ENHANCE TCP PERFORMANCE WITH MULTIPLE PATH ROUTING

BY
FRANK E. WATSON

B.S., University of Colorado, 2000

A thesis submitted to the Graduate Faculty of the

University of Colorado at Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

2005

This thesis for the M.S. of Computer Science degree by

Frank E. Watson

has been approved for the

Department of Computer Science

by

C. Edward Chow, Chair

Jugal K. Kalita

Sudhanshu K. Semwal

Date

Watson, Frank E. (M.S., Computer Science)

Enhance TCP Performance with Multiple Path Routing

Thesis directed by: Professor C. Edward Chow

Latency caused by bottlenecks with in a network can hamper any task regardless of the severity or resources alloted. This thesis talks about a multi-path routing technique to increase bandwidth regardless of a bottleneck. This is done inside the Linux Kernel using IP Tunneling. IP Tunnel is used to mask the route to an intermediate server and the Linux kernel can be modified to alternate and control the IP Tunnel taken automatically without intervention of the user. Taking multiple-routes spreads the hit of a bottleneck but also increases the likelihood of out-of-order packets and duplicate acknowledgment lowering the TCP window size. This thesis also proposes an internal connection-based buffer to solve out of order packets.

Dedication

To Dave Lohman, the Unix system administrator for EAS, for inspiring and helping with a facination with the Linux operation system.

Acknowledgments

For years, I had good people who helped me finish my thesis. If it was not for their help, I would have never finished.

The first on my list is Yu Chi for all his help with IP Tunnels and the Linux Kernel code. I hope in the next few months, I can return a portion of how much you helped me with my thesis.

Thank you Dave Havatin for helping me with learning Lyx and LaTeX and leaving me a copy of your disertation in Lyx. I would not of been able to figure out Lyx and LaTeX otherwise.

Thanks to my Dad and Hwan for letting me live with them for over 8 years while I finished school. Then Dad turning around and virtually buying me a house. Thanks Dad and Hwan!

Thanks also to my second family at Circle Drive Baptist Church where I have grown up these last fifteen years.

Thanks to Dave Nixon at work for letting me take many hours (191 hours of vacation last year) to finish my tests and work on my thesis. Without your understand and tollerance towards my school work, I would still be working on my thesis.

Obviously, I have to thank my advisor. Thank you Dr. Chow for the many encouraging words and the pushes in the right direction when I repeatly hit brick walls.

Thanks to Jim Martin, Dr. Kalita, Dr. Semwal and the rest of EAS faculty and Staff who have been my friends, teachers, and mentors over the last 10 years.

Finally but not least, thanks to my sweetheart and fianc'ee, Kelly. I look forward to being married and spending the rest of my life with you.

Galatians 6:9; Proverbs 3:5

Psalm 115:1

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The following are the objectives of this thesis.

- Develope and study multiple path routing and how it increases bandwidth using intermediate proxies.
- Further improve the multile path routing by developing and studying the effects of a TCP connection based buffering which sorts and filters incoming packets before being processed by the Linux source code.
- To introduce User Mode Linux and show its application for kernel development.
- To study and explain the Linux TCP/IP implementation for the 2.4 kernel series for further research and Kernel development.

The problem of latency in bandwidth during critical operations has been a problem of recent interest.

This thesis builds on the idea of multiple path routing[1] and shows how having multiple connections can be utilized to increase overall bandwidth for a connection through modifications within the Linux kernel. It also shows how with multiple connections weights can be applied to further optimize bandwidth.

1.1 Goals and Modivation for User Mode Linux

Chandra Prakash and Weihorng Wang both worked on the Linux kernel code. Their continual struggles with working on the host machines code propelled the question of an easier way of studing and working with the Linux Kernel.

During Dr. Chow's Computer Communcation class (CS 522), a study on different virtual machines was done by Jeff Rupp and the writer of this thesis. Many virtual machines were studied and a proposed spam black list embedded in the kernel was proposed and developed using User Mode Linux.

User Mode Linux does not have much literature on its setup and configuration. Once configured correctly, User Mode Linux has the ability to create a Linux Virtual Machine. With the User Mode Linux's Virtual Machine, the GDB debugger can be attached to trace through the Linux source code. This debugging feature is currently unque to User Mode Linux. Figuring out how User Mode Linux worked gave researchers a powerful tool for easier understanding and modifying of the Linux Kernel.

This thesis has years worth of work with User Mode Linux and how the networking section of the Linux Kernel operates. The writer hopes this thesis will be a controbution to further Linux kernel development.

1.2 Goals and Motivation for Multiple Path Routing

A scenario was proposed to the academic community last summer of military commanders needing a high resolution photo of a combat area in five minutes. Restraints for the proposal were: the TCP/IP protocol had to be used and no modifications operating system of the intermediate computers. This prompted the idea of using the Linux Kernel to combine the bandwidth of multiple connections. These modifications were only to the client and server machines and used TCP/IP and IP Tunneling.

Methods for increasing bandwidth is a contribution with many applications. This thesis is a stepping stone for other researchers to take the validation of these results to further expand solutions to bandwidth scenario above.

Chapter 2

Techniques for Vitrual Machines

User Mode Linux[2], abbreviated UML, is a part of the sourceforge repository and is a patch to the Linux source which allows a Linux kernel to run at the application level. This thesis project revolves around the use of UML (User Mode Linux) by taking advantage the Linux kernel running at the application level. User Mode Linux also allowed multiple sessions to run on the application layer with both graphical interfaces and networking capabilities.

This section and Appendix A give a general overview of UML (User Mode Linux) then progresses to the more technical issues of how UML runs, networks together, and builds an executable from the the Linux kernel. The next section will talk about what is the application layer.

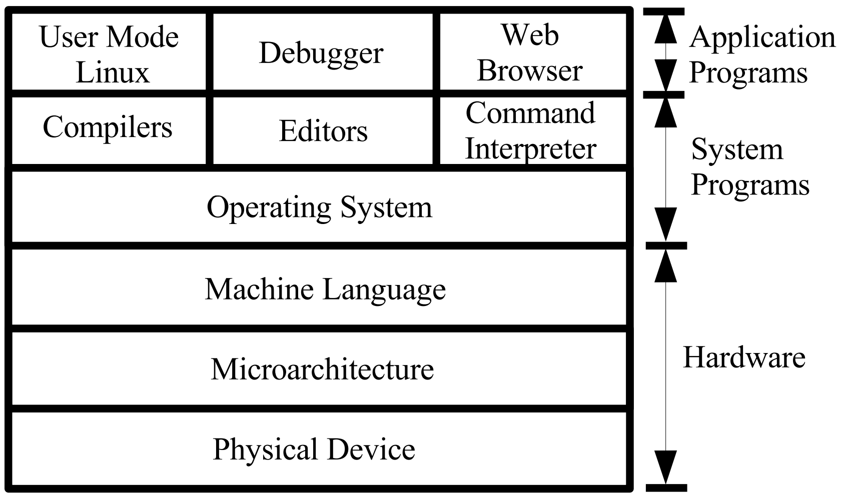


Fig. 2.1. Computer system layers

A computer system consists of three major levels, as shown in Fig. 2.1, are: hardware, system programs, and application programs (or application level)[3].

The significance of running the Linux kernel at the application level is two fold: allows debugging and lowers the risk of damage to the host machine's operating system. Debugging capabilities is a big plus over printk statements (same as printf in C/C++ but at the kernel level). This thesis project goes into the network packets storage structure, called the sk_buff. With debugging, an attached GDB session (or other debugging session) can watch the sk_buff for changes and make observations to packets as they are received. One of the main goals of this thesis is to create multipath router. Using a debugger is helpful in achieving this goal.

Degugging the Linux Kernel is difficult without the infrastructure of an operating system separate from the one currently being debugged. In the current literature, there are only two debuggers mentioned. This includes User Mode Linux and another debugging tool called kgdb. They both use two different methods of debugging.

2.1 Kgdb overview

Kgdb[4] is not the subject of this thesis but deserve some mention. It is a modified version of the popular debugging tool from the free software foundation, gdb. Kgdb involves using two computers through a PPP serial port connection. Kgdb uses the observing operating system as the infrastructure to run Kgdb. The other machine serves as the debugging kernel relays debug information through the PPP serial port connection as shown in Fig. 2.2.

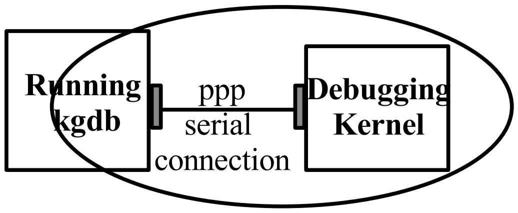


Fig. 2.2. KGdb

While this sounds easier than the User Mode Linux setup, there are set backs like: the use of two machines and how long the debugging machine takes to recover from a kernel crash. Initially looking at the two different options, the needed resources and recovery time is what placed User

Mode Linux as the likely development candidate. Also, User Mode Linux makes it possible to create a cluster of User Mode Linux clusters for network testing on one machine.

2.2 User Mode Linux Overview

UML (acronyms used for User Mode Linux) is ran at the application level. So it can be debugged and does not take the host kernel down if the kernel modifications cause the virtual system to crash. UML is setup in two parts and can have multiple sessions running and has a virtual networking environment to transfer packs between other UML sessions or the host machine. For researchers with a limited amount of machinery the low overhead is extremely helpful. To compile the kernel for a UML takes about a fourth of the time to build compared to a regular kernel.

All of the Linux distributions (Redhat, Slackware, Mandrake and others) include extra kernel options, so a distribution works for all types of machines(Dell's, Hp's, i386's, SUN's, Itaniums, and etc). All the different options are not needed by the kernel nor operating system but is included in kernel by default. Even if the researcher compiled the Linux kernel with no modules and turned on the options needed and created a fast and optimized kernel, the UML kernel still compiles faster. The UML kernel does not have a hardware architecture, everything is done in emulated hardware. For example, the UML kernel relies the host kernel's routing table through a software interface driver (e.g., Tuntap) and does not need an Ethernet driver for networking. UML uses a virtual hard drive. UML does not use hardware (e.g., CD Rom or any other hardware perpetuals). The lack of hardware drivers cuts the compile time down and creates a secilized kernel for only one set of device drivers.

2.2.1 Background

There are two major pieces to the UML architecture as shown in Fig. 2.3 the figure below. This next section will start by explaining the two different components and then describe their significance.

2.2.2 The Root File System: root_fs

First there is the root_fs, which stands for the "root file system". The root_fs is a hard drive image containing the operating system and the other system files. It consists of all the libraries, shells, and

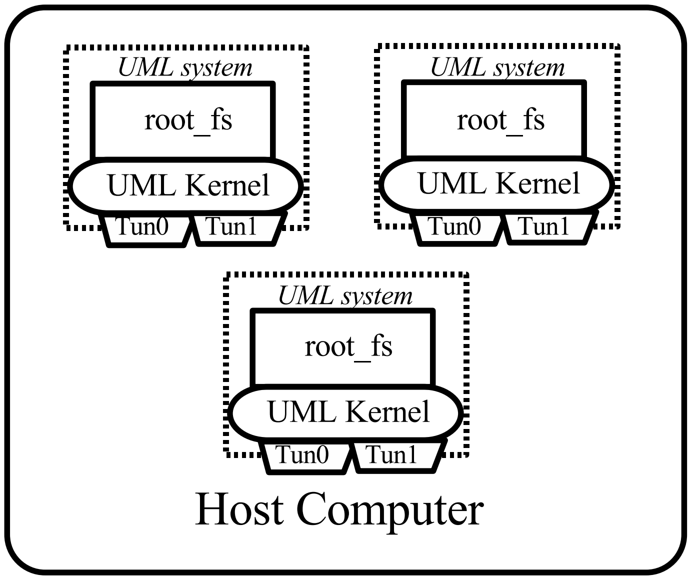


Fig. 2.3. UML architecture with tuntap driver attached to host

applications. At a practical level, the root_fs is a single file of an entire hard drive image with the exception of the Linux kernel (files in the /boot)[1]. Since a root_fs are entire file systems within a file, it is possible to mount to these images using the Unix "mount" command. There is a special option called -o loop which is needed, and can mounted like a device under a Linux platform. The command to mount a root_fs is in Fig. 2.4

```
mount -t <name of the file system> /mnt/floppy -o loop
```

Fig. 2.4. Command to mount a file system

Creating an root_fs is a tedious process and requires the CD's from a preferred Linux distribution. Bigger, more feature intensive root_fs take longer to load. Using a simple root_fs has great merit. For experiments, an older version of Slackware was used; Slackware with an UML kernel takes about one minute to load and has exactly the requirements needed.

[1] The root_fs is very similar to an "ISO image" which makes CD's. A number of software manufactures (Redhat, Sun, and Microsoft) allow developers and system administrators to download these "iso images". An "iso image" is an image byte for byte of a CD Rom disk; all the information is placed into a single file following the iso9660 file format (the file format of CD Roms).

2.2.3 UML Kernel Executable

This root_fs interfaces with the second part of the UML, the Linux executable. What the UML project has done is clever. They have kept kernel development at the front of their design and created the Linux kernel as a stand alone executable which interfaces with the root_fs to create a Linux operating system at the application level. This Linux kernel executable is compiled from a patched Linux source code.

What needs to be understood, the Linux kernel executable interfaces with the root file system. Command line parameters tell the Linux kernel executable how to network interface with the host, how to toggle the debugger on or off (if the skas patch is not installed), how the xwindows on the virtual machines forwards the display and any other hardware or software parameter relating to the host machine.

This Linux kernel is not difficult to compile. The compile time is also a fifth of the normal default compile time. There are a host of advantages for having the UML session partitioned in two sections (Linux executable and root_fs). The UML Linux kernel executable is on average 8 Megabytes. The vmlinuz (virtual machine Linux on the actual Linux host machine) is about 1.4 megs, generally smaller than a disk. The Linux UML executable is bigger, because the UML code is compiled with debug symbols and has all of the virtual device drivers built into the UML kernel.

2.2.4 How UML Could Be Used

User Mode Linux could be used in education for Kernel development and for showing network security concepts. UML is currently used in Dr. Edward Chow's CS 522, Networking Communication, to do routing problems and demonstrates networking concepts without much overhead and security risks. In more advanced classes, the Linux kernel and provide an in depth look at an operating system.

The goal of this section and Appendix A and B is for easier research involving kernel development. Breaking through the complex set up for UML, could be a priceless tool saving both resources and time.

Chapter 3

Technique for Multiple Path Routing

This section represents the discoveries using the User Mode Linux frame work. The main goal is to set up the Linux kernel to do multi-path routing, where the kernel alternates the intermediate destination of the packet but has the same final destination. The Intermediate computer (proxies or gateways) sends the packet to the correct destination. Bottom line, setting up a system where there are multiple paths to the destination.

Within the Linux kernel, the multi-path enhancements are able to send alternate the packets to different intermediate destinations. For example in Fig. 3.1, 10 packets are in a simple HTTP request: six packets send from the client and four return packets from the server. If there are three gateways, the first gateway receives two packets (packets one and seven); second gateway receives two packets (packets three and eight) and; the third receives the remaining two packets (packets four and ten).

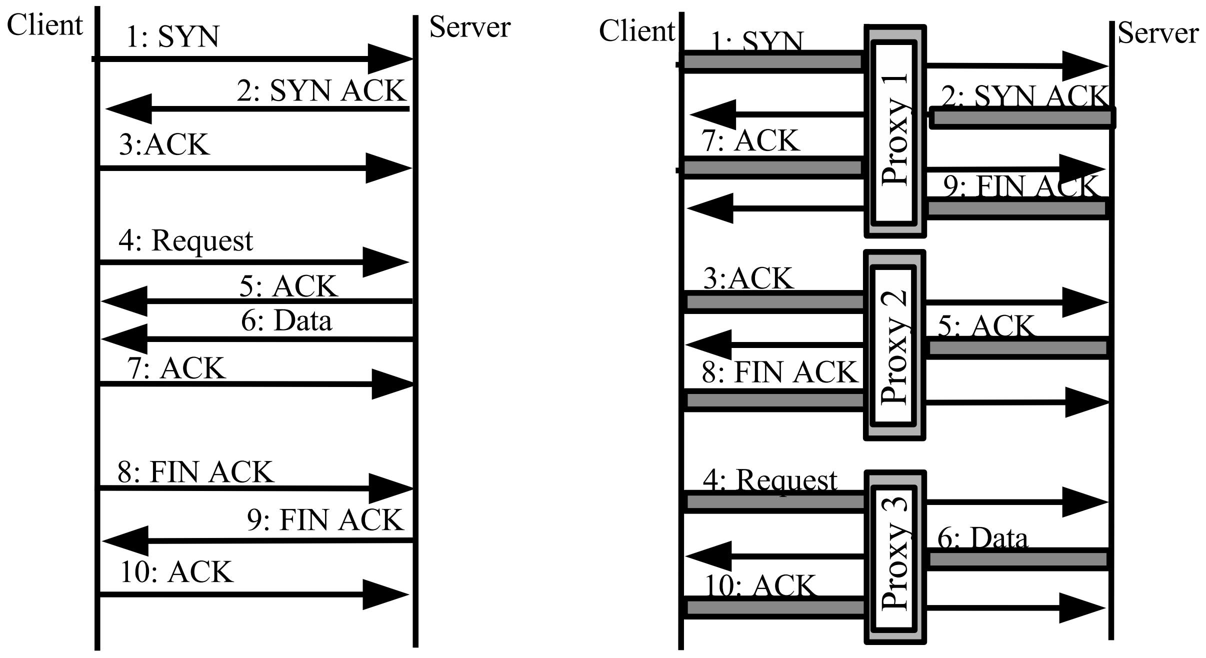


Fig. 3.1. Simple web request (left) and using multi-path routing (right)

The ten packets transmitted are spread over the three proxies. This is multi-path routing. Multi-path routing the ability to spread the packet load between multiple intermediate proxies or gateways so that the destination server does not know about the intermediate gateways or routers. Fig. 3.2 is an illustration of Multi-path routing.

