15 RMI through firewalls

Introduction—Firewalls—SOCKS—HTTP tunnelling—Firewalls and RMI— GIOP Proxies—A note on callbacks—A note on firewalls

15.1 In this chapter

This chapter describes the implications of deploying RMI over the Internet, or over large intranets containing firewalls.[1] If you intend to develop applications which are to be deployed across such networks, you *must* read this chapter.

In our discussions of networks so far we have omitted the topic of firewalls. We have implicitly assumed only the existence of a TCP/IP local area network (LAN).

From one point of view, the Internet is nothing but an extremely large TCP/IP wide area network (WAN). However, making the jump from a LAN to the Internet is not a trivial exercise. There are significant complications which are relevant to RMI.

15.2 Firewalls

In order to prevent office-wide LANs becoming part of the global Internet, a "firewall" is normally placed at the gateway between the LAN and the Internet proper. Like a physical firewall, an Internet firewall's purpose is to provide a high level of security to those on the protected side by preventing dangerous elements from entering.

Figure 15.1 shows a simple view of a firewall.

1. Our discussion is limited to the intersection of RMI and firewalls; it is by no means intended to provide complete coverage of firewalls or network perimeter security techniques in general. This is a large topic. For further information, see Cheswick and Bellovin, *Firewalls and Internet Security.*

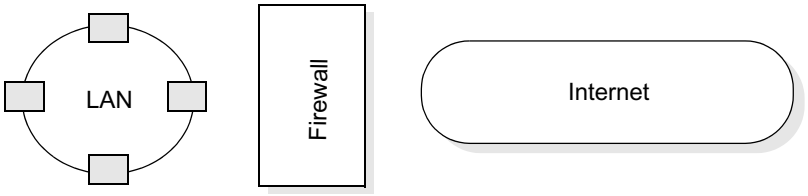

Figure 15.1 Simple view of firewall

The function of an Internet firewall is to block all except authorized communications between the Internet and the inner LAN. Firewalls are of two kinds, usually paired together:

- transport firewalls
- application firewalls.

15.2.1 Transport firewalls

Transport firewalls are generally hardware boxes. They only understand a general transport protocol, typically IP (including TCP and User Datagram Protocol (UDP)), and operate simply by allowing or disallowing connection requests based on the source and target IP address and port number.

Transport firewalls generally block *all* TCP and UDP ports *except* certain "well-known" ones,[2] such as: SMTP (25), HTTP (80), POP3 (110), NNTP (119), SNMP (120), and IMAP (143). Some of the well-known ports such as Telnet (23) are usually blocked. The ports for FTP (20-21) are sometimes blocked, sometimes not. "Anonymous" application-defined ports (1024 and up) are generally blocked, including the ports for *rmid* (1098) and the RMI registry (1099), and all ports allocated by the RMI system for remote objects.

FTP is the Internet File Transfer Protocol. SMTP is the Simple Mail Transfer Protocol used between e-mail servers. Telnet is a protocol and application suite which provides remote terminal access. POP stands for the Post Office Protocol used by e-mail clients. NNTP is the Network News Transfer Protocol. Internet Message Access Protocol (IMAP) is an e-mail retrieval protocol used by e-mail clients.

2. See IETF RFC 1700, as amended.

HTTP is the acronym for HyperText Transfer Protocol, the transport protocol associated with HTML. It is the communications protocol observed between Web browsers and Web servers. It is not to be confused with HTML itself, which is the page markup language of the World Wide Web, and which is transported via HTTP.

15.2.2 Application firewalls

Application firewalls are also known as *proxies*. An application firewall understands a particular application protocol, such as FTP and HTTP, and interposes itself in the conversation between a client behind the transport firewall and a server outside it. To the client, it appears to be the server; to the real server, it appears to be the client. The application firewall ensures that what is going over the connection really is a conversation in the application protocol concerned, and it is controlled by an application-specific configuration which permits or denies access to the outside based on application-specific considerations.

Figure 15.2 illustrates the relationship between transport firewalls and application firewalls.

