A Transport-Level Proxy for Secure Multimedia Streams

KING P. FUNG AND ROCKY K.C. CHANG
The Hong Kong Polytechnic University

To provide secure traversal service, firewalls need more than static packet filtering and application-level proxies. SOCKS is an application-independent transport-level proxy that offers user-level authentication and data encryption. An extended SOCKS UDP binding model with appropriate socket calls is proposed to provide complete support for UDP-based, multimedia streaming applications.

The increasing popularity of multimedia streaming applications, such as RealNetworks' RealPlayer and Microsoft's NetMeeting and Windows Media Player, pose challenges to the Internet infrastructure, particularly in providing secure firewall traversal for video and audio streams. Static packet filtering, the approach used by firewalls today, cannot adequately support the security needs of these applications for several reasons. First, common packet-filtering policies block almost all incoming user datagram protocol (UDP) traffic except for a few services such as the domain name service (DNS), network time protocol (NTP), and Archie.[1] Many multimedia streaming applications, however, employ UDP for data transport because for multimedia streaming, minimal delay and delay jitter is more important than total reliability. Second, the problem with UDP unicast also applies to IP multicast, which supports UDP only. IP multicast, nevertheless, is essential to a scalable solution for Internet-wide multimedia streaming. Finally, in multimedia streaming the UDP ports on the client and server sides are usually dynamically assigned through the application protocol, which is further complicated by the network address translation (NAT) performed by firewalls. As a result, special configurations, such as fixing a particular UDP port for receiving multimedia streams, are often required.[2]

This article investigates the suitability of SOCKS, a transport-level proxy solution adopted by the Internet Engineering Task Force's Authenticated Firewall Traversal Working Group, for supporting multimedia streaming applications.[3] The name SOCKS came from Secure Sockets, originally developed by David Koblas and Michelle Koblas.[4] Specifically, we identify two problems encountered by SOCKS: a mismatch of call sequences between the SOCKS' transport model and multimedia streaming protocols' transport models, and inadequate socket call support for UDP binding. Failure to resolve these problems results in the firewall's blocking of the multimedia streams. We use the real-time streaming protocol (RTSP), an IETF-proposed standard, to illustrate the problems, and we propose an enhanced SOCKS to provide complete support for UDP-based applications, particularly multimedia streaming applications.

 1089-7801/00/$10.00 ©2000 IEEE

Although we consider only RTSP here, the enhanced SOCKS can be applied to other multimedia applications, such as Microsoft's NetShow, because it adopts a transport model similar to RTSP. Moreover, although our focus is the unicast delivery of multimedia streams, SOCKS also supports multicasting.

APPROACHES TO SECURE SERVICE

Special configurations are, as mentioned, one way to offset the inadequacy of static packet filtering in providing secure firewall traversal service. Another approach employs application-level proxies. Such proxies are indeed essential for application protocols such as HTTP and FTP; however, these protocols require additional processing overhead. Future enhancements and modifications to the application protocols also mandate a corresponding update on the proxy. Moreover, additional coding must be implemented to make the applications aware of the proxy server. Managing separate proxies for different applications is also problematic.

Yet another approach employs a stateful inspection engine to filter incoming packets. The engine extracts relevant state information from incoming packets and maintains it in dynamic state tables for evaluating subsequent connection attempts.

Transport-level proxies provide an approach midway between static packet filtering and application-level proxies. Unlike the stateful inspection approach, transport-level proxies do not require sophisticated configuration and management. A transport-level proxy, because it is situated between the application and transport layers, provides a generic transport proxy service to handle all TCP and UDP applications. The generic proxy intercepts socket-call procedures invoked by an application and sets up a proxy connection based on the application's transport requirement.

An application-independent transport-level proxy, providing user-level authentication and data encryption, can offer secure, scalable firewall traversal service for many multimedia streaming applications. SOCKS, the IETF's proposed transport-level proxy solution, is a prime example of such proxies, as we explain.

SOCKS AND RTSP OVERVIEW

SOCKS employs numerous mechanisms to secure traffic flowing through a SOCKS-enabled firewall. A proxy server provides relay services for UDP and TCP traffic with network address and port translation, which hides the true client's identity from outside the firewall. Like most proxies, SOCKS is designed for situations where the client is inside of the firewall and the server is outside. The proxy server, therefore, only allows TCP connection and UDP datagram transmissions initiated from inside the firewall, filtering all other unauthorized incoming traffic. That is, the client (via the proxy client), through the SOCKS protocol, notifies the proxy server of the source and destination socket address pair for the permitted TCP or UDP traffic. In responding to the request, the proxy server "drills a hole" in the firewall for the incoming traffic with the socket address pair.

In this article, *proxy* refers to the SOCKS proxy; thus, proxy server and proxy client refer to the SOCKS server and the SOCKS client, respectively. The unqualified terms *clients* and *servers* refer to application clients and application servers, respectively.

SOCKS refers to SOCKS5 specified in RFC 1928 with Chouinard's proposed UDP extension.[5,6] This extension introduced an enhanced UDP mode on top of the SOCKS5 protocol in order to support incoming UDP traffic, such as multimedia streams. The enhanced UDP mode is assumed wherever we discuss proxy support for UDP. However, as will be seen later, this enhanced UDP mode alone (without our enhanced SOCKS) still cannot provide secure firewall traversal service for incoming multimedia streams. Another Internet draft, proposing a similar UDP extension to SOCKS5, is not discussed here except to note that our enhanced SOCKS can also be implemented with this extension.[7]

Socksified Clients

A client must be "socksified" to use the proxy service. Socksification creates a thin SOCKS layer between the application and transport layers. A proxy client resides in that layer to communicate with a proxy server.

There are two ways to socksify a client.[8] The first method requires recompiling and relinking the applications. The existing SOCKS protocol library provides wrapper programs to interface with the client program. Wrapper programs take control of the standard socket library calls for TCP and UDP; for example, the UDP `bind()` socket call will be replaced by the corresponding `lsUdpBind()` call for SOCKS. The wrapper program then exchanges SOCKS protocol messages with the proxy server; therefore, the wrapper program can be viewed as the proxy client in the SOCKS protocol model. The second method is to perform dynamic library

linking, which makes the socksification process transparent to the applications.

