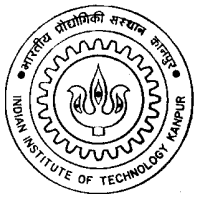
Symmetric TCP Splice: A Kernel Mechanism For High Performance Relaying

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Saugata Das Purkayastha



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

April, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*Symmetric TCP Splice: A Kernel Mechanism For High Performance Relaying*", by *Saugata Das Purkayastha*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

April, 2001

(Dr. Sumit Ganguly)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Acknowledgments

I would like to express my sincere and deep felt gratitude towards my thesis guide Dr. Sumit Ganguly for all the good he has bestowed upon me. Without his encouraging smile and motivating words, I might have found things almost impossible in lot many situations. I am grateful to him for allowing me to work in this field, and the amount of knowledge I have learned from him has been immense. I thank God for having given me an opportunity to work with him.

This work has been a joint work of me and Saibal. We have initially implemented Symmetric TCP Splice and then, we implemented Asymmetric TCP Splice. For thesis defense, we separated our work. Saibal took the Asymmetric TCP Splice, while I took the Symmetric TCP Splice. I thank Saibal for working with me.

I am thankful to Mr. Atul Kumar for his precious support throughout my work. Whether it be a technical discussion or a complex installation problem, he has been patient enough and of immense help to me. I also thank Saibal for his help throughout. At the end I would like to thank all my batch-mates and juniors for constantly encouraging me and also, the Linux-India mailing list for sorting out my problems, during kernel coding.

Abstract

Application layer proxies plays an important role in today's network serving as firewalls and HTTP caches. 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. In this work, we implemented a kernel mechanism called, symmetric TCP splice, to support all class of application layer proxies. In our lab testing, we have seen that, symmetric TCP splice, works as fast as normal router and shows performance improvement of near 25%, over the normal application layer proxy.

Contents

List of Figures

Chapter 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 to encryption servers. Split-connection proxies are attractive because they are backwards compatible with existing servers, allow administration of the service at a single point (the proxy) and typically are very easy to integrate with existing application.

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 work, we implemented a technique called Symmetric TCP Splice, that improves the performance of split connection proxies. The original concept of TCP Splice was introduced by [1]. That work was done to support SOCKS firewall. In this work, we have extended their concept to support general purpose proxies. After we introduce the related concepts we will show, how we can write a HTTP 1.0 proxy using services provided by Symmetric TCP Splice. Our implementation still supports SOCKS, and all our performance testing was based on the SOCKS.

Figure 1.1, shows the architecture of split-connection proxy system. A process running on the client, connects to the proxy. 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.

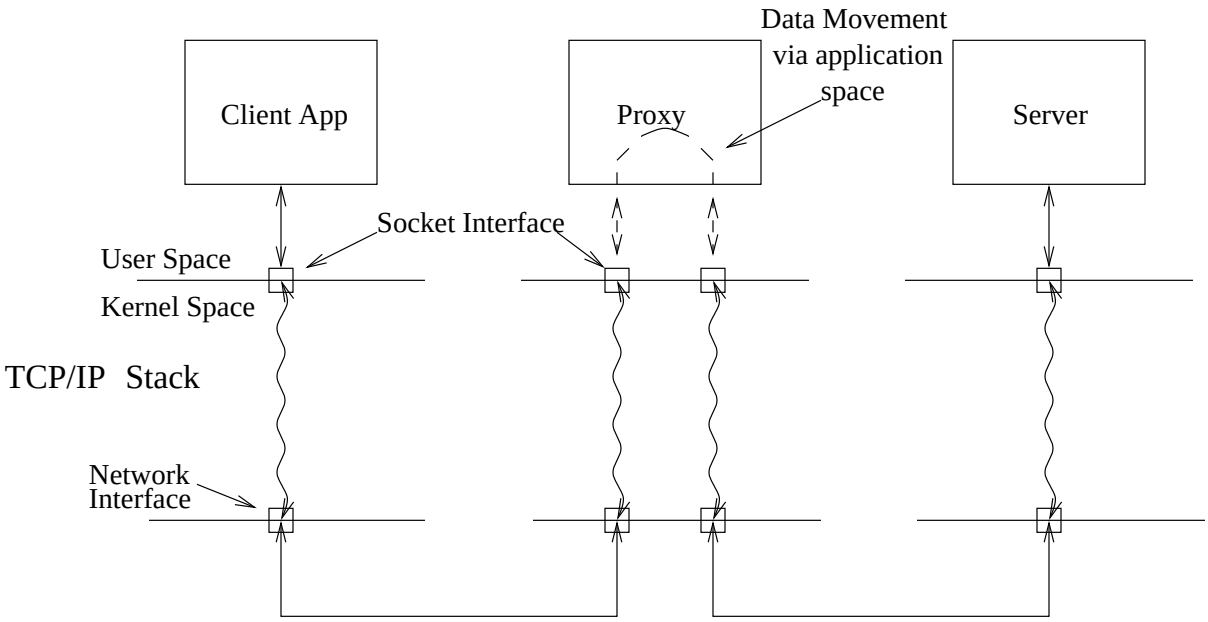


Figure 1.1: Basic architecture of Split Connection Proxy

In order to move data from the server to client, an application layer proxy process reads the data intended for client from the proxy-server connection and writes it into the proxy-client connection. Proxies acting as firewall often need to operate at router speeds and this copying operation cannot be performed at the desired speeds using a general purpose operating system. The cost of moving data twice through the TCP/IP stack, crossing the user/kernel protection boundaries, and the latency of scheduling the process are high enough to make proxies the bottleneck in the system. Our approach is to maximize performance by achieving the tightest possible coupling between the two TCP connections terminating at the proxy.

To support application layer proxies, we have added a generic kernel service

called Symmetric 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 splice set-up 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:

- 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 dramatically improves performance.

- Less book keeping: Proxies using TCP Splice need to maintain less TCP state 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.

1.1 Organization Of The Thesis

In the chapter 2, we give a brief overview of firewall and particularly discuss SOCKS and application layer proxies
In the chapter 3, we present the background required for understanding the implementation of our scheme. We have used Linux kernel (version 2.2.15-4mdk) to implement our scheme and a brief description of the relevant parts is presented here.
Chapter 4, explains the system calls that we have provided to support Symmetric TCP Splice.
Chapter 5, gives the implementation details.
Chapter 6, discusses about some preliminary performance results.
Chapter 7, looks at the conclusions and the future works.

Chapter 2

Firewall

Many organizations connect to the Internet, guarded by firewalls designed to prevent unauthorized access to their private networks. In this chapter, we will discuss the various firewall technologies that are presently used.

2.1 Types Of Firewall

Traditionally, there are three types of firewalls. These are packet filters, application layer proxies and circuit level proxies. A recent advancement in the technology are the firewalls based on stateful packet filters.

2.1.1 Packet Filters

The first type of firewall is called a packet filter. Packet filters use routers to filter information coming to and from a network. Routers check headers of each packet against an access-control list, to determine whether to route or drop the packet. Generally, the header fields that are available to the filter are packet type (TCP, UDP etc), source and destination IP address, source and destination TCP/UDP port. In addition to the information contained in the headers, many filtering implementation also allow the administrator to specify rules based on the interface the packet came in or destined to go out.

2.1.2 Application Layer Proxy

As network configurations became more complex, the need for a more sophisticated level of security emerged. Application-level proxies were developed to meet these requirements. Instead of a direct connection between an internal and external network, application-level proxies serve as a middle-man for Internet services. The proxy intercepts all traffic and relays packets of data back and forth between the external and the internal network, at the application layer.

2.1.3 Circuit Level Proxies

Circuit level proxies are much like application level proxies. However, unlike application level proxies, circuit level proxies control the flow of data at the session layer. Working at session layer means that the proxy actually establishes a virtual circuit between the client and the host on a session by session basis.

A popular circuit level proxy is SOCKS server. The original implementation of TCP Splice was based on SOCKS server. A step by step description, of how client gets its requested object, via a SOCKS server, is given below.

- First, the application client sends the SOCKS server a list of authentication methods that it can support. Next, the SOCKS server examines the server security policy. Network administration's define the server security policies using the SOCKS server configuration. If any of the methods declared by the client meet the defined security policy, the SOCKS server chooses a method of authentication. If none of the methods meet the security policy, the SOCKS server drops communication with the client. After the SOCKS server chooses a method of authentication, it sends a message to the client declaring which method the client should use. Then the authentication process takes place between application client and the SOCKS server.

- After, the client is authenticated, the application client sends a request to the SOCKS server. The request contains the address of the application sever the client wants to connect.

- The SOCKS server, in turn sends a request to the application server address contained in the client request. Using TCP, the SOCKS server communicates with the application server and establishes a proxy circuit between the application client and application server. After the circuit is established, the SOCKS server sends notification to the client application.
- Lastly, with the proxy circuit established, communication between the application client and application server begins. The SOCKS server intercepts each piece of data coming from the client and server and relays it between the two computers.

To summarize, the SOCKS server performs the following basic functions.

- Authenticates clients
- Handles connection requests
- Creates a proxy circuit setup
- Relays data between application

The pseudo-code for SOCKS server is shown in figure 2.2. For more detail about the authentication, request and replies done at various stages, the reader is requested to go though [6].

```
                cli_sock_des=socket();
                if socket returned error
                        return;
                new_cli_sock_des = accept(cli_sock_des);
                if accept returned error
                        close(cli_sock_des); return;
                <authenticate client and read the client request>
                serv_sock_des = socket();
                if socket returned error
                        send appropriate error message to client; goto socket_err;
                connect(serv_sock_des) to server;
                if connect returned error
                        send appropriate reply to client; goto connect_err;
                while(1) {
                        read()  from cli_sock_des, write to serv_sock_des;
                        read() from serv_sock_des, write to cli_sock_des;
                        if(cli_sock_des  and serv_sock_des return EOF)
                                break;
                }
connect_err:    close(serv_sock_des);
  socket_err:   close(new_cli_sock_des);
                <service next request>
```

Figure 2.1: Pseudo-code for a SOCKS Server

Chapter 3

Background

3.1 TCP Background

Before starting discussion about TCP Splice, some background on TCP is required (See [8] for detail). Each TCP segment is typically sent in a single IP packet containing an IP header and TCP header followed by TCP data. The IP header contains the IP address of the packet's source and 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 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 3.1, shows 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 sockets, the normal TCP state machine maintains the following three counters. Using these counters, TCP assigns each byte of data sent over the connection a sequence number and acknowledgement number, so that peer TCP state machine can detect and recover from data loss or duplication.