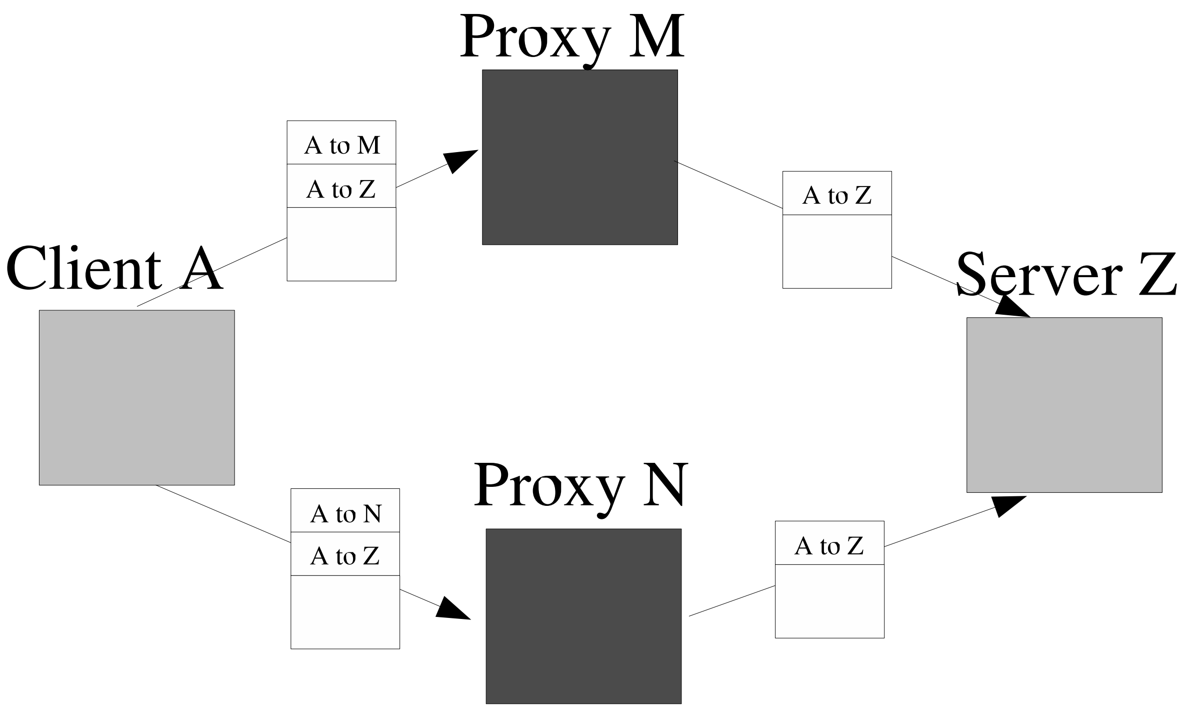


Fig. 3.2. Small network with the first IP header removed after proxy

To follow all the standards for IP (Internet Protocol), the multi-path modifications to the IP header are encapsulated in another IP header. This technique of having two IP headers is called IP tunneling. Fig. 3.3 shows a modified IP tunnel header used to route the packets.

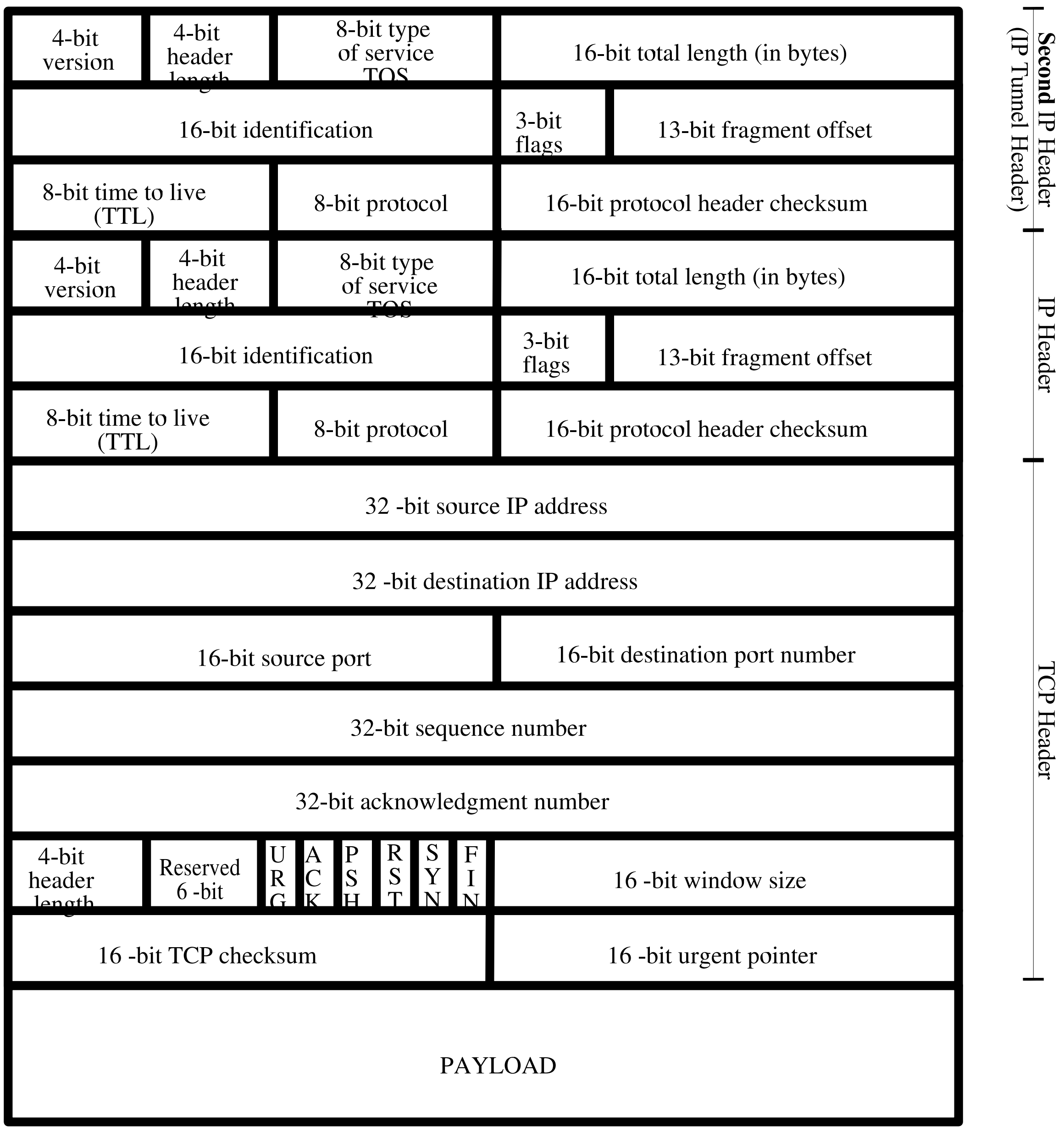


Fig. 3.3. IP tunnel packet header

Four modifications were made to the individual Linux machines and to the actual kernel to achieve multi-path routing. The changes include: creating an IP tunnels to interface with the kernel changes, modifying to the destination entry in the Linux kernel, and packet buffering the incoming TCP packets.

3.1 Iproute2 to create IP tunnels

IP Tunneling is taking the packet and wrapping it with an additional IP Header, so the packet has two IP Headers. Generally the first IP header goes to the first packet destination; the encapsulated header is the final destination.

The main advantage to IP tunneling is hiding the packet's intermediate sender. For this project when the packet is received by an intermediate computer (proxy or gateway), the outer IP header is removed. The intermediate computer removes the first IP header and forwards the packet to the destination (according to the remaining IP header). The remaining packet is repackaged with a new Ethernet frame and sent.

Creating IP tunnels is not trivial, at first. IP Tunnels needs to be turned on in the kernel. The option is under the "network option" in the config menu (or .config file). If the IP Tunnel option is not turned on, an error messages like the ones shown in Fig. C.3 will appear. If this error appears, check the kernel options and make sure that IP Tunnel is selected. Most Linux distributions have IP tunnel turned on as a module.

```
ioctl: No such device
SIOCSIFADDR: No such device
tunl1: unknown interface: No such device
SIOCSIFNETMASK: No such device
SIOCSIFDSTADDR: No such device
```

Fig. 3.4. IP tunnel error messages

There are three commands used to setup IP Tunnel[1]: “ifconfig”, “ip route” and “ip tunnel”. “ip route” and “ip tunnel” are part of the iproute2 suite. The iproute2 suite of utilities was designed to replace the “route” command. Many technical websites state conversion is going along quick because the adaption of the new “ip route”. The “route” command has the routing information laid out differently and using “ip route” takes some adjustments.

```
> ip tunnel add tunl1 mode ipip remote 172.31.0.172 dev eth0
> ifconfig tunl1 172.31.0.169 netmask 255.255.255.255 pointopoint 172.31.0.172 up
```

Fig. 3.5. IP tunnel commands

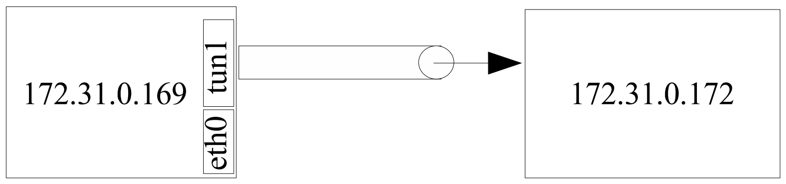


Fig. 3.6. Two computers establishing a tunnel

The commands in Fig. 3.5 are the commands needed to setup an IP Tunnel between 172.31.0.169 (client) and 172.31.0.172 (proxy or gateway). This establishes the tunnel and creates a device driver. Packets are encapsulated with an additional IP header (as shown in Fig. 3.6). The routing table is updated to inform the kernel where packets are heading and if the packets need to be routed through a tunnel. Fig. 3.7 is the command to update the routing table to use the IP tunnel.

```
> ip route add 172.31.0.197 dev tunl1
```

Fig. 3.7. Command to update routing table for ip tunnelling

3.2 Changing the Linux network device driver

Fig. 3.2 shows how the packets are sent to an intermediate proxy (or gateway). The standard direct connections go through the device eth0; the packets destined for an intermediate machine through an IP tunnel are sent to device tunl1. The multipath routing takes advantage of using the INET's underlying device drivers. The modifications to the INET code alternates the IP tunnel addresses. To make the multi-path scheme work, each outgoing packet is sent to different tunnel device (e.g., tunl1, tunl2, tunl3, tunl4, and tunl5). Selecting which packet goes to which tunnel device is done in the Linux kernel in the code handling the IP header[5, 6].

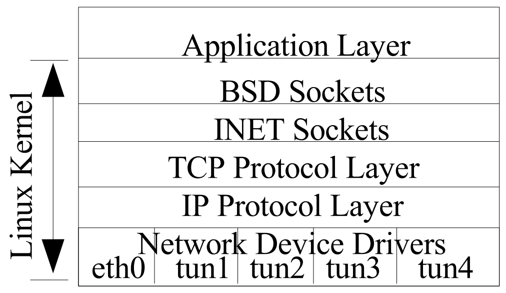


Fig. 3.8. Linux kernel with multiple network devices

The Linux kernel receives a packet and uses pointers to the headers, and payload stored in a structure called sk_buff. There is a socket for each connection called an INET socket and another overseer type socket, called a BSD socket as shown Fig 3.8. The differences between the INET and BSD socket are talked about in detail in Appendix D. The INET socket which is responsible for the connection and has routing information for the IP and Ethernet headers. This information is stored in a structure called the dst_entry.

The dst_entry is complex. It is part of a bigger structure, called rtable as shown in Fig 3.9. Rtable stands for “routing tables”. These routing tables structures work with the computer’s routing table to communicate the packet’s destination and correct header and device information. Each time a packet is sent, the INET socket’s dst_entry is consulted to determine the correct header information. The INET socket also contains the engine maintaining the SEQ and ACK numbers for the connection stored in the tp_pinfo structure. The dst_entry’s settings for a direct connection uses the eth0 (Ethernet) network device. And for an IP Tunnel, the dst_entry uses the tunl (IP Tunnel) network device.

Since the dst_entry is so complex, the kernel modifications relies on values being set in the routing tables. Actual modifications are not done to the dst_entry. If the destination of the packet is not routed through the tunnel device in the routing table and the multi-path routing kernel modifications are activated, the kernel will try to route a packet with a dst_entry set for a direct connect oppose to IP tunnel. This will confuse the kernel and cause it to crash. The information from the routing tables stored in the dst_entry and the way the kernel uses this information should match.

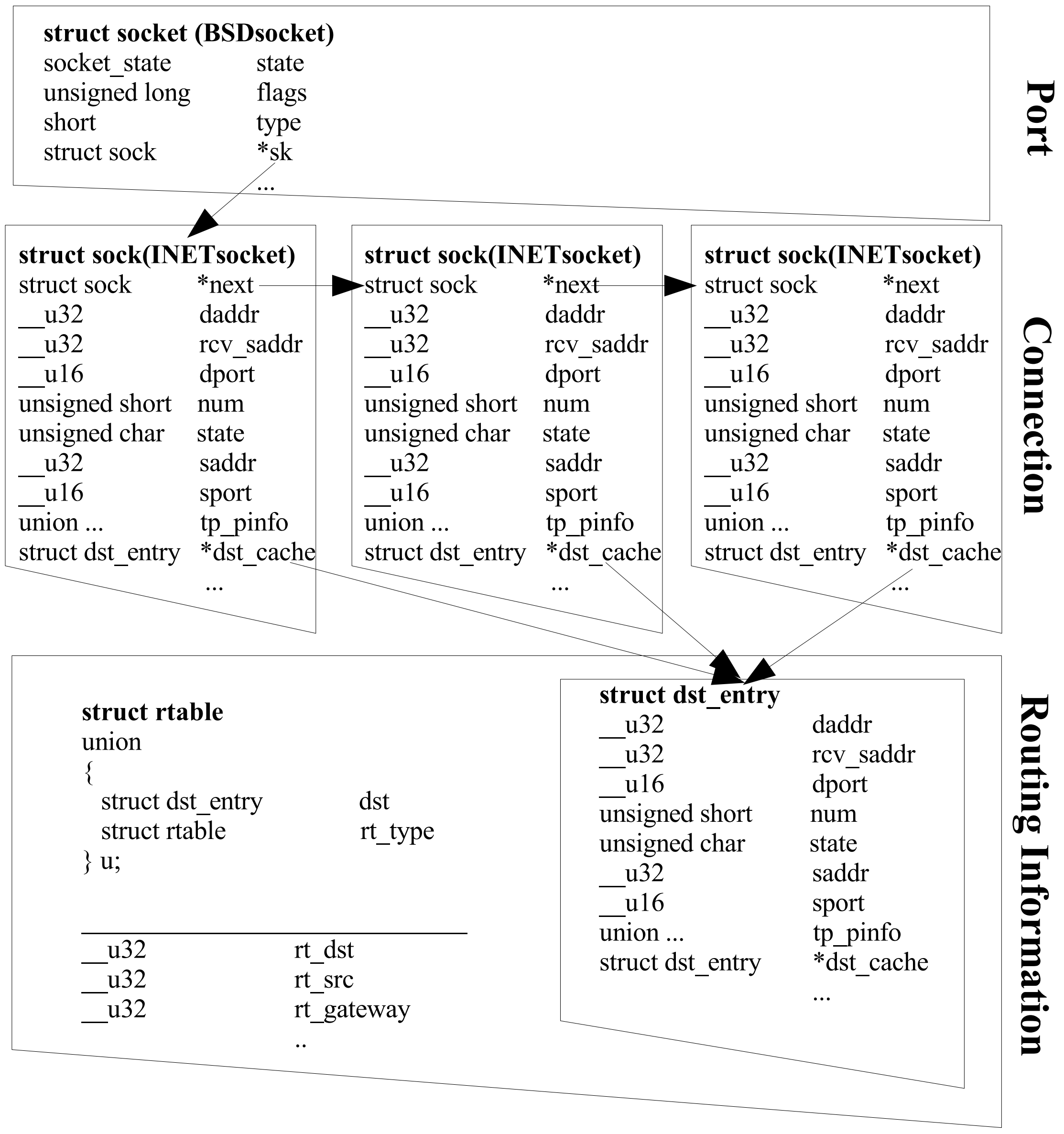


Fig. 3.9. TCP routing and socket structure

3.3 Packet buffering

The Linux kernel stores outbound packets in a queue before flushing the network device. Multipath routing uses multiple network devices and causes packets to arrive out of order. If one or multiple packets get delayed and the packet arrive out of order, the TCP protocol sends another acknowledgment requesting a retransmission of the delayed packet. If this happens three times, the packet window is set to zero. This poses a problem and is a major hit on performance.

