RPC CLIENT: A PATTERN FOR THE CLIENT-SIDE IMPLEMENTATION OF A PIPELINED REQUEST/RESPONSE PROTOCOL

Mark Heuser and Eduardo B. Fernandez

Dept. of Computer Science and Eng.

Florida Atlantic University

Boca Raton, FL 33431

ABSTRACT

This is a design pattern for the client-side implementation of a pipelined request/response protocol. The pattern is an elaboration of the Remote Proxy design pattern presented in the GOF and POSA books. Its intent is to hide the fact that a service is remote and make a remote service convenient to use.

1. INTENT

The RPC Client design pattern is an elaboration of the Remote Proxy design pattern ([3] and [4]) for the case in which

- The proxy is a client-side proxy.
- The communication protocol is a pipelined request/response protocol.
- The execution environment consists of multiple threads of control in a single address space.
- Performance considerations preclude simpler approaches.

The basic intent of the pattern is the same as that for Remote Proxy.

2. MOTIVATION

2.1 Problem

Firewall vendors use the term *proxy* to refer to a networking application interposed between a client and a server. To the client, the proxy appears to be the server, and to the server, the proxy appears to be the client. The proxy's intervention can improve security by restricting the ways in which the client and the server interact.

Consider integrating an HTTP proxy with a third-party URL filter. The URL filter contains a database that categorizes URLs. Categories include pornography, drugs, weapons, sports, job search, and others. Additionally, the URL filter contains an access policy. The access policy defines the conditions under which access to a URL should be permitted (or denied). The access policy considers such factors as the category of the URL, the time of the access, and the identity of the client. When an HTTP client requests a document from an HTTP proxy, the proxy sends the document's URL and the client's identity to the URL filter. The URL filter returns a "permit" or "deny" verdict. If the verdict is "deny," the proxy will

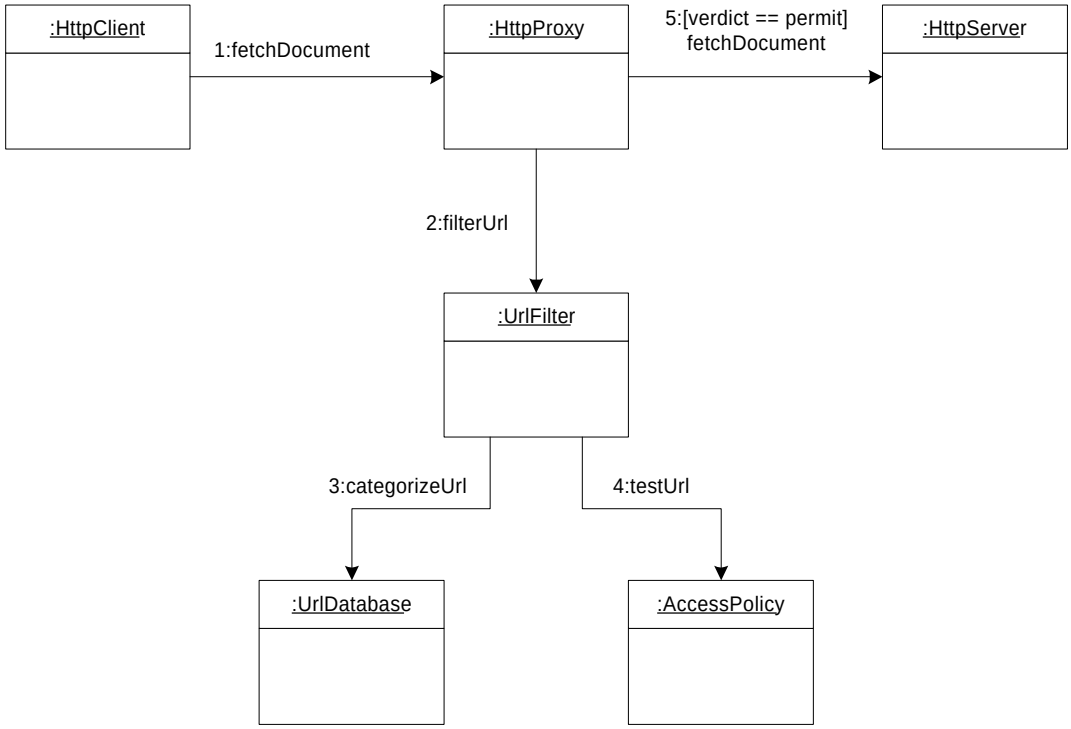

Figure 1: Interaction between an HTTP proxy and a URL filter.

forward a "denied" message to the client. If the verdict is "permit," the proxy will forward the requested document to the client after fetching it from the HTTP server. Figure 1 illustrates the participants and their interactions with a UML collaboration diagram.