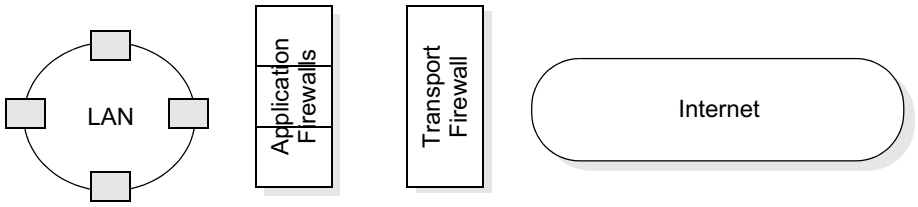

Figure 15.2 Application and transport firewalls

Transport firewalls generally restrict outgoing connections to those originated by an application firewall. The combination of the transport firewall and the application firewalls constitutes the installation's "total" firewall.

15.2.3 HTTP proxies

The best-known type of application firewall is the HTTP proxy.

Applications such as Web browsers can be configured to send HTTP requests via an HTTP proxy server. The functions of the HTTP proxy server are (a) to ensure that the data is indeed HTTP requests and responses, (b) to control which target sites and

ports are allowed, and (c) to forward the request to the target port. By this means, "harmless" applications such as Web browsers can be configured to penetrate the firewall—as long as what is going through it really is HTTP.

The restriction to HTTP works because the HTTP proxy server really is a Web server, and only understands the HTTP protocol.

> HTTP proxies usually also provide a caching service for Web pages: this service is outside the scope of this discussion.

For the purpose of this discussion, HTTP proxy services either:

- allow HTTP to be sent to any port on the target host, or
- allow HTTP to be sent only to the well-known HTTP port 80 on the target host.

Java clients can also be configured to send HTTP requests via an HTTP proxy server, by setting the system properties http.proxyHost and http.proxyPort. These properties control the implementation in Java of HTTP URLs; they also control the operation of RMI clients, as we shall see.

15.2.4 Firewall configuration and control

Firewall configurations are under the control of network administrators. In theory, network administrators can be persuaded to "open" certain ports in order to support RMI. In practice, they are generally rather hard to convince about this: firewall policy critically affects corporate security.

15.3 SOCKS

SOCKS is the title of another "peephole" through the client-side firewall. Logically, a SOCKS server is a general-purpose application proxy, or a "SOCKS proxy". It provides a means of encapsulating an authenticated conversation between a known client inside the firewall and a SOCKS server at the firewall. It permits the client to connect to a service outside the firewall on an arbitrary port, while allowing the network administrator to control which clients may access this service, and without exposing arbitrary client-side ports through the firewall.[3]

3. SOCKS v5 is specicied in RFC 1928.

> The conversation between the client and the SOCKS server is authenticated. However, don't assume that this secures the conversation in any way. The conversation between the SOCKS server and the other end is not authenticated, and that is the part that takes place over the Internet.

As RMI clients use java.net.Socket to connect to RMI servers, and as java.net.Socket will conduct a SOCKS-based conversation if the system property socksProxyHost is set, RMI clients can use SOCKS to get through their own firewalls and communicate with RMI servers in the public internet.

15.3.1 Limitations of SOCKS

SOCKS is a client-side solution only:

- SOCKS cannot address the problem of an RMI server behind its own firewall
- SOCKS cannot handle RMI callbacks: servers outside a firewall cannot execute callbacks to clients located behind a firewall via SOCKS; see also §15.8.

There is no such thing as server-side SOCKS.[4] As long as this remains the case, there can never be any support for it in Java.

15.4 HTTP tunnelling

Most high-level protocols which attempt to solve the firewall issue do so by the technique of "HTTP tunnelling", in which the communications are packaged inside HTTP requests and responses, and sent via the well-known HTTP port 80, which is normally left "open" in firewalls.