A solution to out of order packets is to queue the arriving packets and wait for the delayed packet. This packet buffering is done using the INET socket. The INET socket is connection based, so there is one INET socket per connection and is a structure containing information about the connection (e.g,. sequence number).

When the packet is queued, it is placed on the buffer in order of the sequence number (earliest sequence number are closest to the head) and packets with duplicate sequence numbers are removed. When the packet with the correct sequence number arrives, the packets are fed to the TCP processing code, one packet at a time. The INET code stores the next expected sequence number in the ip_option. If the next packet on the queue does not match what is expected, the buffering continues.

If the packet is lost, the expected packet will never arrive. There are two solutions for missing packet. The TCP timer will time out and send another acknowledgment requesting the packet. Another solution is limit the size of the buffer. If the buffer size becomes greater than a given threshold, the packets are processed as though the buffer did not exist and the TCP engine would send another acknowledgment.

To control the size of the buffer and various parameters of the multi-path routing in the kernel, the proc file system is used to communicate with the kernel from the application level.

3.4 Utilizing the Proc File System

The Linux proc file system has one purpose, to communicate information between the Kernel and the user to fine tune the Kernel's performance. The proc file system is not stored on the hard drive but is entirely in memory and starts and disappears with the Kernel. At startup the kernel populates the proc file system and mounts it to a directory for the user and applications to have access. For example, the IP routing table use the proc to communicate the gateway and dev information needed to route incoming and outgoing packets properly. When the command "route" is executed, the ârouteâ program displays a file in the proc file system, /proc/net/route, to the screen. The route command just makes the contents of this file look more eye appeasing to the user.

This project uses the proc file system[7] to turn off and on the multipath routing and control other variables used for scold loging and buffering. Within the Linux code, a proc directory can be created in the /proc using the command “proc_mkdir.” Fig. 3.10 is the code to make a proc directory and populate an entry.

```
#define MAXDATALEN 20
char bufferOnVal[MAXDATALEN + 1];
static int initScoldProc(void){

  // creates the directory in the proc file system under /proc/sys/net/ipv4/mulipath
  multiPathDir = proc_mkdir("sys/net/ipv4/multipath", NULL);
  // creates a proc entry /proc/sys/net/ipv4/multpath/bufferOn
  bufferOnProc = create_proc_entry("bufferOn", 0644, multiPathDir);
  // sets the proc struct to use the following functions read, and write
  // and sets the data value to char array, bufferOnVal.
  bufferOnProc->data = bufferOnVal;
  bufferOnProc->read_proc = procReadBufferOn;
  bufferOnProc->write_proc = procWriteBufferOn;
}
```

Fig. 3.10. Kernel code for a proc file system entry

Towards the end, pointers to the functions “procReadBufferOn” and “procWriteBufferOn” are set the “read_proc” and “write_proc.” These methods are shown in the next frame. The file system has an API for developers to use the proc file system. Fig. 3.11 shows the two functions pass in for read and write. These functions read in a pointer “data” and and type casts the value so the user can easily have read and write access to the data.

```
static int procWriteBufferOn(struct file *file, const char *buffer, unsigned long count, void *data)
{
 int len; char * inData = (char*) data;
 if (count > MAXDATALEN) {
 len = MAXDATALEN;
 } else{
  len = count;
 }
 // retrieve data from user
 if (copy_from_user(inData, buffer, len)) {
  return -EFAULT;
 }
 // write data to variable
 inData[len] = '\0';
 sscanf(inData, "%d", &bufferOn);
 //printk (KERN_CRIT "bufferOn: %d\n", bufferOn);
 return len;
}

static int procReadBufferOn(char *page, char ** start, off_t off, int count, int *eof, void *data){
 int len;
 char *outData = (char *) data;
 len = sprintf (page, "%s", outData);
 //printk(KERN_CRIT "Read bufferOn %d", bufferOn); return len;
}
```

Fig. 3.11. Kernel code for a proc read and write methods

To use the proc values values at the application layer, a “cat” command can show the contents of a variable. The proc value can be written to using “echo” with a redirection. Two examples are shown in Fig. 3.12

```
root@walrus:/proc/sys/net/ipv4/multipath# ls
bufferOn bufferSize
root@walrus:/proc/sys/net/ipv4/multipath# cat bufferOn
0
root@walrus:/proc/sys/net/ipv4/multipath# echo "1" > bufferOn
root@walrus:/proc/sys/net/ipv4/multipath# cat bufferOn
1
```

Fig. 3.12. Setting proc values

3.5 Testing using TC bandwidth limiting

Rate limiting is a condition needed to test the multi-path routing. Tc is a tool used for maintaining traffic[8]; it is an interface to the IP chains in the kernel. Of the different types of command line interfaces to IP chains, and tc seems to be the easiest to set up. The tc utility can be downloaded from the maintainers' website[9] and is included in newer version of Iproute2 (the software used to setup the IP Tunnels)[10].

The tc utility works per device. Each device can be assigned a total bandwidth. The bandwidth can also be partition between different outgoing IP address using qdiscs. For example, eth0 (default Ethernet device driver) is set to two different destinations 128.198.60.172 and 128.198.60.173. The total bandwidth for eth0 is 1024 kb/s. The total bandwidth can be partitioned into a qdisc based on destination address. For this example, there is a one to two ratio used for the two classes. So the qdisc with packets outbound to 128.198.60.172 (341 kbit/sec) has twice bandwidth as the qdisc with packets bounded to 128.198.60.173 (683 kbit/sec).

```
tc qdisc add dev eth0 root handle 1: htb default 12
tc class add dev eth0 parent 1: classid 1:1 htb rate $maxSpeed ceil $maxSpeed
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 341kbps speed ceil 1024kbps speed
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 683kbps speed ceil 1024kbps speed
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 1024kbps speed ceil 1024kbps speed
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 match ip dst 128.198.60.172 ip flowid 1:10
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 match ip dst 128.198.60.173 ip flowid 1:11
```

Fig. 3.13. Setting up tc qdisc to limit bandwidth

The Fig. 3.13 shows the commands to set up the rate limiting using tc's qdiscs. There are three sets of commands: set up the root for the eth0, set up each class, and setup each filter.

Finally there are additional kernel requirements when setting up the qc utility. Accroding the tc utilities' website[9], the following QoS (Quality of Service) kernel options need to be set. These QoS kernel options are listed in Fig. 3.14

```
#
# QoS and/or fair
queueing
#
CONFIG_NET_SCHED=y
CONFIG_NET_SCH_CBQ=m
CONFIG_NET_SCH_HTB=m
CONFIG_NET_SCH_CSZ=m
CONFIG_NET_SCH_PRIO=m
CONFIG_NET_SCH_RED=m
CONFIG_NET_SCH_SFQ=m
CONFIG_NET_SCH_TEQL=m
CONFIG_NET_SCH_TBF=m
CONFIG_NET_SCH_GRED=m
CONFIG_NET_SCH_DSMARK=m
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_QOS=y
CONFIG_NET_ESTIMATOR=y
CONFIG_NET_CLS=y
CONFIG_NET_CLS_TCINDEX=m
CONFIG_NET_CLS_ROUTE4=m
CONFIG_NET_CLS_ROUTE=y
CONFIG_NET_CLS_FW=m
CONFIG_NET_CLS_U32=m
CONFIG_NET_CLS_RSVP=m
CONFIG_NET_CLS_RSVP6=m
CONFIG_NET_CLS_POLICE=y
```

Fig. 3.14. Kernel options needed for tc to run

Chapter 4

Performance Analysis

Hundreds of tests were done using a socket connection Perl script. The Perl script would establish a connection and transmit configuration parameters for both the client and server. The script would wait a given time to verify the settings were accepted on the server and use apache bench as a web benchmark. Apache bench contains statistical information. For many tests, the webpage was retrieved up to 5 times and if the mean and median were not within a given threshold, the test would be invalidated.

These results were collected and verified. The results below are of the average bandwidth given in kilobytes per second. A diagram of the setup will help visualize the test scenario. The computers in the testbed and used for these results were 10 machines (with the same configuration) running Linux Fedora Core One with 600 MHz Pentium III processors and 256 megabytes of memory with a modified 2.4.24 Linux kernel. Kernel modifications were in the INET network implementation, and additional proc values were added to enable access to these kernel values.

4.1 Baseline results

These are the baseline results with no bandwidth limiting, weighting different paths, or buffering incoming packets. The remainder of the chapter tests deals with these different scenarios. These results are straight baseline results. Below in Fig 4.1 shows the four different types of connections.

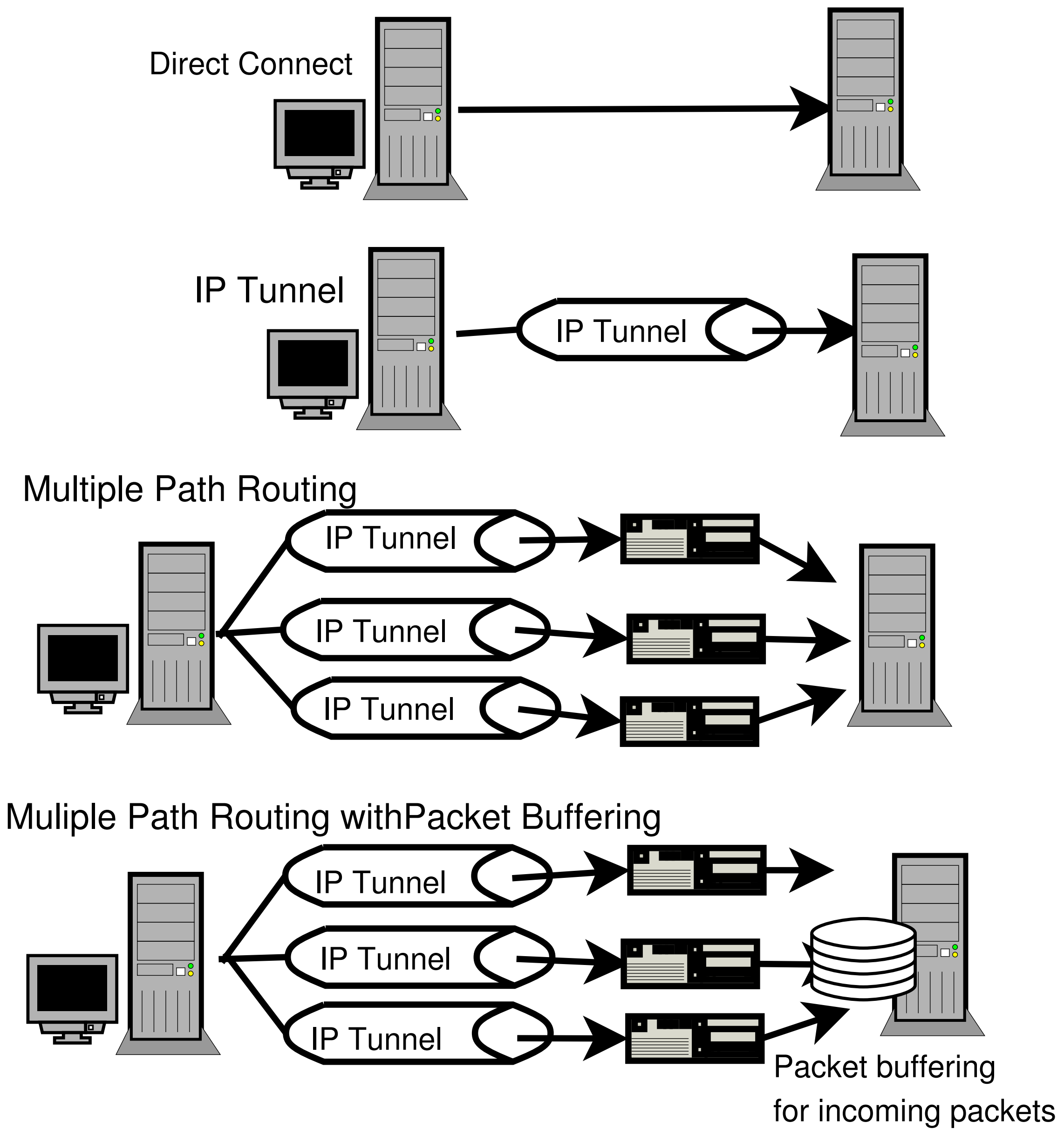


Fig. 4.1. Types of connections

Direct connection is a connection without any modifications. IP Tunnel is a connection using packet with two IP headers. The first header goes to the intermediate proxy (or gateway) and is stripped off. The second header contains the destination's IP address and is routed to the destination by the intermediate proxy.

Multiple path routing uses IP tunneling and kernel modifications to switch the intermediate server. Multiple Path Routing with Packet Buffering has a buffer before the TCP processing kernel code. This buffer sorts out of sequence packets.

Below are the results both in Table 4.4 and Fig. 4.2 displaying the control bandwidth for the different types of connections.

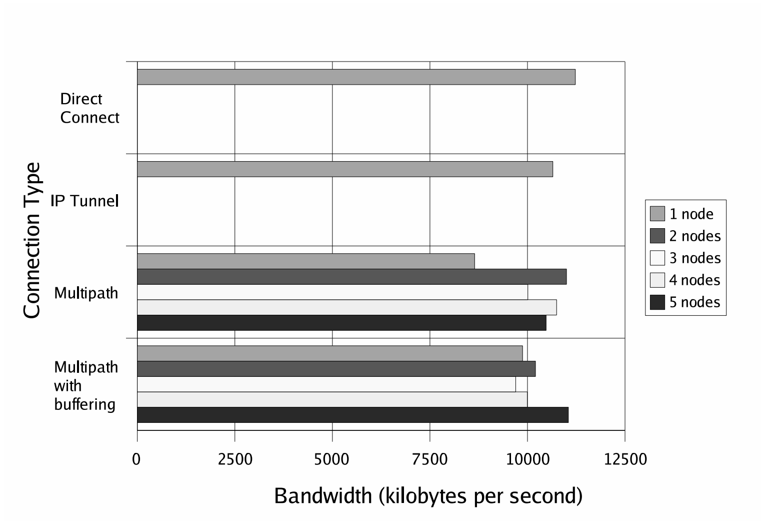


Fig. 4.2. Baseline results

The multiple path tests are done over five intermediate servers (or nodes). Since there is no kernel code modifications for the Direct Connection and IP Tunnel connections only one node is listed. This is to represent no intermediate node for these two connections.

	0 or 1 node	2 nodes	3 nodes	4 nodes	5 nodes
Direct Connect	11228.4				
Direct Connect (Other Compute)	11398.82				
IP Tunnel	10651.68				
IP Tunnel (Other Computer)	11244.47				
Multipath	8645.98	11001.57	10010.8	10748.66	10478.11
Multipath with buffering	9875.86	10206.51	9699.83	9996.66	11049.27

Table 4.2. Baseline results

The Direct Connection and the IP Tunnel were also benchmarked on other machines to further test the control. These two additional results were added to the table but are not in the Fig. 4.2. These additional baseline tests were done on Pentium III's with 1 Gigahertz processors.

4.2 Bandwidth controlled scenario

The first scenario is to rate the overall connection speed to a set bandwidth and see how the result bandwidth between the server and the client is affected. The purpose of this test is to eliminate any additional factors assisting the TCP processing (e.g., high bandwidth allowing for quick recovery for out of sequence packets); it also provides a more real world scenario because long distance Internet connections generally do not have 100 megabit bandwidth. It least not at the time of the writing of this thesis.

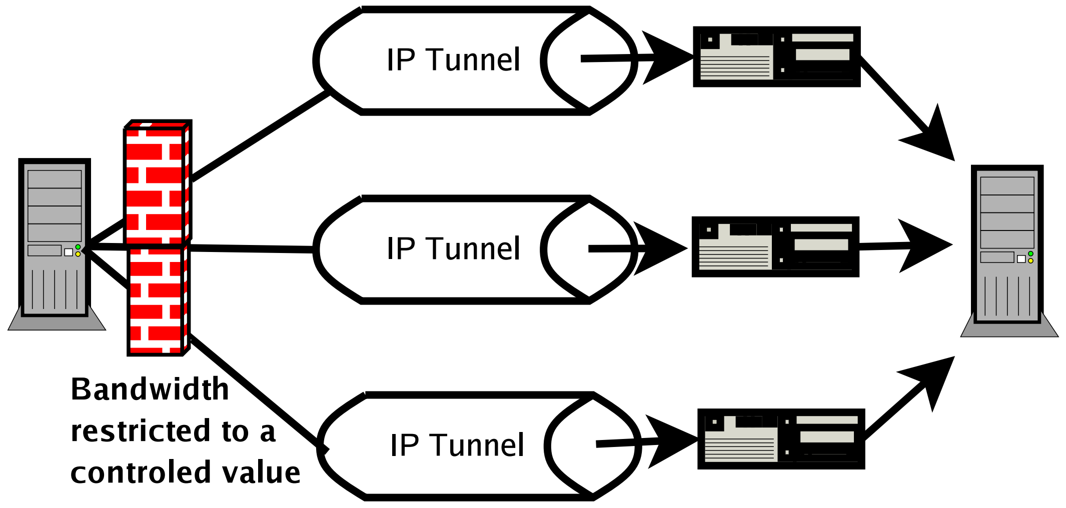


Fig. 4.3. Bandwidth limited scenario

The next set of graphs will be broken up based on connection. The first two graphs will not have any intermediate nodes. There are two sets of bandwidth in addition to the number of nodes. The bandwidth for each bar (located in the key) is the bandwidth the connection is bandwidth rate limited. The x-axis is the result bandwidth between the client and the server.