The interface to the URL filter consists of a pipelined request/response protocol over a TCP connection. For example, the manufacturer of the URL filter could choose this interface in order to avoid integrating source code with other vendors' software. This means that the proxy sends a URL to the filter in a request message, and the filter sends a verdict back to the proxy in a response message. Each request message contains a distinguishing value known as a request identifier. Furthermore, each response message echoes the request identifier from the

corresponding request message. This arrangement allows the proxy to send new requests before previous requests have completed, and to associate responses with the proper requests.

2.2 Forces

The software that implements the client side of the URL filter protocol must resolve the following forces:

- Efficiency. URL filtering directly affects the performance of the HTTP proxy. A simple approach that used a different TCP connection for each request would result in poor performance. Instead, it is necessary to multiplex requests from several threads over a single TCP connection, and to demultiplex responses from that connection back to the respective threads.

- Thread safety. The HTTP proxy uses multiple threads to handle multiple, simultaneously active sessions. These threads will compete for access to a single TCP connection when sending request messages. Furthermore, one thread must serve as the demultiplexor.

- Encapsulation. The manufacturer of the URL filter could have chosen a different type of interface. For example, the URL filter could have been packaged as a library that accessed the URL database directly. In order to make the replacement of the URL filter as easy as possible, the HTTP proxy must not be aware of the manufacturer's choice of interface.

- Extensibility. It should be possible to take advantage of new features in a future version of the URL filter without having to rework existing software.

- Reusability. The software should be reusable by other proxies that encounter URLs.

- Usability. The software should be easy to use.

Note that many of the functional problems to be solved have nothing to do with URL filtering. These problems include aspects such as how to identify servers, connection to servers, recovery, fault tolerance, etc. Those aspects will not be discussed here.

2.3 Solution

The general structure of an RPC is shown in Figure 2. This uses two kinds of messages: request and response. Request messages contain a request identifier, a procedure identifier, and the procedure's arguments. Response messages contain a request identifier, a procedure call status, and the procedure's results. Clients send unsolicited request messages to servers. Each request message is distinguished by its request Id. Servers echo request Ids in the response messages sent back to clients. The request Id is contained in a common header shared by all response messages.

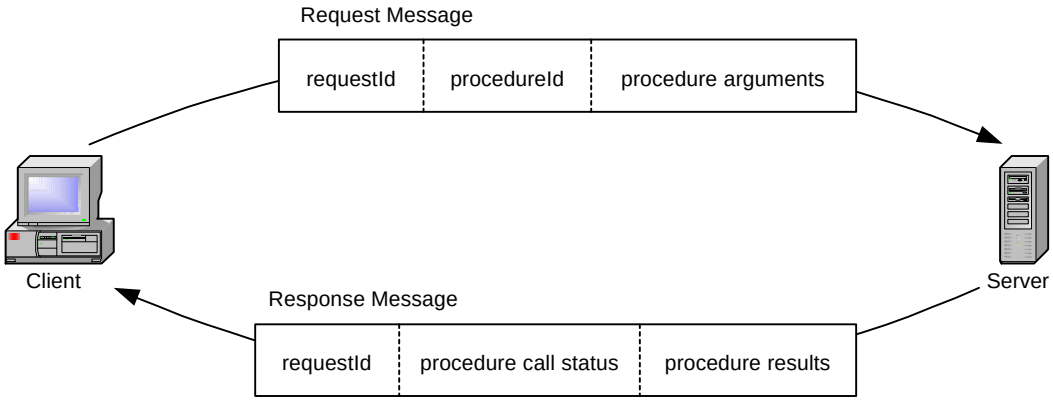

Figure 2: RPC communication pattern.

It should be possible to separate the service-dependent (URL filtering) aspects of the software from the service-independent aspects, resulting in two distinct layers. The service-independent layer should be able to solve functional problems found in the client-side implementation of any RPC-like protocol. A solution that considers these forces is based on the use of a remote proxy to implement the RPC.

Figure 3 shows the structure of the pattern. The RpcConnection class describes how to connect to the server, while the stub handles the distribution aspects. In the Consequences section (Section 8), we show how this pattern balances the forces described earlier.

3. APPLICABILITY

Use the RPC Client pattern when:

- A client needs to communicate with a server using a pipelined protocol.
- An RPC protocol may be needed for compatibility reasons.
- The execution environment consists of multiple threads of control in a single address space.
- Performance considerations make simpler approaches unacceptable.