This is rather like enclosing a sealed addressed envelope inside another sealed addressed envelope, with the understanding that the inner envelope is to be posted to the inner addressee when received by the outer addressee (the recipient of the outer envelope). Consider the example of an over-supervised girl (Alice) trying to write to her boyfriend (Bob) when her outgoing mail is scrutinized by her parents. Alice

4. Notwithstanding statements which have appeared in the RMI FAQ and mailing list: apparently these statements refer to a defunct RBIND command, which does not appear in RFC 1928 or any other public SOCKS protocol document.. SOCKS does have a BIND command, which supports a one-shot client-side callback. RMI callbacks are not one-shot, and neither Java nor RMI knows a client-side callback when it sees one, so this feature is not usable by RMI.

seals a letter to her boyfriend inside a letter to an approved girlfriend (Tracey). The letter to Tracey gets through the parental "firewall", and Tracey posts the inner envelope to Bob on receipt.

HTTP tunnelling only works through firewalls with HTTP proxies.

15.4.1 Limitations of HTTP tunnelling

HTTP tunnelling is a client-side solution only:

- it cannot address the problem of an RMI server behind its own firewall
- it cannot handle client-side callbacks: servers outside a firewall cannot execute callbacks to clients located behind a firewall via SOCKS; see also §15.8
- it cannot address the problem of an RMI server behind a firewall
- an HTTP server must be present at the server end, and an HTTP proxy server is usually required at the client-side firewall.

15.5 Firewalls and RMI

OK, so what does all that have to do with RMI? When an RMI client first connects to a remote object, the default RMI client socket factory tries to create a client socket—a java.net.Socket—directly connected to the host in which the required remote object is running.

If there is an intermediate firewall in the way, this connection attempt will fail quite quickly, with either a java.net.NoRouteToHostException or a java.net.UnknownHostException.[5] If the system property http.proxyHost is set, RMI's default transport then tries, in order, two HTTP tunnelling techniques to connect to the remote host.

1. Direct forwarding over HTTP.
2. Indirect forwarding over HTTP.

5. Or, unfortunately, a ConnectionException, depending on the firewall vendor. The reasons for this are abstruse: see §15.9.

In both cases, an HTTP POST request is sent to the HTTP proxy. The proxy address is formed from the system properties http.proxyHost, and http.proxyPort if set, otherwise port 80 is assumed.

15.5.1 Direct forwarding over HTTP

In this technique, an HTTP request is sent to the HTTP proxy, to be forwarded directly to the actual target port.

The direct-forwarding technique involves:

- the client
- the local HTTP proxy and any intermediate proxies
- the remote RMI server.

Direct forwarding constructs a URL of the form:

```
http://host[:port]
```

where *host* is the target host and *port* is the target port. This will work if all intermediate firewalls and HTTP proxies permit HTTP connections to arbitrary ports. Most firewall configurations do *not* permit this, in which case indirect forwarding must be tried.

15.5.2 Indirect forwarding over HTTP and CGI

In this technique, an HTTP request is sent to the HTTP proxy, to be forwarded to port 80 of the target machine, and from there to a "CGI script".[6] This assumes that (a) there is an HTTP server listening on that port, and (b) the RMI CGI script is installed with the HTTP server (see §15.5.3). Indirect forwarding constructs a URL of the form:

```
http://host:80/cgi-bin/java-rmi?forward=port
```

6. A CGI (Common Gateway Interface) script is an executable program which is invoked by a Web server to handle URLs in a certain format.

where *host* is the target host, *80* is the standard HTTP port, and *port* is the real target port at which the RMI service required is listening. This URL invokes the CGI script, which forwards the call to the target RMI server port on the same machine.

The indirect-forwarding technique involves:

- the client
- the local HTTP proxy and any intermediate proxies
- the remote HTTP server
- the remote RMI CGI script
- the remote RMI server.

If either of these strategies succeeds, the target machine can unpack the RMI call from the surrounding HTTP protocol, execute the call, and embed the result in an HTTP response sent back along the same path.

These techniques together amount to an "RMI over JRMP over HTTP" protocol, or RMI/HTTP for short.

For further information about RMI/HTTP see the RMI specification, §§3.5, 10.4.

15.5.3 The *java-rmi* CGI script

The *java-rmi* CGI script acts as the target of an indirect forwarding request. It is invoked with its input and output connected to the source socket which had been accepted by the Web server; it makes a target socket connection to the specified port on the local machine, reads the incoming data from its source socket, sends it to the target socket, reads the reply from the target, and writes it back to the source. It then closes the source and target sockets and exits.