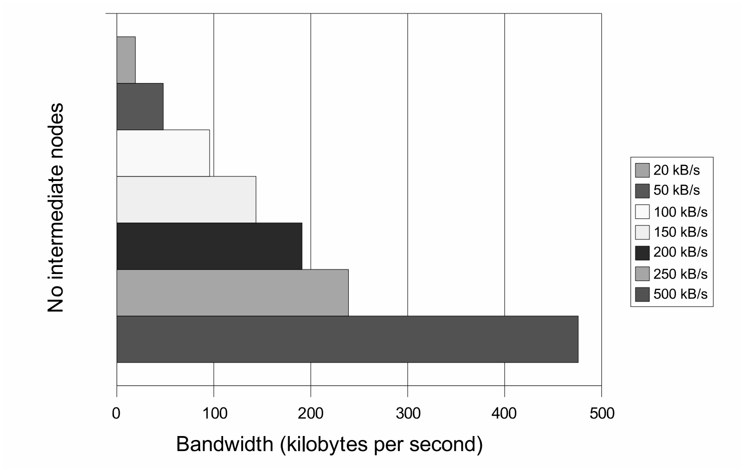


Fig. 4.4. Direct connection with bandwidth rate limiting

	20 kB/s	50	100	150	200	250	500
Direct Connect	19.16	47.89	95.68	143.57	191.1	238.93	475.91

Table 4.4. Direct connection with bandwidth rate limiting

For direct connection scenario baseline, the resultant bandwidth shows a coorlation to the bandwidth restrictions. For example, 19.16 kB/s is the resultant bandwidth for a restriction set at 20 kB/s. This verifies the restrictors works with a small devation of less than 9 percent.

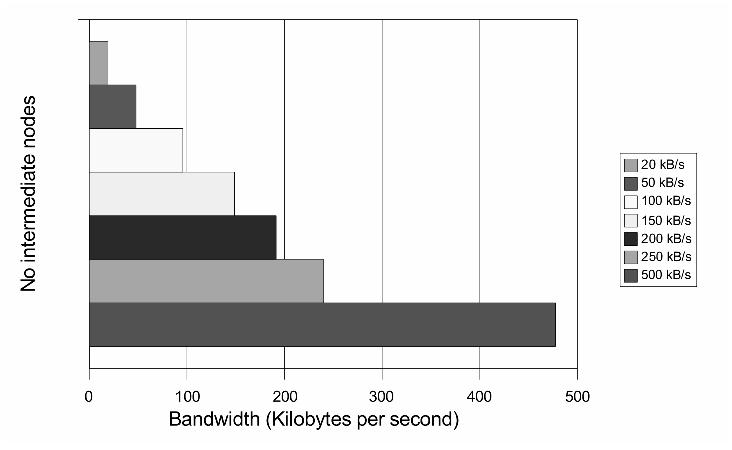


Fig. 4.5. IP Tunnel with bandwidth rate limiting

	20 kB/s	50	100	150	200	250	500
IP Tunnel	19.18	47.91	95.84	148.8	191.5	239.81	477.58

Table 4.6. IP Tunnel with bandwidth rate limiting

One observation shown in the results from the direction connection and applying IP Tunnel scenarios, IP Tunnel does not impose a noticable performance cost.

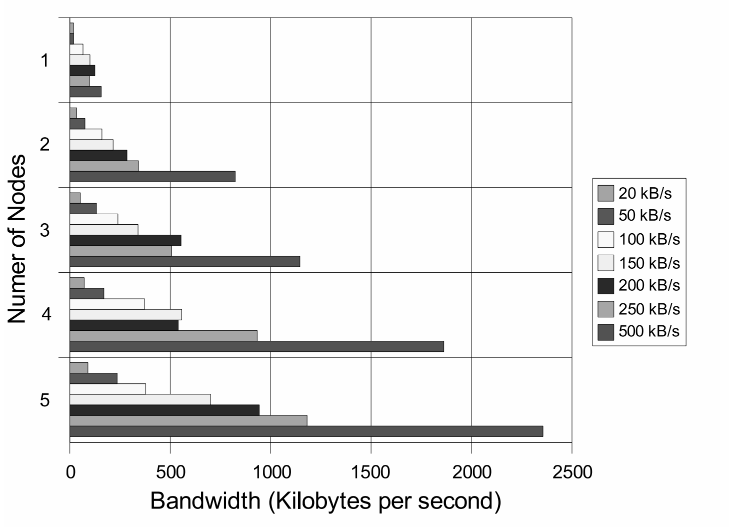


Fig. 4.6. Multiple path routing with bandwidth rate limiting

# of node	20 kB/s	50	100	150	200	250	500
1	18.59	18.94	65.78	100.45	124.91	98.13	156.43
2	34.52	75.72	160.09	215.88	284.61	342.22	823.66
3	52.62	132.81	239.66	339.72	553.93	507.35	1145.6
4	71.8	169.92	372.87	557.24	539.76	932.85	1861.99
5	90.27	235.85	378.31	700.82	943.65	1181.53	2355.65

Table 4.8. Multiple path routing with bandwidth rate limiting

One of the most important observations reported by this thesis is how the multiple path Linux code modifications utilaizes the other nodes. For example, notice how the bandwidth increase as additional nodes are added. In some cases the additonal node increases the bandwidth by a multiple of the number of nodes. One node is 18.59 k/Bs and three nodes is 52.62 kB/s, almost three times the bandwidth.

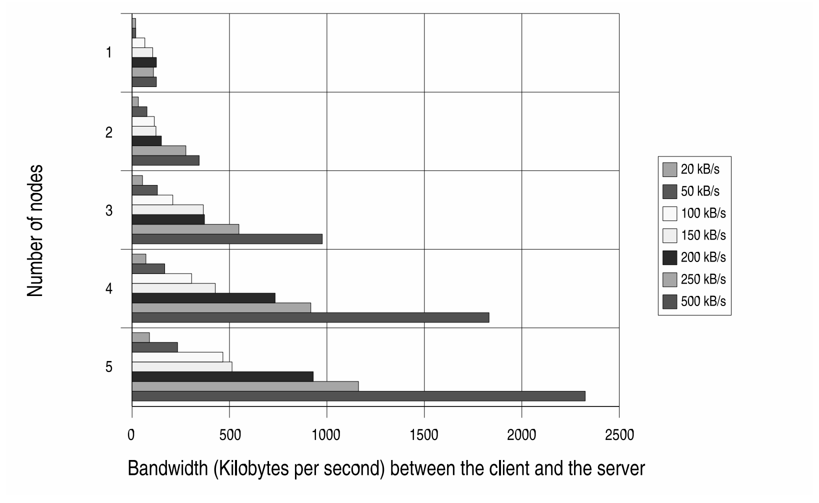


Fig. 4.7. Multiple path routing with buffer and bandwidth rate limiting

# of node	20 kB/s	50	100	150	200	250	500
1	18.59	19.29	65.75	105.93	124.91	110.03	124.86
2	34.25	76.65	114.26	122.87	150.46	276.04	344.46
3	53.6	129.88	209.03	365.87	371.56	548.52	976.49
4	70.97	167.49	305.79	427.14	733.86	917.5	1831.78
5	89.35	233.55	466.35	513.21	929.92	1162.14	2325.29

Table 4.10. Multiple path routing with buffer and bandwidth rate limiting

The data from the multiple path with a buffer was a baffle. The research team believed if a buffer was applied to the muliple kernel modifications the bandwidth would increase[11]. Multipath routing has a performance hit because of out of sequence packets. A solution to out of sequence packets was to create a buffer. The results in Fig. 4.6 and Fig. 4.7 show buffering based on the connection causes an additional overhead hit and lowers the bandwidth.

4.3 Weighted scenarios

These tests were done for two reasons: to see the effect of one connection having lower bandwidth and how applying a weight to the number of packets going to a given connection affects the result bandwidth. Kernel modifications allow redirection of packets and can allow a weight, a number of packets a destination receives, to assigned to each of the intermediate proxies. A weight of 1:6 means the first proxy receives one packet while the other proxies receive six packets of the seven packets sent. This weighted distribution is shown in Fig. 4.8.

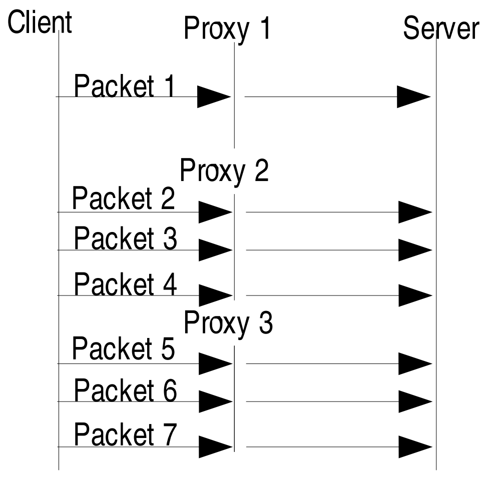


Fig. 4.8. Weighted packet distribution for a 1:6 ratio

The purpose for the weighted packet distribution is out of sequence packets. Multiple path routing causes a number of out of sequence packets; when a path to an intermediate proxy is a lower bandwidth then the other connections, out of sequence packets happen more frequently and is more of a real world problem. The idea of one path at a lower bandwidth than the other paths is illustrated in Fig. 4.9

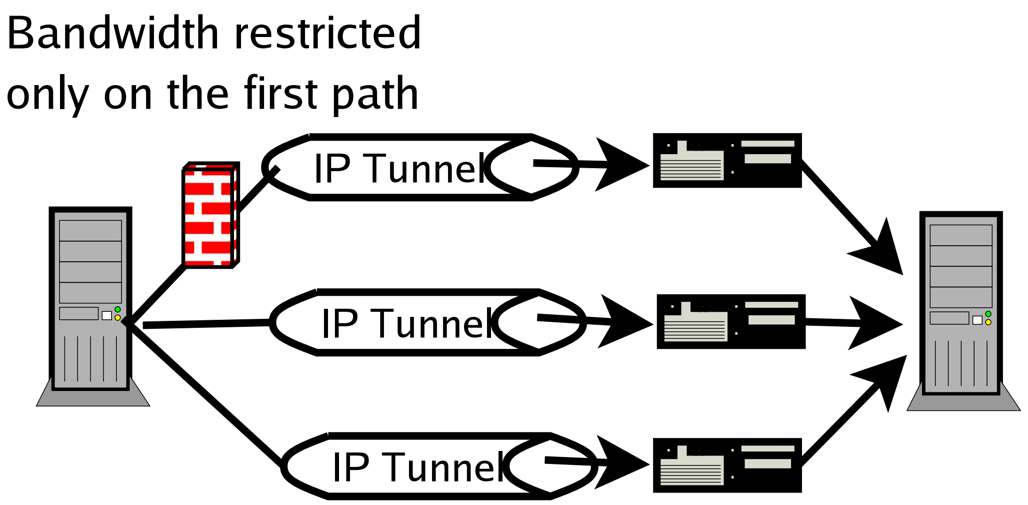


Fig. 4.9. One path with bandwidth lower then the other paths

4.3.1 Weighted paths and one path with small bandwidth

This scenario takes restricts the first path by a controlled bandwidth and applies a weight the number of packets per each path.

In the graphs, the first path is bandwidth controlled by the bandwidth in the key. The bandwidth on the x-axis is the result bandwidth between the client and server. The y-axis is the weight ratio for the packets going to each of the nodes. The first ratio value goes to the first node; remaining ratio is equally given to the remaining paths. For these tests, only two and five nodes were tested. A one node test would be a repeat from Tables 4.8 and 4.10. Since packet buffering is also being studied, the last two tests are with packet buffering.

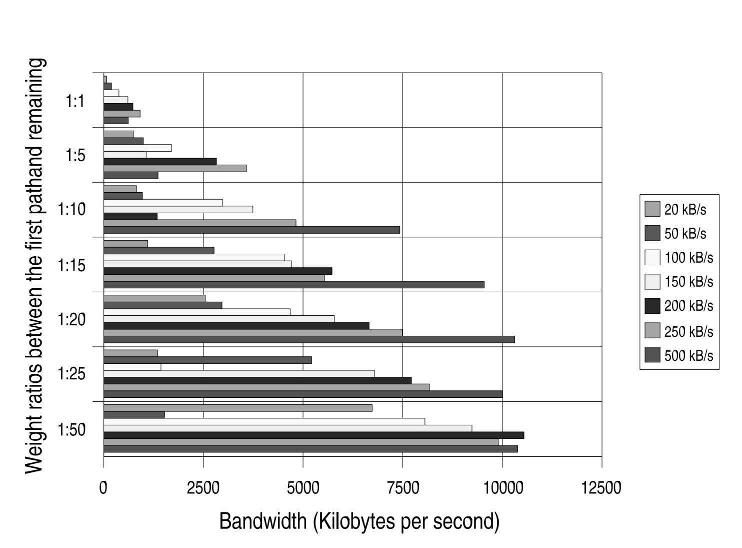


Fig. 4.10. Two nodes with no buffer and weighted packet distribution

Weight ratio	20 kB/s	50	100	150	200	250	500
1:1	74.73	194.04	379.29	605.41	731.56	912.91	615.69
1:5	744.43	993.16	1698.55	1064.84	2827.98	3579.81	1360.96
1:10	821.44	970.64	2980.17	3741.78	1341	4823.39	7429.3
1:15	1099.2	2766.55	4537.04	4715.12	5726.8	5537.9	9547.39
1:20	2545.89	2965.74	4677.69	5783.55	6657.01	7490.8	10312.84
1:25	1351.84	5215.93	1434.78	6791.28	7720.03	8171.29	10010.05
1:50	6735.84	1526.95	8058.24	9238.52	10543.52	9898.79	10386.47

Table 4.12. Two nodes with no buffer and weighted packet distribution

The results in Fig. 4.10 shows bandwidth incerases with the number of packets weighted towards a paths without any latency. This in addition to multiple path routing, the results in Fig. 4.10 show additional bandwidth gains by placing a weight on each of the connections. These results show the potential of writing a scheduler to apply weights. The scheduler could change weights based on current bandwidth readings[1] to further increase bandwidth.

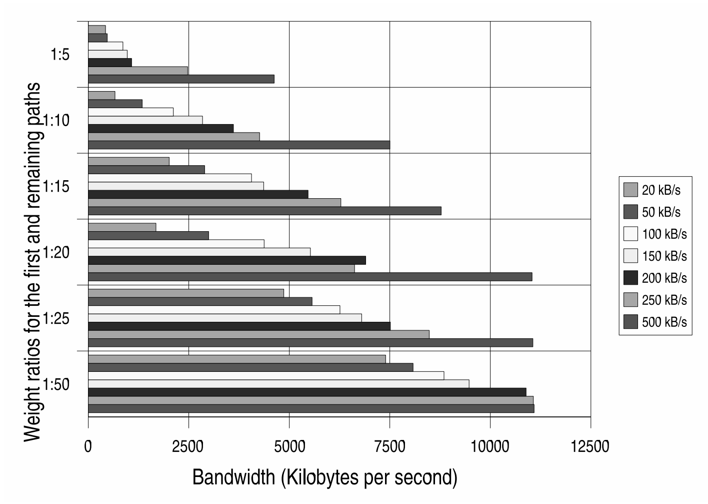


Fig. 4.11. Five nodes with no buffer and weighted packet distribution

Weight ratio	20 kB/s	50	100	150	200	250	500
1:5	429.16	473.08	861.53	969.8	1077.58	2471.8	4623.66
1:10	664.92	1340.77	2116.51	2841.25	3610.58	4262.41	7502.99
1:15	2011.95	2894.95	4061.25	4362.48	5467.36	6282.97	8777.42
1:20	1683.33	2994.02	4374.42	5525.86	6901.93	6622.7	11038.23
1:25	4862.64	5565.77	6260.65	6800	7514.78	8481.29	11058.5
1:50	7395.51	8078.38	8846.95	9471.17	10889.49	11070.3	11089.29

Table 4.14. Five nodes with no buffer and weighted packet distribution

Fig. 4.11 is another set of results showing the benifits of applying a weight to the packets intermediate proxy and is also another reference showing the benifits of multiple path routing where increasing the number of intermeidate proxies increases the overall bandwidth.