4. STRUCTURE AND PARTICIPANTS

Figure 3 illustrates the participants in the RPC Client pattern. RpcClient, RpcConnection, and RpcStub form an abstract, service-independent layer, while ConcreteClient,

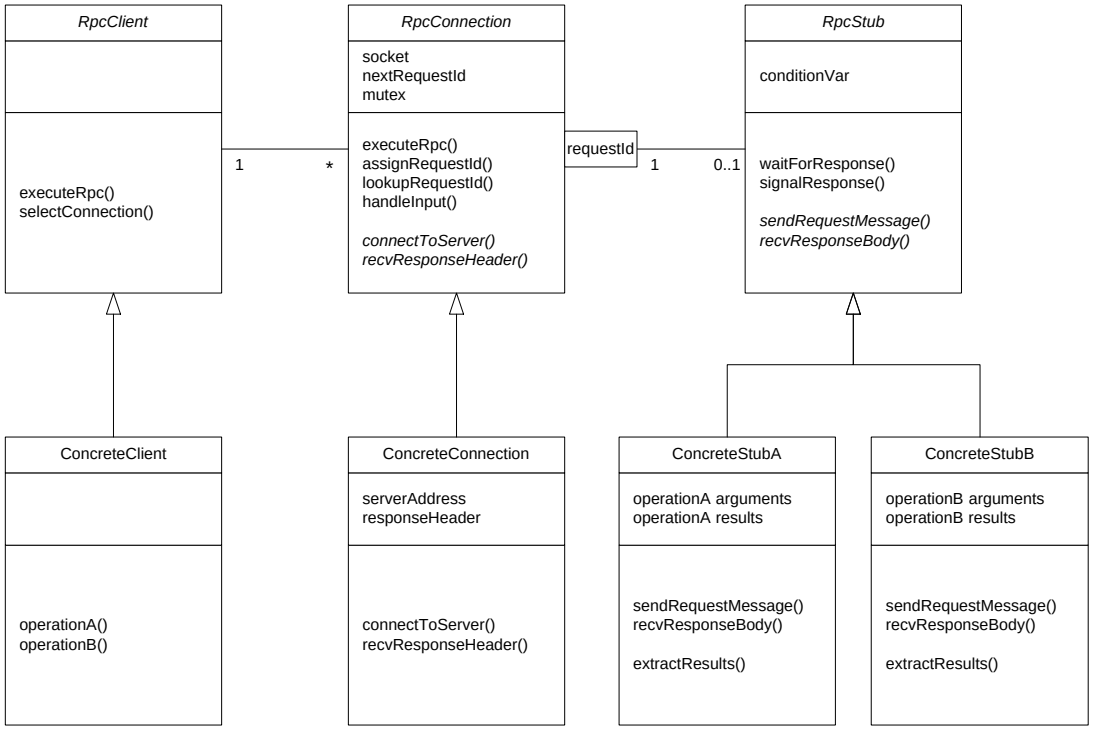

Figure 3: UML class diagram for the RPC Client pattern.

ConcreteConnection, and the ConcreteStub classes form a concrete, service-dependent layer.

- **RpcStub**. RpcStub is an abstract base class representing an invocation of a remote procedure. Its primary responsibility is thread synchronization. RpcStub provides a way for a thread to await the completion of a remote procedure call. It also declares two abstract operations: sendRequestMessage() and recvResponseBody().

- **ConcreteStubA**, **ConcreteStubB**. ConcreteStubA and ConcreteStubB are concrete, service-dependent classes derived from RpcStub. Each (StubA/StubB) represents an invocation of a particular remote procedure (operationA()/operationB()). Their responsibilities include marshaling procedure arguments and unmarshaling procedure results. Each contains the arguments and results of its procedure, and implements the sendRequestMessage() and recvResponseBody() operations declared in RpcStub. sendRequestMessage() marshals the procedure's arguments and sends a request message.

recvResponseBody() receives the body of a response message and unmarshals the procedure's results.

- **RpcConnection**. RpcConnection is an abstract base class representing a single connection to a server. Although the pattern permits multiple connections to the same server, such usage is not the norm. RpcConnection is responsible for overseeing the execution of a remote procedure, including such tasks as allocating request Ids, establishing connections, serializing access to connections, and matching responses to requests. It contains the socket used to communicate with the server, and the next available request Id. It also declares two abstract operations: connectToServer() and recvResponseHeader().

- **ConcreteConnection**. ConcreteConnection is a concrete, service-dependent class derived from RpcConnection. Its primary responsibility is to implement the connectToServer() and recvResponseHeader() operations declared in RpcConnection. connectToServer() establishes a connection with a server. By implementing this operation in ConcreteConnection, RpcConnection remains independent of transport protocols and addressing formats. recvResponseHeader() receives the header of a response message and returns the request Id contained therein. Given the request Id, RpcConnection can match the response to a request. ConcreteConnection contains the server's network address and a response message header.

- **RpcClient**. RpcClient is an abstract base class representing an RPC-based service. It is responsible for assigning a procedure call to a server (selectConnection()) and for recovering from network and server failures.

- **ConcreteClient**. ConcreteClient is a concrete, service-dependent class derived from RpcClient. It provides the public interface seen by users; namely, operationA() and operationB().

5. COLLABORATIONS

A remote procedure call executes in three distinct phases: sending the request message, receiving the response message, and extracting the results. The first and last phase execute in the same thread, while the middle phase executes in a different thread.