When a socksified client executes a TCP connect() socket call for TCP-based applications or a UDP bind() socket call for UDP-based applications, the proxy client intercepts the call. If the local policy determines to redirect the call to the proxy server, the proxy client first establishes a TCP connection to the proxy server at port 1080. The TCP connection serves as a control channel for negotiating authentication methods and encryption options, and for client authentication. After that, the proxy client formulates a corresponding SOCKS command: a SOCKS connect command for TCP connect(), and a SOCKS Enhanced UDP Bind command for UDP bind() (the corresponding command, per RFC 1928, in SOCKS5 UDP binding is UDP associate), and sends it to the proxy server through the control channel.

SOCKS for TCP-Based Applications

Within TCP-based applications, the TCP control channel is also used for TCP data relay—an in-band model—after the SOCKS connect command is successful. On the left-hand side of Figure 1, the client's TCP data is redirected to the proxy server's internal relay socket, which is on the same side as the local client's.

The proxy server relays the data to the intended remote socket using an external relay socket, which is on the same side as the remote server. The *remote socket* and *local socket* are associated with the remote server and local client, respectively. The proxy server actually creates a new TCP connection to the remote socket for relaying TCP data between the local client and the remote server. Thus, the client is unaware of SOCKS' presence, and the remote server thinks that the proxy server is the final destination.

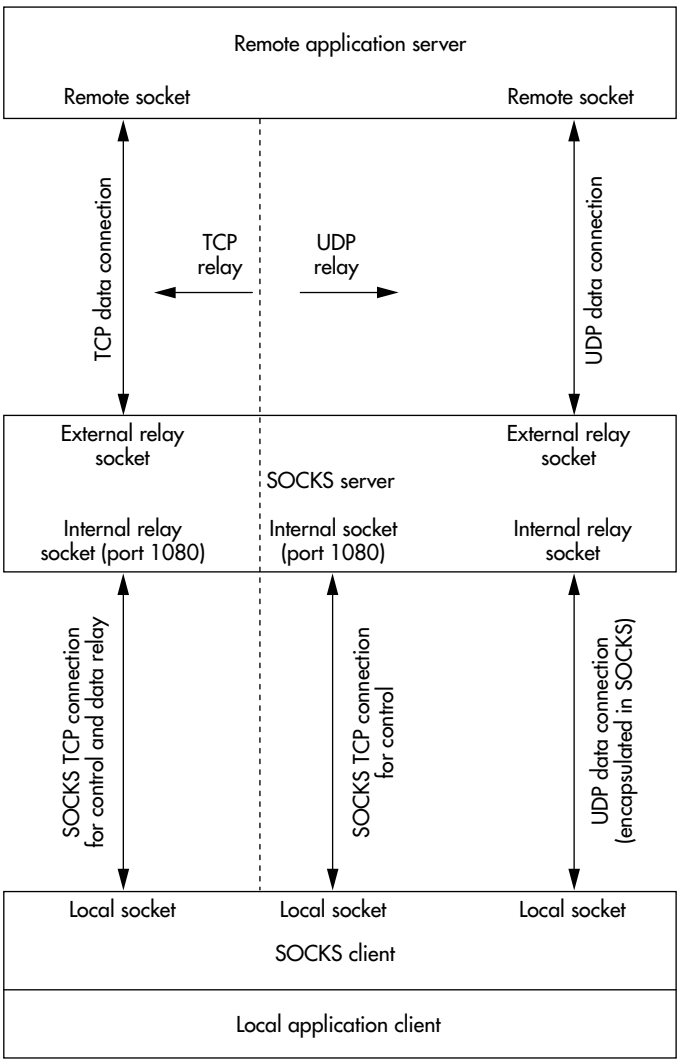

Figure 1. The SOCKS transport model. SOCKS uses a single TCP channel for both SOCKS control messages and data transfer for TCP-based applications. However, SOCKS uses a TCP channel only for SOCKS control messages for UDP-based applications. UDP datagrams are transported separately with SOCKS encapsulation.

SOCKS for UDP-Based Applications

SOCKS uses an out-of-band transport model for UDP-based applications—the TCP connection between the proxy server's internal socket and proxy client is used only for SOCKS control messages, not for data transfer. Specifically, the following messages are exchanged between the proxy client and proxy server in the TCP control channel:

- The proxy client sends a SOCKS Enhanced UDP Bind command to the proxy server. The command is accompanied by a local socket address from which the proxy client will send UDP datagrams for this UDP association, and by a remote socket address that the proxy client wishes to send to or receive from.
- In response to the SOCKS Enhanced UDP Bind command, the proxy server provides an internal and an external relay socket address it will use for this UDP association. The internal relay socket lets the proxy client relay UDP datagrams to the proxy server. The external relay socket lets the proxy server send or receive on behalf of the proxy client for this UDP association.

If the SOCKS Enhanced UDP Bind command is successful, the proxy server will have set up an inter-

nal UDP data relay connection between its internal relay socket and the proxy client. The proxy server then uses an external relay socket to relay the UDP data between the local client and the remote server. The original UDP association between the local and remote sockets now consists of a UDP association between the local and the internal relay sockets, and a UDP association between the external relay and the remote sockets. As a result, the proxy server hides the true client from the remote server, and the proxy client hides the proxy server from the client.

RTSP's Transport Model

A common model for delivering multimedia streams generally involves a TCP connection and one or more UDP associations. The TCP connection serves as a control channel between a client and a server, and the server's TCP port is well known. The UDP associations are used for delivering multimedia streams from a streaming server to the client. The UDP ports on both sides are usually dynamically assigned (perhaps within a predefined or configurable range in an actual implementation, such as the RealPlayer application[9]), and the assigned addresses are communicated through the TCP control channel between client and server.