The next two results are from tests done with a connection based buffer with a maximum buffer size of 100 packets. The buffer stores packets until the correct and expected packet arrives and sents the sorted packets to be processed.

[1] The Linux kernel has timers and the infastructure to montior and determine the current bandwidth. Two examples are the TCP timers for sending duplicate acknowledgements and the IP Chains for restricting bandwidth to a set value.

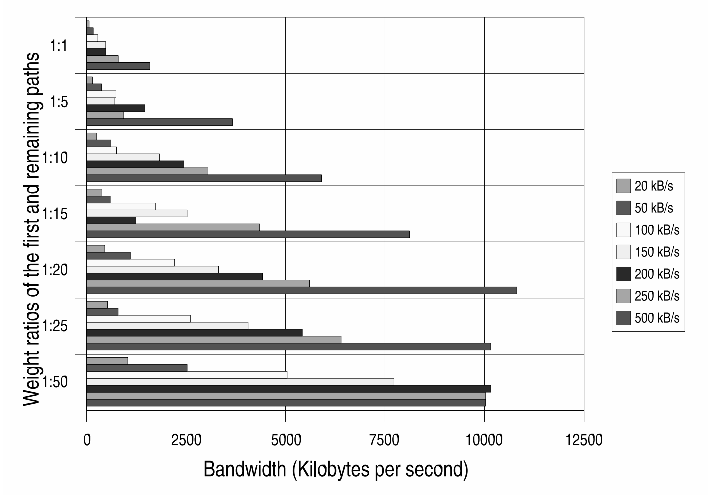


Fig. 4.12. Two nodes with buffer and weighted packet distribution

Weight ratio	20 kB/s	50	100	150	200	250	500
1:1	65.56	170.42	285.01	482.97	484.43	798.32	1591.96
1:5	149.46	378.63	740.57	695.41	1467.37	937.25	3666.94
1:10	246.87	614	755.12	1836.36	2449.2	3051.8	5903.01
1:15	387.14	597.32	1729.88	2527.56	1228.37	4349	8113.38
1:20	459.41	1100.52	2213.77	3316.02	4421.19	5602.23	10812.94
1:25	526.93	793.25	2609.29	4059.65	5421.48	6397.25	10157.18
1:50	1039.59	2530.88	5040.78	7726	10158.86	10020.19	10026.31

Table 4.16. Two nodes with buffer and weighted packet distribution

The results in Fig. 4.12 are the a continuation of results disproving the efficency of buffering the incoming packets. Fig. 4.12 does further solidifies the efficency of placing a weight on the intermediate proxies by testing these results with a buffer.

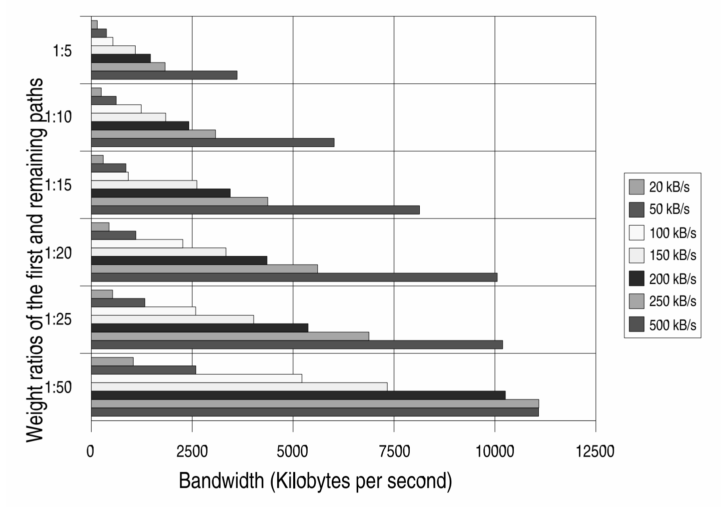


Fig. 4.13. Five nodes with buffer and weighted packet distribution

Weight ratio	20 kB/s	50	100	150	200	250	500
1:5	150.08	376.09	535.07	1091.85	1467.25	1828.28	3612.16
1:10	246.84	616.7	1238.35	1842.64	2417.22	3078.82	6017.99
1:15	295.54	857.57	917.48	2615.94	3441.39	4373.17	8133.43
1:20	439.95	1104.84	2266.62	3337.03	4354.55	5606.78	10058.13
1:25	527.3	1328.11	2585.6	4024.4	5370.11	6879.43	10193.91
1:50	1041.07	2590.06	5218.48	7334.42	10258.47	11088.64	11082.53

Table 4.18. Five nodes with buffer and weighted packet distribution

Fig. 4.13 compared to Fig. 4.11 demonstrates buffering the incoming packets cases a bandwidth penality greater than the duplicate acknowledgement from without a buffer. Ethereal network taps have shown the buffering does actually significantly reduce the dupicate acknowledgements caused by the multiple path technique to a problem explained further in Section 3.3.

4.3.2 Weighed paths, one path with small bandwidth, and additional paths bandwidth controlled

Early results indicated a smaller overall bandwidth would validate time penalty with buffering and sorting incoming packets. This lead to this additional set of tests where one node is heavily bandwidth limited and weights applied to each path. There are one sets of tests taken: 100 kilobytes per second overall bandwidth.

The results below have an overall bandwidth of 100 kilobytes per second on each node and the first node is further bandwidth restricted based on the bandwidth in the key. These results are for two and five nodes. Y-axis represents the weight ratio applied to the first path and the remaining paths. The X-axis is the result bandwidth between the client and server.

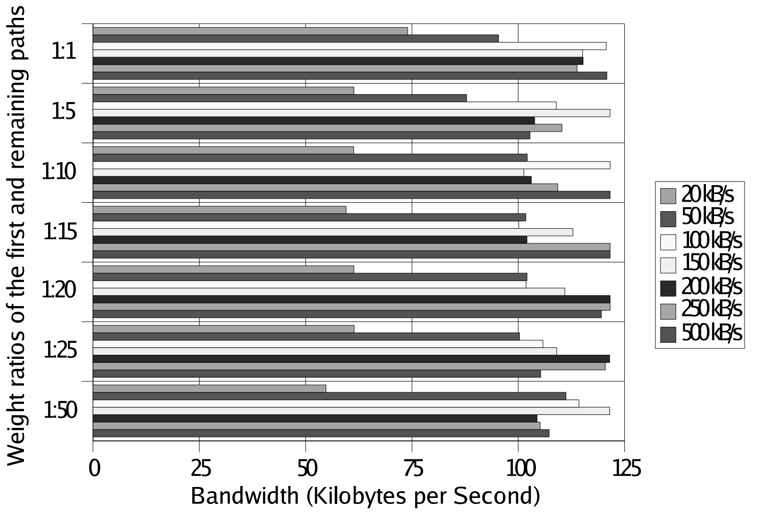


Fig. 4.14. Two nodes, without buffer, first node bandwidth variable, other nodes 100 kB/s

Weight ratio	20 kB/s	50	100	150	200	250	500
1:1	74.02	95.36	120.71	115.16	115.25	113.83	120.84
1:5	61.37	87.88	108.97	121.59	103.87	110.27	102.76
1:10	61.3	102.12	121.61	101.3	103.08	109.29	121.61
1:15	59.55	101.79	100.13	112.86	102.06	121.61	121.61
1:20	61.37	102.06	101.84	110.95	121.61	121.61	119.55
1:25	61.43	100.29	105.81	109.06	121.51	120.48	105.3
1:50	54.81	111.21	114.28	121.51	104.42	105.15	107.26

Table 4.20. Two nodes, without buffer, first node bandwidth variable, other nodes 100 kB/s

Fig. 4.14 shows no major impact with rate limiting the remaining connections. With virtual machines where the overal bandwidth is restricted by the virtual network device, even slower bandwidths (4-5 kB/s) were initially seen and gave the theory excessive bandwidth provides the TCP protocol a method for self correcting from extra duplicate acknowledgements. These results are proven inconclusive by Fig. 4.14 showing there is greater than single digit bandwidth as demonstrated with the virtual machines.

Additional tests were done with buffering and were similar to the differences in Fig. 4.13 Fig. 4.11. These results were the final proof buffering the incoming packets was more of a cost than a benifit.

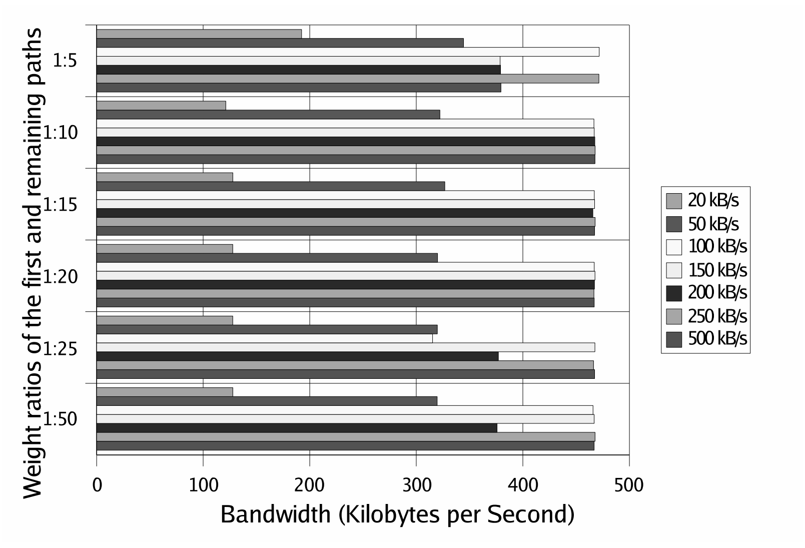


Fig. 4.15. Five nodes, with buffer, first node bandwidth variable, other nodes 100 kB/s

Weight ratio	20 kB/s	50	100	150	200	250	500
1:5	192.4	344.35	471.66	378.65	379.09	471.37	379.47
1:10	121.29	322.16	466.71	466.98	467.59	467.85	467.74
1:15	127.86	326.75	467.07	467.33	465.71	467.85	467.36
1:20	127.8	319.98	467.02	467.7	467.22	466.74	467.01
1:25	127.8	319.79	315.31	467.62	377.06	466.33	467.37
1:50	127.82	319.6	465.88	466.93	375.83	467.78	466.94

Table 4.22. Five nodes, with buffer, first node bandwidth variable, other nodes 100 kB/s

The thesis writer debated on the necessity of the results in Fig. 4.15. The main point of Fig. 4.15 and Fig. 4.14 shows under further constraints how multiple path routing and placing a weight on the intermediate packets increases bandwidth.

4.4 Buffer results

These last set of results answer the question if the overall bandwidth is affected by available buffer size. In the experiments with debug turned on, the buffer never appeared to go above a size of 20. These tests restrict the overall bandwidth on a one node connection and change the buffer size.

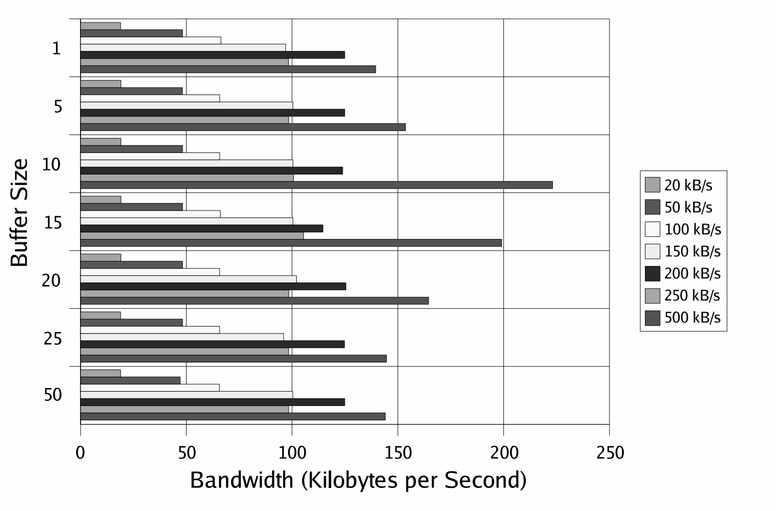


Fig. 4.16. Buffer size changing with controlled bandwidth

Buffer Size	20 kB/s	50	100	150	200	250	500
1	18.93	48.25	66.4	96.97	124.91	98.32	139.51
5	19.12	48.23	65.78	100.39	124.91	98.4	153.59
10	19.12	48.25	65.78	100.38	123.93	100.68	223.16
15	19.12	48.25	66.11	100.39	114.58	105.44	199.05
20	19.1	48.25	65.76	102.05	125.43	98.46	164.55
25	18.94	48.25	65.78	96.03	124.77	98.39	144.63
50	18.93	47.1	65.65	100.32	124.91	98.39	144.08

Table 4.24. Buffer size changing with controlled bandwidth

Fig. 4.16 shows the bandwidth is control by the restriction placed on the connection and not by changing the buffer size.

Additional bandwidth limiters were requested in addition to HB, the techique to restrict bandwidth using ipchains described in Section 3.5. Tests with a 10Mbit hub were performed. These results were very poor because of the number of collisions and gave inconstancies.

4.5 Summary of results

The main results for the scenarios show multiple path routing increases bandwidth, and weighted multiple path routing increases bandwidth further.

4.5.1 Multiple path routing

The results from Sections 4.2 and 4.3 show the bandwidth increase as Multiple path routing is turned activated within the kernel and intermediate proxies are added. The bandwidth increases almost proportionally with the addition of proxy nodes. Further tests were done to make sure the TCP was not compensating with all the excessive bandwidth results are shown in 4.14, and Fig. 4.22. These results restricted the overall bandwidth between proxy nodes and show TCP compensates for out-of-order sequences.

4.5.2 Weighted multiple path routing

Tests in Section 4.3 also showed the potential to further increased bandwidth by applying a weight to each path going to the intermediate proxies. As the ratio changes, more packets going to paths with higher bandwidth, the overall bandwidth increases. Fig. 4.10 and Fig. 4.11 show the increase in bandwidth as the packet ratio changes.

4.5.3 Buffering

Buffering takes a performance hit and only under unreliable, and low bandwidth increases the overall bandwidth. This can be proven by looking at results in Sections 4.2 and 4.3 with those including buffering and those same conditions without buffering (e.g., Fig. 4.11 and Fig. 4.13.

Since these results differed from the intial results done on virtual machines, a couple of theories arrose from researchers.

- The first packet in the TCP's three way hand shake was always going to the slow path and could slow the overall connection speed. This theory was disproven with unpublished results where the first packet went to a higher bandwidth path.
- The having unlimited bandwidth (100 megabits per second) on the remaining connections would allows the TCP protocol to recover from out of order sequences. Other tests done with

the overall bandwidth capped, showed the TCP protocol still performed well even without excessive bandwidth.

The results in Section 4.4 also show the size of the buffer does not affect bandwidth. So the tests with the buffer show, buffering the incoming packets takes a higher penality the advantage of removing duplicate packets and is a good for future references. The connection-based buffering technique described in section 3.3 might also benfit another project where bandwidth speed is less of an issue.

Chapter 5

Conclusion and future work

This last chapter is a high level summery of discoveries of this thesis and possible future work. The contributions include:

- The performance increase with multiple path routing.
- Weighted multiple path routing further increases bandwidth performance.
- A technique for connection based buffering technique to remove out of sequence packets.
- Documentation on debugging the Linux kernel with an IDE or emacs editors, an additional tool for other kernel developers.
- Documention explaining networking packets processing by the Linux TCP/IP sutie, INET.

5.1 Work contributions

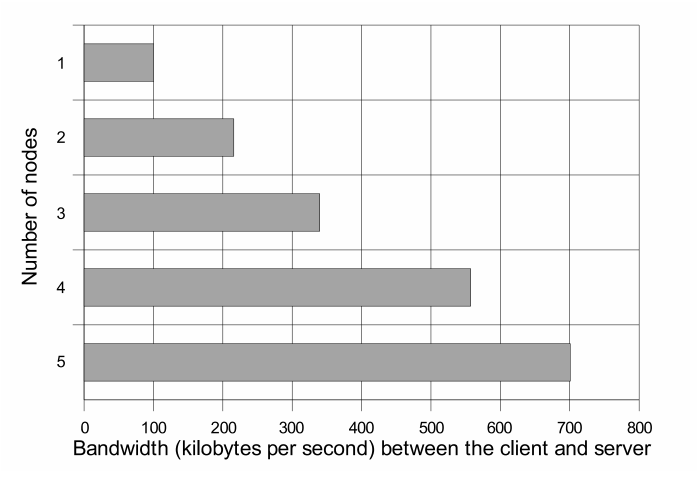


Fig. 5.1. Multiple path routing with bandwidth rate limiting to 150 kB/s

# of node	Bandwidth for each path is restricted to 150 kB/s
1	100.45 kB/s
2	215.88 kB/s
3	339.72 kB/s
4	557.24 kB/s
5	700.82 kB/s

Table 5.2. Multiple path routing with bandwidth rate limiting to 150 kB/s

Fig. 5.1 and Table 5.2 is a reproduction from Section 4.2 to show how the bandwidth increases as the number of nodes in multiple path routing increases. This bandwidth increase is shown better in the table. Each path is restricted by a given bandwidth listed in the column headers of the table.