Sending the Request Message

This phase covers the transmission of the request message. It begins when a thread invokes a remote operation, and ends when the thread blocks to await the response message. The collaborations are described in the following paragraphs and are illustrated in Figure 4.

- When operationA() is invoked on ConcreteClient, a ConcreteStubA object is created to hold the operation's arguments and results.

- ConcreteClient selects a ConcreteConnection to execute the operation, and passes the stub object to the connection.

- ConcreteConnection assigns a request Id to the operation and, if necessary, establishes a network connection.

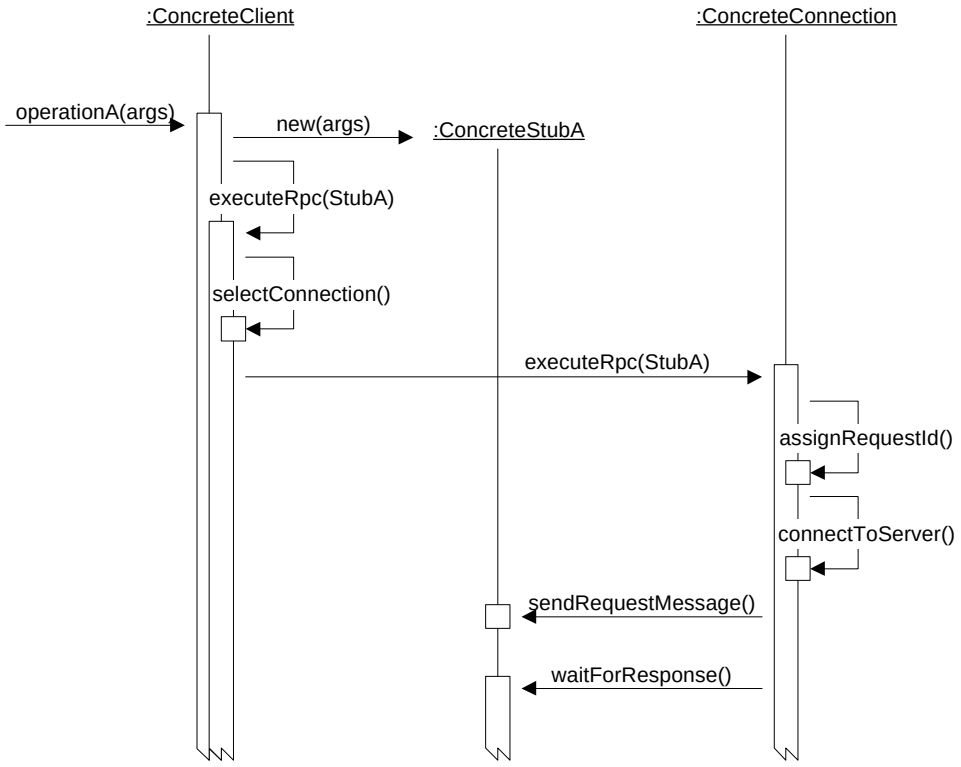

Figure 4: Sending the request message.

- ConcreteStubA sends the request message to the server and blocks the calling thread.

Receiving the Response Message

This phase covers the reception of the response message. It begins when data is available on a connection, and ends when the calling thread is awakened. The collaborations are described in the following paragraphs and are illustrated in Figure 5.

- When data is available on a connection, ConcreteConnection's handleInput() operation is invoked. ConcreteConnection receives the header of the response message and uses the request Id contained therein to match the response to the appropriate request.

- ConcreteStubA receives the response body, unmarshals the procedure's results, and awakens the calling thread.

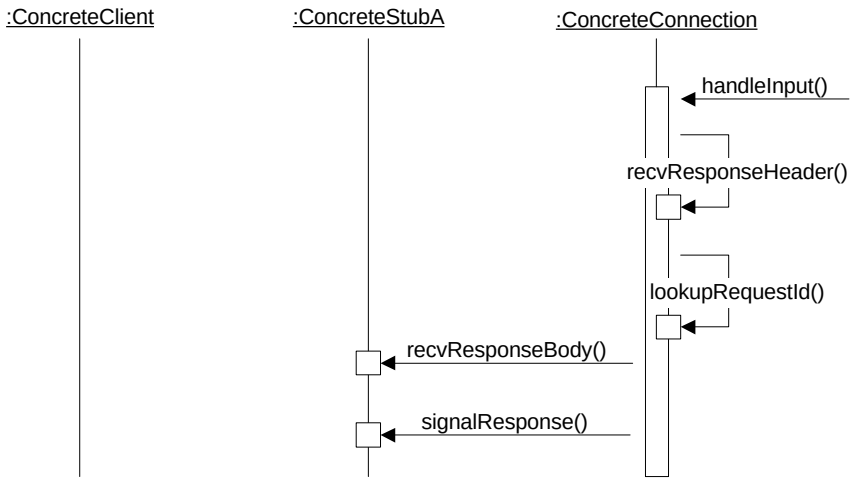

Figure 5: Receiving the response message.

