RC 21139 (03/17/98)
Computer Science/Mathematics

IBM Research Report

TCP Splicing for Application Layer Proxy Performance

David Maltz

IBM Research Division &
Dept. of Computer Science
Carnegie Mellon University
dmaltz@cs.cmu.edu

Pravin Bhagwat

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York
pravin@watson.ibm.com

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

TCP Splicing for Application Layer Proxy Performance

David A. Maltz
IBM Research Division &
Dept. of Computer Science
Carnegie Mellon University
dmaltz@cs.cmu.edu

Pravin Bhagwat
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York
pravin@watson.ibm.com

Abstract

Application layer proxies already play an important role in today's networks, serving as firewalls and HTTP caches — and their role is being expanded to include encryption, compression, and mobility support services. Current application layer proxies suffer major performance penalties as they spend most of their time moving data back and forth between connections; context switching and crossing protection boundaries for each chunk of data they handle. We present a technique called TCP Splice that provides kernel support for data relaying operations which runs at near router speeds. In our lab testing, we find SOCKS firewalls using TCP Splice can sustain a data throughput twice that of normal firewalls, with an average packet forwarding latency 30 times less.

KEYWORDS: TCP, Firewalls, SOCKS, Application Layer Proxies, Performance

1. INTRODUCTION

Many designs for internet services use *split-connection proxies*, in which proxy machine is interposed between the server and the client machines in order to mediate the communication between them. Split-connection proxies have been used for everything from HTTP caches to security firewalls to encryption servers.[14] Split-connection proxy designs are attractive because they are backwards compatible with existing servers, allow administration of the service at a single point (the proxy), and typically are easy to integrate with existing applications.

While attractive in design, modern split-connection proxies typically suffer from three related problems: they have poor performance; they add a significant latency to the client-server communication path; and they potentially violate the end-to-end semantics of the transport protocol in use. In this paper, we explain the details of a new, general technique called TCP Splice that improves the performance of split-connection proxies, and we show the performance improvements we created by adding TCP Splice to a SOCKS[7] application layer firewall.

Figure 1 shows the architecture of split-connection proxy system. When processes running on the client attempt to connect to a server machine, a client library intercepts the connection attempt, redirecting it to first make a connection to the proxy machine. The proxy machine then makes a second connection to the server, splitting the logical connection between server and client into two pieces. The split nature forces all traffic between client and server to flow through the proxy, which allows the proxy to manipulate the data and provide its service.

With the exception of proxy services running on custom hardware, all proxy services known to the authors are implemented as user level processes that we term *application layer proxies.* Application layer proxies read data intended for the client from the proxy-server connection, modify it if needed, and write it into the proxy-client connection. Likewise, data intended for the server is read from the proxy-client connection, modified if needed, and written into the proxy-server connection.

To support application layer proxies, we add a simple, generic kernel service called TCP Splice which has an easy-to-use application programming interface. The TCP Splice takes care of all data forwarding operations directly in the kernel, leaving the set-up, tear-down and logging tasks specific to each type of proxy in the user level application where they are easy to modify or amend as needed. The TCP Splice technique improves the current state of the art in three ways:

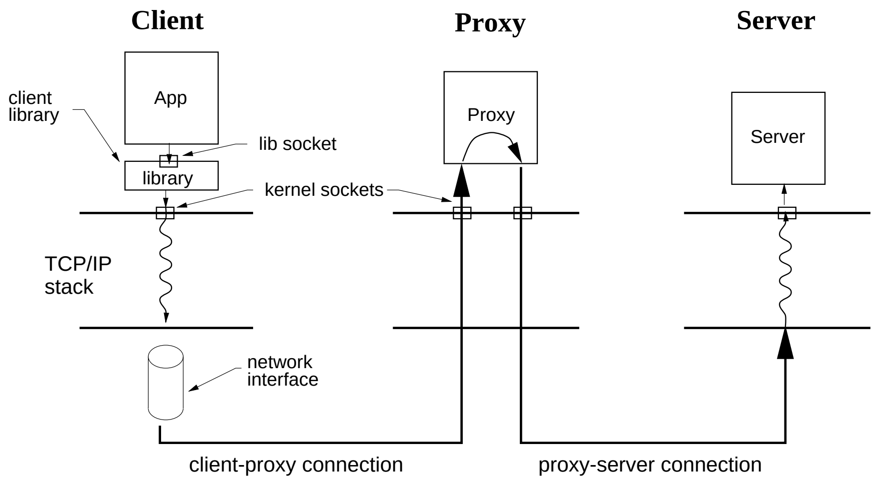

Figure 1 Basic architecture of split connection application layer proxies. The client library redirects connection attempts to the proxy, which relays data between client and server.

- Performance: A proxy or firewall using TCP Splice acts like a layer 3.5 router; it does not incur either transport or application layer protocol processing overhead for each packet it processes. The reduced complexity and code path length dramatically improves throughput.
- Less book keeping: Proxies using TCP Splice need to maintain less TCP state information for each of the connections that pass through them, and the proxy does not have to buffer any packets.
- Better end-to-end semantics: TCP Splice enables two ends of the connection to communicate as peers, allowing control information to flow end-to-end. Aside from other advantages, this provides the connection with true TCP reliability semantics and correct urgent data handling.

In the sections that follow, we first provide an overview of SOCKS firewalls and application layer proxies in general. We then give some background material on the innards of TCP needed for the explanation of of TCP Splice, and present the technique along with its performance.

2. BACKGROUND

SOCKS [7] is a protocol for use by application layer proxies implementing a network firewall function, and its implementation [4] is typical of all application layer proxies. Figure 3 shows pseudocode for a typical SOCKS server. The implementation style requires at least two protection boundary crossings and at least 4 passes over all the packet data[1] per chunk of data relayed, which greatly limits the performance of the proxy.