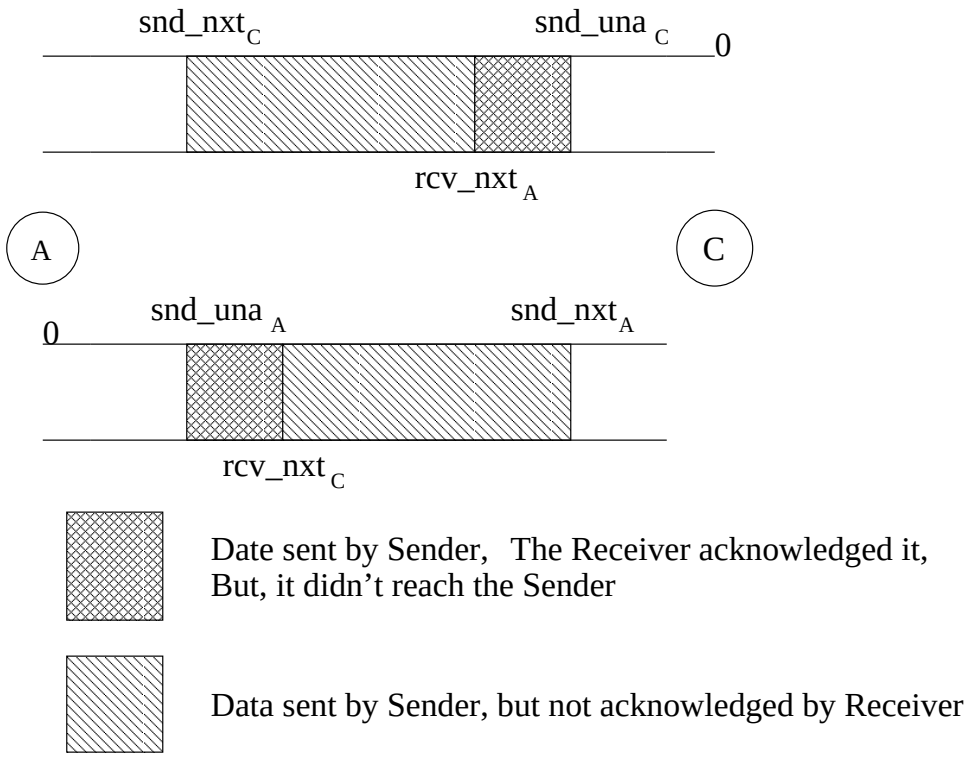


Figure 3.1: A normal TCP connection between two sockets, A and C, with state counters labelled

- 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 expect 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 that snd_nxt have not yet been sent. Data bytes with sequence numbers less that rcv_nxt have been received by TCP stack, but, perhaps not yet read by the application. We say that

data is acknowledged when the sender of the data receives an acknowledgement for it: snd_una will be less than rcv_nxt whenever an ACK is in flight, delayed or lost.

In addition to these variables, Linux keeps following queues to queue incoming and outgoing TCP segments.

- write_queue: This is used for queueing all the outgoing segments. When, the application writes some data, kernel constructs a sk_buff structure (discussed later), with this data, and queues it in the write queue. It is removed from this queue, when acknowledgment for these data bytes arrives.

- recv_queue: This is used for queueing TCP segments which are received in-order.

- out_of_order_queue: This is used for queueing TCP segments which are received out of order.

3.2 Overview of Linux Ipv4 Stack Implementation

The Linux Ipv4 stack is implemented in a layered fashion with each layer responsible for some well defined networking functionality [9]. This is illustrated by the following figure 3.2.

Just like the network protocols themselves, Linux implements the Internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software concerned only with BSD sockets. Supporting this is the INET socket layer, this manages the communication end points for the IP based protocols TCP and UDP. When UDP packets are transmitted, Linux neither knows nor cares if they arrive safely at their destination. TCP packets are numbered and both ends of the TCP connection make sure that transmitted data is received correctly. The IP layer contains code implementing the Internet Protocol. This code prepends IP headers to transmitted data and understands how to route incoming IP packets to either TCP or UDP. Underneath the IP layer, supporting all of the Linux's networking are the networking devices, for example PPP and Ethernets. Network devices do not always represent physical

devices, some like the loopback device are purely software devices. Unlike standard Linux devices that are created via the mknod command, network devices appear only if the underlying software has found and initialized them.

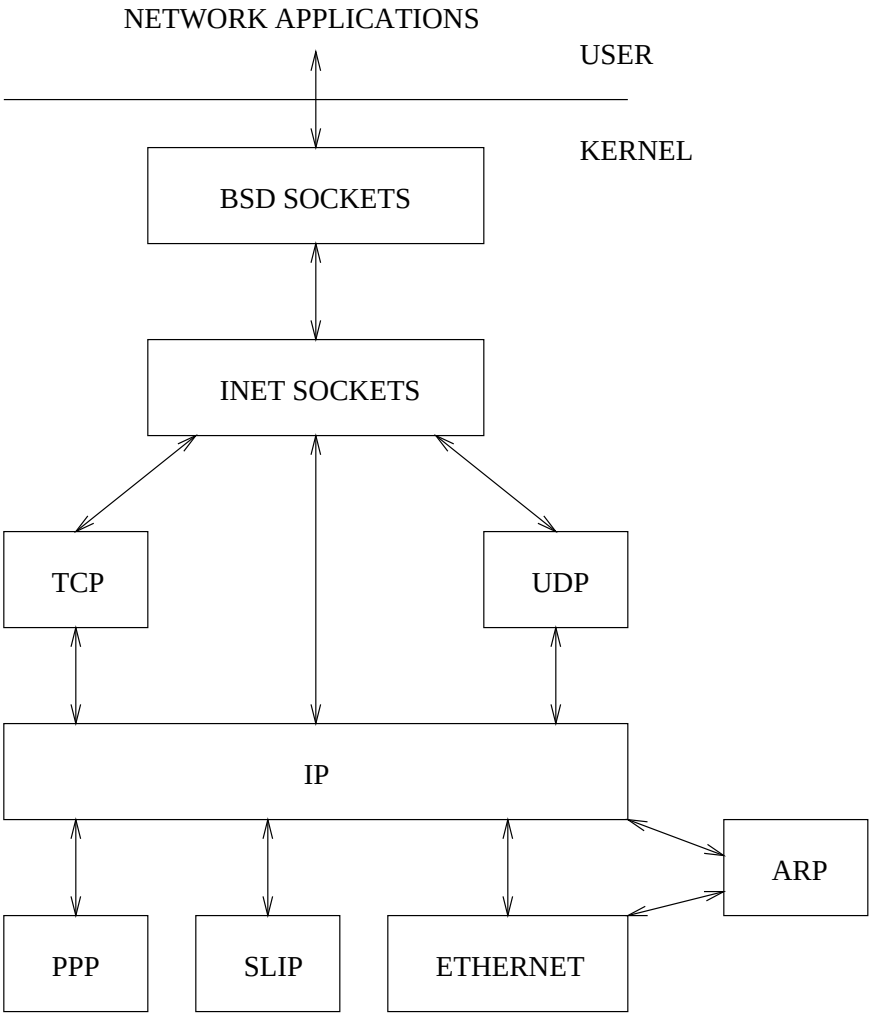


Figure 3.2: Linux TCP/IP network layer

This layering will result in protocol headers/trailers being added and stripped as a packet moves along the stack. The figure 3.3 gives an example of packet structure.

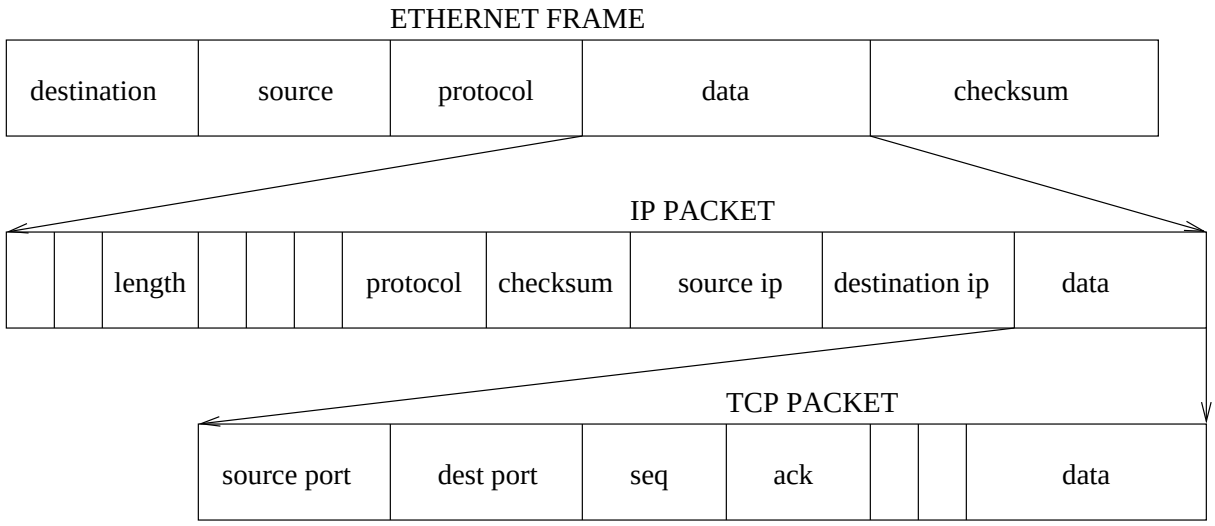


Figure 3.3: Network packet

3.3 Socket Buffers - sk_buff

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data [9]. This make passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses socket buffers or sk_buffs to pass data between the protocol layers and the network device drivers. sk_buffs contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions.