Extracting the Results

This phase covers the extraction of the procedure's results from the ConcreteStubA object. It begins when the calling thread resumes execution, and ends when the thread returns from operationA(). The collaborations are described in the following paragraphs and are illustrated in Figure 6.

- When the calling thread resumes execution, it returns from the executeRpc() operations in ConcreteConnection and ConcreteClient. ConcreteClient extracts the procedure's results from ConcreteStubA and deletes the stub.

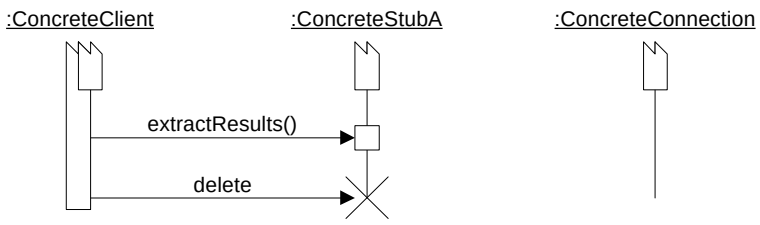

Figure 6: Extracting the Results.

6. VARIANTS

If it is not necessary to use multiple servers for fault tolerance or load balancing, the class diagram can be simplified by merging the RpcClient and RpcConnection classes.

7. EXAMPLE

Figure 7 shows how the pattern can be applied to the URL filtering problem. The example employs the simplification suggested in the previous section.

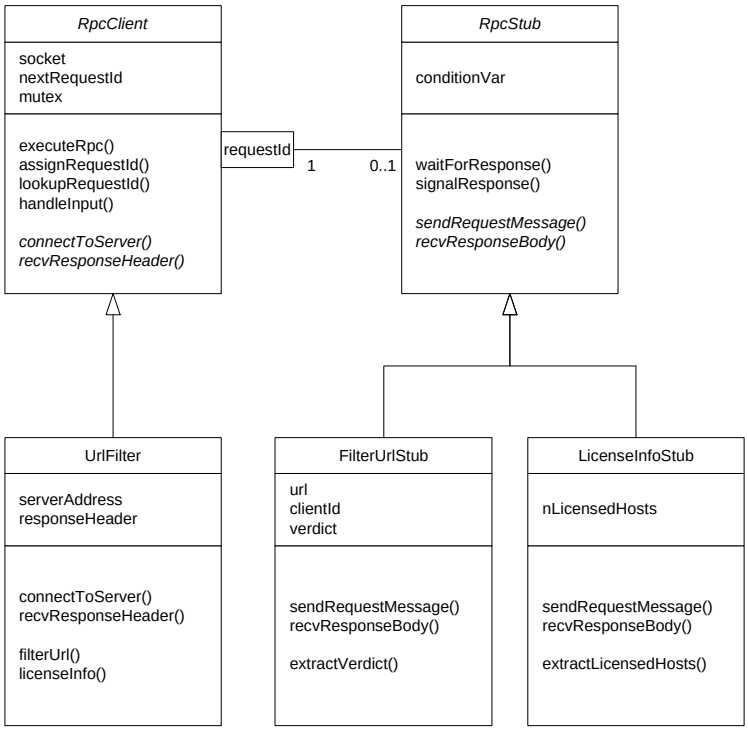

Figure 7: UML class diagram for the URL filter.

8. CONSEQUENCES

Benefits

- The RPC Client pattern hides the fact that a service is provided remotely, and makes the remote service as easy to use as a local service.

- The RPC Client pattern separates service-dependent issues and service-independent issues into distinct layers. The service-independent layer should be reusable by different service-dependent layers.

- The RPC Client pattern is extensible. To add a new remote procedure, one need only add a new operation to ConcreteClient and derive a new class from RpcStub.

- The RPC Client pattern should give good performance in a multithreaded execution environment, although we don't have experimental data to prove it.

Liabilities

- The RPC Client pattern is only applicable to RPC-like communication protocols in a multithreaded execution environment. More general patterns are described in Section 11.

- An application that relies heavily on remote procedure calls should use a standard RPC toolkit, such as Sun RPC, DCE RPC, or CORBA. This pattern may, in that case, overlap with some existing functions.

- To simplify programming, the RPC Client pattern uses a synchronous procedure call model wherein the calling thread blocks until it receives a response from the server. In some cases, performance can be improved with an asynchronous procedure call model that allows the calling thread to overlap execution with the server.

- Some consideration must be given to limiting the data copying overhead that occurs during the final phase of remote procedure call execution. See Figure 6.

9. IMPLEMENTATION

The following paragraphs discuss some implementation issues.

- Because the lifetime of ConcreteStubA matches the lifetime of operationA(), ConcreteStubA can be allocated in the activation record of operationA(). Allocation from the stack is faster than allocation from the heap.