The intuition behind TCP Splice is that we can change the headers of incoming packets as they are received and immediately forward them, rather than passing packets up through the protocol stack to user space, only to have them passed back down again. The effect is to have the proxy relay packets as if it were a layer 3.5 router (see 2). Authentication, logging, and other tasks are done by the proxy in user space as normal, but the data copying part of the proxy — where the performance is normally lost — is replaced by single `ioctl()` call to set up

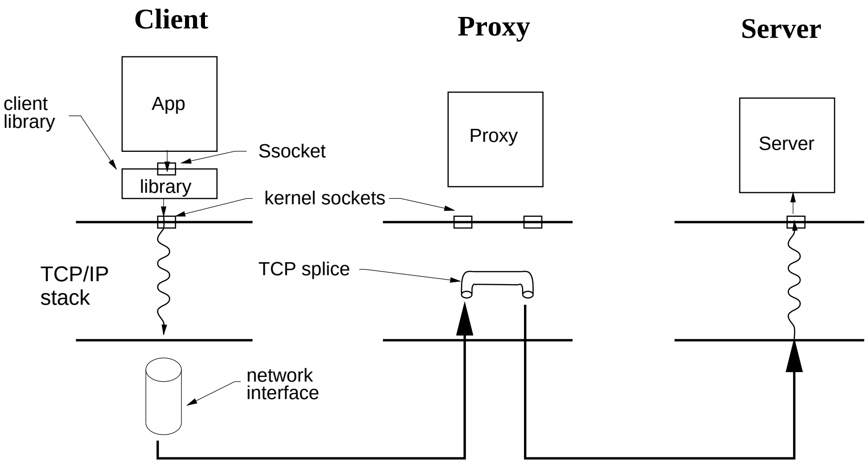

Figure 2 The architecture of a proxy using TCP Splice. Data relay between the client and server is performed in-kernel by the TCP Splice.

[1]two for checksumming and two to copy in/out of the kernel

the splice. After the splice is initiated, the user level proxy can go on to other tasks. Figure 4 shows the pseudo code for our splicing SOCKS server.

3. TCP SPLICE

The use of TCP Splice to support mobility is explained in [8], but that paper elides the details required for a correct, high speed implementation. Figure 5 shows how the splice operations fit in to the normal SOCKS message exchange. We focus on clients making connections to servers, but SOCKS and TCP Splice also support clients accepting connections, such as used with FTP in active mode.

When a client application calls `connect()`, the client library traps the call and converts it to a call to `Sconnect()`, which opens a connection to the proxy as shown. The proxy and client library engage in an authentication negotiation (eliminated from the diagram), and the client

```
c = accept() client connection;
  <authenticate client>
s = socket();
connect(s) to server;
send(c) OK message;
while (1) {
  read() from c, write() to s;
  read() from s, write() to c;
  if (c and s return EOF) {
    close(c); close(s);
    break;
}
<service next request>
```

Figure 3 Pseudocode for a SOCKS firewall, which is typical of most application layer proxies.

```
c = accept() client connection;
  <authenticate client>
s = socket();
ioctl(s, INTEND_SPLICE, c);
connect(s) to server;
n = sizeof(OK message);
/* splice c and s after n more
   bytes of data are sent */
ioctl(c, PRESPLICE, {s, n})
send(c) OK message;
close(s);
close(c);
<service next request>
```

Figure 4 Pseudocode for a SOCKS firewall using TCP Splice for the data relay. The forwarding loop is removed, allowing the proxy to immediately continue with other actions.

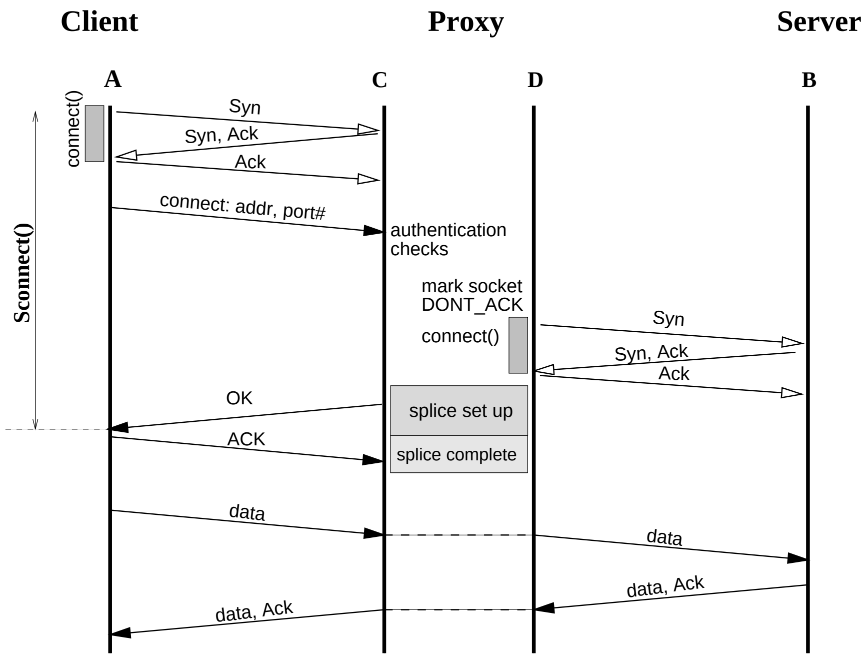

Figure 5 Packets exchanged during a SOCKS connection setup showing when splice operations take place.

then sends the proxy the address of the server to which it wants to connect. The proxy opens a second socket, marked **D** in the figure, and tells the transport layer it intends to splice the connection terminating at that socket with the connection terminating at socket c using the `INTEND_SPLICE ioctl()`. This call also sets a TCP Splice specific `DONT_ACK` flag bit on the socket, which marks the beginning of the server to client splice. Any data sent by the server after the flag is set are queued at the proxy for relay to the client, rather than being read and processed by the proxy application. The proxy then connects to the request server using the normal `connect()` call.

When the proxy application has completed any exchanges it wishes to have with the client, it calls `ioctl()` to initiate the splice specifying the two sockets to be spliced together (**C** and **D**) and the number of additional data bytes the proxy application wishes to write on the **A-C** connection before it is spliced and closed to the proxy application. For the SOCKS protocol, this is the length of the 'connect succeeded' message the proxy sends to client library; allowing the client to return from the `Sconnect()` call to the client application. In our general model, we refer this final message from proxy to client as the 'OK' message. The 'OK' message acts as a synchronization point for the client — data received after it are from the server, not the user level proxy process. Its length can be 0 if desired, in which case the splice is completed immediately. Once the splice is set up, the splice code relays any queued packets from the server and unsets the `DONT_ACK` flag. When an ACK of the OK message arrives at the proxy, the splice is complete.

3.1. TCP Background

Before describing how segments are relayed by TCP Splice, some background on TCP is required (see [13] for more detailed information). Each TCP segment is typically sent in a single IP packet containing an IP header and TCP header followed by the TCP data. The IP header contains the IP address of the packet's source and its destination. The TCP header contains which port on the destination machine the packet is intended for, and which source port it came from. These four pieces of information uniquely identify which TCP connection the packet is part of. The TCP header also contains a sequence number field which indicates where in the connection the data in this segment belong, and an acknowledgment field indicating how many bytes of data the segment's sender has received from its peer.

Figure 6 depicts a normal TCP connection with data in flight between endpoints. Each normal TCP connection is point-to-point and terminates at a *TCP socket* which is named by an address and a port number. A TCP connection is uniquely identified by the names of the two sockets at its endpoints. For each TCP socket, the normal TCP state machine maintains the following three counters. Using the counters, TCP assigns each byte of data sent over the connection a sequence number so TCP can detect and recover from data loss or duplication.