Figure 3.4 shows the sk_buff data structure, each sk_buff has a block of data associated with it. The sk_buff has four data pointers, which are used to manipulate and manage the socket buffer's data. (The reader is referred to include/linux/skbuff.h of Linux source for details).

head points to the start of the data area in memory. This size is fixed when the sk_buff and its associated data block is allocated.

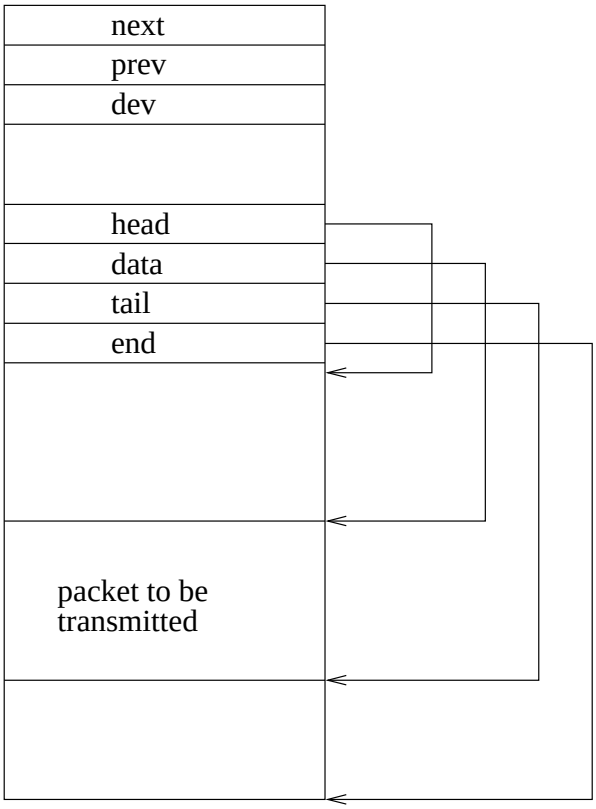


Figure 3.4: Network socket buffers

data points at the current start of protocol data. This pointer varies depending on the protocol layer that currently owns the sk_buff.

tail points at the current end of the protocol data. Again, this pointer varies depending on the owning protocol layer.

end points at the end of the data area in memory. This is fixed when the sk_buff is allocated.

There are two length fields len and truesize, which describe the length of the current packet and total size of the data buffer respectively. The sk_buff handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. These safely manipulate the data, tail and len fields in the sk_buff.

push: This moves the data pointer towards the start of the data area and increments the len field. This is used when adding data or protocol headers to the start of the data to be transmitted.

pull: This moves the data pointer away from the start, towards the end of the data area and decrements the len field. This is used when removing data or protocol headers from the start of the data that has been received.

put: This moves the tail pointer towards the end of the data area and increments the len field. This is used when adding data or protocol information to the end of the data to be transmitted.

trim: This moves the tail pointer towards the start of the data area and decrements the len field. This is used when removing data or protocol tails from the received packet.

The sk_buff data structure also contains pointers that are used as it is stored in doubly linked circular lists of sk_buff's during processing. There are generic sk_buff routines for adding sk_buffs to the front and back of these lists and removing them.

The primary goal of the sk_buff routines is to provide a consistent and efficient buffer handling method for all of the network layers, and by being consistent to make it possible to provide higher level sk_buff and socket handling facilities to all the protocols.

3.4 Interrupt Handling

Interrupts are used to allow the hardware to communicate with the operating system. There are two types of interrupt in Linux: fast and slow interrupts. Slow interrupts are the usual kind. Other interrupts are legal while they are being dealt with. After a slow interrupt has been processed, additional activities requiring regular attention

are carried out by the system - for example, the scheduler is called as and when required. A typical example of slow interrupt is the timer interrupt. Fast interrupts are used for the short, less complex tasks. While they are handled other interrupts are blocked. A typical example is the keyboard interrupt [9].

3.5 Bottom Half Handling

There are often times in a kernel, when you do not want to do work at this moment. A good example of this is during interrupt processing. When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work to be done later on [9].

The figure 3.5 shows the kernel data structures associated with bottom half handling. There can be up to 32 different bottom half handlers. bh_base is a vector of pointers to each of the kernel's bottom half handling routines. bh_mask and bh_active have their bits set according to what handlers have been installed and which of the bottom halves are active. If bit N of bh_mask is set then the Nth element of bh_base contains the address of a bottom half routine. If bit N of bh_active is set then Nth bottom half handler routine should be called as soon as possible. After each system call or slow interrupt, if no further interrupt is running, a check is made to see whether any of the bottom half is active and if active, appropriate bottom half handler is called. Some of the kernel's bottom half handlers are device specific, but others are more generic.

We will now discuss some interesting bottom halves:

IMMEDIATE_BH: This is a generic bottom half used by several device drivers to register tasks to be done later. Registration is done by queueing the task in *tq_immediate*, which is one of the Task Queues[1] maintained by the kernel.

[1]The Task Queue is a list of tasks, to be executed later. Linux kernel maintains four such queues. These are *tq_scheduler*, *tq_timer*, *tq_immediate*, *tq_disk* [9].

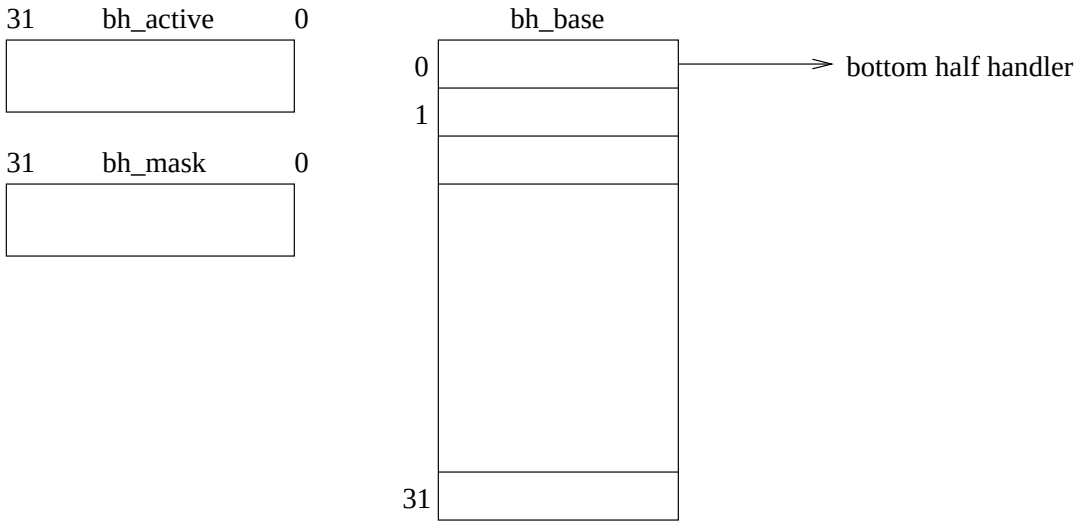


Figure 3.5: Network bottom half handling

After registering a task this bottom half is marked active.

TQUEUE_BH: This bottom half is activated at each timer tick if a task is registered in *tq_timer*, which is like *tq_immediate* another Task Queue maintained by kernel. TQUEUE_BH is always executed later than IMMEDIATE_BH.

TIMER_BH: This bottom half is marked by *do_timer*, the function in charge of the clock tick. The function that this bottom half executes is the one that drives the kernel timers.

CONSOLE_BH: The console performs tty switching in a bottom half. This operation can involve process control. For instance, switching between the X Window system and text mode is controlled by the X server. Moreover, if the keyboard driver asks for a console change, the console switching can not be done during the interrupt. It also can not be done while a process is writing to the console. Using a bottom half fits the task because bottom halves can be disabled at driver's will; in this case, CONSOLE_BH is disabled during a console write.

NET_BH: Network drivers should mark this queue to notify the upper network layers of events. We will discuss more about NET_BH in the next section.

3.6 Network Packet Handling

3.6.1 Receiving IP packets

At boot time, the network drivers are built into the kernel and initialized. This results in a series of device data structures linked together in the dev_base list. Each device data structure describes its device and provides a set of callback routines that the network protocol layers call when they need the network driver to perform work. These functions are mostly concerned with transmitting data and with the network device's address. When a network device receives packet from its network it must convert the received data into sk_buff data structure. These received sk_buff are added onto the backlog queue by the network driver. If the backlog queue grows too large, then the received sk_buff are discarded. Next the driver marks the NET_BH.

When the network bottom half handler is run, it processes any network packets waiting to be transmitted before processing the backlog queue of sk_buff. It determines which protocol layer to pass the received packets to. As the Linux networking layers are initialized, each protocol registered itself by adding a packet_type (see include/linux/netdevice.h of the Linux source) data structure onto either the ptype_all list or into the ptype_base hash table. The packet_type data structure contains the protocol type, a pointer to a network device, a pointer to the protocol's receive data processing routine and finally, a pointer to the next packet_type data structure in the list or hash chain. The ptype_all chain is used to snoop all packets being received from any network device and is not normally used. The ptype_base hash table is hashed by protocol identifier and is used to decide which protocol should receive the incoming network packet. The network bottom half matches the protocol types of incoming sk_buff against one or more of the packet type entries in either table. The protocol may match more than one entry, for example when snooping all network traffic, and in this case the sk_buff will be cloned. The sk_buff is passed to the matching protocol's handling routine. For example, IP packets are passed onto ip_rcv.