- Figure 5 shows the handleInput() operation being invoked by an external agent when data becomes available on a connection. There are at least three choices for the external agent.

    1. A dedicated thread. Each RpcConnection object could dedicate a thread to the demultiplexor role. Although this approach is simple, it is also costly.

    2. A Reactor [1]. Each RpcConnection object could be registered with a Reactor (RpcConnection would have to be derived from EventHandler). The thread executing the reactor's event loop would invoke the handleInput() operation. This approach allows a single thread to serve as the demultiplexor for several RpcConnection objects, but creates dependencies on Reactor and EventHandler.

    3. A client thread. Data can arrive on a connection only while a remote procedure call is in progress. Instead of blocking to await the completion of its call, one thread could assume the role of demultiplexor. The demultiplexor must be able to recognize when its response message arrives, and if other remote procedure calls are still in progress, must nominate another thread for the job. Although this approach adds a small amount of complexity, it does not increase resource usage and it does not create additional dependencies on other objects.

- As shown in Figure 4, each thread that invokes a remote procedure sends a request message. In order to guarantee the proper interleaving of multiple request messages from multiple threads over a single connection, request message transmission must be serialized. This task is the responsibility of RpcConnection.

  As shown in Figure 5, response message reception is serialized by limiting the number of demultiplexors at any given time to one.

  Request message transmission and response message reception do not need to be serialized, and in fact should not be serialized if performance is important. However, the two algorithms may contend for access to the data structure that implements the mapping from request Ids to RpcStubs. For example, a mapping may be added during request message transmission, and deleted during response message reception.

  The preceding discussion shows the need for two mutexes[1]: one to serialize request message transmission and one to serialize map changes.

  Because of its length, we show our sample code as an Appendix.

10. KNOWN USES

The motivation section describes a use of the pattern in CyberGuard's firewall product. General uses of the Remote proxy are shown in [3],[4], and [5].

11. RELATED PATTERNS

- RPC Client is an elaboration of Remote Proxy.

- The Strategy pattern could be used to vary the load balancing or error recovery algorithms employed by the RpcClient class.

- RPC Client uses the Half-Sync/Half-Async architectural pattern presented in [2]. The RpcConnection and RpcStub classes implement the queueing layer between the synchronous and asynchronous task layers.

REFERENCES

[1] D.C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J.O. Coplien and D.C. Schmidt, eds.), pp. 529-545, Reading, MA: Addison-Wesley, 1995.

[2] D.C. Schmidt and C.D. Cranor, "Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design 2* (J. Vlissides, J.O. Coplien, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[1] A mutex is a synchronization object that provides exclusive access to shared data.

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

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY: John Wiley & Sons, 1996.

[5] A.R.Silva, F.A.Rosa, and T. Goncalves, "Distributed Proxy: A design pattern for distributed object communication", *Procs. of PLOP'97*, http://jerry.cs.uiuc.edu/~plop/plop97