- `snd_nxt`: The sequence number of the next data byte to be sent.
- `snd_una`: The sequence number of the first unacknowledged data byte (equivalent to the sequence number of the greatest ACK received).
- `rcv_nxt`: The sequence number of the next byte of data the sockets expects to receive (equivalent to one more than the greatest consecutive sequence number received so far).

These counters define a *sequence space* associated with the socket. Without loss of generality, we assume for this explanation that each sequence space begins numbering at 0. Data bytes with sequence numbers greater than `snd_nxt` have not yet been sent. Data bytes with sequence numbers less than `rcv_nxt` have been received by the TCP stack, but perhaps not yet read by the application. We say that data is acknowledged when the sender of the data receives an acknowledgment for it: `snd_una` will be less than `rcv_nxt` whenever an ACK is in flight, delayed, or lost.

3.2. Mapping Sequence Spaces

When two connections are spliced together, the data sent to the proxy on one connection must be relayed to the other connection so that it appears to seamlessly follow the data that came

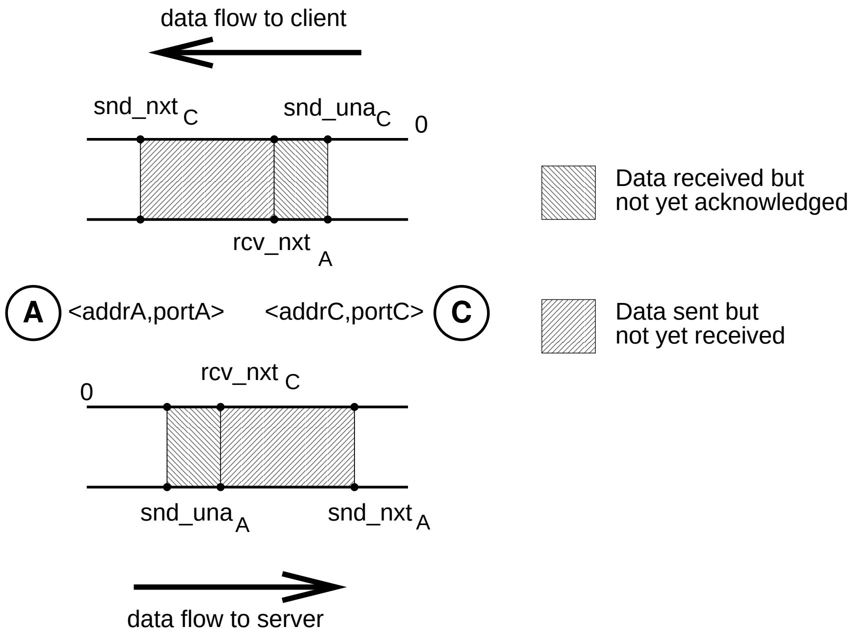

Figure 6 A normal TCP connection between sockets **A** and **C** with state counters labeled.

before it. The seamless nature of the data must be preserved even if there are data or ACK packets in flight at the time the splice is made, or if data must be retransmitted. Since all data bytes in TCP are assigned a sequence number in the sequence space of their connection as described above, we achieve a seamless splice by mapping the sequence numbers from one connection's sequence space to the other connection's sequence space.

We call the sequence number of the next new byte to be received from **A** the *splice initial receive sequence number* (`splice_irs`) and the sequence number **B** next expects the *splice initial send sequence number* (`splice_iss`). Together, the pair $<$`splice_irs`,`splice_iss`$>$ define a mapping between the sequence number spaces of the spliced connections from **A** to **B** where data with sequence number `splice_irs` $+N$ on the **A** to proxy connection maps to sequence number `splice_iss`$+N$ on the proxy to **B** connection. A second pair similarly defines the mapping of the splice connection from **B** to **A**.

Figure 7 shows how `splice_irs` and `splice_iss` are chosen for the logical connection from server to client. The proxy assumes any packets it receives after the `DONT_ACK` mode is set on socket **D** are intended to be relayed to the client. It chooses `splice_irs` to be the sequence number it next expected to hear at that time, so the next byte will be the first byte relayed across the splice. This byte of data, when it arrives, should be the first byte of data the client sees after the OK message, so `splice_iss` is set to the next sequence number after the OK message. Choosing `splice_iss` requires the splice code be told how many bytes long the OK message will be when the splice is initiated, since the OK message may not have been written when the splice is initiated.

Figure 8 shows how `splice_irs` and `splice_iss` are chosen for the logical connection from client to server. The `rcv_nxt` counter at socket **C** holds the sequence number of the byte that falls immediately after the end of the client-proxy authentication exchange, and is the sequence number of the first byte of data the client application will send to the server. This byte of data should be mapped to the next byte of data that the server expects, which is tracked by socket **D**'s `snd_nxt` counter at the time the splice is initiated.

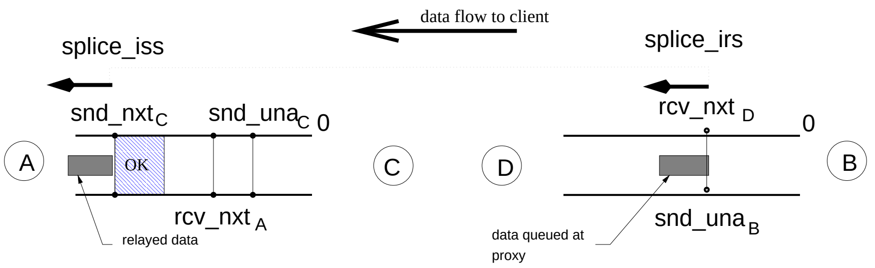

Figure 7 Choice of base points for mapping data flowing from server to proxy to client.

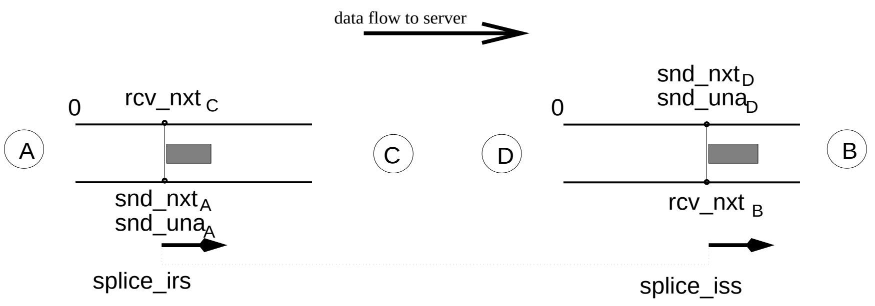

Figure 8 Choice for base points for mapping data flowing from client to proxy to server.