Multiple path routing takes advantage of the bandwidth of addtional intermediate proxies and increases the overall bandwidth. These results are a begining of a solution to the problem posed in Section 1.2. The technique for multiple path routing is described in Section 3.1 and Section 3.2.

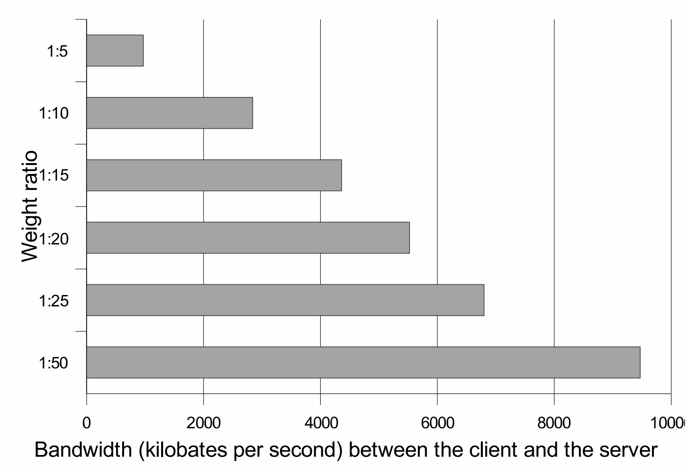


Fig. 5.2. Weighted Multiple path routing with the first path bandwidth limited to 150 kB/s and using five nodes

Weight ratio	Bandwidth for the first path is restricted to 150 kB/s
1:5	969.8 kB/s
1:10	2841.25 kB/s
1:15	4362.48 kB/s
1:20	5525.86 kB/s
1:25	6800 kB/s
1:50	9471.17 kB/s

Table 5.4. Weighted Multiple path routing with the first path bandwidth limited to 150 kB/s and using five nodes

Fig. 5.2 and Table 5.4 are cropped versions of results from Section 4.3. Fig. 5.2 is respresentive of all results demonstrating as the weighted ratio increases to higher bandwidth paths, the overall bandwidth increases.

Section 3.3 describes the technique for creating a connection based buffer. Connection based buffering is a good concept but still needs more work to be benifial for multiple path routing. Connection based buffering significantly reduce the number of duplicate acknowledgements as shown with ethereal network taps, but Fig 5.3 shows the bandwidth is lower with connection based buffering and multiple path routing. The connection based buffering techniques has a perfomance hit caused by waiting for the correct packet before processing the buffer. This delay increases the overall window and in higher bandwidth environments decreases the overall bandwidth speed.

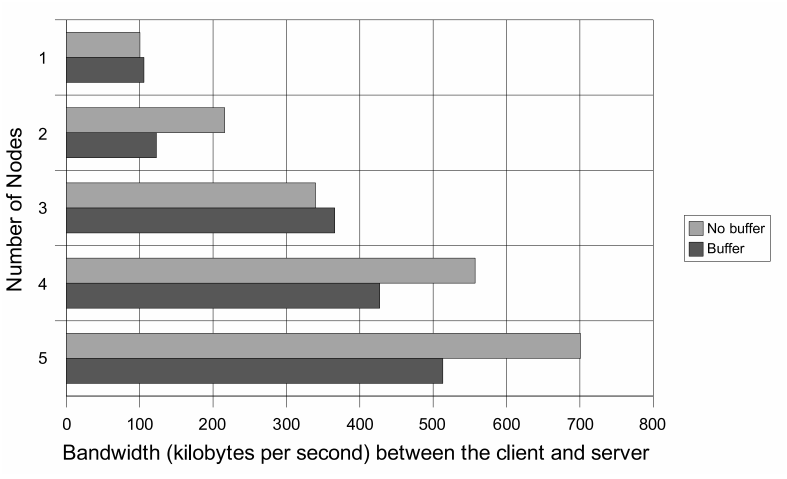


Fig. 5.3. Multiple path routing vs multipath routing using a connection based buffer

# of node	No buffer	Buffer
1	100.45 kB/s	105.93 kB/s
2	215.88 kB/s	122.87 kB/s
3	339.72 kB/s	365.87 kB/s
4	557.24 kB/s	427.14 kB/s
5	700.82 kB/s	513.21 kB/s

Table 5.6. Multiple path routing vs multipath routing using a connection based buffer

5.2 Additional work

Appendix A and B talk about how to setup a virtual operating system, User Mode Linux, to debug and study the Linux kernel. Many months went into figuring out the online documenation and experimenting with configurations so the virtual machine would work properly.

Appendix C and D documents the study of the Linux kernel's TCP/IP suite, INET. Considering the lack of documentation for the INET code, the writer believes it could be very popular and a useful tool to understanding and teaching the TCP/IP protocol suite.

5.3 Future work

While there are many applications fo expand a technique for utlitilizing bandwidth over multiple connections, here are a few the writer would like to share.

5.3.1 A scheduler for applying the weights to each connection path

Using the technique talked about in Section 4.3, a scheduler could be written to manage and change the weights based on the available bandwidth. The bandwidth per connection needs to be stored in a structure. The bulk of the data could be routed through faster connections and calculations could be done to find the weight peaking the bandwidth. The algorithm used could also write to the scheduler and send packets later in the window anticipating a delay and utilitizing the slow connection path without incurring out of sequence packets. The results shown in Section 4.8 demonstrates the possiblity of raising the bandwidth to fully utilize the combined bandwidth of all the connection paths.

5.3.2 One-time pad crytograpy using the INET code

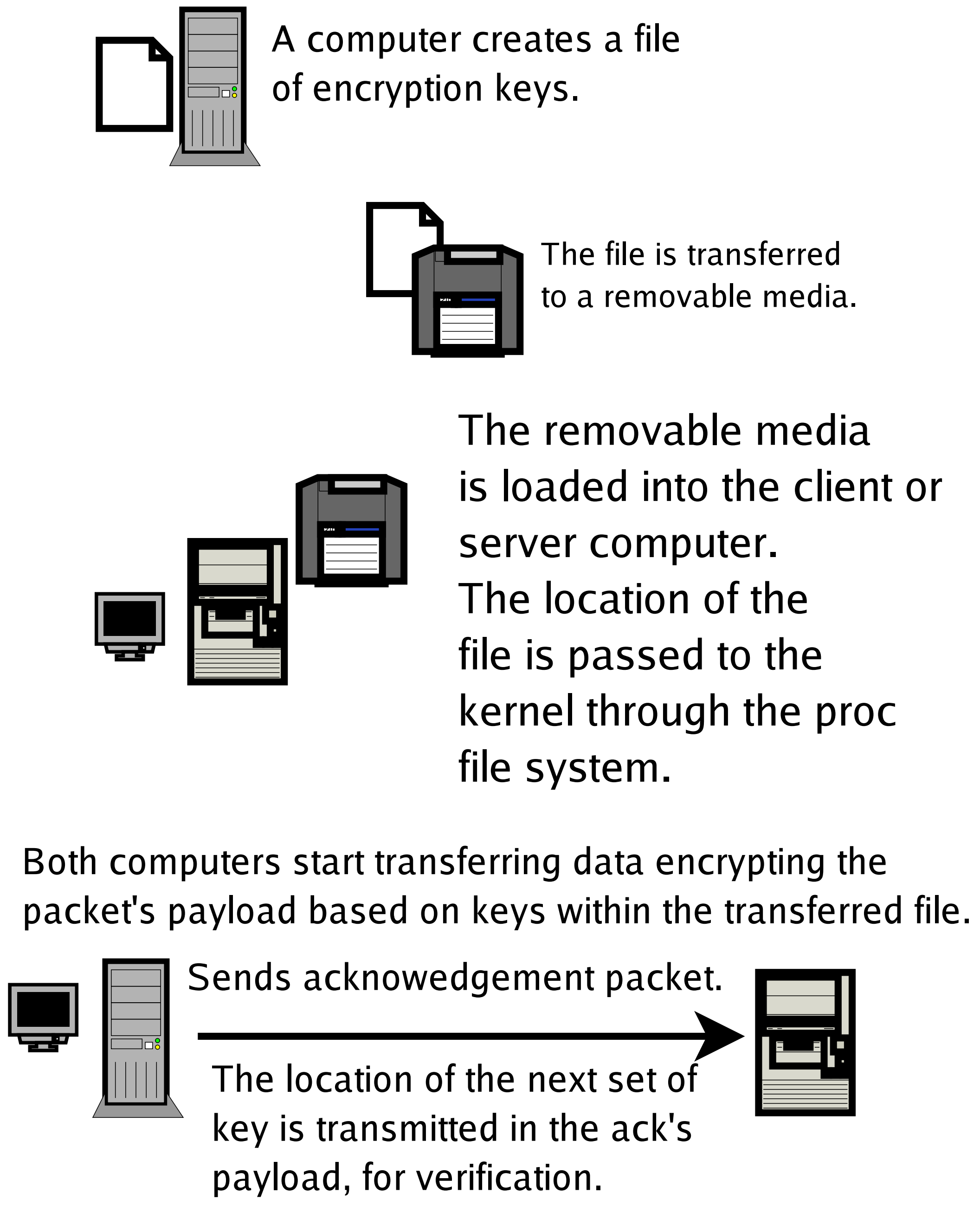

A computer creates a file of encryption keys.
The file is transferred to a removable media.
The removable media is loaded into the client or server computer. The location of the file is passed to the kernel through the proc file system.
Both computers start transferring data encrypting the packet's payload based on keys within the transferred file.
Sends acknowedgement packet.
The location of the next set of key is transmitted in the ack's payload, for verification.

Fig. 5.4. One time pad with the INET code modifications

The techniques for virtual machines and modifying the TCP/IP protocol explained in A, C, and D could be the baseline for creating a secure protocol based on the TCP/IP and one-time pad cryptography[12].

A random generated file containtaing multiple keys is stored on a removeable media (i.e., one gigabyte flash card). The flie's location and a switch for using this random file are passed to the kernel through the proc. Section 3.4 talks about how proc can input data to the kernel. The TCP protocol has acknowledgment packets with no data in the payload. Both computers (server and client) have access to the random generated file. The acknowledgement packet can carry a pointer to the current key. The INET code can be modified to encrypt the payload based on the current key. When the packet is received, the INET code knows the key to decrypt the data's content.

Since the keys change every four to five packets received (roughly 1500 bytes per packet or 6000 bytes). Using a one gigabyte removable device and encryption keys of 128 bits every four packets. A one gigabyte flash card should be able to send rougly 6.25 terabyes of data (excluding the overhead for the packet transfer) before being recharged with a new random file. Fig. 5.4 graphically illistrates this idea of using one time pad with INET modifications.

Each acknowledgement sends the location of the next key to be used. Since acknowledgments are sent every four to five packets during an established connection, the next sent packet uses the next key in the location. This ensures each packet is encrypted with a unque key and data within the acknowledgements are merely pointers to verify where each side is pointing to within the file.

5.3.3 Pornography filter within the kernel

Kernel modifications can force an Internet user to check a website for decency without user intervention. If the check returns saying the site is no allowable, the kernel can return an invalid message and block contiuation of the connection.

A protocol similar to ARP could be developed to check a website by sending the IP address to an online database. When the online database checks the website, a confirmation can be sent to the computer, similar to DNS. The Linux kernel can also cache the list of acceptable sites to further speed up validation.

Bibliography

[1] Y. Cai. (2004, October) Ip over ip tunnel. [Online]. Available: http://cs.uccs.edu/ scold/iptunnel.htm

[2] J. Dike. User mode linux. [Online]. Available: http://user-mode-linux.sourceforge.net

[3] A. S. Tanenbaum, *Modern operating systems*, 2nd ed. Prentice-Hall, Upper Saddle River, 2001.

[4] D. Grothe. (2001) Kgdb: linux kenrel source level debugger. [Online]. Available: http://kgdb.linsyssoft.com/

[5] Y. Cai, “Linux kernel enhancements for multipath collection,” September 2004, uccs network research seminar.

[6] C. Chow, C. Y., D. Wilkinson, and G. Godavari, “Secure collective defense system,” in *Global Telecommunications Conference*, no. 4. GLOBECOM ’04. IEEE, Dec 2004, pp. 2245 – 2249.

[7] E. Mouw. (2001, June) Linux kernel procfs guide. [Online]. Available: http://www.kernelnewbies.org/documents/kdoc/procfs-guide.pdf

[8] M. Devera. (2002, may) Htb linux queuing discipline manual - user guide. [Online]. Available: http://luxik.cdi.cz/ devik/qos/htb/manual/userg.htm

[9] B. Hubert. Linux advanced routing and traffic control. [Online]. Available: http://lartc.org/

[10] F. Echantillac, F. Chanussot, and P. Primet. (2004, Febuary) A tool for routing and traiffic control. [Online]. Available: http://perso.ens-lyon.fr/francois.echantillac/Docs/index.html

[11] S. Bohacek, J. Hespanha, J. Lee, C. Lim, and K. Obraczka, "Tcp-pr: tcp for persistent packet reordering," *23rd International Conference on Disributed Computing Systems*, pp. 222–23, 2003.

[12] Wikipedia. One-time pad from wikipedia, the free encyclopedia.wikipedia. Wikipedia. [Online]. Available: http://en.wikipedia.org

[13] L. Torvalds. The linux kernel archives. [Online]. Available: http://www.kernel.org

[14] D. Coulson. (2002) User-mode-linux community site. [Online]. Available: http://usermodelinux.org/index.php

[15] G. Beekmans. Linux from scratch. [Online]. Available: http://www.linuxfromscratch.org

[16] R. Binns. Uml builder. [Online]. Available: http://umlbuilder.sourceforge.net/

[17] J. Brokmeier, "Run linux on linux learn how to build user mode linux filesystems," *Linux magazine*, no. 60, January 2004.

[18] V. L. Systems. e2fsprogs. [Online]. Available: http://e2fsprogs.sourceforge.net/ext2.html

[19] J. Dike, "Running linux on linux," *Linux magazine*, no. no. 28, 2001.

[20] M. Krasnyansky. (2001) Universal tun/tap device driver frequently asked question. [Online]. Available: http://vtun.sourceforge.net/tun/faq.html

[21] J. Ward. (2003, April) Compiling the linux kernel on redhat 7.1. [Online]. Available: http://www.jsward.com/linux/redhat-kernel.html

[22] K. Lowe. Kernel rebuild guide. [Online]. Available: http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html

[23] D. Seager. (2001, February) Linux software debugging with gdb. [Online]. Available: http://www-106.ibm.com/developerworks/library/l-gdb/

[24] J. Zawodny, "Benchmarking with apache bench," *Linux Magazine*, no. no. 49, 2003.

[25] I. S. Institute. (1981, September) Rfc 793, transmission control protocol (tcp/ip). [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[26] B. Hall. (2001, October) Beej's guide to network programming. [Online]. Available: http://www.ecst.csuchico.edu/ beej/guide/net/bgnet.pdf

[27] D. Comer, *Interworking with tcp/ip principles*, 4th ed. Prentice-hall, upper saddle river, 2000, vol. Vol. 1.

[28] A. Cox, "Kernel korner: network buffers and memory management," *Linux journal*, no. no. 30, 1996.

[29] M. Rio, M. Goutelle, T. Kelly, R. Hughes-Jones, J.-P. Martin-Flatin, and Y.-T. Li, "A map of the network code in linux kernel 2.4.20," DataTAG, Tech. Rep., March 2004. [Online]. Available: http://datatag.web.cern.ch/datatag/papers/tr-datatag-2004-1.pdf

[30] G. Herrin. (2000, May) Linux ip networking, a guide to the implementation and modification of the linux protocol stack. [Online]. Available: http://www.kernelnewbies.org/linux-net.pdf

[31] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. VerWornerz, *Linux kernel programming*. Addison-Wesly, London, 2002.

[32] H. Welte. (2000, October) Linux network buffers. [Online]. Available: http://gnumonks.org/ftp/pub/doc/skb-doc.html

[33] A. Rubini, *Linux device drivers*. O'Reilly and assocates, Sebastopol, 2001.

[34] V. Guffens and G. Bastin. (2002, May) Modeling of the linux switch architecture. [Online]. Available: http://www.auto.ucl.ac.be/ guffens

[35] G. Insolvibile, "Inside the linux packet filter," *Linux journal*, no. 94, April 2002.

[36] E. Troan, "Introduction to system calls," *Linux magazine*, 1999.

[37] Whatis.com, the leading it encyclopedia and learning center. [Online]. Available: http://whatis.techtarget.com/

[38] H. Welte. (2000, January) The journey of a packet through the linux 2.4 network stack. [Online]. Available: http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html

Appendix A

Comments on User Mode Linux and Testing Perl Script

The general setup is fairly straight forward. The UML web-page hosted by sourceforge has vast amounts information about setting up UML[2]. The trick is setting up the network section with debugging. The documentation for setting up the network section is not detailed.

