ dynamic cast operator would require a significant additional code base. There are few situations where a programmer may benefit from typecasting to a subclass, so a way to support the sub-classing is necessary.

Because RMI is supposed to impose an interface boundary which restricts the access to remote objects in a specific way, a server may not want the client to have access to the functions exposed in the subclass. Implementing a generic dynamic cast would defeat this boundary. To allow for sub-classing in the scenarios where it would be useful, the programmer can define a simple function which will allow for sub-classing and preserve the RMI interface abstraction. This is an example of the code a programmer would use to accomplish this.

```
class SuperClass;
class SubClass;

class SuperClass {
public:
    virtual operator SubClass*() {
            return NULL;
}};

class SubClass {
    operator SubClass*() {
            return this;
}};
```

Using this method, the necessary subclass type-casting can be done, even over an RMI connection, without disturbing the RMI functionality.

Network Robustness Guarantee

- methods with a return value are guaranteed to have been invoked
- methods from a single connection are guaranteed to be invoked in the order they are called
- (Thus all RMI methods called before a method with a return value are invoked before that method returns)
- All methods will be invoked unless communication with the server becomes impossible, in which case an error will be propagated to the client application.

For RMI we need to send discrete chunks of data with guaranteed delivery and guaranteed packet ordering within a connection. For these purposes, TCP streaming sockets work very well. On top of the TCP protocol we added the restriction that each message is preceded by its size in bytes. This way, variable sized messages are able to be read by a socket class that does not understand their content. There is some redundant encoding here, as the stream was coded as multiple packets that each have a size attached. However, the major alternative, UDP, does not have any guarantees about packet timing. Implementing guaranteed delivery with timeouts and a specialized connection protocol in UDP would be more efficient, but significantly more complicated, as well as a great risk introducing subtle bugs.

Communication Format

During development we have used a message format that was easy to read and debug. For example, integers were converted to hexadecimal strings. For increased efficiency it is possible to code arguments in a more condensed binary form that is more difficult to read. An application programmer could switch between the two formats with a macro, allowing him to debug with an ASCII call log in the first mode, and then use the second for his release executable

Efficiency

Despite the high level feel of an RMI application, they are surprisingly efficient. An RMI method call begins with a virtual function call to the appropriate method, which marshals the arguments into a buffer. This buffer is then sent across the network where the server unmarshals the arguments, makes a hash table lookup to locate the correct object, and make a virtual function call into the actual method. Compared with the inherent delay of making a kernel call to write data to the network and latency of network communication, the overhead of a few virtual function calls, and the computation required to compose packets are insignificant.

What does have a significant impact on performance is our reliance on TCP. We currently use TCP streams for their guaranteed packet delivery; however, it is possible to build a guaranteed packet delivery protocol layer on top of UTP without incurring the extra overhead that a more sophisticated protocol adds.

In order to gauge the efficiency of our implementation, we calculated the maximum sustained call rate. In order to do this, we set up a test and measured the time required to perform one hundred thousand member function calls with no arguments and no return values. We tested a localhost connection so that we could measure the computational efficiency of RMI rather than network latency. If the client does not have to block for a return value, then the latency of the network is less important, but the overhead of communication is still present. We found that the time per call was a bit less than 10 microseconds. This is about fifty times greater than the time required to make a virtual function call to a non RMI instance of the same class. Even though it requires proportionally more overhead, the overall overhead of RMI is relatively small when compared to the time required to execute the code in a sufficiently complicated function.

Discussion of a sample peer-to-peer application

In order to justify our RMI toolkit, we implemented a demonstration Peer-to-Peer application. Our Peer-to-Peer system consists of many different nodes which each have a list of the available resources for download. Each server also maintains a list of siblings. During queries, these sibling lists are used to propagate queries throughout the network in a decentralized fashion.

While our Peer-to-Peer demo is not novel for its features, it is quite unique in the simplicity and brevity of its source code. The entire project was under one hundred lines of code. Our simple RMI implementation definitely made this task much more pleasant. For example, when a query is sent out, a reference to a ResultSet is sent with it. If a server finds a match to the query, it simply invokes the add(Resource* r) method on the result set. Since the return value of this method is void, the server is not required to block while waiting for a response.

This same interaction could have occurred in an event based application, but would have been more complicated. Instead of a result set, a query would contain a query number and an originating host address. The owner of the information would compose a message and send it back to the client. When the client receives the message, it must decode it and eventually call a function equivalent to add(…) that will append the new result to the list of results. The computer performs the same steps because our system is also an implementation of event based programming; however, in our case all of the details have been abstracted out. The id's and system calls are wrapped as objects, allowing the programmer to concentrate solely on adding extra functionality.

References

This section is not formatted according to any normal specification for bibliographies. This is a temporary unsorted list of the documents that we worked with for the project.

[1] Shankar, Ajai "RPC and C++: Build a Template Library for Distributed Objects Containing Multiple Interfaces" November 2000. (http://msdn.microsoft.com/msdnmag/issues/1100/RPC/default.aspx)

[2] "Distinct ONC RPC / XDR for C or C++ and Borland® Delphi™" November, 4th 2002 (http://www.onc-rpc-xdr.com/products/rpc/rpc-windows-vc-c-delphi.asp)

[3] Coëtmeur, Alain "ORPC : a RPC in C++. Reference manual." February 7th 1996 (http://www.medusa.uni-bremen.de/intern/orpc/orpc.html)

[4] Kidd, Eric "XML-RPC for C/C++" 2001 (http://xmlrpc-c.sourceforge.net/)

[5] "RPCgen" 2000 (http://netbula.com/oncrpc/rpcgen.html)

[6] Sals, Richard "DCE: Notes on Free RPC Source" (http://www.opengroup.org/dce/mall/dce-rpc-readme.htm#top)

[7] "JavaTM 2 Platform, Standard Edition, v 1.4.0 API Specification" 2002 (http://java.sun.com/j2se/1.4/docs/api/)

[8] Birrell, Andrew D. and Bruce Jay Nelson "Implementing Remote Procedure Calls" ACM Transactions on Computer Systems Vol. 2, No. 1, February 1984 (http://godzilla.cs.nwu.edu/~srg/Papers/7-26-01/p39-birrell.pdf)