3.3. Relaying Segments

As each segment is received at a spliced socket, the segment's headers are altered to address the segment to the socket at the other end of the spliced connection. The segment's TCP headers are altered so the segment will be intelligible to the end system when it arrives — the segment will look like a continuation of the normal TCP connection the end-system first started with the proxy. To alter a segment for forwarding, the proxy needs only the state from the two sockets located on it (labeled **C** and **D** in the figures). In the discussion below, all the variables referred to are those kept by the proxy. Below, the socket a segment arrives on is labeled `in` and the socket it will be sent out from is labeled `out`.

Alter IP header

- Change source and destination address to that of outgoing connection.
- Remove IP options from incoming packet.
- Update IP header checksum.

Alter TCP header

- Change source and destination port numbers to match outgoing connection.
- Map sequence number of packet from incoming sequence space to outgoing space.
  `seq_num = (seq_num - in->splice_irs) + out->splice_iss`
- Map ACK number of packet from incoming sequence space to outgoing space.
  `ack_num = (ack_num - in->splice_iss) + out->splice_irs`
- Map TCP options as needed.
- Update TCP header checksum.

It is possible to update the TCP and IP header checksums in fail-safe manner, thereby saving the time required to first check the checksum of a received packet and then recompute the checksum after the headers have been altered (see [1]). Our current implementation does not use fail-safe checksum updates, but verifies and computes the checksum as normal by making two passes over the entire packet, so TCP Splice can potentially perform even better than reported here. With checksum updates, TCP Splice would perform *no* passes over the packet other than the operation to copy the packet from the network interface card onto the machine and the operation to copy it back to the network interface card.

The TCP urgent pointer is represented as an offset from the segment's sequence number; hence, it is not changed during the mapping procedure. By directly mapping and relaying packets, TCP Splice preserves the complete semantics of TCP's urgent pointer, so urgent data can be used in both the inline and out-of-band modes provided by the Berkeley Sockets API. Normal application layer proxies require urgent data to be used in the inline mode only as they will drop urgent data delivered out-of-band if it arrives faster than it can be relayed.

While relaying a segment, one special check must be made to ensure the segment does not contain data with sequence numbers less than the splice base point `splice_irs`. If such a segment is received, the data up to `splice_irs` is simply chopped out of the segment and the remainder of the segment relayed. If a segment containing data before `splice_irs` was relayed intact, the data would overwrite the OK message that lies just before `splice_irs` and confuse the client SOCKS library. The dropped data is only a retransmission of data received and processed by the proxy before the splice was set up, anyway, so dropping it is harmless.

3.4. TCP Options

The TCP options that will be used on a connection are negotiated exactly once when the SYN packets that establish the connection are exchanged. In the case of split-connection using TCP Splice, this initial SYN exchange is with the proxy machine, not the server node. Since packets will flow almost directly from client node to server node once the splice is in place, the proxy must either negotiate compatible options on both connections, or have a way to convert between the options both nodes accept. This is difficult since the proxy does not know which server the client wishes to connect to when it accepts and negotiates options for the client-proxy connection. Not knowing the intended server, the proxy can not simply negotiate compatible options.

We use a combination of three solutions to solve the options negotiation problem. First, we can simply not splice together connections that have negotiated incompatible options and force the proxy to forward data through application space as before, though this forfeits all the improvements of TCP Splice. Second, we can have the proxy only support the minimally acceptable common set of options. Third, certain options can be mapped or stripped[2] from packets being spliced between connections, allowing the proxy to support the options at the cost of a small additional per-packet overhead.

The following sections describe how we handle options negotiation for the most common TCP options. In reality, most of the options listed below are rare in practice, with few of them appearing in actual use in traces we have examined, so our exercise to support them is largely academic. Supporting only the minimum common subset of options including Timestamps and Maximum Segment Size appears to be sufficient for deployment on corporate networks today.

3.4.1. Timestamps

The proxy can safely negotiate to use the Timestamp option [5] on any connections it creates, since if it is called on to splice together a connection that uses timestamps with one that does not, it can simply strip the Timestamp option from all packets as they pass through the proxy. Even if both connections participating in a splice use Timestamps, special handling of the option is required to prevent an end-system from having echoed to it timestamps it did not generate, since such timestamps could seriously confuse the round trip time estimator. We handle the TCP Timestamp option in the same fashion as sequence numbers: we map the timestamps used on the proxy-mobile connection to those used on the proxy-server connection based on the last timestamps sent and received by the proxy before the splice was established. To map the two fields contained in each Timestamp option, we use the equations:

- `TSval_out = (TSval_in - in->lastrecv_TS) + out->lastsent_TS`
- `TSecho_out = (TSecho_in - in->lastsent_TS) + out->lastrecv_TS`

where `lastrecv_TS` and `lastsent_TS` represent the last TimeStamp that was received or sent on that socket before the TCP Splice was established.

In reality, it is only necessary to map the Timestamp options which contain echoes of proxy-generated timestamps, which would confuse an endsystem receiving them while expecting to receive timestamps based on its own clock. Since packets containing proxy-generated timestamps will be cleared from the system within one window's worth of packets after the splice is completed, the overhead of mapping could be saved if there was anyway to identify when the last of these packets had cleared the network.

3.4.2. Window Scale

Our proxy supports the Window Scale TCP option, but advertises a window scale of 0 to any client node which connects to it. When the proxy makes the proxy-server connection, it propagates the client node's proposed Window Scale option. If the server does support the option, it will be able to send data to the client using the client's scaled window, though the

[2] Options can be stripped from packets either by simply replacing the option's bytes with the TCP NOP (0x01), or by actually moving later data up and on top of the option. Which choice is best is a tradeoff between proxy and network loading.

client will believe the server only supports the default 64KB window. This is compatible with the typical client-server model in which most data flows from server to client. If the server does not support the Window Scale option, the proxy must unscale the client's advertised receive window when relaying packets to the server. If a Window Scale option is received in later segments sent by the client node, the proxy must strip these options, though the client node is prohibited by RFC1323 from sending additional segments with Window Scale options.

3.4.3. Selective Acknowledgements

A proxy using TCP Splice can support the Selective Acknowledgement (SACK) option [9] in a manner similar to Timestamps. If called upon to splice together a connection use SACK with one that is not, the splice simply strips all SACK options from the segments as it relays them. When relaying segments between two connections that both use SACK, the acknowledgment block edges contained in the SACK option are mapped in the same way as the segment's cumulative ACK field.