A.1 Installing the UML utilities

The first thing is to install the UML tools. The UML tools allow the virtual networks to talk to each other and provides additional support tools.

To install the UML utilities, download the uml_utilities_<build number>.tar.bz2 from the user mode linux download page (http://user-mode-linux.sourceforge.net/dl-sf.html). With root privilege, uncompress the tar.bz2. This will uncompress the source into a directory "tools". Change the directory into tools and execute a “make all”.

```
tar xvjf uml_utilities<build number>.tar.bz2
cd tools; make all
make install
```

Fig. A.1. Unpackaging UML tools

This will install the UML tools on the host machine. Again these tools are needed for the UML sessions to communication with each other.

There are five main tools. It is not necessary to call them up or use them by command line prompt. The UML session needs the tools installed and will use them in the background. The five main tools are:

- uml_moo - merge COW(Copy On Write) file with its backing file
- uml_mconsole - UML management console
- uml_switch - switch daemon
- uml_net - setuid helper for network setup
- tunctl - create and control TUN/TAP interfaces

A.2 Compile the kernel the UML kernel

The Linux sourced code is available at the kernel.org repository[13]. At the time of the experimenting with UML the newest kernel was 2.4.18. The Linux kernel is intimidating . Unpackaged the code is about 300 MegaBits without any builds.

he program "patch" updates the source code. For any Perl fans, this is the same patch invented by Larry Wall. It is used for maintaining source code and for distributions of versions. Patch takes a diff file (differences of the original file and newer file) and makes the original file a newer version. Take the UML patch file and apply it to the root level of the UML source code. The UML patch is available from the UML sourceforge web-page (the file name looks similar to: uml-patch-2.4.19-51.bz2). The bz2 extension represents bzip2 format (a compression format similar to zip files. Fig. A.2 shows how to uncompress the file type and apply the uncompressed patch file.

```
bunzip2 uml-patch-2.4.19-51.bz2
patch -p1 < uml-patch-2.4.19-51 # at the root level of the source code
```

Fig. A.2. Installing UML patch file

Notice the file name does not have the .bz2 extension. The patch command can actually work from a couple of different levels. The -p1 means at the root level of the source code. -p0 means one directory above (../ directory) and the updates are redirected into the directory.

Once the patch is applied, the next step is to compile the kernel code. This is going to involve two commands. These commands are shown in Fig. A.3 and need to be executed at the top level of the patch Linux source tree.

```
make xconfig ARCH=um
make linux ARCH=um
```

Fig. A.3. UML Make command

The GUI menu screen in Fig. A.4 should pop up.

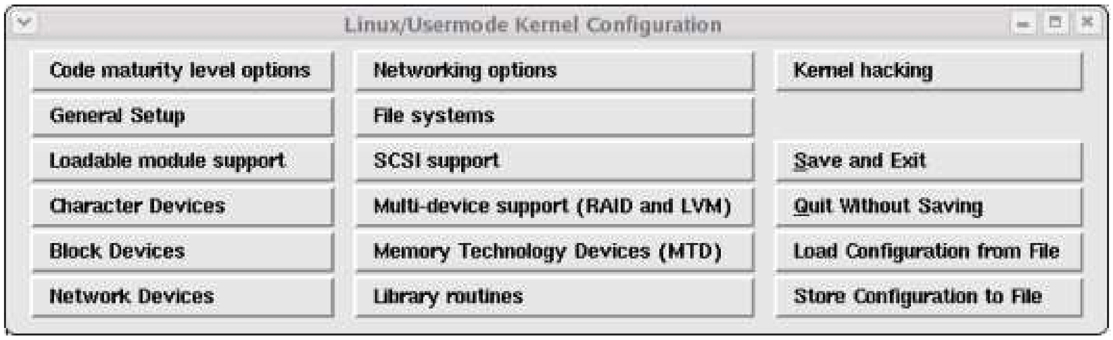


Fig. A.4. Uml make config screen

Just the default is good. To tweak the kernel a little more, click the "kernel hacking" button and turn on all four options. This adds some extra information in the back-traces and also ensures the debugging component of the kernel is working properly. In the newer kernel (2.4.18+), especially with the skas mode patch into the host kernel setting these debug options breaks the compile. Turning on all the options for the "kernel hacking" in kernel version before 2.4.17 gave additional debug information to be used while debugging the kernel. This thesis' experiments are based on the 2.4.26 kernel which works with the skas host patch.

After setting the desired options, hit the "save and exit" button and exit the GUI. This will save the configuration into a file called, ".config." ".config" is a hidden file.

There were two commands mentioned (patch -p1 < uml-patch-2.4.19-51; make xconfig). If the next command does not execute properly, try “make dep ARCH=um.” There have been instances where the UML kernel would not compile completely. "make dep ARCH=um" fixes the problem.

“make linux ARCH=um” creates the Linux executable at the top level of the Linux kernel source tree. The compile should take almost 10 minutes the first time the UML Linux kernel is compiled. Once the kernel is compiled; the second compile only builds modified objects. So, future recompilation do not take as long. Once the Linux executable is create the just copy the "linux" file to the same directory as the root_fs.

A.3 Creating the root_fs

If an experimenter is bold enough to create their own root_fs, there are tools to assist in this venture. For those who prefer the simple, a root_fs’ can be downloaded from the UML’s sourceforge website[2, 14].

Slackware is probably the purest Linux distribution, meaning the programs and libraries are not "enhanced" for a specific Linux kernel and architecture. A root file systems build with RedHat are the worst violators of changing or enhancing source code to perform better. UML is an architecture built on virtual software devices, not optimized for RedHat’s enhancements.

One way to make a custom root_fs is to create an empty root system file (maybe 400-600 megs) with the "dd" command (dd if=/dev/zero of=new_filesystem seek=100 count=1 bs=1M). Then mount the empty root_fs using "mount <name of empty file> <mount directory> -o loop" and follow the instructions from the document "linux from scratch"[15]. This will create a personalized Linux distribution. Creating a root_fs from a distribution might be better.

There are four tools are available to create root_fs[1] [2]; the two most common are uml builder[16] and mkrootfs[17]. Both tools use RPM distributions (i.e., Redhat, Suse, and Mandrade). Mkrootfs is a very simple tool for advanced users and is command line. Mkroofs is mainly used for creating multiple root_fs from a script. Uml Builder is a simple GUI interface for single root_fs creation.

Uml Builder has a step by step interface allowing for networking and xwindow setup. At the end of the setup, the GUI creates the script file for turning on the xwindows. As a side note, the script

[1] http://user-mode-linux.sourceforge.net/fs_making.html has more information about these tools

file for turning on xwindows is not an easy script file. Unlike turning on networking and debugging, the script for setting up xwindows is extremely complicated, is sparely documented on the web, and is pages long. If a script for xwindows is needed, consider the twenty minute configuration time with uml builder oppose to hacking the instructions.

A.4 Resizing the root_fs

There are times when resizing the file system is necessary because the default size is not enough. To increase the file size just do these three command in the frame below. The file size in this example is increased to 640 MB, but the 640 MB can be changed to a more desirable size. resize2fs is part of the e2fsprogs utility suite[18] for ext2 file format.

```
dd if=/dev/zero of=root_fs bs=1 count=0 seek=640MB
e2fsck -f root_fs
resize2fs -p root_fs
```

Fig. A.5. Resizing a uml filesystem

A.5 Parameters for running UML

If the root_fs is in the same directory where the Linux executable is launched, the user can merely type "./linux" and UML should start. Parameters are not really needed. The parameters are used for networking options, redirecting the location of the root_fs, and setting the memory. The main advantage for this appendix is the debugger setup. When debugging UML, have the Linux executable and the source at the top level of the source tree. If the UML exectuable is located somewhere else, the debugger (gdb) cannot find the source code. Fig. is an example of a script used to run UML.

```
./linux umid=windom mem=128M debug=go ubd0=root_fs udb2=swap
```

Fig. A.6. Running uml

This example is without network connection (networking will talked about in the next section).

The ./linux, represents in this current directory. There are RPM binary versions of UML (which do not allow for debugging and other good stuff). The RPM binary version of UML installs the Linux executable in the /usr/bin. Now while this is nice and good, it interferes with the running of the Linux executable. The directory paths are confused and UML no longer had debugging capability. This is what happens when there are two versions of the same file both in the path. This is not an uncommon problem, so make a note. Run the Linux executable with the current directory symbol in the front. It is an extra "./".

The umid=windom is a label given to the Linux kernel. The UML utilities references the UML session by the label passed in with umid=<id>. The UML utilities controls the networking, monitors UML sessions, and provides a jail program to catch hackers.

The mem=128M is the memory UML will be reserving. This is defaulted to 128 megabits, but can be as low as 16 M (16 megabit) without xwindows. Most of the machines used for the experiments only had 512 megabits and with three machines at 128, it left the host operating system with 128. Not good.

The debug=go is an important switch. This switch does not work if the skas is patched into the host machine's kernel.

The udb0=root_fs is the name of the root_fs. If this option is left blank, the Linux kernel executable will default to root_fs.

The last option, udb2=swap is the swap space for the UML. The "swap" is assigned to udb2 and is not a file but use the host kernel's swap.

A.6 Network Setup

There are two more optional parameters which were not mentioned. These options are for the networking. While UML has five different network drivers it can utilize, the two main virtual network drivers are Tuntap and Ethertap.

A UML session requires two IP addresses per device driver[19]. The first IP address needs to be in the command line prompt, and the second IP address is used by the "ifconfig" utility within the UML session. The Ethertap and Tuntap virtual network drivers need two IP addresses to interface with the IP Tables and the virtual machines. In Fig. A.7 is a copy of what the routing tables look likes after the UML interface drivers (tuntap drivers) are interfaced with the host machine's networking components.

```
[root@walden uml5]# route -n
IP routing table Destination Gateway Genmask Flags Metric Ref Use Iface
128.198.60.172    0.0.0.0          255.255.255.255   UH 0 0 0 tap1
128.198.60.173    0.0.0.0          255.255.255.255   UH 0 0 0 tap0
128.198.60.128    0.0.0.0          255.255.255.128   U  0 0 0 eth0
127.0.0.0         0.0.0.0          255.0.0.0         U  0 0 0 lo
0.0.0.0           128.198.60.129   0.0.0.0           UG 0 0 0 eth0
```

Fig. A.7. Routing table with tuntap entries

On the far right, there are two interfaces labeled tap0 and tap1. These interfaces are used by the UML session stating 128.198.60.172 and 128.198.60.173 are to be routed to the virtual machines using the tuntap drivers.

The next section will talk about the first of the two drivers Ethertap more in depth. While it is possible to have two UML sessions using either Tuntap or Ethertap, Ethertap is the older of the two and is no longer recommended in UML development. It still deserves some mention.

A.6.1 Ethertap

Fig. A.8 is a command line using Ethertap.

```
./linux umid=windom eth0=ethertap,,,128.198.60.133 mem=60M udb2=swap
```

Fig. A.8. Uml run with ethertap

eth0=ethertap,,,128.198.60.133 is the command line parameter which interfaces with the IP routing table entry for Ethertap. Ethertap is another device driver similar to the Ethernet driver and provides packet reception and transmission for the application level. Ethertap does not allow for multiple network interfaces in a UML session and is not as secure as Tuntap.

A.6.2 Tuntap

Tuntap like Ethertap is a virtual network device driver and interfaces with the host machine's IP routing table[20]. Tuntap does allow for a UML session to have multiple interfaces to the host machine's routing table. For a while, the use of Joseph Mac's Linux Virtual Machine (LVS) was pondered. LVS requires two network interfaces to do a NAT (Network Address Translation). The first network interface was for the back-end servers and the other for general Internet access. Having one UML session with two network interfaces allows for the creation of many virtual Local Area Networks (LAN's) on one machine.

Fig. A.9 is a command line using tuntap using a single interface:

```
./linux umid=windom eth0=tuntap,,,128.198.60.172 mem=128M //
debug=go ubd0=root_fs udb2=swap
```

Fig. A.9. Uml run with tuntap

The umid=windom is used for assigning an ID to the UML session. As a side note, the ID to the UML session is used by the UML utilities tools to monitor and give external commands to the UML session from the host machine. Fig. A.10 is the command line parameters for two network interfaces.

```
./linux umid=windomCM eth0=tuntap,,,128.198.60.162 eth1=tuntap,,,128.198.60.163 //
mem=128M debug=go ubd0=root_fs udb2=swap
```

Fig. A.10. Uml run with two tuntap interfaces

Add an additional "eth(n)" driver (where n is the number of the device) gives the UML session another network device. If there is another Tuntap driver, an additional IP address needs to be assigned through the command line.

A.7 Once inside the UML session

Run the commands in Fig. A.11 at the UML prompt to gain network connectivity.

```
ifconfig eth0 172.31.0.169
route del -net 172.31.0.0 dev eth0 netmask 255.255.0.0
route add -host 172.31.0.101 dev eth0
route add default gw 172.31.0.101
```

Fig. A.11. Ifconfig commands for setting up the network device

Where 172.31.0.169 is the IP address of the UML session (not the same IP address as the one passed in). 172.31.0.0 representing the subnet. 172.31.0.101 represents the host machine's IP address. This IP address will be used as a gateway for the UML session. A script file can set all the network options once UML session starts. To adding a second network interface, add the command line parameters in Fig. A.12 setting the eth1 interface.

```
ifconfig eth1 192.68.0.100
route del -net 192.68.0.0 dev eth1 netmask 255.255.0.0
route add -host 192.68.0.100 dev eth1
```

Fig. A.12. Ifconfig commands for setting up a second network device

A.7.1 Inside of Slackware – configuring the Slackware root_fs

The Slackware root_fs can be used just "as is" without any modifications. There are many tweaks that can be done to the Slackware root_fs. These tweaks include: setting the number of xterm windows spawning off, setting up the hostname, configuring dns, adding additional programs (like ssh), and how to set the routing table to automatically start up without a script or typing in the routing commands.

A.7.2 spawning off xtermals (setting the number of xterminals)

Slackware is a pure version of Linux. The maintainers of Slackware do not make excessive modifications to Linux kernel, libraries, or programs to run well together. Not adding additional modifications makes Slackware work well with the non-conforming UML kernels. The only problem with having a "pure version", is not having the configuration utilities found in the RedHat distributions.

Without these UI configuration utilities modifications are done by editing files.

Spawning off xterminals are found in the /etc/inittab. Open up the /etc/inittab file with an editor. Scrolling down the /etc/inittab, do a search on "spawn" and grab the section listed below.

Like most scripting languages or Linux config files the "#" means comment. In Fig. A.13, the c0: is the terminal the UML session is called on. The remaining c1-c6 are spawned xtermals. When c1-c6 are uncommented, the UML session will bring up additional xtermals connected into the running UML sessions.

```
# These are the standard console login getties in multiuser mode:

c0:1235:respawn:/sbin/agetty 38400 tty0 linux
#c1:1235:respawn:/sbin/agetty 38400 tty1 linux
#c2:1235:respawn:/sbin/agetty 38400 tty2 linux
#c3:1235:respawn:/sbin/agetty 38400 tty3 linux
#c4:1235:respawn:/sbin/agetty 38400 tty4 linux
#c5:1235:respawn:/sbin/agetty 38400 tty5 linux
#c6:12345:respawn:/sbin/agetty 38400 tty6 linux
```

Fig. A.13. Spawning xterminals

A.7.3 Setting up DNS

Setting up the DNS consists of three parts: correctly configuring the host name, setting up the host file, and assigning the DNS servers. The first involving modifying a file called HOSTNAME (all caps). The HOSTNAME file is in the /etc directory. The file only consists of the full DNS name of the UML session. For example, an UML session with the name walden.uccs.edu would only have walden.uccs.edu inside the HOSTNAME file.

The host file (/etc/hosts) is the first place the DNS server checks for a listing (by default). Unlike the HOSTNAME file this is not unique to Slackware, generally all Linux distributions have this file. Open and add all the DNS entries for your private network. The format is space delimited: <IP address> <Full DNS name> <nickname>

The last file is the /etc/resolv.conf. The resolv.conf lists all the available DNS servers. The format is also space delimited: nameserver <IP address of the DNS server>. Word of caution the

DNS resolver is sensitive the to the host machine's firewall. If the firewall is turned on, the answer from the DNS server will not be able to make it through the firewall. The firewall is not set up to handle TAP's IP address and rejects the packet designated for TAP driver.

A.7.4 Installing and uninstalling programs

To install a Slackware program, either mount the root_fs or turn on the host machines ftp server to get the package onto the root_fs. Fig. A.14 shows the command to mount the root_fs.

```
mount root_fs /mnt/root_fs -o loop
```

Fig. A.14. Mounting a root file system

/mnt/root_fs is directory where the file system should be mounted. Pending on the version of Slackware, a mirror site should have the packages needed. The Slackware directory structure is awkward at first. The âdirectory listingâ file should mention where a package is kept. For example the ssh package would be under the "n" directory (for network). The package should end in .tgz extension. Fig. A.15 shows the commands to install and uninstall slackware packages.

```
installpkg xf_bin.tgz # installs a package
removepkg xf_bin.tgz # uninstalls a package
# xf_bin is the package name
```

Fig. A.15. Installing and uninstalling slackware packages

A.7.5 Configuring the network on startup

Under Slackware root_fs, the network configuration file is located in /etc/rc.d. Before 9.1, fill in the IP address (IPADDR field), netmask(NETMASK FIELD), and gateway (GATEWAY field) in the rc.inet1. If there are questions, there are BOLD comments instructing what fill in. For Slackware 9.1, fill in the network parameters in the /etc/rc.d/rc.inet1.config. Filling in the config file will save time by avoiding the manually network setup (using ifconfig).