The IETF's Multiparty Multimedia Session Control Working Group developed RTSP to serve as a common platform for managing and controlling multimedia streams.[10] RTSP's transport model adopts the common approach just described.[11] An RTSP client first establishes a TCP connection to an RTSP server at port 554. (The connection could be, but seldom is, implemented as UDP instead of TCP.) The message exchange sequence between client and server in this control channel is as follows:

1. An RTSP client sends a setup request to an RTSP server. The request specifies the URL of the requested stream and the destination socket address for receiving the stream, which is associated with the RTSP client in the absence of a proxy service.
2. The RTSP server responds with the source socket address of the streaming server that will deliver the requested stream. This socket address may or may not be associated with the RTSP server.
3. The RTSP client sends a play request for the requested stream, and the streaming server with the source socket address then starts sending the requested UDP stream to the destination socket address.

The data packet used for the multimedia streams can be based on the real-time protocol (RTP) or other proprietary protocol, such as RealNetworks' real data transport (RDT). Data packets are sent using unicast UDP, multicast UDP, or inline TCP (via the TCP control channel). Moreover, when RTP is used for the data transport, another UDP channel is set up between the RTSP client and RTSP server for the real-time control protocol (RTCP). An application-level proxy for RTSP is described elsewhere.[9]

PROBLEMS RECEIVING MULTIMEDIA STREAMS

Figure 2 shows RTSP's transport model via SOCKS. SOCKS establishes two TCP connections—one for internal and the other for external—for RTSP's control channel, and establishes another internal TCP connection plus UDP associations for receiving multimedia streams. SOCKS can relay the RTSP's control channel between a local RTSP client and a remote RTSP server through the SOCKS' TCP connections. After the internal TCP connection is set up and the local client is authenticated, the local client can send different RTSP methods to the remote server through RTSP's control channel.

Even with the enhanced UDP mode, two problems prevent SOCKS from receiving multimedia streams. The first problem results from a mismatch of RTSP and SOCKS call sequences, as depicted in Figure 3 (on page 62). Specifically, an RTSP client is required to include its local socket address for receiving the requested stream's remote socket address in its setup request to an RTSP server. The local socket is associated with the RTSP client in the absence of intermediate proxies. But with the SOCKS proxy separating the client and server, the local socket is referred to the proxy server's external relay socket for this UDP association. However, according to the SOCKS protocol, this information is available only after a successful SOCKS Enhanced UDP Bind command is executed.

On the other hand, the proxy server requires the proxy client to provide the remote socket address along with the SOCKS Enhanced UDP Bind command before it will reply with its external relay socket address. But the remote socket address is available only after the local client sends the setup request to the RTSP server, according to the RTSP protocol we described.

As a result, the SOCKS Enhanced UDP Bind command and the RTSP setup request are mutually dependent on each other to provide the needed information. The proposed, enhanced UDP

mode thus only provides fields for the remote socket address of the streaming source, but the actual address information is not available by the time the Bind command is sent to the proxy server.[6] This problem causes the firewall to reject the requested multimedia stream because the proxy server fails to learn the stream's correct remote socket address.

The second problem concerns the interface between an RTSP client and a proxy client. The UDP bind() socket call does not have an argument for specifying a socket address. Therefore, the RTSP client is unable to pass the remote socket address to the proxy client through the socket call, even if the remote socket address were available.

We have examined the only publicly available SOCKS source code, made available by NEC Corporation.[12] This implementation made several workarounds to resolve these two problems. Without the remote socket address field in the UDP bind() call and the remote socket address itself, the implementation assumes that the remote socket's network address is the same as the destination network address in the previous SOCKS TCP connection opened by the same proxy client. That is, the streaming server and the RTSP server are assumed to reside in the same machine.

This assumption is obviously not always correct; for example, a collection of identical streaming servers can provide a scalable multimedia service, or the video and audio servers can reside in different machines. In some cases, multiple streams can even be sent concurrently to improve the application's performance.

As for the UDP port of the remote socket, the implementation simply sets it to any port; thus the proxy server checks only the source address of the incoming UDP datagrams. This workaround clearly compromises network security because the security check applies only to the host level. Therefore, new solutions that solve the two problems for all scenarios without compromising the firewall's security are clearly needed.

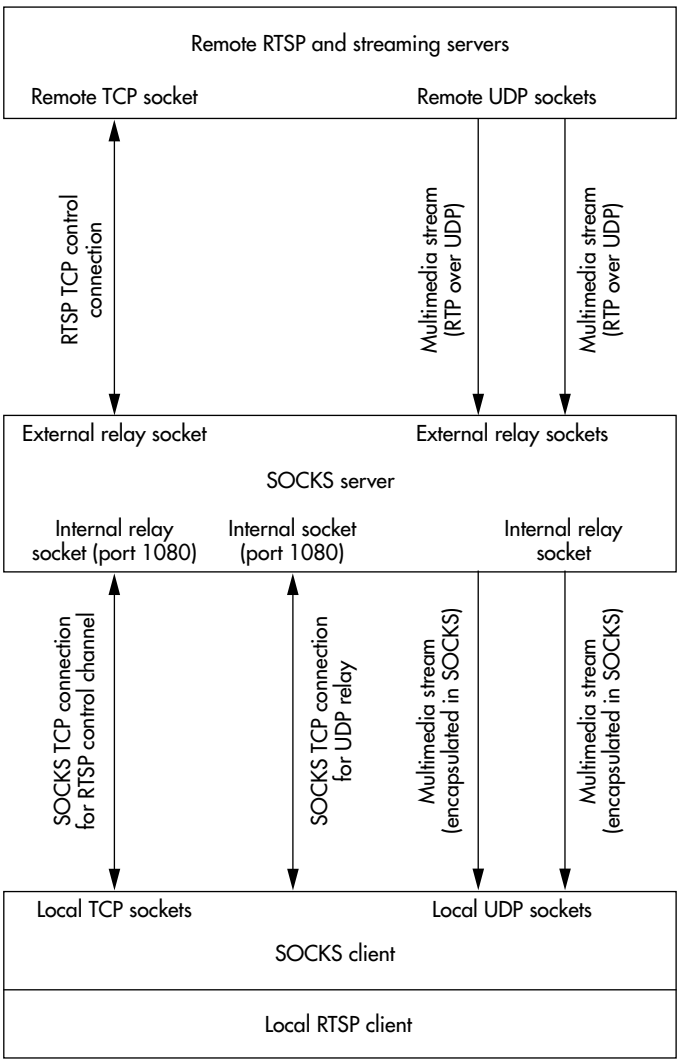

Figure 2. RTSP's transport model via SOCKS. SOCKS must handle both the RTSP control channel and UDP-based multimedia streams. Between proxy client and proxy server, SOCKS therefore uses a single TCP channel for the RTSP control channel, and another TCP channel for the UDP-based multimedia streams.