3.6.2 Sending IP Packets

Packets are transmitted by applications exchanging data or else are generated by the network protocols. Whichever way the data is generated, an sk_buff is built to contain the data and various headers are added by protocol layers as it passes through them.

The sk_buff need to be passed to a network device to be transmitted. First though the protocol, for example IP, need to decide which network device to use. This depends on the best route for the packet. For computers connected by modem to a single network, say via the PPP protocol, the routing choice is easy. The packet should either be sent to the local host via the loopback device or to the gateway at the end of the PPP modem connection. For computers connected to an Ethernet the choice are harder as there are many computers connected to the network. For every IP packet transmitted, IP uses the routing tables to resolve the route for the destination IP address. Each IP destination successfully looked up in the routing tables returns a rtable data structure describing the route to use. This includes the source IP address to use, the address of the network device data structure and sometimes, a prebuilt hardware header. This hardware header is network device specific and contains the source and destination physical address and other media specific information. If the network device is an Ethernet device, the hardware header would be as shown in figure 3.3 and source and destination address would be physical Ethernet address. The hardware header is cached with the route because it must be appended to each IP packet transmitted on this route and constructing it takes time. The hardware header may contain physical addresses that have to be resolved using the ARP protocol. In this case the outgoing packet is stalled until the address has been resolved. Once it has been resolved and the hardware header built. The hardware header is cached so that future IP packets sent using this interface do not have to ARP.

3.6.3 Forwarding IP packets

The packets from a source and to a destination other than this machine are forwarded to the next hop machine provided that this machine is configured to be

an ip_forwarder and there is a valid routing table entry for the destination machine. This machine will act as a router. The ip_forwarding can be controlled via a configuration parameter or a sysctl variable using the proc system interface. The function ip_rcv checks if a particular packet is for the machine itself. If yes, it will provide the packet to the appropriate upper level protocol's receive handler. If this is not the case, and the ip_forwarding is enabled, it will call the ip_forward routine. The ip_forward routine will consult the routine table to find the next hop for this destination and transmits the packet to that machine.

Chapter 4

TCP Splice

We mentioned in chapter 1, that Symmetric TCP Splice takes care of all the data forwarding operations directly in kernel. The intuition behind Symmetric TCP Splice is that we can change the headers of the 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. Using Symmetric TCP Splice, the architecture of Split Connection Proxies is shown in figure 4.1. Compare it with figure 1.1, of Split Connection Proxy without splice. The effect is to have the proxy relay packets as if it were a layer 3.5 router. 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 lost, is replaced by a system call, splice(). After the splice, the proxy can go to other tasks. In this chapter we will first discuss all the system calls, that we have introduced, for supporting Symmetric TCP Splice and later using these system calls, we will show the pseudocode of SOCKS server and HTTP 1.0 proxy.

4.1 System Calls For Supporting Symmetric TCP Splice

In the discussions to follow, client and server connection means, TCP connection established between proxy and client and between proxy and server respectively. The

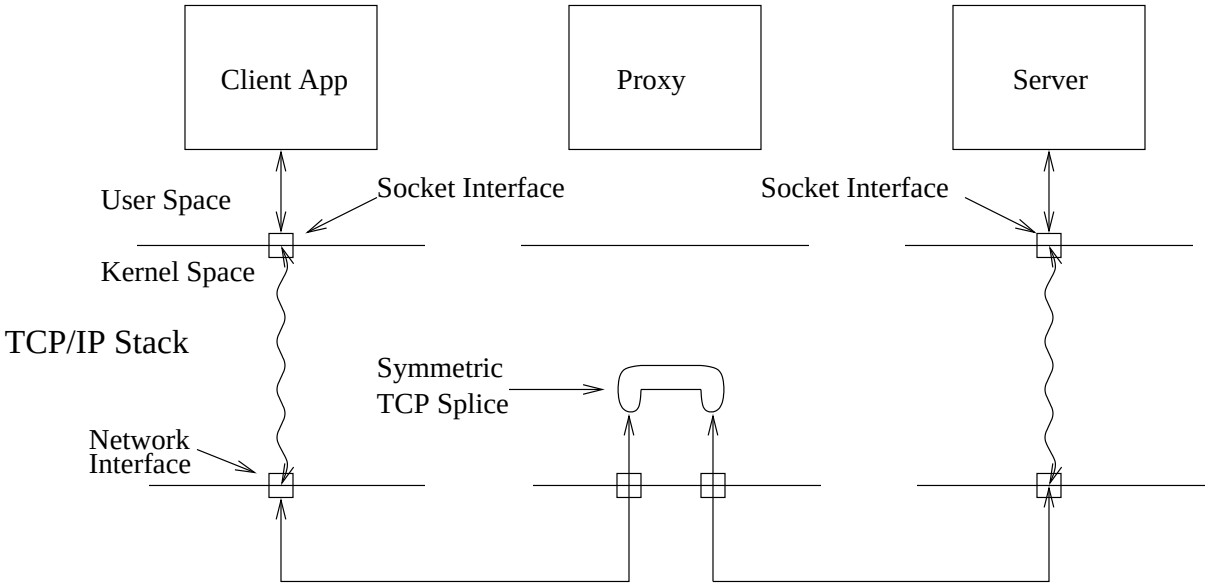


Figure 4.1: Basic architecture of Split Connection Proxy with Symmetric TCP Splice

client interface means the proxy end of the client connection. And server interface means the proxy end of the server connection. Also, *client socket descriptor* and *server socket descriptor* means, the socket descriptors associated with the client interface and server interface respectively. Followings are the system calls, that we have provided to support Symmetric TCP Splice.

- **init_splice():** This system call is called at the beginning, with *client socket descriptor*. This should be called, before the client connection is established, i.e, at some place between socket() and accept() system call, for the *client socket descriptor*.

- **intend_splice():** This system call is called with *client socket descriptor* and the *server socket descriptor*, that are intended to be spliced later. The client connection should be established at that point, but the server connection should not be established, i.e, we have done only socket() system call, for the *server socket descriptor*, but not connect(). After intend_splice is called, reading from the *client socket descriptor* is not allowed.

- **splice():** This system call is called with *client socket descriptor*, *server socket descriptor* and a *length* parameter. The client and server connections must be established. The *length* parameter is the number of bytes that can be written to the *client socket descriptor* after the splice system call is made. This system call *sets up* the splice, between the client and server connections and from that point on, data transfer between the client and server connections are done inside the kernel. After splice system call is made, writing more than *len* bytes of data to the *client socket descriptor* or writing any data to the *server socket descriptor* is not allowed.

It should be noted that, the application is not allowed to read from the *server socket descriptor* at any point of time. Now with this primitives, we will show the pseudocode for both SOCKS server and HTTP 1.0 proxy.

4.2 SOCKS Server

The SOCKS server pseudocode is shown in the figure 4.2.

First the SOCKS server opens a *client socket descriptor* for accepting connection request from client. Then it does init_splice with this *client socket descriptor* and waits for client connection. After accepting client connection, it authenticates the client and if authentication is successful, it goes on to read the client request, which is typically, the remote host IP address and port number, to which the client wants to connect. SOCKS verifies the request, and if everything is ok, then it opens *server socket descriptor* and does a intend_splice with the *client socket descriptor* and *server socket descriptor*. Then it connects to the server and splices the client and server connections. Then it sends a "ok" message to the client and closes the descriptors. Note that when a descriptor, which is spliced, is closed, kernel just frees the process tables entry for this descriptor and does not send fin to the peer. Next, the SOCKS server goes on to serve the next request.

For more details about the authentication, request and reply messages used in SOCKS, the reader is requested to see [6].

```
                    cli_sock_des=socket();
                    if socket returned error
                            return;
                    init_splice(cli_sock_des);
                    if init_splice returned error
                            close(cli_sock_des); return;
                    new_cli_sock_des = accept(cli_sock_des);
                    if accept returned error
                            close(cli_sock_des); return;
                    <authenticate client and read client request>
                    serv_sock_des = socket();
                    if socket returned error
                            send appropriate reply to client; goto socket_err;
                    intend_splice(new_cli_sock_des , serv_sock_des);
                    if intend_splice returned error
                            send appropriate reply to client; goto intend_splice_err;
                    connect(serv_sock_desc) to server;
                    if connect returned error
                            send appropriate reply to client; goto connect_err;
                    n = sizeof  "OK"  mesage;
                    splice(new_cli_sock_desc , serv_sock_desc , n);
                    If splice returned no error
                            send(cli_sock_desc , "OK"  message);
                    else if splice returned error
                            send appropriate reply to client;
intend_splice_err :
      connect_err :  close(serv_sock_desc);
        socket_err :  close(new_cli_sock_desc);
                    <service next request>
```

Figure 4.2: Pseudocode of SOCKS server using Symmetric TCP Splice for data relay

4.3 HTTP 1.0 Proxy Server

The HTTP 1.0 proxy server pseudocode is shown in the figure 4.3.

First the proxy opens a *client socket descriptor* for accepting connection request from client. Then it does init_splice with this *client socket descriptor* and waits for client connection. After accepting client connection, it authenticates the client and if authentication is successful, it goes on to read the client request, which, typically is a HTTP Get request. It checks if it can serve the request from its cache. If so, it serves the request from its cache. Otherwise, it will have to establish a connection to the server for the requested object. For that it first opens *server socket descriptor* and does intend_splice with the *client socket descriptor* and *server socket descriptor*. Then it connects with the server and forwards the client request to the server. After that it splices the two connections. Then, it closes both the descriptors and goes on to serve next request.

For more details about the appropriate request and reply messages used in HTTP 1.0, the reader is requested to see [7].