The *java-rmi* CGI script is distributed as *java-rmi.exe* on Windows platforms and *java-rmi.sh* on Unix-like platforms including Linux and Solaris. It is really a script only on Unix-like platforms: on Windows platforms it is a fully-fledged executable file. It is distributed in the bin directory of the JDK (the topmost such directory, not the jre/bin directory), but it must be installed by copying it to where the Web server expects to find it, typically the Web server's cgi/bin directory. Check your Web server's documentation for details.

15.5.4 The RMI servlet handler

The *java-rmi* CGI script is also distributed as a "servlet".

A *servlet* is a Java class which extends the abstract javax.servlet.http.HttpServlet class, and acts as an integrated extension of a Web server, typically as an "alias" for a CGI request. Servlets are supported by all major Web servers.

The purpose of the RMI servlet handler is the same as that of the *java-rmi* CGI script—to accept indirect RMI forwarding requests. Like all CGI scripts, the *java-rmi* CGI script suffers from fairly extreme overheads. Each invocation of the script requires the Web server to invoke a new process, which in turn executes first the script itself and then the Java runtime (to execute the real redirector, which is a Java program). By contrast, the RMI servlet handler is installed once, instantiated on demand, and reused for each new request, rather than being reloaded each time. It is only discarded after a period of inactivity, according to the Web server's rules for Java servlets. The efficiency gains over the script version are very significant.

From JDK 1.2, the RMI servlet handler is distributed with the JDK.[7] If you have an earlier version of the JDK, the servlet can be obtained from Sun's Java Web site.[8] The servlet itself is not JDK-specific.

Unlike the CGI script, which merely has to be physically present when invoked, the RMI servlet handler must be explicitly installed in your Web server somehow. Typically, a Web server has a facility for defining servlets as "aliases" for CGI scripts. Consult the RMI servlet handler documentation and your Web server's documentation, for further details.

15.5.5 Authenticated HTTP tunnelling

HTTP servers can be set up to require authentication from a client. You can make use of this feature to implement an initial level of security at your server-side HTTP Server. Alternatively, your RMI clients may encounter this issue when traversing intermediate HTTP proxies. When HTTP authentication is required you must use the services of java.net.Authenticator to supply the authentication response.

7. in the docs/guide/rmi/archives directory.
8. http://java.sun.com/products/jdk1.3/docs/guide/rmi/archives/rmiservlethandler.zip .

15.5.6 Disabling RMI/HTTP

It is possible to *disable* the client's attempts at HTTP tunnelling by setting the system property java.rmi.server.disableHttp to true. Its default value is false.

15.5.7 Limitations of RMI/HTTP

RMI's HTTP tunnelling suffers from quite a number of limitations which, depending on your circumstances, you may find more or less severe. Some of these are as stated in §15.4.1, being inherent in the technique, and some of them are specific to RMI's and its implementation of HTTP tunnelling:

1. It cannot address the problem of an RMI's server behind a firewall.
2. It cannot handle RMI's callbacks (see §15.8).
3. An HTTP server must be present at the server end, and an HTTP proxy server is usually required at the client-side firewall.
4. Usually, either the *java-rmi* CGI script or the RMI's servlet handler must be installed at the server-side HTTP server.
5. It is firewall-dependent (see §15.9).
6. When the server is known to be beyond a firewall, the initial direct connection attempt is really a waste of time, yet no facility is provided—other than the deprecated practice of using the hidden RMIHttpToXXXSocketFactory classes directly—for configuring the client *only* to use HTTP tunnelling.
7. It is implemented by the default RMI socket factory: if an RMI's client installs its own, or if an RMI's server defines a client socket factory, HTTP tunnelling behaviour is lost.[9]
8. It is up to 10 times as slow; at its worst if you are using the script rather than the servlet.