3.4.4. Maximum Segment Size

The Maximum Segment Size (MSS) option [13] is sent in all SYN packets (and only SYN packets) to tell the peer TCP machine what is largest TCP segment this TCP machine is capable of handling. Because forcing the proxy to resegment the data would be a costly operation, our proxy uses a two tiered approach to handle the MSS option.

If the connecting client node supports Path MTU Discovery [10] (as indicated by the presence of a Don't Fragment (DF) flag in the IP header), then the proxy advertises whatever MSS it feels appropriate in the SYN packet it sends to the client node. Should the server advertise a smaller MSS on the proxy-server connection than the proxy has advertised on the client-proxy connection, the proxy can generate ICMP Destination Unreachable messages with the "Fragmentation needed, DF flag set" code to the client node to force its segment size to be equal to the MSS accepted by the server.

If the client node does not support Path MTU Discovery, then the proxy simply advertises the minimum MSS of 536.

4. EVALUATION

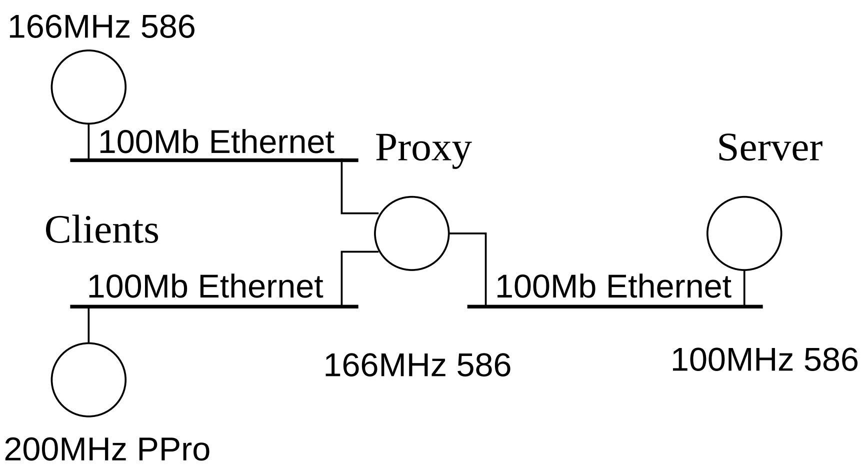

Figure 9 Test network topology. All machines running BSDI BSD/OS 3.0.

To evaluate the performance of TCP Splice, we benchmarked both an unmodified SOCKS server and a SOCKS server modified to use TCP Splice as shown in figure 4. In a separate experiment, we configured the proxy as a router and benchmarked its performance without the overhead of SOCKS. In the discussion below we call the unmodified system *SOCKS*, the system with our TCP Splice modifications *TCP Splice*, and the system in which the proxy acts like a router *IP Forward*. We used the publicly available distribution of SOCKS5 0.15.0 from NEC at `ftp.nec.com`. All the experiments were run on the 100Mbps test network shown in figure 9. All the machines ran BSDI BSD/OS 3.0.

All send and receive TCP buffers were set to 16KB since this is a typical value. Experimentation showed that the relative performance of the three system was unchanged by varying TCP buffer size, though, as expected, the absolute performance increased with increasing buffer size. The length of input and output interface queues on the proxy was increased to 150 to provide the network with extra buffering. The extra buffering is needed only to prevent TCP synchronization between the packet sources, which is commonly known to cause paradoxical decrease in TCP throughput. A production network would not require such buffering since the offered load would originate from significantly more than two machines, and RED [2] could be implemented at the proxy or other router.

For traffic generation and performance monitoring we used two applications: `netperf`, which is publicly available from [6], and `loadclient`, which we implemented ourselves. Netperf is a network benchmarking tool providing an interface for choosing socket buffer sizes, measuring throughput and response time between two end systems, and recording CPU utilization. Since the netperf client sends all data to the server over a single TCP connection, it is difficult to project how the proxy would scale when many connections are routed though it. To overcome this limitation, we implemented a traffic generator called `loadclient` that opens multiple connections through the proxy and then pushes data through them as rapidly as possible. In all the results reported here `loadclient` was used to generate workload and netperf was run on the proxy to measure CPU utilization.

To establish the baseline performance of the test network, we first configured the proxy to be a normal router (without any firewall function), and used `loadclient` to measure the maximum attainable throughput from the clients (the sources) to the server (the sink). To compare the performance of SOCKS versus TCP Splice, we used a SOCKSified `loadclient` to direct the traffic load through the firewall functions of the proxy — the same client was used to test both the SOCKS and TCP Splice firewalls. A comparison of throughput, response time, and CPU utilization numbers reveal the performance advantage of using TCP Splice over SOCKS. Since IP Forward numbers provide an upper bound, comparison with those numbers allows us to quantify the overhead of using TCP Splice and SOCKS.

In analyzing the performance of TCP Splice we will make three points.

1. The TCP Splice proxy can support substantially higher throughput than an Application Layer Proxy. Furthermore, the proxy using TCP Splice can sustain throughput comparable to the equivalent hardware acting as a router.
2. The TCP Splice proxy adds significantly less latency to the overall client-server connection than an Application Layer Proxy.
3. The TCP Splice proxy uses fewer proxy machine resources to do to the same work as an Application Layer Proxy.

All throughput measurements reported here are the average throughput of a minute long collection period. CPU utilization numbers are the average of three different samples collected over ten second periods while throughput tests were running. Except where noted, standard deviations are insignificantly small and are not reported.

4.1. Throughput Comparisons

In the first experiment, a SOCKSified `loadclient` opens a given number of connections through the proxy and then pushes data through them as rapidly as possible. The loadserver accepts connections from the client through the proxy, and then reads data from the connections. The number of bytes per second read by the loadserver, summed over all the connections, is what we report as the throughput of the proxy, since all those bytes of data had to flow through the proxy.

Figure 10 shows a comparison of the throughput of the SOCKS and TCP Splice proxies when running on the test network shown in figure 9. Both the IP Forward and TCP Splice are faster than SOCKS by a factor of more than two. Interestingly, TCP Splice performance is comparable to IP Forward. This is because at 70Mbps load, proxy is not CPU constrained (as shown in Figure 11); the real bottleneck is the 100 Mbps Ethernet which could not transport more data due to collision and MAC layer encapsulation overheard.