4.4 Discussion About Caching At Proxy

One of the important jobs of any proxy server is its capability, to serve from its cache, because, it improves the response time at the client end. When it copies data from server connection to client connection, it keeps a copy at its cache for later use. Using TCP Splice, its not possible to cache data directly, since, the data do not go up to the application layer. For supporting such service, recently a new concept of TCP Tapping is introduced. As packets are relayed as in normal TCP Splice, a copy of the data is also kept in a local socket buffer reassembly queue, called the Splice TCP buffer, which the application can read. For more information about TCP Tap, reader is requested to go through [2].

```
                    cli_sock_des=socket();
                    if socket returned error
                            return;
                    init_splice(cli_sock_des);
                    if init_splice returned error
                            close(cli_sock_des); return;
                    new_cli_sock_des = accept(cli_sock_des);
                    if accept returned error
                            close(cli_sock_des); return;
                    <authenticate client>
                    read the HTTP request from client;
                    check  in cache for the request;
                    if found in cache then serve from cache; goto served_from_cache;
                    serv_sock_des = socket();
                    if socket returned error
                            send appropriate reply to client; goto socket_err;
                    intend_splice(new_cli_sock_des , serv_sock_des);
                    if intend_splice returned error
                            send appropriate reply to client; goto intend_splice_err;
                    connect(serv_sock_des) to server;
                    if connect returned error
                            send appropriate reply to client; goto connect_err;
                    forward the HTTP request  to server;
                    splice(new_cli_sock_des , serv_sock_des , 0);
                    if splice returned error
                            send appropriate reply to client;
intend_splice_err :
     connect_err :  close(serv_sock_des);
      socket_err :
served_from_cache:  close(new_cli_sock_des);
                    <service next request>
```

Figure 4.3: Pseudocode of HTTP 1.0 proxy server using Symmetric TCP Splice for data relay

Chapter 5

Implementation

The Symmetric TCP Splice, is implemented as a state machine. In this chapter, we will first explain the state machine in detail.

5.1 Symmetric TCP Splice State Machine

From the point, the application does intend_splice with *client socket descriptor* and *server socket descriptor*, we associate a *splice-state* with each of the TCP connections. In general, a splice is associated with a pair of TCP connections, one of which is the client connection and the other is the server connection. The *splice-states* transitioned by the client interface and server interface is shown in the figure 5.1. Although, the *splice-state* transitioned by the client and server interface are almost same, they are shown separately for convenience.

Now we will explain each *splice-state*,

- **CLOSE:** This *splice-state* is entered only by the server interface. It is entered, when the application calls intend_splice. It is the starting point of the server interface *splice-state* transition. Server interface changes its *splice-state* to ESTABLISHED, when the Server interface TCP state enters the ESTABLISH state.

- **ESTABLISHED:** This *splice-state* is entered both by client interface and

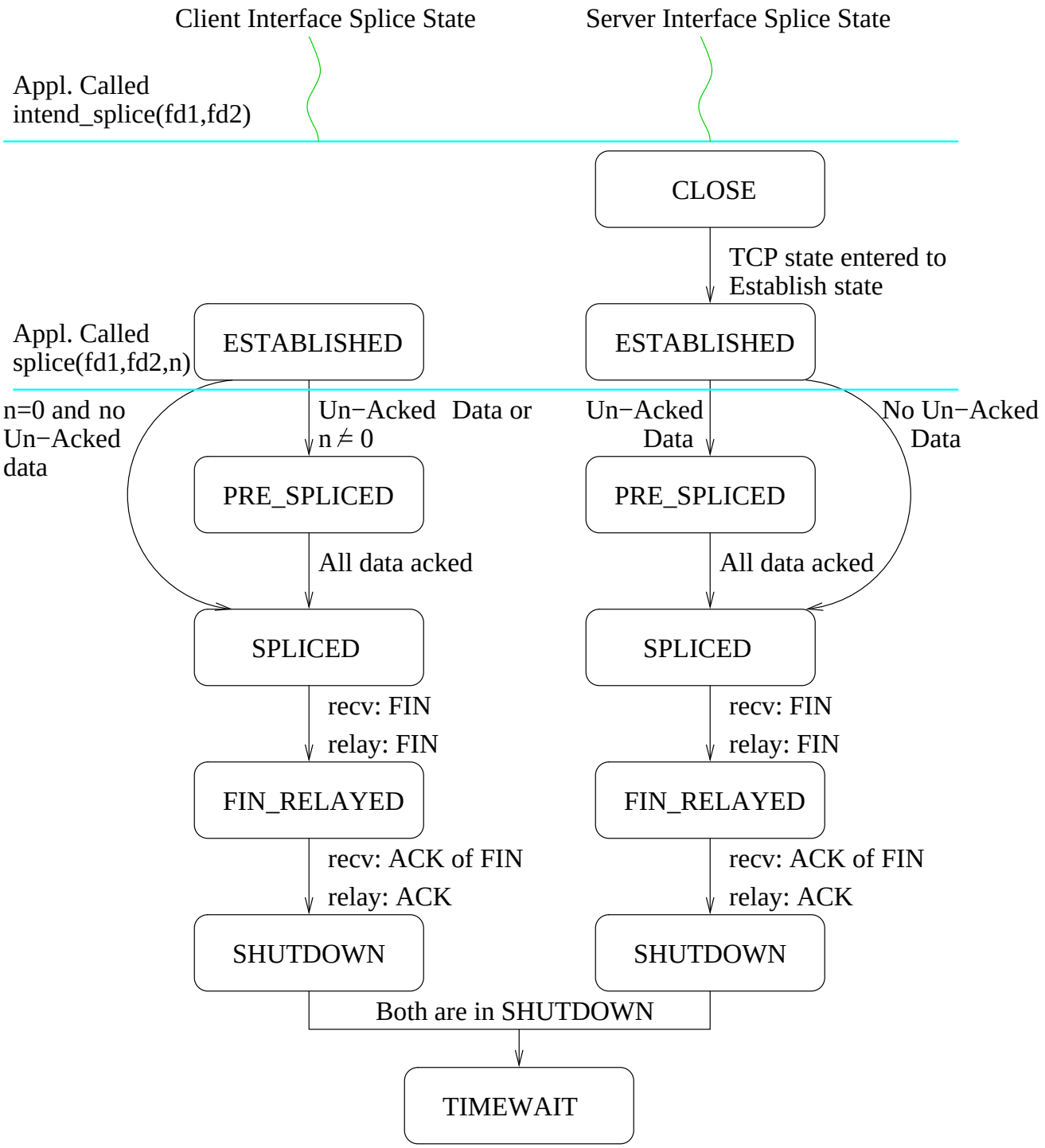


Figure 5.1: State diagram for spliced TCP connections

server interface. The client interface enters this *splice-state*, when the application calls intend_splice. It is the starting point of the client interface *splice-state* transition. When the client interface *splice-state* is at ESTABLISHED, all TCP segments from client are queued and are not given to application, instead they are transferred to the server connection when the splice is *set up*. The client interface leaves this *splice-state* when the application calls splice. From ESTABLISHED client interface changes its *splice-state* to SPLICED, if all data sent by proxy to client are acknowledged, i.e, the client interface write queue is empty and the application notifies that, it will not write any more data at the *client socket descriptor* after splice system call, by setting the third parameter of the splice system call to zero. Otherwise, client interface *splice-state* changes to PRE_SPLICED. When the client interface changes *splice-state* from ESTABLISHED and if the server interface *splice-state* is not PRE_SPLICED then, all the TCP segments from client, that were queued at this *splice-state*, are forwarded to the server connection.

The server interface *splice-state* enters ESTABLISHED from CLOSE. Similar to the client interface when the server interface *splice-state* is at ESTABLISHED, all TCP segments from server are queued. Server interface leaves this *splice-state* when the application calls splice. From ESTABLISHED the server interface *splice-state* moves to SPLICED, if the server interface write queue is empty, i.e, all data sent from proxy to server are acknowledged. Otherwise, it moves to PRE_SPLICED. When the server interface *splice-state* changes from ESTABLISHED and if the client interface *splice-state* is not PRE_SPLICED then all the TCP segments from server, that were earlier queued, are forwarded to client connection.

- **PRE_SPLICED:** This *splice-state* is entered by client interface from ESTABLISHED. Any TCP segments from server are queued, when the client interface *splice-state* is at PRE_SPLICED. This is done due to the reason that, may be the client connection is congested. So, instead of forwarding the segments and making the situation worse, we queue it. From PRE_SPLICED client interface changes its *splice-state* to SPLICED, when the application had

written the data bytes at the *client socket descriptor*, it was supposed to write after the splice system call and all data are acknowledged at the client interface, i.e, the write queue at the client interface is empty. When the client interface *splice-state* changes from PRE_SPLICED all the TCP segments from the server, that were queued previously, are relayed to the client connection.

PRE_SPLICED, is entered by the server interface *splice-state* from the ESTABLISHED. When the server interface *splice-state* is at PRE_SPLICED, any TCP segments from the client are queued. This is done due to the same reason of congestion as explained above. From PRE_SPLICED, server interface *splice-state* moves to SPLICED, when all data are acknowledged at the server interface, i.e, the write queue at the server interface is empty. When the server interface *splice-state* changes from PRE_SPLICED all TCP segments from client, that were queued previously, are relayed to the server connection.

- **SPLICED:** Client interface *splice-state* enters SPLICED, from either ESTABLISHED or from PRE_SPLICED as explained above. When the client interface *splice-state* is at SPLICED any TCP segments from server connection are relayed to client connection, after properly updating of the TCP and IP headers, as explained later in this chapter. From SPLICED client interface *splice-state* moves to FIN_RELAYED when fin is relayed from server connection to client connection.

  Server interface *splice-state* enters SPLICED from either ESTABLISHED or PRE_SPLICED as explained above. When the server interface *splice-state* is at SPLICED all TCP segments from client are relayed to server connection, after proper modification of the TCP and IP headers. From SPLICED server interface *splice-state* moves to FIN_RELAYED when fin is relayed from client connection to server connection.