ENHANCED SOCKS FOR RTSP-BASED APPLICATIONS

Our solutions to the above problems make no assumptions about the remote socket address. Our goal is to solve the problems without compromising security and to leave the application protocol unchanged insofar as possible—the RTSP client and server call sequences must be preserved. These requirements, therefore, led us to extend the SOCKS UDP association model to handle the scenario in which the remote socket address is unavailable during a UDP association. Moreover, the extended UDP association model should support current UDP associations.

We classify the UDP binding types that SOCKS must support into UDP Open, UDP Listen, and Two-step UDP Open, as Figure 4 shows.

All three UDP bindings are initiated by a local client. UDP Open creates a UDP association between a local socket and a known remote socket; for example, a local DNS client queries a remote DNS server. UDP Listen corresponds to the connectionless server mode in which a local socket is

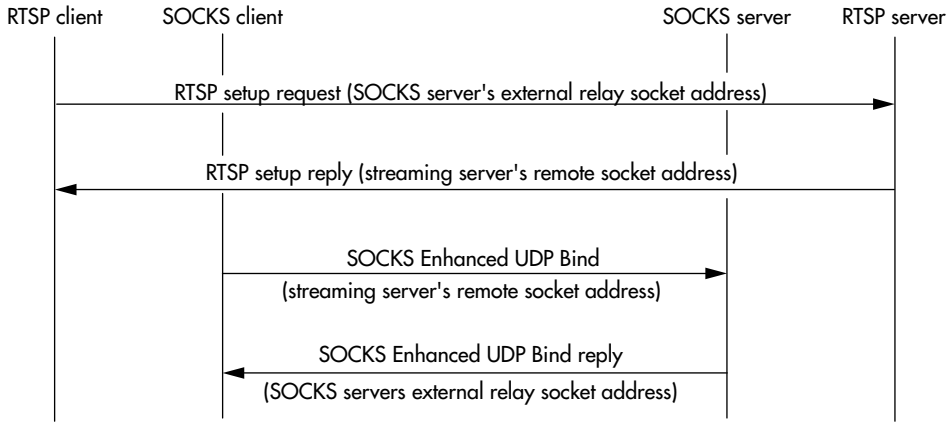

Figure 3. A protocol call sequence mismatch problem: The RTSP client cannot receive the streaming server's remote socket address without first receiving the SOCKS server's external relay socket address, and the client cannot receive the SOCKS server's external relay socket address without first receiving the streaming server's remote socket address.