9. It can be regained by a certain amount of extra programming. You have to arrange your own socket factory to return a "wrapped" socket which extends Socket, but which delegates its I/O to a Socket supplied by RMI's default socket factory, i.e. the result of RMISocketFactory.getDefaultSocketFactory, after doing whatever else it has to do to the data stream. In cases such as SSL, where a second socket factory mechanism is used to obtain the actual socket, it may not be possible to stack sockets together in this way.

9. It implies a new TCP connection per RMI's call,[10] whereas RMI over TCP reuses an established connection where possible: establishing a TCP connection requires a three-way handshake, and closing it requires a four-way handshake.

10. It was unusably unreliable prior to JDK 1.2.2.

15.5.8 Summary

RMI attempts two kinds of HTTP tunnelling if the following conditions are met:

- the initial direct-connect attempt failed with a NoRouteToHostException or UnknownHostException
- the system properties http.proxyHost and http.proxyPort are set
- the system property java.rmi.server.disableHttp is not set
- the default client socket factory is in use.

The HTTP-tunnelled RMI request will be understood at the server if the following conditions are met:

- the default server socket factory is in use, or
- Java JDK 1.2.2 or later is in use at the server.

15.6 GIOP Proxies

OMG has adopted a submission for the CORBA 3.0 specification on CORBA/Firewall Security.[11] This submission specifies (a) a CORBA application firewall—a GIOP proxy, and (b) a bidirectional extension of GIOP to handle callbacks into clients behind firewalls. An extension to handle secured proxy connections is also proposed.

We saw in Chapter 14 that RMI can be made to communicate via IIOP instead of JRMP, and that GIOP is the data part of the IIOP protocol.[12] When implementations of the CORBA 3.0 specification incorporating GIOP proxies appear, and when RMI/IIOP conforms to CORBA 3.0, GIOP-proxying and bidirectional GIOP may between them

10. In Sun's implementation, RMI HTTP clients reuse TCP connections to the same URL within a short timeout period. However, a new TCP connection to the target is still initiated per RMI call by the CGI script or servlet at the Web server.
11. *Joint Revised Submission CORBA/Firewall Security.*
12. Formally speaking, IIOP is the application of GIOP to TCP/IP.

provide a complete solution to the RMI firewall problem—for RMI/IIOP services only.

15.6.1 Benefits

Unlike the current RMI/HTTP implementation, the CORBA/Firewall Security proposal takes account of servers behind one or more firewalls (as well as clients behind firewalls), i.e. true inter-enclave GIOP communications across arbitrary numbers of firewalls protecting both clients and servers.

The CORBA/Firewall Security proposal does not use an error-code fallback strategy to redirect traffic via the proxy. Instead, clients and servers behind firewalls are expected to be configured with proxy information. For this reason, the proposal does not rely on firewall properties, unlike RMI, and therefore does not suffer from the incompatibility between RMI's expectations of firewall errors and actual implementations of firewalls.

15.6.2 Limitations

The GIOP proxy solution will only be applicable to those RMI users prepared to use the IIOP protocol, and therefore prepared to lose activation and DGC from their RMI applications, as discussed in Chapter 14.

The GIOP proxy and bidirectional GIOP solutions will not be usable until:

- CORBA 3.0 is approved
- RMI/IIOP is upgraded to conform to it
- GIOP proxies become available and installed in quantity.

15.7 The RMI Proxy

The RMI Proxy is a commercial product which addresses Java RMI over the Internet. In the terms of this chapter, it is a JRMP proxy—an application proxy for the JRMP protocol.

The RMI Proxy executes in the application-proxy host; RMI servers and clients are made aware of it via an rmi.proxyHost property and a ProxyNaming class which substitutes for java.rmi.Naming.

15.7.1 Benefits

The RMI Proxy supports most of the firewall configurations supported by the GIOP Proxy described above:

- clients behind firewalls
- servers behind firewalls
- client-side callbacks
- nested client-side and server-side firewall enclaves
- clients behind any combination of RMI Proxies and SOCKS servers.

It provides extensive security management, based on the Java 2 Security Model. Access to specific RMI interfaces can be controlled down to the level of individual methods, via combinations of client hostname/IP address, codebase, and interface/method name.