- **FIN_RELAYED:** Client interface enters this *splice-state* from SPLICED. However, there may be special transitions to FIN_RELAYED from either ESTABLISHED or PRE_SPLICED, not shown in figure 5.1. When the server

interface *splice-state* changes from ESTABLISHED or when the client interface *splice-state* changes from PRE_SPLICED, it relays TCP segments from server, that were queued, to the client connection. If a queued fin packet is relayed then the client interface *splice-state* changes to FIN_RELAYED. When the client interface *splice-state* is FIN_RELAYED all TCP segments from server are relayed to the client connection, after modifying the TCP and IP headers. From FIN_RELAYED client interface *splice-state* moves to SHUTDOWN when the acknowledgement, of the fin forwarded, is received from client. When client interface *splice-state* changes to SHUTDOWN check is made to see whether the server interface *splice-state* is also SHUTDOWN and if it is true then, both the interface *splice-state* are changed to TIMEWAIT.

Server interface enters this *splice-state* from SPLICED. However, there may be special transitions to FIN_RELAYED from either ESTABLISHED or PRE_SPLICED similar to the client interface *splice-state* transition as explained above. When the server interface *splice-state* is FIN_RELAYED all TCP segments from client are relayed to the server connection, after modifying the TCP and IP headers. From FIN_RELAYED client interface *splice-state* changes to SHUTDOWN when acknowledgement, of the fin forwarded, is received from server. When server interface *splice-state* changes to SHUTDOWN check is made to see whether the client interface *splice-state* is also SHUTDOWN and if it is true then, both the interface *splice-state* are changed to TIMEWAIT.

- **SHUTDOWN:** From FIN_RELAYED both client interface and server interface reaches this *splice-state*. When the client interface *splice-state* is SHUTDOWN all TCP segments from the server connection are relayed to the client connection, after TCP and IP header modification. Similarly, when the server interface *splice-state* is SHUTDOWN all TCP segments from the client connection are relayed to the server connection.

- **TIMEWAIT:** This *splice-state* is entered when both the client and server interface *splice-state* are in SHUTDOWN, i.e, both fin and it's acknowledgements are relayed in either direction. Now, either client or server TCP state

machine is in TIMEWAIT. This TCP state is maintained for the 2MSL time[1]. To support this, we keep both the client and server interface *splice-state* at TIMEWAIT, resembling to the normal TCP state machine TIMEWAIT state. At this *splice-state*, TCP segments from either connection is relayed to the other connection, after modification of the TCP and IP headers. After 2MSL time we tear down this splice between the client and server connection by removing all the related kernel data structures, we maintained for this spliced connections and set the TCP state of both the connection to CLOSE.

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

5.2 Relaying Segments

As each segment is received at a spliced interface, we verify the checksum of the segment, alter the segment's TCP and IP headers, to address the segment to the other end of the spliced connection and forward it. In this section we will discuss, the TCP and IP header fields that need to be changed, before relaying the segment to the other end of the spliced connection.

5.2.1 Alter IP Header

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

[1]MSL or maximum segment lifetime is the time a segment can exist in the network before being discarded. RFC 793 specifies it to be 2 minutes. 2MSL wait provides protection against delayed segments from an earlier incarnation of a connection from being interpreted as part of a new connection that uses same local and foreign IP addresses and port number

5.2.2 Alter TCP Header

- Change source and destination port numbers to match outgoing connection.
- Map the sequence numbers.
- Map TCP options as needed.
- Update TCP header Checksum.

The TCP sequence number and TCP option mapping is discussed later in this chapter. TCP checksum is computed two times for a TCP segment that is relayed, first, when the segment is received, to check whether the segment reached the proxy without error and second, when we forward the segment. For efficiency, during verifying checksum of the incoming segment, we saved the partial checksum of the TCP segments data portion. And when we recompute the checksum of the segment, after altering the TCP and IP headers, we start as usual with the TCP headers, but, we do not compute again partial checksum over the data portion of the segment and instead, use the saved partial checksum and compute the checksum. This saves one extra pass over the data portion of the TCP segment.

The TCP urgent pointer is represented as an offset from the segment's sequence number and it is not changed during the mapping procedure. By directly mapping and relaying segments, TCP Splice preserves the complete semantics of TCP's urgent pointer.

5.3 Mapping Of Sequence Numbers

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 before it. Since all data bytes in TCP are assigned a sequence number in the sequence space of their connection, we achieve a seamless splice by mapping the sequence numbers from one connection's sequence space to the other connection's sequence space.

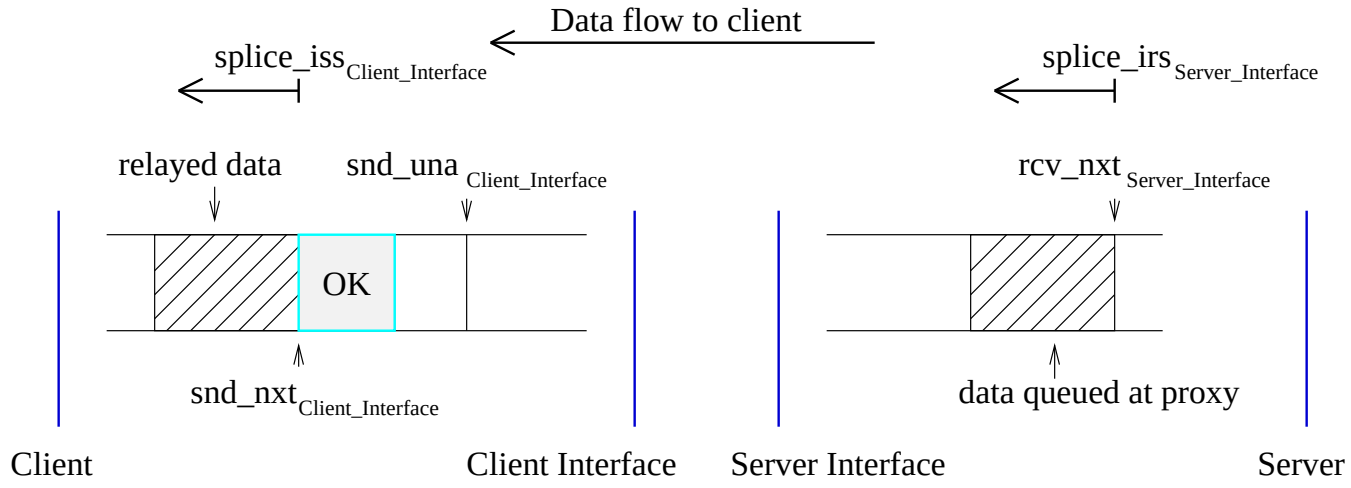


Figure 5.2: Choice of base points for data flowing from server to proxy to client

We call the sequence number of the next data byte to be sent to client from client interface, $splice\_iss_{client\_interface}$ and the sequence number client interface next expects from client, $splice\_irs_{client\_interface}$. Similarly, we call the sequence number of the next data byte to be sent to server from server interface, $splice\_iss_{server\_interface}$ and the sequence number server interface next expects from server, $splice\_irs_{server\_interface}$. The initial values of all the above four splice parameters are shown in the figure 5.2 and figure 5.3.

Now for all TCP segments from client, the sequence number, seq_num_{in} and acknowledgement number, ack_num_{in} are mapped to seq_num_{out} and ack_num_{out} as per following equations,

$$seq\_num_{out} = (seq\_num_{in} - splice\_irs_{client\_interface}) + splice\_iss_{server\_interface}$$

$$ack\_num_{out} = (ack\_num_{in} - splice\_iss_{client\_interface}) + splice\_irs_{server\_interface}$$

Similarly, for all TCP segments from server, the sequence number, seq_num_{in} and acknowledgement number, ack_num_{in} are mapped to seq_num_{out} and ack_num_{out} as per following equations,

$$seq\_num_{out} = (seq\_num_{in} - splice\_irs_{server\_interface}) + splice\_iss_{client\_interface}$$

$$ack\_num_{out} = (ack\_num_{in} - splice\_iss_{server\_interface}) + splice\_irs_{client\_interface}$$

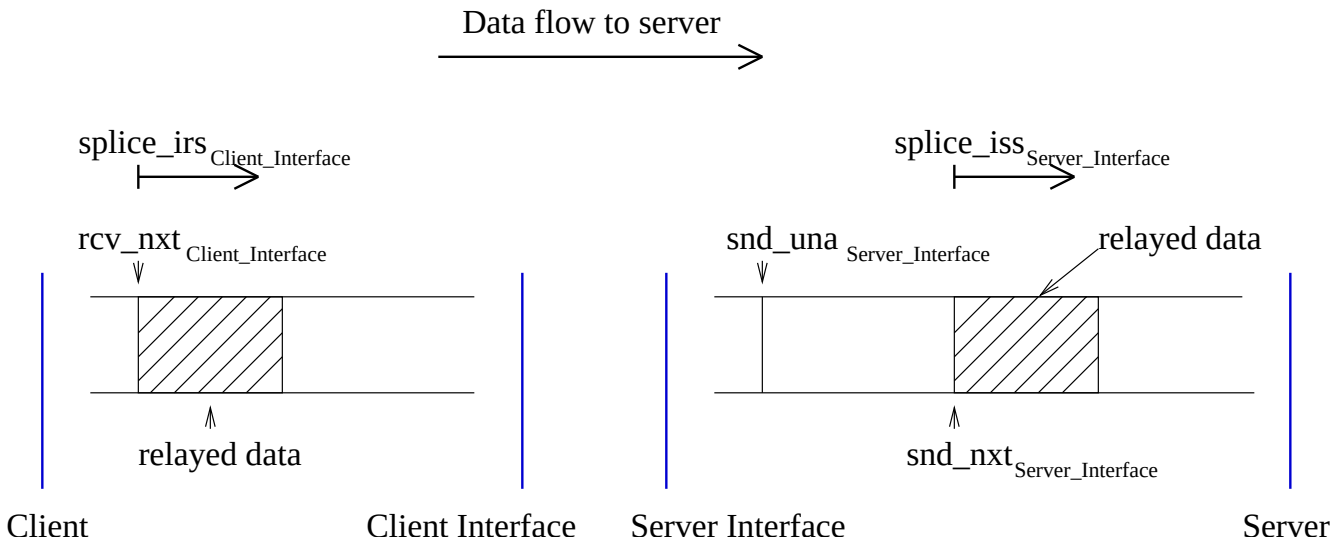


Figure 5.3: Choice of base points for data flowing from client to proxy to server

5.4 TCP Option

The TCP Option 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 with the server node. Since, packets will flow almost directly from client node once the splice is in place, the proxy must 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, when it accepts and negotiates options for the client connection. Not knowing the intended server, the proxy can not simply negotiate compatible options.