4.2. CPU Utilization Comparison

Figure 11 compares the CPU utilization required to achieve the throughput shown in figure 10. Clearly, the SOCKS proxy is CPU limited at 33 Mbps. The TCP Splice proxy, on the other hand, provides twice the throughput and remains only around 80 percent utilized. Under the same load condition, proxy's CPU utilization is below 70% in the IP Forwarding mode. Our current implementation of TCP Splice recomputes the TCP layer checksum after modifying the TCP/IP header, causing an extra pass over the data. With an optimized implementation (i.e., with incremental checksum computation) it is possible to bridge the 15% performance gap between IP Forward and TCP Splice. The extra CPU cycles could be used to carry more traffic, or for additional proxy services such as encryption or compression of the carried data. Figure 12 compares the CPU utilization required to support a throughput of approximately 33Mbps, which is the largest throughput the SOCKS proxy could support, clearly indicating the opportunity to

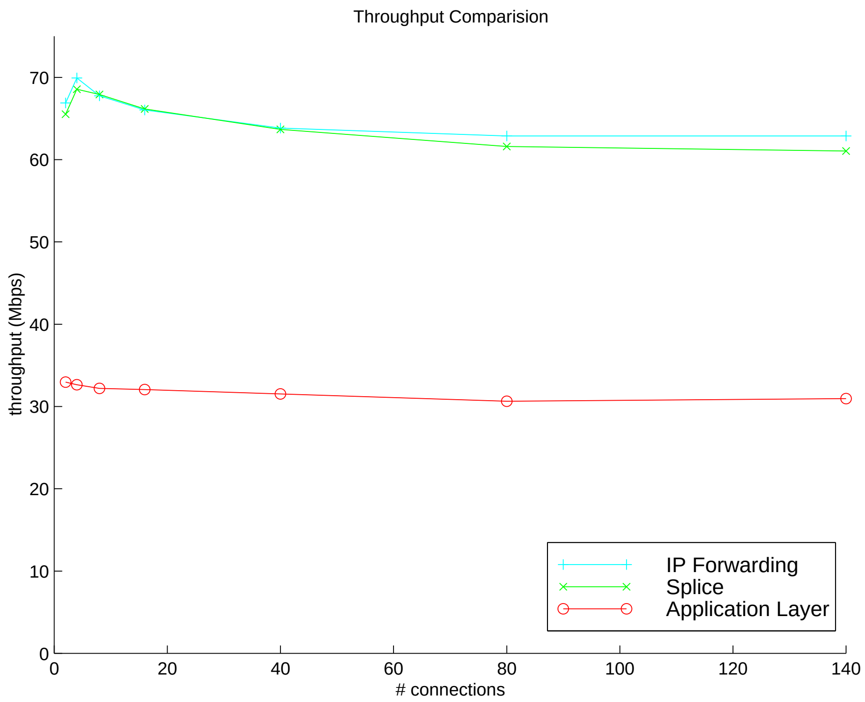

Figure 10 Throughput in Mbps of the SOCKS and TCP Splice proxies compared against IP Forwarding throughput.

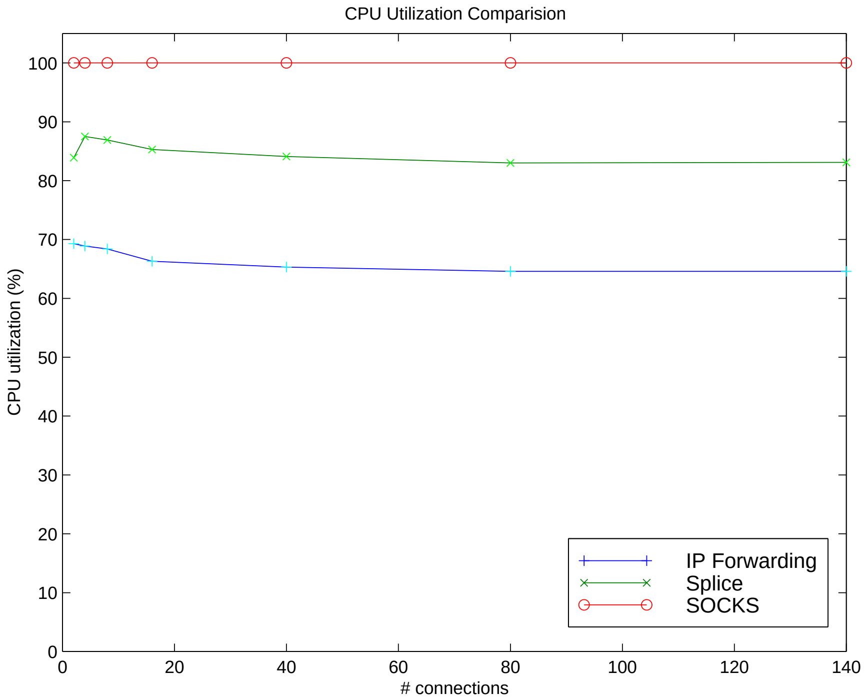

Figure 11 CPU utilization under varying number of connections.

either add additional functionality to a Splice proxy, or to reduce the processing power required of the proxy hardware allowing a cheaper system to be used.

4.3. Latency Comparison

Figure 13 shows a comparison of the forwarding latency of TCP Splice, SOCKS, and IP Forward. A single connection from client to server was set up, and packet arrival and departure times were recorded using `tcpdump`. The difference in time between when each byte was received on the client-proxy connection and sent on the proxy-server connection is plotted as a function of the byte's sequence number. Table 1 shows the mean, variance, and standard deviation of all the samples shown in Figure 13. The average forwarding latency for TCP Splice is 102 microseconds, which is marginally greater that IP Forwarding latency of 92 microseconds. The variance of both methods is comparable and small. Slightly high value of IP Forward variance reported in table 1 is a random occurrence and is not statistically significant.

Contrasted with TCP Splice, SOCKS has an average forwarding latency of 3.2 milliseconds, and the variance is quite large. The first packet relayed by SOCKS sees no queuing delay (since there are no queued packets yet), and so estimates the forwarding latency of the raw code path

Table 1 Forwarding latency comparison between TCP Splice, SOCKS, and IP Forwarding

	mean	variance	std. deviation
TCP Splice	0.1027	0.0019	0.0435
SOCKS	3.2273	0.5680	0.7536
IP Forward	0.0925	0.0040	0.0629