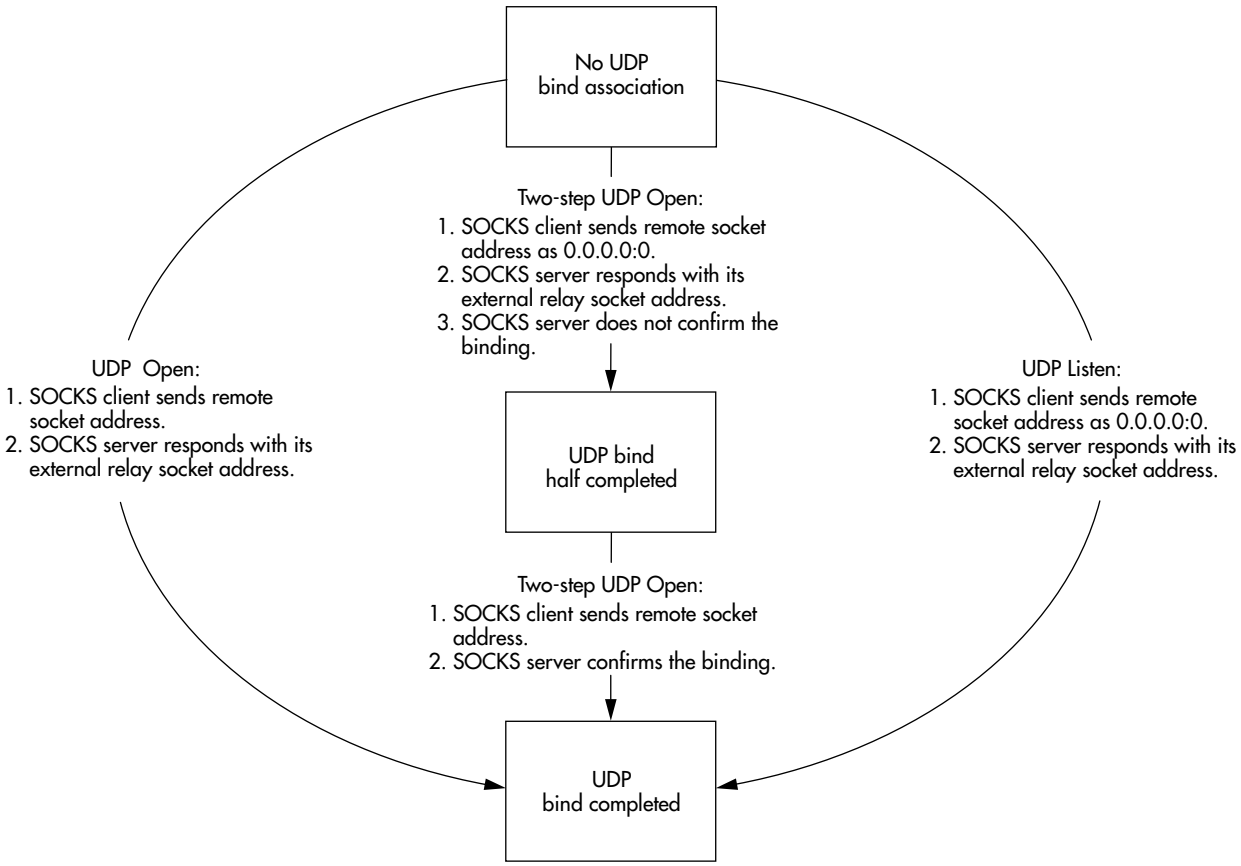

Figure 4. A state diagram for three types of UDP binding that the enhanced SOCKS provides—UDP Open, UDP Listen, and Two-step UDP Open. UDP Open supports, for example, a local DNS client querying a remote DNS server. UDP Listen supports, for example, a local DNS server waiting for requests from remote DNS clients. Two-step UDP Open supports, for example, a local RTSP client receiving multimedia streams from a remote streaming server.

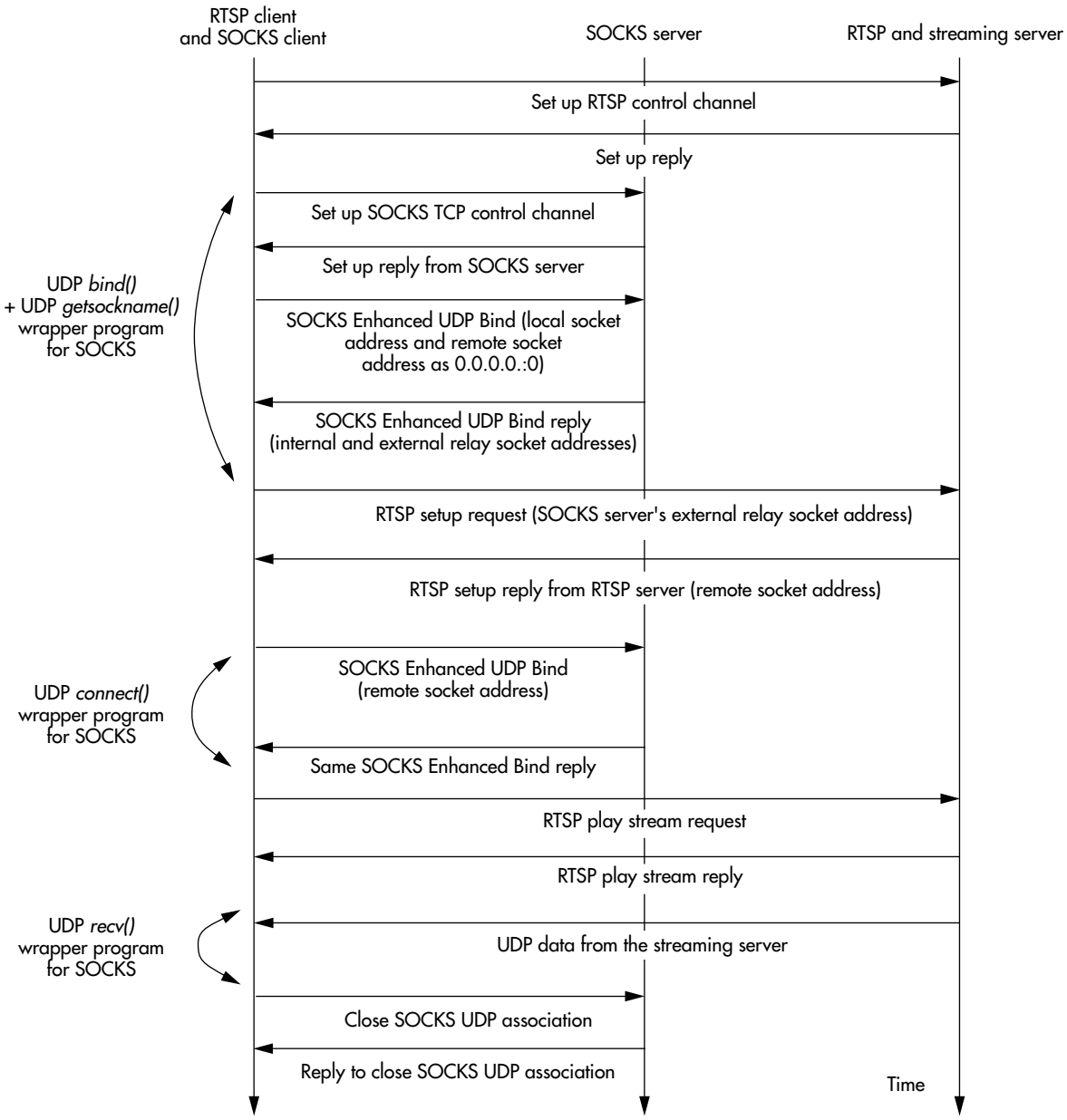

Figure 5. A call sequence for an RTSP session via the enhanced SOCKS. This figure depicts three types of protocol interaction: (1) Between RTSP client and RTSP server (via RTSP), (2) between RTSP client and proxy client (via system socket calls), and (3) between proxy client and proxy server (via enhanced SOCKS protocol). The diagram does not explicitly show interaction between proxy server and RTSP server.

open to any remote sockets and so does not have an explicit one-to-one UDP association: for example, a local DNS server waits for requests from remote DNS clients. SOCKS5, as detailed in RFC 1928, supports only UDP Open.