APPENDIX : SAMPLE CODE

```
/*
 * Client software for ad hoc RPC.
 *
 * INTENT: RpcStub and RpcClient together constitute a service-independent
 * layer/framework for the client-side implementation of an RPC-like
 * protocol.
 *
 * CONTEXT: An RPC-like protocol, or a pipelined request/response protocol,
 * is one that consists of two kinds of messages: request and response.
 * Clients send unsolicited request messages to servers.  Request messages
 * contain a distinguishing value known as a request Id.  Request Ids are
 * generated by the client.  Servers send response messages to clients.  A
 * response message echoes the request Id from a prior request message.
 * All response messages have a common header containing the request Id.
 * Communication occurs over a reliable, bidirectional, byte stream
 * (SOCK_STREAM).
 *
 * The execution environment consists of multiple threads of control in a
 * single address space.
 */

/*
 * CLASS: RpcStub
 *
 * PURPOSE: RpcStub is an abstract base class that represents an invocation
 * of a remote procedure.  Its primary responsibility is thread
 * synchronization.  RpcStub provides a way for the thread making a remote
 * procedure call to await the completion of the call.
 *
 * Service-dependent subclasses represent an invocation of a particular
 * remote procedure.  They should contain the procedure's arguments and
 * results, and should implement the mashaling and unmarshaling routines
 * declared here: SendRequestMessage() and RecvResponseBody().
 *
 * COLLABORATORS: RpcClient
 */
class RpcStub : public dll_node
{
        RpcStub (RpcStub const &);
        RpcStub& operator= (RpcStub const &);

        int     st_error;         // status
        size_t  st_requestId;     // request Id
        int     st_wakeupReason;  // wakeup reason
        ThreadAutoResetEvent  st_event;  // wakeup event

public:
        // Construct/destroy an RpcStub object.
```

```
        RpcStub ();
        virtual ~RpcStub () {}

        // Get/set status.
        int Error () const { return (st_error); }
        void Error (int error) { st_error = error; }

        // Get/set request Id.
        size_t RequestId () const { return (st_requestId); }
        void RequestId (size_t requestId) { st_requestId = requestId; }

        /* Wakeup reasons.
         */
        enum {
               WR_DONE = 1,  // you're done
               WR_RETRY,     // try try again (error recovery)
               WR_DISPATCH   // assume the role of dispatcher
        };

        // Wait for an RPC to complete.  Return the wakeup reason.
        int WaitForResponse ();

        // Signal the completion of an RPC.
        void SignalResponse (int reason);

        /* SendRequestMessage(), defined in a class derived from RpcStub, sends
         * a request message to a server via frame buffer "fb."  Return 0 if
         * successful, an error code otherwise.
         */
        virtual int SendRequestMessage (SocketFrameBuf &fb) = 0;

        /* RecvResponseBody(), defined in a class derived from RpcStub,
         * receives a response body from a server via frame buffer "fb."  The
         * response header has already been received and is referenced by
         * "hdr."  Return 0 if successful, an error code otherwise.
         */
        virtual int RecvResponseBody (void *hdr, SocketFrameBuf &fb) = 0;
};

RpcStub::RpcStub ()
{
        st_error = 0;
        st_requestId = 0;
        st_wakeupReason = 0;
}

void
RpcStub::SignalResponse (int reason)
{
        st_wakeupReason = reason;
        st_event.Signal ();
}

int
RpcStub::WaitForResponse ()
{
        st_event.Wait ();
        return (st_wakeupReason);
}

/*
 * CLASS: RpcClient
 *
 * PURPOSE: RpcClient is an abstract base class that represents an
 * RPC-based service.  Its responsibilities include allocating request Ids,
 * handling network and server failures, and coordinating the use of a
 * single stream socket by multiple threads.
 *
 * Service-dependent subclasses provide the public interface of a
```

```
 * particular RPC-based service.  They should contain the header of a
 * response message and the network address of a server, and should
 * implement the abstract routines declared here: RecvResponseHeader() and
 * ConnectToServer().
 *
 * COLLABORATORS: RpcStub
 */
class RpcClient
{
        RpcClient (RpcClient const &);
        RpcClient& operator= (RpcClient const &);

        ThreadMutex     cl_sendMutex;      // socket serializer
        StreamSock      cl_serverSocket;   // connection to server
        SocketFrameBuf  cl_serverBuf;      // socket frame buffer
        bool            cl_serverUp;       // connection is up
        StopWatch       cl_downInterval;   // time spent down

        ThreadMutex     cl_stubMutex;          // stub serializer
        size_t          cl_nextRequestId;      // next request ID
        dll<RpcStub>    cl_stubList;           // list of pending stubs
        size_t          cl_maxStubListLen;     // max queue length
        RpcStub         *cl_stubDispatcher;    // current dispatcher
        size_t          cl_nDispatchSwitches;  // # dispatcher switches

        int OpenConnection ();
        void CloseConnection (int error);
        int DispatchStubs ();

protected:
        /* ConnectToServer(), defined in a class derived from RpcClient,
         * establishes a connection with the RPC server.  Return 0 if
         * successful, an error code otherwise.
         */
        virtual int ConnectToServer (StreamSock &s) = 0;

        /* RecvResponseHeader(), defined in a class derived from RpcClient,
         * receives a response header from a server via frame buffer "fb."
         * Return 0 if successful, an error code otherwise.  If successful, an
         * opaque pointer to the header is returned in "hdr," and the request
         * Id is returned in "requestId."
         */
        virtual int RecvResponseHeader (SocketFrameBuf &fb, void *&hdr,
                size_t &requestId) = 0;

        /* Execute the specified RPC.  Return 0 if successful, an error code