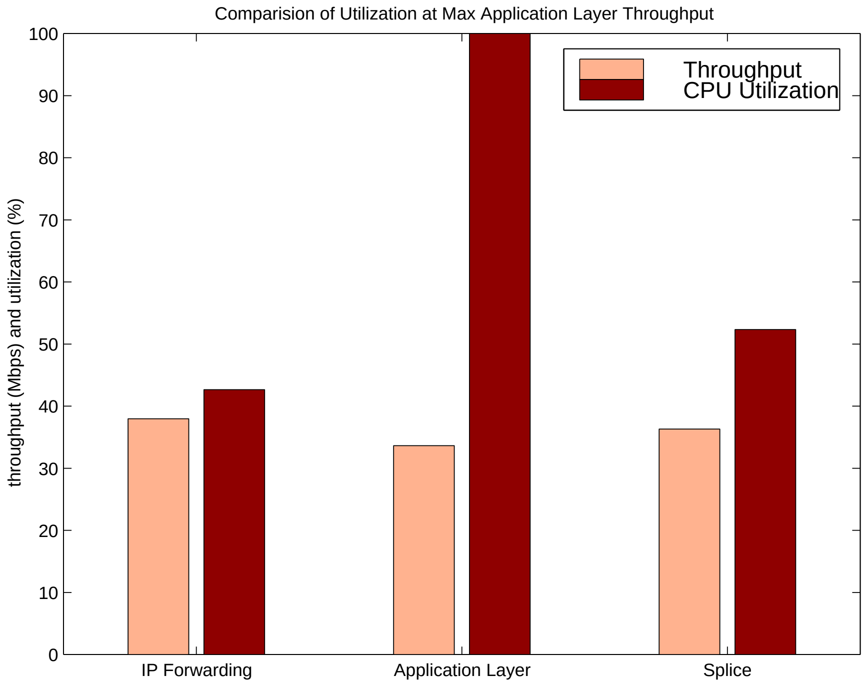

Figure 12 CPU utilization for approximately equal throughputs.

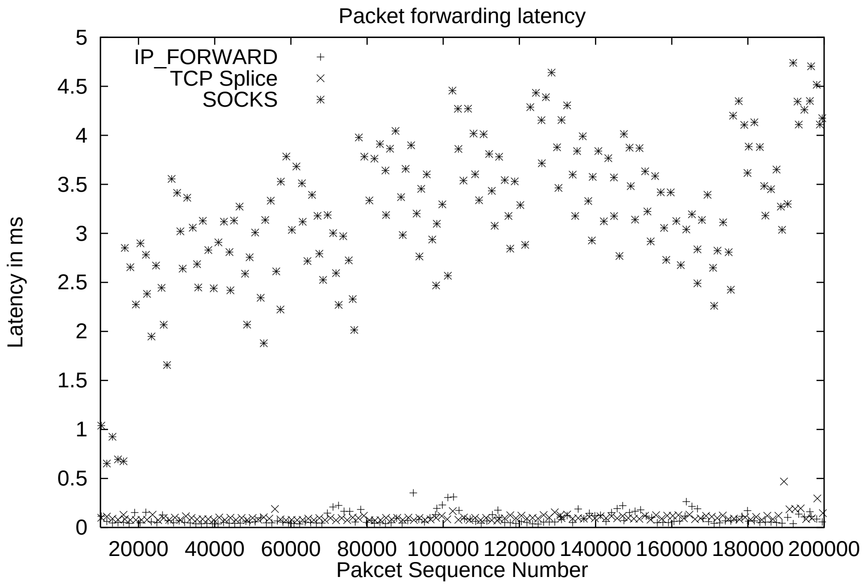

Figure 13 Distribution of Forwarding latency of TCP Splice, SOCKS, and IP Forwarding

length. SOCKS' path length is then about 740 microseconds, compared to TCP Splice's 102 microseconds.

As more data arrives at the SOCKS proxy, a queue of packets waiting to be relayed builds up in the socket buffers of the proxy, so the per byte forwarding latency grows. The latency drops again as the queue is drained, and the variation presumably indicates when the proxy process is scheduled.

5. RELATED WORK

A major contribution of TCP Splice is its ability to splice together two connections which were independently set up with the proxy and that have already carried arbitrary traffic with the proxy application. This feature is unique to TCP Splice, and it obviates the need for a separate control connection, thereby making TCP Splice more general and powerful than previously existing techniques. There are dominant applications, such as the Netscape web browser, and procotols, such as the SOCKS authenticated firewall traversal procotol, which assume the ability to send in-line control and authentication information. With TCP Splice, these applications and protocols can be supported easily in a backwards compatible fashion.

The addition of a constant offset to TCP sequence numbers was first proposed for TCP in order to support the FTP protocol in Network Address Translation (NAT) gateways[1]. While this is a primitive form of linear sequence number mapping, TCP Splice significantly extends the use of sequence number mapping and works correctly even if data is in flight at the time of the splice, or data is later retransmitted. TCP Splice also provides the necessary synchronization primitives to enable Application Layer Proxies to use sequence number mapping as a generic primitive that increases performance.

From the publicly available information, the Cisco PIX Firewall[3] uses sequence mapping techniques similar to those that underlie TCP Splice to implement what they call "cut-through proxy". However, TCP Splice is a significantly more general technique. The PIX firewall is a stateful firewall that traps and queues outgoing SYN packets until they are authenticated by an out-of-band authentication protocol, which requires a separate control connection to the client machine. If the authentication succeeds, the queued packets are released and their headers modified in a kind of advanced NAT. The PIX firewall is never a party to the communication with the client, and can not exchange information in-line with the client. TCP Splice, on the other hand, allows two arbitrary connections on which arbitrary information has been exchanged to be spliced together. This generality makes TCP Splice a primitive on which many proxies, not just firewalls, can be built.

For application layer proxies that modify data on the fly, the sequence number mapping tricks of TCP Splice can be used to push the modification operation into the kernel, and also the data to be modified while maintaining the end-to-end reliability semantics of TCP. We are currently looking at using TCP Splice to increase the throughput of encryption and compression services offered by our firewall.

6. CONCLUSION

A commonly used technique for building application layer firewalls involves inserting a TCP proxy in the communication path of the two communicating end points. In the majority of cases, the purpose of the firewall is only to authenticate the originators of connections, exercise some form of excess control, and monitor the network activity. After the authentication phase is over, the proxy effectively operates like a bi-directional application layer relay, copying data back and forth between the two connections. Under medium to high load conditions this becomes a major performance bottleneck.

We have shown that, using the technique of TCP Splice, we can overcome the performance bottleneck of application layer firewalls. Our lab experiments show that with TCP Splice we can double the throughput of a SOCKS firewall, while reducing the packet forwarding by a factor of 30. Besides performance, TCP Splice also provides better end-to-end semantics, since TCP segments and ACKs are exchanged end-to-end over a spliced connection.

With the TCP Splice technique, it is possible to build inexpensive and faster application layer proxies, including SOCKS firewalls. Furthermore, with an optimized implementation (e.g., with the incremental checksum) it is possible to push the performance envelope beyond current limits. We are in the process of developing an industrial strength SOCKS firewall solution based on TCP Splice.

7. ACKNOWLEDGEMENTS

We would like to thank Srini Seshan for his comments and critiques while designing TCP Splice, and Mark Davis and Karen von Kaenel for their useful pointers to experimental data.