Additionally, we introduce a new, Two-step UDP Open to solve the mismatch problem stated earlier. The final result of the Two-step UDP Open is the same as that of UDP Open, but the two steps under UDP Open are performed separately in Two-step UDP Open. As a result, the first step in Two-step UDP Open is the same as UDP Listen, because the proxy client cannot learn the actual remote socket address in this step. Although the proxy server returns its external relay socket address at the completion of this step, the proxy server does not con-

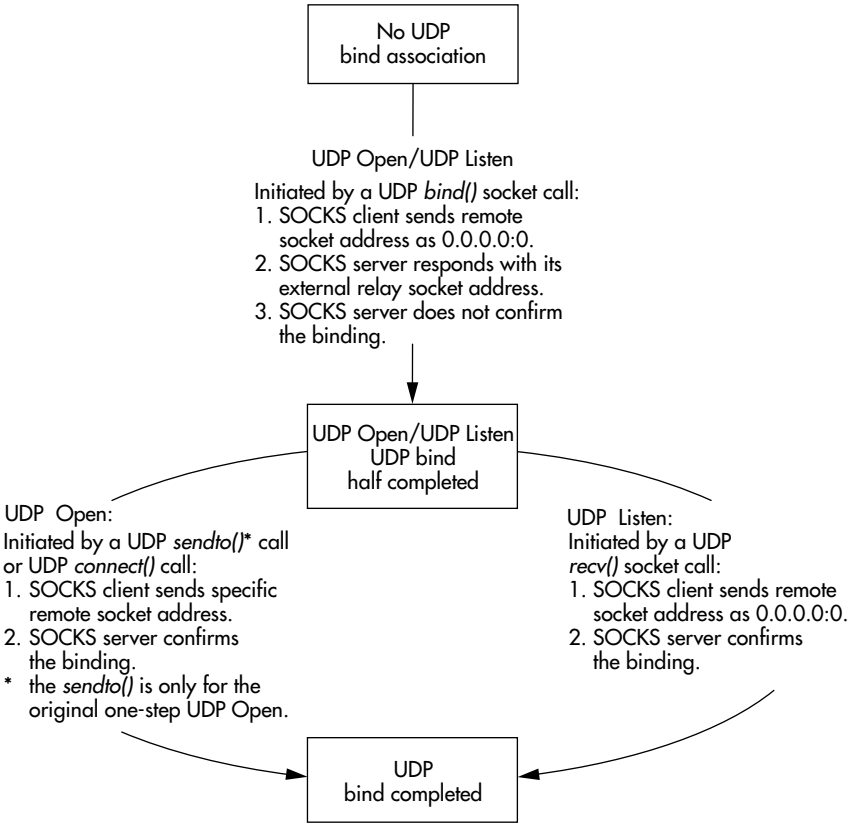

Figure 6. A state diagram for the Extended SOCKS UDP Binding. Unlike the state diagram in Figure 4, this Extended SOCKS UDP Binding goes through a two-step binding for all three types of UDP binding. The advantage of this approach is to provide a consistent framework for different types of UDP binding while leaving SOCKS command set unchanged.

firm the binding until it obtains the intended remote socket address from the proxy client in the second step. With the proxy server's external relay socket address, the RTSP client can obtain the streaming server's remote socket address through RTSP's control channel. The proxy client subsequently sends the remote socket address to the proxy server; thus, the UDP binding is confirmed in the second step and the mismatch problem is solved.

Next, the interface between an RTSP client and a proxy client must be modified to support the Two-step UDP Open. The modified socket interface procedure consists of two socket calls: UDP bind() and UDP connect(), which correspond to the two steps in the Two-step UDP Open, respectively. The UDP bind() socket call triggers the first SOCKS protocol exchange sequence, in which the proxy server replies with its external relay socket address. The UDP connect() call, on the other hand, triggers the second SOCKS protocol exchange sequence to confirm the UDP binding.

Figure 5 shows the combined call sequences for RTSP and SOCKS, plus the corresponding socket calls. To simplify the figure, we show the RTSP's control channel between client and server as a logical TCP connection, similarly for the play request and reply and for the UDP data transfer.

Enhanced SOCKS for Complete UDP Support

The Two-step UDP Open complicates the UDP binding process. One problem is the need to differentiate the SOCKS Enhanced UDP Bind command for the first step of the Two-step UDP Open from that for UDP Listen because the respective SOCKS messages are identical. One solution is to create a different SOCKS Enhanced UDP Bind command for UDP Listen, but this would complicate the SOCKS command set. Our alternative is to split UDP Open and UDP Listen into two steps. This results in an extended SOCKS UDP binding model that provides a consistent framework for different types of UDP binding.

Figure 6 is a state diagram for the extended SOCKS UDP binding. In the first step, a UDP bind() socket call triggers the SOCKS Enhanced UDP Bind command with 0.0.0.0:0 as the remote socket address. The second step for UDP Open, appropriately labeled UDP Open, combines the UDP Open and Two-step UDP Open in Figure 4. The second step for the UDP Listen, on the other hand, goes through a similar state transition as in Figure 4.

A proxy server can distinguish the two steps in the extended SOCKS UDP binding, based on a Transaction ID (TID) used in the enhanced UDP mode and the remote socket address. The TID is assumed to be the same for the first and second steps of a given UDP binding. When a proxy server receives a SOCKS Enhanced UDP Bind command, it first checks if it received the TID in the previous command. A new TID implies the first step; a used

TID, the second. Moreover, the original one-step binding can be distinguished from two-step binding by means of a specific, remote socket address in the bind command. As a result, the original one-step UDP Open can be supported concurrently with the extended UDP binding. The extended binding process is summarized below:

- A bind command with a remote socket address equal to 0.0.0.0:0 and an unused TID implies the first step of the new extended binding process.
- A bind command with a remote socket address equal to 0.0.0.0:0 and a used TID implies the second step of the new extended binding process. This binding corresponds to UDP Listen.
- A bind command with a remote socket address not equal to 0.0.0.0:0 and a used TID implies the second step of the new extended binding process. This binding corresponds to UDP Open.
- A bind command with a remote socket address not equal to 0.0.0.0:0 and an unused TID implies the original one-step UDP Open.