Attempts to penetrate the RMI Proxy via protocols other than JRMP do not succeed, and RMI/JRMP calls which fail the security configuration are not transmitted to the server, but instead cause a java.rmi.AccessException at the client. As this exception extends java.rmi.RemoteException, there is no major coding impact on clients or servers.

15.7.2 Limitations

The RMI Proxy is subject to the following limitations:[13]

- no support for RMI/IIOP
- no support for RMI socket factories
- no support for secure conversations.

Some of these features are actively planned; some are subject to features in future JDK releases; the remainder are under review.

Disclosure: The RMI Proxy was conceived, designed, and implemented by Esmond Pitt, one of the authors of this book, who has a financial interest in this product. For further information see the RMI Proxy home page at http://www.rmiproxy.com.

13. at the time of writing (March 2001).

15.8 A note on callbacks

In general, servers outside a firewall cannot execute callbacks to clients located behind a firewall, because the callback is itself a server behind a firewall.

> The RMI multiplexing protocol[14] was a temporary solution to this problem, now obsolete. It provides multiple virtual RMI connections over a single physical TCP connection. It is layered over JRMP, and hence cannot support RMI/IIOP; it does not scale; it cannot time-out a connection. It was introduced into the JDK 1.02 pre-release of RMI to overcome extreme applet security restrictions in Netscape's browser JVM, which have long since been corrected. Client-side initiation of the Multiplexing Protocol is disabled as from JDK 1.2.2, although the protocol is still understood by RMI servers so as to support legacy clients. For more information see the evaluations for Bug Ids 4183204 and 4257730 in the Java Developer Connection Bug Parade.

Enhancements being planned for JDK 1.4 include an improved RMI protocol which can handle more than one RMI call at a time per connection. It was not clear at the time of writing whether this protocol will be bidirectional, which would solve the callback problem.

It should also be noted that firewall experts consider client-side callbacks to be intrinsic security risks.[15]

15.9 A note on firewall implementations and RMI

When certain brands of firewall reject a connection request for a blocked port in the range 1024 or above, the client's JVM throws a java.net.ConnectException rather than the anticipated exceptions: either java.net.NoRouteToHostException or java.net.UnknownHostException as described in §15.5. In this case the RMI HTTP fallback mechanism described in that section is not triggered. This section describes what is really going on.

1. The documents which specify the TCP/IP protocol state (a) that a host "must" respond to a connection request to a listening port with a TCP RST segment if the connection is not "allowed by the user and the system",[16] and (b) a gateway

14. RMI specification, §10.6.
15. Cheswick and Bellovin, *Repelling the Wily Hacker*, §C.4 p. 254.
16. IETF RFC 793.

"may" respond to a connection request to any port "blocked by administrative action" with an ICMP Destination Unreachable message.[17]

2. The distinction in these specifications between listening ports and blocked ports is significant, as is the distinction between hosts and gateways, and between "must" and "may". This has led to different interpretations.
3. Most TCP/IP implementations and router, gateway, and firewall vendors have adopted (b) as the standard behaviour for firewalls, treating them as gateways rather than as hosts.
4. However, some firewalls handle a connection request for a blocked port in the application-defined range (1024 upwards) by sending a TCP RST response instead, meaning "port not active". Apparently, some other firewalls don't respond to such requests at all.
5. When processing responses to socket connection requests, Java maps a TCP RST response into a ConnectException, and it maps an ICMP Destination Unreachable response into a NoRouteToHostException.

It is undesirable for RMI to treat ConnectException as a possible indication of a firewall, because this exception—and the underlying TCP RST—is the response "normally" expected when no process is listening on the port. It is usually issued by the target host, possibly after quite a wait while the request and the response traverse the Internet. It would therfore be most undesirable to apply the HTTP-tunnelling technique to *every* refused connection, for performance reasons alone—it would at least triple the wait just referred to.

For these reasons, when an RMI client attempts to open a blocked application-level port through such a firewall, it does not get either of the exceptions which initiate the two phases of RMI/HTTP, so it never attempts an RMI/HTTP proxy connection.

17. IETF RFC 792, and RFC 1122, §3.2.2.1.