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 the options including Timestamps and Maximum Segment Size appears to be sufficient for deployment on corporate networks today.

5.4.1 Timestamp

Before, we go on describing, how we handled Timestamp option [4], we will discuss about the Timestamp option handling in Linux.

Timestamp option is used for two purposes. It provides

1. Protection against wrapped sequence numbers, PAWS.

2. Better rtt calculation.

The Timestamp option carries two 32 bit Timestamp fields, Timestamp Value and Timestamp Echo Reply. The Timestamp Value field (TSval) contains the current value of the Timestamp clock of the TCP, sending the option. The Timestamp Echo Reply field (TSecr) is only valid if the ACK bit is set in the TCP header. If its valid, it echoes a TSval, that was sent by the remote TCP in the Timestamp option. Using TSecr, from the acknowledgement segments, the sender calculates rtt. Also, seeing the TSval, the receiver can detect the wrapped sequence numbers. It has been noted that, in a gigabit network, the 32-bit sequence numbers can wraparound in 17 seconds. Whereas, 32-bit Timestamp Value, wrap around at a much lower frequency. It is found that, even at the highest possible Timestamp counter frequency, the sign bit of the TSval, wraparounds only every 24 days. Therefore, if the TSval of the segment, is less than the most recent Timestamp received from the peer, this segment is old and must be discarded. The segment might be discarded later in the input processing because the sequence number is "old"; but PAWS test is intended for high speed connections, where the sequence numbers can wrap quickly.

Linux maintains following variables for Timestamp option handling,

- **ts_recent:** When a TCP segment is received, it's sequence number is compared with, the last acknowledgment sent over the connection. If they are equal and this segment is not a duplicate segment, then the TSval of the segment is copied to ts_recent.

- **tcp_time_stamp:** This is current value of *jiffies*, a global variable which is incremented on every timer interrupt.

When we relay the segments, we map the TSval and TSecr, of the segment and forward to the other connection, just like as we did for the sequence number. We initialize, the base points for the Timestamp mapping as shown below,

$last\_recvTs_{client\_interface} = ts\_recent_{client\_interface}$

$last\_sentTs_{client\_interface} = tcp\_time\_stamp$

$last\_recvTs_{server\_interface} = ts\_recent_{server\_interface}$

$last\_sentTs_{server\_interface} = tcp\_time\_stamp$

Now we map the Timestamp Value, $TSval_{in}$ and Timestamp Echo Reply, $TSecr_{in}$, of the segment to $TSval_{out}$ and $TSecr_{out}$ respectively.

For the segments going from client to server, we use the following equations,

$TSval_{out} = (TSval_{in}$ - $last\_recvTs_{client\_interface}) + last\_sentTs_{server\_interface}$

$TSecr_{out} = (TSecr_{in}$ - $last\_sentTs_{client\_interface}) + last\_recvTs_{server\_interface}$

Similarly, for the segments going from server to client, we use the following equations,

$TSval_{out} = (TSval_{in}$ - $last\_recvTs_{server\_interface}) + last\_sentTs_{client\_interface}$

$TSecr_{out} = (TSecr_{in}$ - $last\_sentTs_{server\_interface}) + last\_recvTs_{client\_interface}$

In addition to the above rules, if one connection does not support Timestamp option and the other supports, and are spliced together, we simply strip the Timestamp option from the segments, that are passed from the connection supporting this option to the connection not supporting it. Stripping is done by overwriting the Timestamp options by NOP option. Offcourse, if neither of the connections supports Timestamp options, we do not need to do anything.

5.4.2 Window Scale

Window Scale option [4] avoids the limitations of a 16-bit window size field in the TCP header. Larger windows are required for what is called *long fat pipes* networks with either high bandwidth or a long delay. The Window Scale option defines a 1-byte shift count. This shift count can have any values between 0 and 14. Window Scale option can only appear in the SYN packets.

To support Window Scaling option, Linux uses following variables,

- **rcv_wscale:** This is the scale factor for the windows to be advertised to the peer. When TCP decides to advertise a window, it is right shifted, by rcv_wscale times and is placed in the Window field of the TCP header.
- **snd_wscale:** This is the scale factor for unscaling any advertised window from the peer. Unscaling is done by left shifting the window advertised, by snd_wscale times.

Both the above parameters are fixed at the connection setup time. The rcv_wscale is calculated from the socket buffer size and is advertised, in the Window Scale option, in the SYN segment. The peer saves that value as its snd_wscale. Internally, Linux maintains window sizes as 32 bit values, not 16 bit.

To handle the window scale, we do the following things,

1. We set the rcv_wscale for the client interface to 0. This setting of rcv_wscale is done during init_splice system call.
2. We set the rcv_wscale of the server interface to the snd_wscale of the client interface, during the intend_splice system call, i.e, we forward the window scale shown by client to server.

The above steps ensures that, if server does support this option, it will be able to send data to the client using the client's scaled window, though the 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 window scale option, the proxy unscales the window advertised by the client. The unscaled window is clamped at 65535, i.e, if after unscaling the resultant window becomes greater than 65535, it is set to 65535.

5.4.3 Selective Acknowledgements

Selective acknowledgement (SACK) [5] is used to convey extended acknowledgment information over an established connection. Specifically, it is to be sent by a data receiver to inform the data transmitter of non-contiguous blocks of data that have

been received and queued. The data receiver is awaiting the receipt of data in later retransmissions to fill the gaps in sequence space between these blocks. At that time, the data receiver will acknowledge the data normally by advancing the left window edge in the Acknowledgement Number field of the TCP header.

It is important to understand that the SACK option does not change the meaning of the Acknowledgment Number field, whose value still specify the left window edge, i.e, one byte beyond the last sequence number of fully-received data. However, SACK provides additional information that the data transmitter can use to optimize retransmission.

This option contains a list of blocks of contiguous sequence space occupied by data that has been received. Those sequence numbers are mapped in the way similar to the normal sequence number mapping. In addition, if one connection, supporting SACK, is spliced with one that does not, we strip the SACK options that are passed from, the connection supporting it, to the one not supporting it.

5.4.4 Maximum Segment Size

The Maximum Segment Size (MSS) option is sent in the SYN packet to tell the peer TCP machine what is the largest TCP segment, this TCP machine is capable of handling.

Before we go on describing the MSS option handling, we will show the important variables, used by Linux, for supporting MSS option.

- **user_mss:** This variables can be set by the user with setsockopt system call. It specifies the maximum segment size for this socket. It does not count for the TCP options, but includes only bare TCP header.
- **mss_clamp:** This variable stores the MSS negotiated at the connection setup.
- **mss_cache:** This is the current effective sending MSS, including all TCP options except for SACKS.

Now we will discuss, how the MSS value is calculated, which is advertised in the SYN segment during the connection setup. The mss_clamp, which is a 16-bit

unsigned variable, is initialized to 65535. Then TCP checks whether, the user_mss is set or not. If set, it initializes the mss_clamp to user_mss. Next it computes mss_cache, by subtracting the TCP and IP headers length and TCP options length, other than SACKS, from the MTU[2]. Next, a check is made to see, if mss_clamp is greater than the sum of mss_cache and TCP options length (excluding SACKS). If yes then, the mss_clamp is set to the sum of mss_cache and TCP options length (excluding SACKS). In the SYN segment, TCP advertises mss_clamp as the MSS. When, the peer TCP receives the SYN segment, it notes the MSS. Next it checks, whether, the user_mss (at the peer TCP) is set. If set then, the peer TCP set the mss_clamp to the minimum of MSS seen in the SYN segment and the user_mss. Next it checks the MTU of the interface, through which the connection request is received, and finds out the maximum TCP segment size possible, by subtracting the TCP and IP header length from it. Then the peer TCP, takes the minimum of the mss_clamp and maximum TCP segment size computed above, and set mss_clamp to it. Later in the SYN/ACK segment it advertises this mss_clamp.

Now that we have some insight about the MSS option negotiation, we will explain how MSS is negotiated with Symmetric TCP Splice.

In Symmetric TCP Splice, we set the user_mss of the server interface to the mss_clamp of the client interface, during intend_splice. As a result of this step, the MSS known to server, will be acceptable to the client also. However, if the MSS negotiated on the server connection is less than the MSS negotiated on the client connection and if the client sends a TCP segment of size more than the MSS of server connection, we fragment that TCP segment, into smaller fragments, until the sizes of all the new TCP fragments, becomes less the the MSS of the server connection. Though this introduces an overhead, but, such situation will be rare, because, in normal situation, the TCP segments form client, will be "get" requests and the size of those segments will be very small compared to the MSS negotiated. But, no such fragmentation is required for the server to client segments.