         * otherwise.
         */
        int ExecuteRpc (RpcStub *pStub);

public:
        // Construct/destroy an RpcClient object.
        RpcClient ();
        virtual ~RpcClient ();

        // Return the total number of RPCs performed.
        size_t TotalRpcs () const { return (cl_nextRequestId); }

        // Return the maximum observed length of the RPC queue.
        size_t MaxRpcQueueLength () const { return (cl_maxStubListLen); }

        // Convert an error code into a string.
        char const *ErrorMsg (int code);
};

TimeValue const MIN_DOWN_TIME (15, 0);

RpcClient::RpcClient ()
{
```

```
        cl_serverUp = 0;

        cl_nextRequestId = 0;
        cl_maxStubListLen = 0;
        cl_stubDispatcher = 0;
        cl_nDispatchSwitches = 0;

        cl_downInterval.Set (MIN_DOWN_TIME);
        cl_downInterval.Start ();
}

RpcClient::~RpcClient ()
{
        assert (cl_stubList.empty ());
        CloseConnection (-1);
}

int
RpcClient::OpenConnection ()
{
        // assert (caller holds cl_sendMutex);

        // If the connection is up, there is nothing to do.

        if (cl_serverUp)
                return (0);

        // Re-connect at most once every MIN_DOWN_TIME seconds.

        if (cl_downInterval.TotalTime () < MIN_DOWN_TIME)
                return (RpcError::ERR_SOCKET);

        // Ask derived class to connect to the server.

        int error = ConnectToServer (cl_serverSocket);
        if (error) {
                cl_downInterval.Reset ();
                return (error);
        }

        cl_serverBuf.Attach (cl_serverSocket);

        // Mark the connection up.

        cl_serverUp = 1;
        return (0);
}

void
RpcClient::CloseConnection (int error)
{
        // assert (caller holds cl_sendMutex);

        // If the connection is down, there is nothing to do.

        if (! cl_serverUp)
                return;

        // Close the connection.

        cl_serverSocket.Close ();
        cl_serverUp = 0;

        // Wake all waiting threads.

        cl_stubMutex.WriteLock ();
        while (1) {
                RpcStub *pStub = cl_stubList.pop_front ();
                if (pStub == 0)
```

```
                        break;
                pStub->Error (error);
                pStub->SignalResponse (RpcStub::WR_DONE);
        }
        cl_stubMutex.Unlock ();
}

int
RpcClient::ExecuteRpc (RpcStub *pStub)
{
        // assert (caller holds no locks);

        while (1) {
                cl_sendMutex.WriteLock ();

                // Assign a request Id to the stub object and put the
                // object in the list of pending stubs.

                cl_stubMutex.WriteLock ();
                pStub->RequestId (cl_nextRequestId++);
                cl_stubList.push_back (pStub);
                if (cl_stubList.size () > cl_maxStubListLen)
                        cl_maxStubListLen = cl_stubList.size ();
                cl_stubMutex.Unlock ();

                // Send the request message to the server.

                int error = OpenConnection ();

                if (error == 0)
                        error = pStub->SendRequestMessage (cl_serverBuf);

                if (error) {
                        cl_stubMutex.WriteLock ();
                        cl_stubList.remove (pStub);
                        cl_stubMutex.Unlock ();

                        CloseConnection (error);
                        cl_sendMutex.Unlock ();
                        return (error);
                }

                // Wait for the response message.

                int wakeupReason = 0;

                if (cl_stubDispatcher == 0) {
                        ++cl_nDispatchSwitches;
                        cl_stubDispatcher = pStub;
                        cl_sendMutex.Unlock ();
                        wakeupReason = DispatchStubs ();
                }
                else {
                        cl_sendMutex.Unlock ();
                        wakeupReason = pStub->WaitForResponse ();
                        if (wakeupReason == RpcStub::WR_DISPATCH)
                                wakeupReason = DispatchStubs ();
                }

                if (wakeupReason == RpcStub::WR_DONE)
                        return (pStub->Error ());

                // not currently used
                assert (wakeupReason == RpcStub::WR_RETRY);
        }

        // Should never get here.
        // assert (0);
        // return (-1);
```

```
}

int
RpcClient::DispatchStubs ()
{
	// assert (caller holds no locks);

	int error = 0;

	while (1) {
		// Receive a response header.

		void *hdr;
		size_t requestId;

		try {
			error = RecvResponseHeader (cl_serverBuf, hdr, requestId);
		}
		catch (...) {
			error = RpcError::ERR_RESOURCE;
		}

		if (error)
			break;

		// Find the matching stub object.

		RpcStub *pStub = 0;

		cl_stubMutex.WriteLock ();
		for (dll<RpcStub>::iterator it (cl_stubList.begin ());
			it != cl_stubList.end (); ++it)
		{
			if ((*it).RequestId () == requestId) {
				pStub = cl_stubList.remove (it);
				break;
			}
		}
		cl_stubMutex.Unlock ();

		if (pStub == 0) {
			error = RpcError::ERR_PROTOCOL;
			break;
		}

		// Receive the response body.

		try {
			error = pStub->RecvResponseBody (hdr, cl_serverBuf);
		}
		catch (...) {
			error = RpcError::ERR_RESOURCE;
		}

		pStub->Error (error);

		// Check whether the dispatcher's duty is done.

		if (pStub != cl_stubDispatcher)
			pStub->SignalResponse (RpcStub::WR_DONE);
		else {
			cl_sendMutex.WriteLock ();
			if (cl_stubList.empty ())
				cl_stubDispatcher = 0;
			else {
				++cl_nDispatchSwitches;
				cl_stubDispatcher = cl_stubList.back ();
				cl_stubDispatcher->SignalResponse (RpcStub::WR_DISPATCH);
			}
```

```
                cl_sendMutex.Unlock ();

                return (RpcStub::WR_DONE);
            }
        }

        assert (error);
        cl_sendMutex.WriteLock ();
        CloseConnection (error);
        cl_stubDispatcher = 0;
        cl_sendMutex.Unlock ();

        return (RpcStub::WR_DONE);
}
```