Another important, related issue is the interface between an RTSP client with a proxy client. For UDP Listen, a client may invoke a UDP recv() socket call; for UDP Open, a UDP sendto() call or a UDP connect() call, because both have arguments for specifying a remote socket address. As for the Two-step UDP Open, we propose that the client invoke a UDP connect() socket call because a connect() socket call simply associates a UDP port with a fixed, remote socket address. This association matches very well with the concept of a multimedia stream, which is specified by a fixed pair of UDP socket addresses. Moreover, the connect() socket call will not affect RTSP's operation because each multimedia stream uses a distinct UDP port.

RTSP Requirements

As we've said, an important criterion in the enhanced SOCKS design is leaving the application protocol unchanged. The enhanced SOCKS requires no changes in RTSP other than enforcing certain requirements that should have been stated explicitly in the specification and an additional UDP connect() socket call after the UDP bind() socket call. These new requirements do not affect RTSP's transport establishment procedure for direct connectivity without intermediate proxies.

The first requirement is to stipulate that both the RTSP server and client use the fields of source socket address and destination socket address in the RTSP setup request and reply messages. The use of these fields is not currently mandatory. For example, the RTSP specification states that the streaming server's source address can be specified if different from the RTSP server's. The specification describes one example where the source socket address was not specified in the transport header.

In another specification example, both the source and destination address were ignored in the transport header because they were assumed identical to those for RTSP's control channel. The missing information in the transport header for both cases can be derived from other available information in the absence of firewalls. However, the derived addresses might become invalid if the RTSP session goes through SOCKS. To conclude, an RTSP client must supply a destination socket address in the setup request message to receive the requested stream. Similarly, an RTSP server must supply a source socket address in the setup reply message to identify explicitly the source of the requested stream.

The second requirement concerns the corresponding UDP socket calls for an RTSP client to properly interface with a proxy client. The current RTSP specification does not specify any socket call requirements, and the current RTSP implementation invokes only a bind() socket call to set up a multimedia stream. In the extended SOCKS UDP binding, a connect() socket call must be invoked once the remote socket address is obtained. In fact, it is good practice to add the additional connect() socket call after the bind() socket call for RTSP-based applications. The reason is that the UDP port for a multimedia stream is always bound to a single UDP port in a streaming server. Therefore, it is a one-to-one UDP binding, instead of a one-to-many UDP binding.

These two requirements must be tied tightly with the RTSP request and RTSP reply sequence, so that information exchange is correctly sequenced between an RTSP client and a proxy client. The UDP bind(), UDP connect(), and UDP recv() socket calls must be executed in the right sequence between the RTSP request and RTSP reply, as indicated in Figure 5. The UDP bind() and the UDP getsockname() socket calls must be executed before the RTSP setup request. The UDP connect() socket call must be executed after receiving the RTSP setup reply but before sending the RTSP play request. The UDP recv() socket call must be executed after receiving the RTSP play reply.

PROTOTYPE

We implemented a prototype for the enhanced SOCKS, including the necessary changes in the

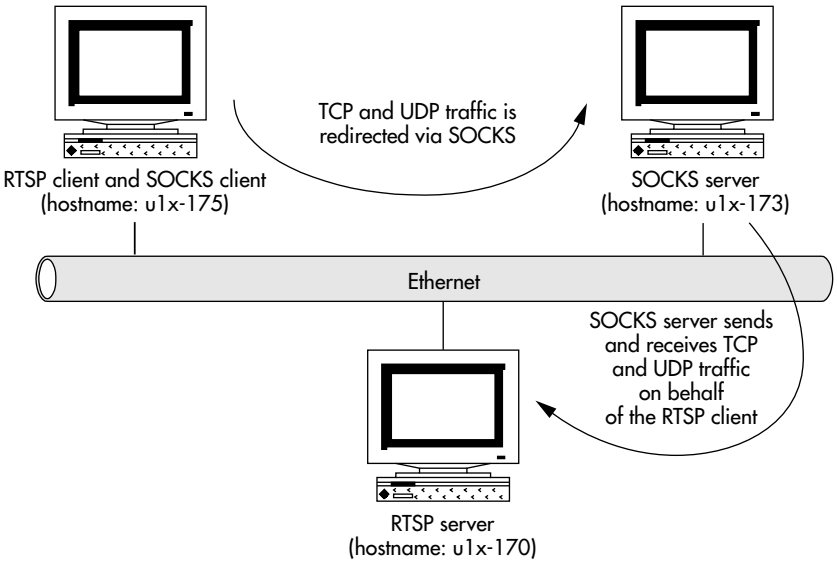

Figure 7. Testbed setup for the enhanced SOCKS.

RTSP client and server. We based our implementation on the NEC-released source code for SOCKS5 with additional enhancements for UDP.[12] We modified the source code to implement the enhanced UDP mode proposed by Chouinard and the enhanced UDP binding model that we propose.[6] We modified or added about 800 lines of code for the enhanced SOCKS. We based the source code for RTSP, however, on the RTSP reference implementation provided by RealNetworks, with fewer than 200 lines of code modified or added.[13] Altogether, we analyzed roughly 8,000 lines of code.

We tested the prototype in a LAN environment, as Figure 7 shows. The RTSP client, RTSP server, SOCKS client, and SOCKS server were all running Sun Microsystems' Solaris 2.6. The RTSP client and SOCKS client ran on the same Sun Solaris machine (hostname u1x-175 in Figure 7) while the SOCKS server ran on another Solaris machine (hostname u1x-173), acting as a firewall. All the TCP and UDP traffic between the RTSP client and server went through the SOCKS server. The RTSP server ran on another Solaris machine (hostname u1x-170).

We based the test on the event sequence depicted in Figure 5. The RTSP client initiated the RTSP TCP control connection to the RTSP server via the SOCKS server. A `get rtsp.wav` command issued from the command line interface to the client triggered RTSP setup and play requests. In the RTSP setup request, the RTSP client set up the UDP transport for the stream file rtsp.wav, and the two-step SOCKS UDP binding was performed. In the RTSP play request, the RTSP client initiated a play request and the RTSP server delivered the audio stream rtsp.wav to the RTSP client. The RTSP client application then closed. Detailed traces of the tests are described elsewhere.[14]

The two-step SOCKS UDP binding was successful, judging by the screen captures we obtained from the RTSP server, SOCKS server, and RTSP client. Each step mapped correctly to the corresponding wrapper socket call program. The UDP port binding at the SOCKS server was communicated from the SOCKS layer to the RTSP client. The source address and source port for the UDP stream were communicated from the RTSP client to the SOCKS layer.