[2]There is a limit on the number of bytes an Ethernet frame can carry. This is size is called Maximum Transfer Size or MTU. For Ethernet, MTU is 1500. Most types of networks have such upper limit

Chapter 6

Performance

6.1 Experiments

To evaluate the performance of Symmetric TCP Splice, we compared the response time from the client end with, Symmetric TCP Spliced SOCKS server, normal SOCKS server and Router. The normal SOCKS server and Router was run on the unmodified kernel. The test network is shown in the figure 6.1. Both Proxy and Web server was run on Linux version 2.2.15-4mdk, while the Client was run on Linux version 2.2.14-5.0.

For performance testing, we have written the client, Web server and SOCKS server. The client followed the SOCKS protocol, as discussed earlier. The Web server was a typical HTTP like Web server. The client first establishes a connection with the SOCKS server. And then sends the Web server's IP address and port number to the SOCKS server. The SOCKS server, accepts the client connection and forks a child process, *Child_SOCKS*. The *Child_SOCKS* reads the Web server IP address and port number, from the client and establishes a connection with the Web server. Next, the *Child_SOCKS* splices the two connections, sends the "ok" message to the client and exits. The client waits for the "ok" message and then sends the "get" request to the Web server. This "get" request passes through the kernel of the SOCKS server and reaches the Web server. The Web server after reading the request, sends back the response and closes the connection. We measured the time

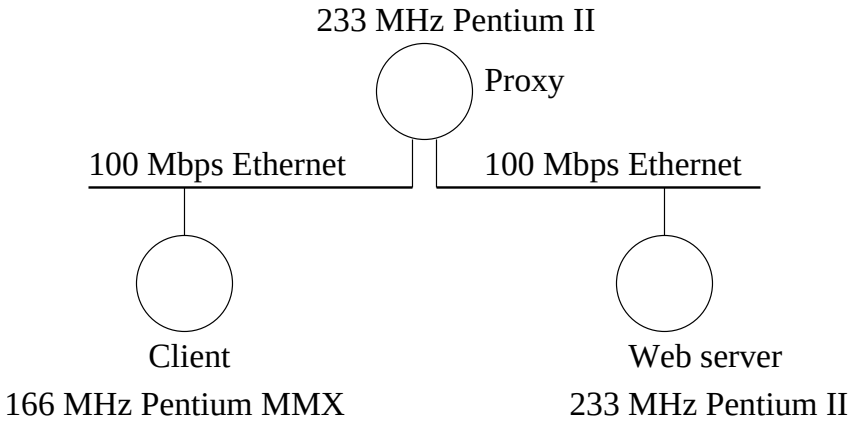


Figure 6.1: Test Network Topology

difference, when the client sends the get request and when it gets back the complete response. We call this the response time in the following discussion.

For testing the response time with Router, we enabled IP forwarding in our proxy machine. The client connects to the Web server directly and sends the "get" request. In this case also, we measured the time difference, when the client sends the "get" request and when it gets back the complete response from Web server.

We have considered two test cases. In the first case, we compared the response time of Spliced SOCKS server, with normal SOCKS server and Router, at minimum load condition, where only one connection exist between the client and SOCKS server and between SOCKS server and Web server. The result is shown in the figure 6.1.

In the second case, we tested the Spliced SOCKS server's performance at high load. Here, the client forks a number of processes and each of them establishes connection with the SOCKS server and follow exactly the same procedure, as described above to send the request and get back the response. The response time is then averaged over the number of connection. We compared the result with the normal SOCKS server. The result for data of different sizes is shown in the figure 6.2.

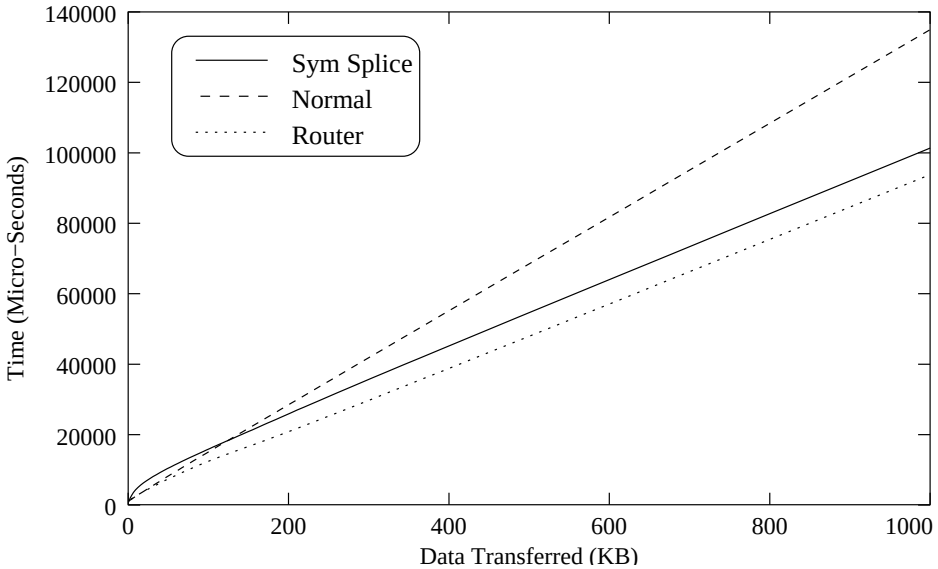


Figure 6.2: Performance Analysis At Low Load

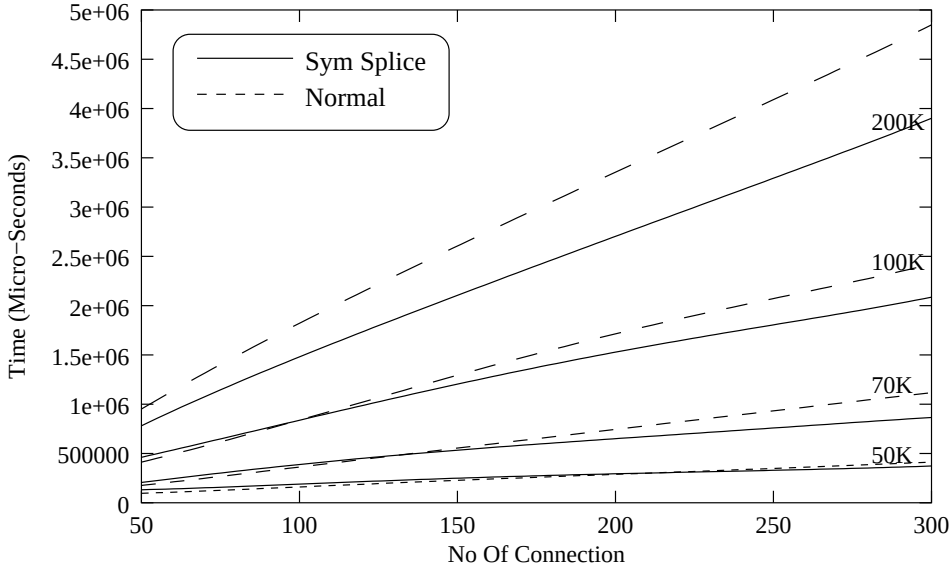


Figure 6.3: Performance Analysis At High Load

6.2 Analysis

From the result, it is seen that, the Symmetric TCP Splice does not perform well for transferring small sized data, but, performs well when the transfer size is high. In fact for transferring data over 1M, the improvement goes above 27% over the normal SOCKS server. It is also, interesting to note that, the spliced SOCKS server, works as fast as Router, when transferring large data.

At high load, the overhead of context switching has affected the response time, of normal SOCKS server and thats why, the performance of Symmetric TCP Splice is better than the normal SOCKS server even when transferring data of smaller size.

Chapter 7

Conclusion And Future Work

We have presented a method for transferring data between two TCP connections inside the kernel. For this we have modified the Linux kernel to suit our purpose. We have also shown how to write SOCKS server and HTTP 1.0 proxy, using the system calls, which we have provided to support Symmetric TCP Splice. And we have shown that response time at the client end, with proxy supporting Symmetric TCP Splice is better than the normal proxy. We have also shown that the proxy with Symmetric TCP Splice works as fast as Router. And more importantly Symmetric TCP Splice performs better with high load.

7.1 Future Work

The Symmetric TCP Splice, as implemented, instructs very rigid form of application programming. For example, we can read from client socket descriptor before intend_splice, but not after that. Also, we can not read from server socket descriptor at all. As an extension of this work, we have also, implemented Asymmetric TCP Splice, where, we have given the flexibility of splicing at any point of time after both the server and client connections are established. That scheme can be included here also. But the main limitation of Symmetric TCP Splice is that, it can not be used with any protocol, that supports persistent connection, like HTTP 1.1. In HTTP 1.1, the client opens a connection with the server and sends multiple

"get" requests over the same connection, without closing it. With Symmetric TCP Splice, we splice for the lifetime of the connections. So, the application layer proxy, supporting HTTP 1.1, after reading the first "get" request will splice and after that, following "get" requests from client, will not reach the application proxy and will pass through the kernel, of the proxy, to reach the server directly. So, after serving first request, any kind of access restriction on the requests from client or even the service from proxy cache is no longer possible. To overcome this drawback, we have implemented Asymmetric TCP Splice, to support persistent connection, where the application gets back the control, after the request is served by the server. But very few protocols support persistent connection. So, the current implementation of Symmetric TCP Splice can support most of the commonly used proxies.

Bibliography

[1] D. Maltz P. Bhagwat, TCP Splicing for application layer proxy performance. IBM Research Report, March 1998.

[2] D. Maltz P. Bhagwat, Improving HTTP proxy caching proxy performance with TCP Tap. IBM Research Report, March 1998.

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

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

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

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

[7] T. Berners-Lee, R. Fielding and H. Frystyk. Hypertext Transfer Protocol-HTTP/1.0. Internet Request For Comments RFC 1945, May 1996.

[8] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, MA, 1996

[9] Michael Beck et al. *Linux Kernel Internals*. Addison-Wesley Publishing Company, 1996