References

[1] K. Egevang and P. Francis. The IP network address translator (NAT). Internet Request For Comments RFC 1631, May 1994.

[2] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[3] Cisco Systems Inc. Cisco's PIX firewall series and stateful firewall security. Web White Paper, May 1997. Available as http://www.cisco.com/warp/public/751/pix/nat_wp.htm.

[4] NEC U.S.A. Inc. Introduction to SOCKS. Web White Paper, 1996. Available as http://www.socks.nec.com/introduction.html.

[5] Van Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. Internet Request For Comments RFC 1323, May 1992.

[6] Rick Jones. The public netperf homepage. Web White Paper. Available as http://www.cup.hp.com/netperf/NetperfPage.html.

[7] M. Leech, David Koblas, et al. SOCKS protocol version 5. Internet Request For Comments RFC 1928, April 1996.

[8] David A. Maltz and Pravin Bhagwat. MSOCKS: an architecture for transport layer mobility. In *infocom-98*. Computer Science Press, 1998. To appear.

[9] Mathew Mathis, Jamshid Mahdavi, Sally Floyd, and A. Romanov. TCP selective acknowledgement options. Internet Request For Comments RFC 2018, October 1996.

[10] J.C. Mogul and S.E. Deering. Path mtu discovery. Internet Request For Comments RFC 1191, November 1990.

[11] Vern Paxon. Automated packet trace analysis of TCP implementations.

[12] Jon Postel. Transmission control protocol. Internet Request For Comments RFC 793, September 1981.

[13] W. Richard Stevens. *TCP/IP Illustrated, The Protocols*, volume 1. Addison-Welsley, 1994.

[14] Bruce Zenel and Dan Duchamp. General purpose proxies: Solved and unsolved problems. In *Proceedings of Hot-OS VI*, May 1997. read as http://www.mcl.cs.columbia.edu/ baz/ps/hot-os-vi.ps.

A. TCP Splice States

The states transitioned by sockets in a TCP Splice are shown in figure 14. The initial splice setup stages are explained in section 3. After the splice `ioctl` is made, the two sockets leave the normal TCP *ESTABLISHED* state and are linked together from then on.

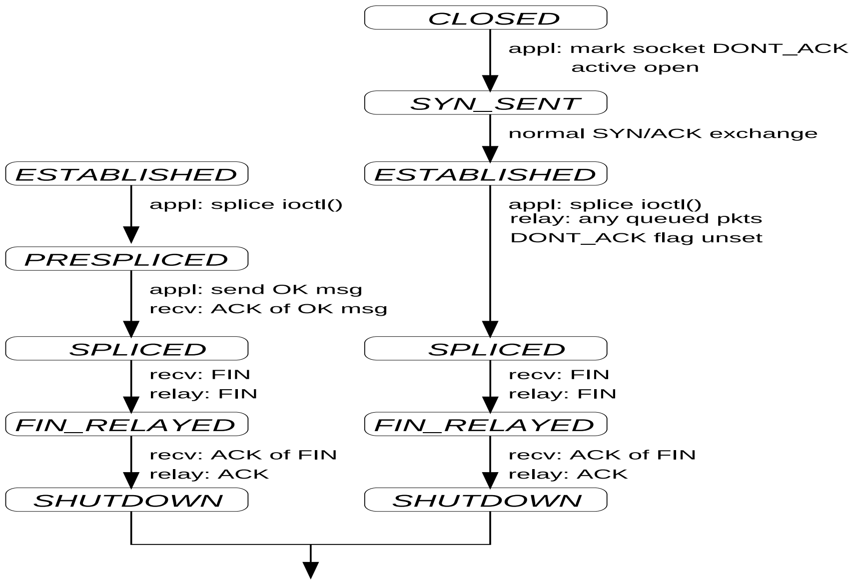

Figure 14 State diagram for spliced TCP sockets. The states transitioned by the client-side socket are on the left, and the states transitioned by the server-side socket are on the right.

Refering back to figure 5, the client-side socket, **C** enters a *PRESPLICED* state where it remains until the OK message has been acknowledged by the client. Data received from the client while the socket is *PRESPLICED* are queued until the splice completes, when they are relayed. ACKs sent by the client with sequence numbers less the splice base point `splice_iss` are interpreted as requests for retransmission of data written into the connection by the proxy and are processed by the proxy's normal TCP state machine. When an ACK of the OK message is received, the proxy can be certain all the data the proxy wrote to the client has been received and no more retransmissions of that data will be needed. The splice code then frees any socket buffers associated with socket **C**, shuts off the normal TCP state machine, and enters the *SPLICED* state.

Since the server-side socket, **D**, does not have to wait for an ACK before entering the *SPLICED* state, it transitions to *SPLICED* immediately upon the splice `ioctl()`. The splice code relays any of the packets that had been queued while the socket had its `DONT_ACK` flag set, shuts down the sockets normal TCP state machine and frees any socket buffers associated with it.

Since sockets entering the *SPLICED* state are completely cut off from the upper layers of the protocol stack, the splice must be able to tear itself down when the client and server close their connection. As TCP segments are passed through the splice, they are examined for indications that the end systems are closing their connections. When a segment with the FIN bit set is received by a spliced socket, the socket transitions to the *FIN_RELAYED* state and relays the FIN. When an ACK for this FIN is relayed through the socket, the socket transitions to the *SHUTDOWN* state. When both spliced sockets are in the *SHUTDOWN* state, it implies both end systems have closed their end of the connection and acknowledged the other endpoint's close. The splice is then torn down and the sockets freed.

If the splice code receives a TCP Reset segment from either side, it relays the Reset, and then tears down the splice. Any future packets received at the proxy on those two connections will result in the proxy originating and sending a Reset to the packet's source.

It is reasonably well known that network servers and proxies require improved TIME_WAIT handling in order to support large number of connections. The issue arises because when a normal TCP connection is closed, the socket initiating the close must keep the socket in a state called TIME_WAIT for a period of twice the maximum segment lifetime (MSL), defined to be 2 minutes, in order to give any segments from the connection still in the network time to drain.[12] The presence of a socket in TIME_WAIT prevents the end-system from reusing the port number and IP address pair the old connection used until the 2 MSL period is over, at which time the socket is freed. If the port number and IP address pair are reused before 2 MSL is over, packets from the old connection that had been drifting in the network could show up and confuse the new connection.

During the TIME_WAIT period, it is wasteful of proxy resources to retain an entire socket only to record that a particular port number/address pair can not be reused. In fact, during evaluation we frequently found that the real SOCKS firewall ran out of socket structures when handling more than 40 connection set-ups and tear-downs a second. When a TCP Splice is torn down, the port number/address pair of the server socket can be stored in a simple list while the sockets themselves are freed.