On the RTSP side, the external address binding at the SOCKS server was communicated from the RTSP client to the RTSP server. The RTSP server sent the UDP stream's source address and source port to the RTSP client. Although there is an additional step in the extended UDP binding process, performance did not noticeably degrade. Even in a more involved network environment, we do not expect significant performance degradation as a result of the additional binding step.

MULTICAST-ENABLED SOCKS

IP multicasting is an efficient way to send a multimedia stream to a number of receivers. The Internet draft that put forward the enhanced UDP mode also proposed two approaches to extend SOCKS for supporting IP multicasting— multicast-to-unicast mode and multicast-to-multicast mode.[6] The former mode provides a more secure control by devolving multicast packets into unicast packets delivered to each receiver in the group. The latter mode, on the other hand, provides less control but is more scalable and efficient. The extended UDP binding model we propose does not affect the two multicast support modes.

CONCLUSIONS

The firewall-segmented Internet today inevitably poses challenges to designing correct transport-level proxies and application protocols. We have learned

important lessons regarding these designs in this work, and we believe that our experience can be extended to form sound protocol design principles.

First, application protocols are generally designed without firewalls, or with respective application proxies, in mind. With a transport-level proxy, like SOCKS, this assumption no longer holds. As a result, some missing application protocol information, which is not mandatory in the specification, may cause the firewall to reject incoming traffic. We have illustrated this point using RTSP in this article.

Application protocol designers and implementers, therefore, should take into consideration the transport-level security requirement. Specifically, the client and server should exchange the source and destination socket address at the application level and provide them to the transport layer. One way to achieve this is to standardize the function call library such that the source and destination socket address must be provided before data transfer is allowed. A prime example of this approach is to create an RTP function call library that enforces such a standard.

Second, application protocols are becoming very complex, and they may involve multiple TCP connections and UDP associations for a single application session. Moreover, TCP and UDP ports may be assigned dynamically through the application protocols. RTSP-based multimedia streaming considered in this article, the emerging voice-over-IP standard, and multimedia conferencing are notable examples. A transport-level proxy must handle such complex interactions. We expect that our half-binding approach can resolve similar mismatch problems with other application protocols. ■

REFERENCES

1. D.B. Chapman and E.D. Zwicky, *Building Internet Firewalls*, O'Reilly & Assoc., Sebastapol, Calif., 1995.
2. RealNetworks, "Configuring RealPlayer Version G2 to Work from behind a Firewall," http://service.real.com/firewall/configRP6.html#udp, 1999.
3. IETF's Authenticated Firewall Traversal Working Group charter, http://www.ietf.org/html.charters/aft-charter.html, Aug. 1998.
4. D. Koblas and M. Koblas, "SOCKS," *Proc. UNIX Security III Symp.*, Usenix Assoc., Berkeley, Calif., 1992, pp. 77-83.
5. M. Leech et al., "SOCKS Protocol Version 5," RFC 1928, http://www.ietf.org/rfc/rfc1928.txt, Mar. 1996.
6. D. Chouinard, "SOCKS V5 UDP and Multicast Extensions to Facilitate Multicast Firewall Traversal," Internet Draft, http://www.socks.nec.com/draft/draft-ietf-aft-mcast-fw-traversal-01.txt, Nov. 1997.
7. M. VanHeyningen, "SOCKS Protocol Version 5," Internet Draft, http://search.ietf.org/internet-drafts/draft-ietf-aft-socks-pro-v5-05.txt, June 2000.
8. Stardust Forums, "SOCKS—The Border Service Enabler: A Technology Backgrounder," http://www.stardust.com/events/socks98/socks6.pdf, Sept. 1998.
9. RealNetworks, "Using RTSP with Firewalls, Proxies, and Other Intermediary Network Devices," version 2.0/rev. 2, http://docs.real.com/docs/proxykit/rtspd.pdf, 1998.
10. IETF's Multiparty Multimedia Session Control Working Group charter, http://www.ietf.org/html.charters/mmusic-charter.html, Mar. 1999.
11. H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," RFC 2326, http://www.ietf.org/rfc/rfc2326.txt, Apr. 1998.
12. Source code for SOCKS5, http://www.socks.nec.com/cgi-bin/download.pl, 1998.
13. Source code for the RTSP Reference Implementation, http://www.realnetworks.com/devzone/library/rtsp/index.html, 1998.
14. K.P. Fung, "SOCKS5-based Firewall Support for UDP-based Applications," master's thesis, The Hong Kong Polytechnic Univ., Dept. of Computing, Hong Kong, PRC; http://www2.comp.polyu.edu.hk/~csrchang/MSc/Billy.pdf, June 1999.

King P. Fung is an associate responsible for network project implementation in the information technology group of Morgan Stanley Dean Witter. He has many years' industrial experience in the design and support of enterprise internetworking infrastructure and firewall infrastructure for various global financial institutions. Fung earned a BSc, with honors, in electronic engineering from the Chinese University of Hong Kong in 1990, and an MSc in information technology from the Hong Kong Polytechnic University in 1999.

Rocky K.C. Chang is an assistant professor in the Department of Computing at the Hong Kong Polytechnic University. His research interests include IP multicast routing protocols, high-performance TCP design, security protocols and issues, and performance modeling. Chang received a PhD in computer system engineering from Rensselaer Polytechnic Institute in 1990. He was with IBM Thomas J. Watson Research Center from 1991 to 1993.

Readers can contact the authors at Dept. of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong; voice (852)-2766-7258, fax (852)-2774-0842; csrchang@comp.polyu.edu.hk.