A.8 Debugging with UML

Starting about UML kernel patch 2.4.19, Jeff Dikes (the maintainer of UML) came up with a the Skas (Separate Kernel Address Space) patch which causes the UML session to run in an entirely different host space from its processes[2]. It also allows for a debugger to attach to the main process and step through the Linux source code. While debugging was in the previous tt mode (Tracing Thread), a separate debug window using GDB would come up with the correct thread attached. The newer skas mode allows for outside debuggers to attach to the kernel.

A.8.1 Installing the skas patch on the host machine

When recompiling a kernel, the default settings are missing device drivers which most distributions include as modules. Three things are needed to install the skas patch. First, get a Redhat kernel source tree as close to the skas patch version as possible. linux-2.4.22-1.2115.nptl (from Fedora core 2) works with host-skas3.patch[2]. Second, download a Linux kernel from the Linux Kernel Repository[13] compatible with the skas patches listed at the UML website. For this example, kernel 2.4.24 works. Once the kernel is downloaded , and unpackaged (for kernel-2.4.24.tar.bz2 type: tar xvjf kernel-2.4.24.tar.bz2; for kernel-2.4.24.tar.gz: tar xvzf kernel-2.4.24.tar.gz). Change into the top directory of the kernel code (cd linux-2.4.24) and type "make mrproper". This is a clean, just like "make clean" but stronger. Apply the skas patch, "patch -p1 < host-skas3.patch". Copy from the Redhat source kernel a config file (kernel-2.4.22-i686.config) located under linux-2.4.22-1.2115.nptl/configs. Copy this config file to the top of the 2.4.24 kernel root tree and rename it to “.config” (cp ../linux-2.4.22-1.2115.nptl/configs/kernel-2.4.22-i686.config .config)[21]. Then type "make old config". A bunch of options should come up. Set all of the them to the default (just hit return) except the âproc_mmâ, set it to true. The âproc_mmâ is the skas patch. Once this is done type "make dep; make bzImage; make modules; make modules_install". This will build the kernel and install the modules. Next you need to install the kernel (cp arch/i368/boot/bzImage /boot/vmlinuz-2.4.24), set the initrd (mkinitrd /boot/initrd-2.4.24.img 2.4.24), and add a grub entry (just make a copy of a previous entry and change the initrd and vmlinuz files to match the other one). Then reboot. For additional help doing this, see the howto kernel document[22].

A.8.2 Crashing gracefully

This section contains three tips which can help immensely when crashing the User Mode Linux session. These tips are: set a breakpoint in panic.c:panic function, verify the User Mode Linux processes are killed, and check the file system using fsck.ext2.

User Mode Linux's kernel modifications has the scheduler call the panic.c:panic function to prompt the user when a crash occurs. This call to panic.c is good for the kernel developer in many ways: setting a breakpoint within the panic.c:panic function will create a back trace letting the developer know what caused the kernel crash and gives what values were on the stack when the crash occurred.

When a crash occurs, the User Mode Linux's processes might still be alive and attached to the root_fs. It is always a good idea to do a process status (ps command) and pipe it with grep, looking for âlinuxâ or the UML session's ID name. Fig. A.16 shows an example.

```
[frank@walden bin]$ ps aux | grep linux
frank 3332 pts/2 S /bin/sh /home/frank/linuxRs2/runRS2
frank 3333 pts/2 S /home/frank/linuxRs2/linuxRS2 (feline) [/sbin/rmmod]
frank 3335 pts/2 T [linuxRS2]
frank 3340 pts/2 S /home/frank/linuxRs2/linuxRS2 (feline) [/sbin/rmmod]
frank 3341 pts/2 S /home/frank/linuxRs2/linuxRS2 (feline) [/sbin/rmmod]
frank 3592 pts/4 S /bin/sh /home/frank/linuxRs/runRS
frank 3593 pts/4 S /home/frank/linuxRs/linuxRS (b2b) [/sbin/rmmod]
frank 3595 pts/4 T [linuxRS]
frank 3600 pts/4 S /home/frank/linuxRs/linuxRS (b2b) [/sbin/rmmod]
frank 3601 pts/4 S /home/frank/linuxRs/linuxRS (b2b) [/sbin/rmmod]
frank 3675 pts/5 S /bin/sh /home/frank/linuxRs3/runRS3
frank 3676 pts/5 S /home/frank/linuxRs3/linuxRS3 (feline) [/sbin/rmmod]
frank 3678 pts/5 T [linuxRS3]
frank 3683 pts/5 S /home/frank/linuxRs3/linuxRS3 (feline) [/sbin/rmmod]
frank 3684 pts/5 S /home/frank/linuxRs3/linuxRS3 (feline) [/sbin/rmmod]
frank 12852 pts/8 S grep linux
```

Fig. A.16. Checking for a uml process

Now there are three different UML sessions running: feline's, b2b's, and walrus. To kill the process, grab the process numbers which is the number after the owner of the process (in the example above the number after "frank"). Type "kill -9 <process numbers>". All the process numbers for all the selected UML session needs to be terminated. Using the example above, the command to kill the feline UML session would be: "kill -9 3333 3340 3341 3676 3683 3684". If all the UML process are not terminated after a UML kernel crash and the UML is restarted, a message like the one in Fig. A.17 appears.

```
VFS: Cannot open root device "ubd0" or 62:00
Please append a correct "root=" boot option
Kernel panic: VFS: Unable to mount root fs on 62:00
```

Fig. A.17. Locked file system panic message

In conclusion, the most common error causing a UML session to not start (when everything else is working) is another process still attached to the root_fs. Using a "kill -9" on the UML process fixes this problem.

The final tip is to use "/sbin/fsck.ext2 -p <root_fs name>" to check your root_fs systems. It is about five times faster than letting the UML session do the disk check. So when the kernel crashes and all the processes are terminated, type in "/sbin/fsck.ext2 -p <root_fs name>", and it will check the file system a lot faster.

A.8.3 Using a .gdbinit

Having a file called ".gdbinit" with the commands to by pass all the beginning breaks, expedites the debugger's startup.

Before opening the gdb program, put the file â.gdbinitâ at the top of the level of the UML source. Inside the ".gdbinit" type the two lines in Fig. .

```
handle SIGSEGV pass nostop noprint
handle SIGUSR1 pass nostop noprint
```

Fig. A.18. Stopping gdb from breaking at signals

The .gdbinit is the default script gdb looks for when started. This is helpful because without these two commands, gdb or any other debugger would stop every second or two. The ".gdbinit" file can be put in the home directory or use gdb -x <name of the file> to have gdb read the initial script.

A.8.4 Debugging in gdb

Once gdb starts, it is really hard to break it. It is recommended to start gdb without the Linux executable file appended in the command line. Otherwise, the UML session might have to be rebooted (hitting control-c a couple of times might work).

Start gdb. Once gdb is loaded, set it least one breakpoint. Then type "file," then the path to the Linux executable. For example: "file /home/frank/linuxSrc/linux". Gdb will automatically load the Linux executable with the correct parameters. When the program breaks, print the different variables or back-trace to better understand the kernel. For emacs user, gdb also interfaces with emacs[23].

A.8.5 Gdb debugging inside Eclipse

Install Eclipse. First download eclipse from one of their mirror sites[2].

Unzip the eclipse package using the unzip command "unzip eclipse-platform-3.0-linux-gtk.zip" also download and install the CDT plug-in. A recommended site is: ftp://eclipse.mirrors.tds.net/pub/eclipse.org/tools/
2.0-linux.gtk.x86.zip. The plug CDT in should be unzipped at the same directory level as the eclipse zip file (was unzipped).

Creating a project. File -> New->Project; In the new project window as shown in Fig. A.19.

[2] An example eclipse file would be at ftp://eclipse.mirrors.tds.net/pub/eclipse.org/eclipse/downloads/drops/R-3.0-200406251208/eclipse-platform-3.0-linux-gtk.zip

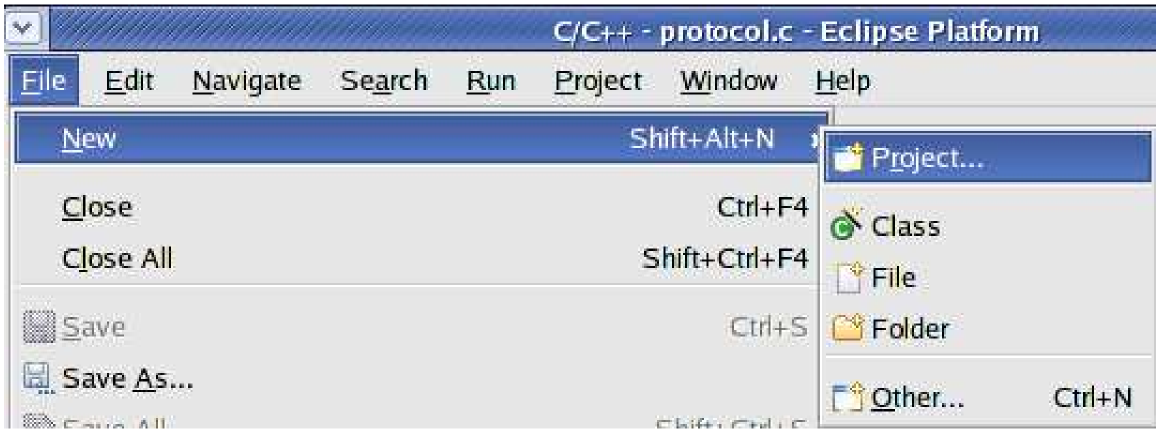


Fig. A.19. New eclipse project

Select C->Standard Make C Project. Then hit "next" as shown in Fig. A.20.

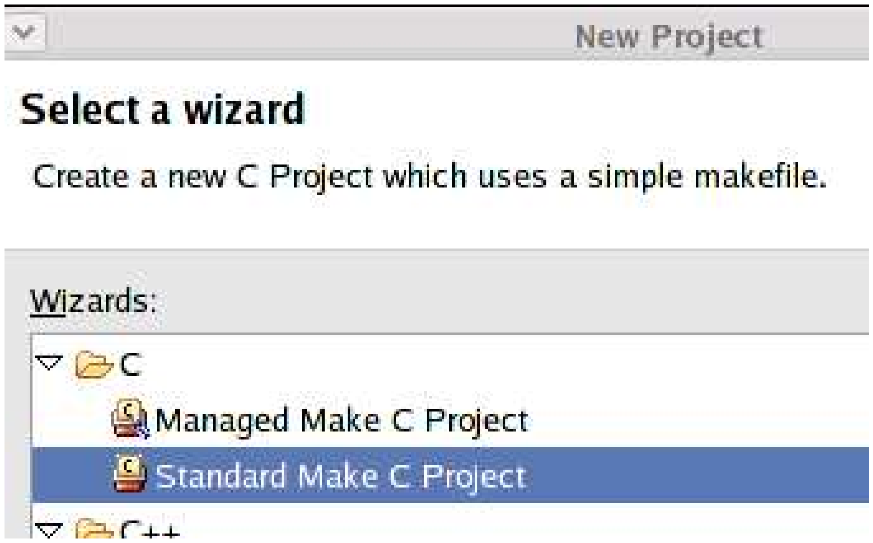
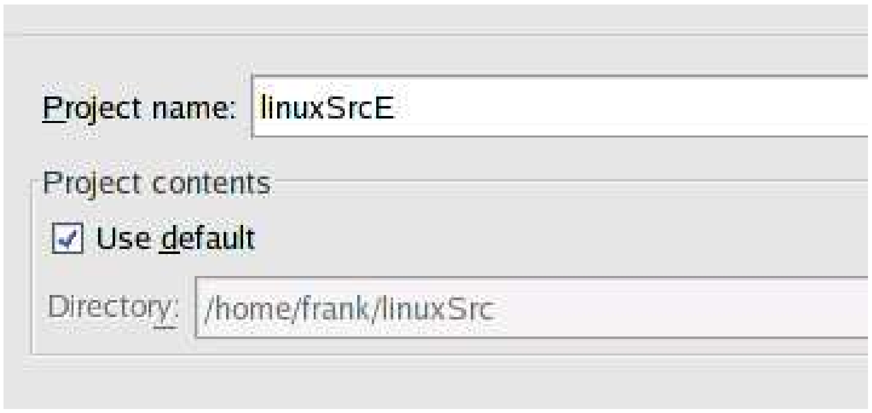


Fig. A.20. Starting a standard make c project

Project name is the directory where your Linux source code lives. For example, “/home/frank/” is the workspace directory. The top level of my Linux source tree is /home/frank/linuxSrcE. The project is linuxSrcE (just linuxSrcE – no additional directory names) as shown in Fig. A.21. Leave "Use default" clicked on, then hit "finish".



Project name: linuxSrcE

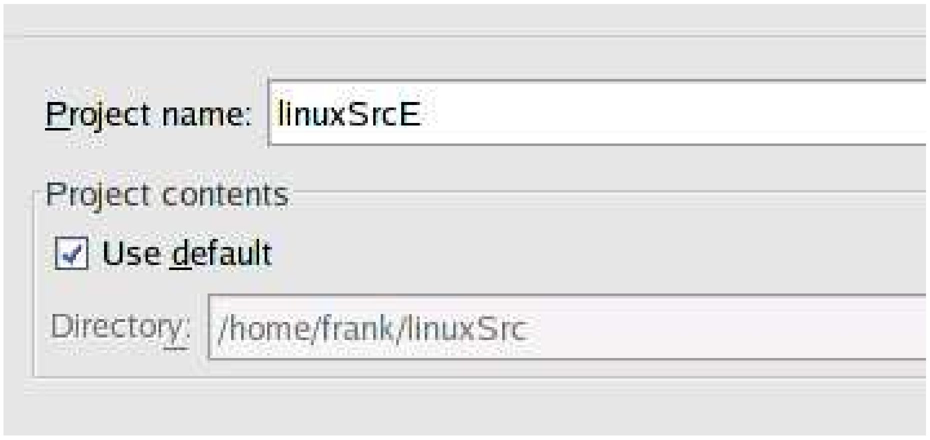
Project contents

Use default: ✓

Directory: /home/frank/linuxSrc

Fig. A.21. Setting up a project

The progress window C/C++ index should be indexing the Linux tree as shown in Fig. A.22. The process should take about fifteen minutes to half an hour. The indexer will find all your c, h, and make files files and place them in the project.



Project name: linuxSrcE

Project contents

Use default: ✓

Directory: /home/frank/linuxSrc

Fig. A.22. Code indexer

De-select project->build automatically. Otherwise, the framework will try and rebuild the source code everything the code is saved.

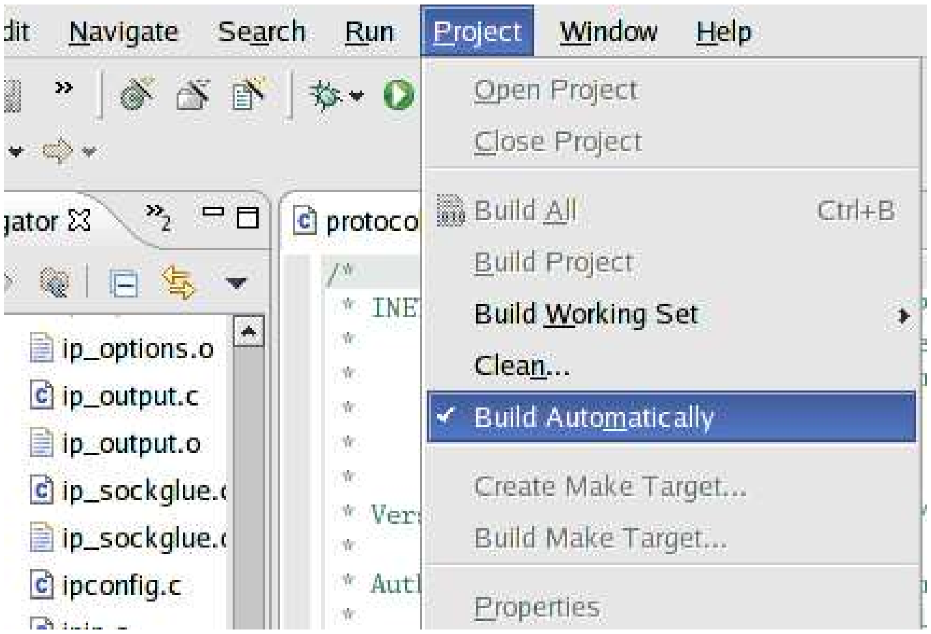


Fig. A.23. Deselect build automatically

Adding a run / debug. Click the run icon ->run shown in Fig. A.24. Another screen should come up.

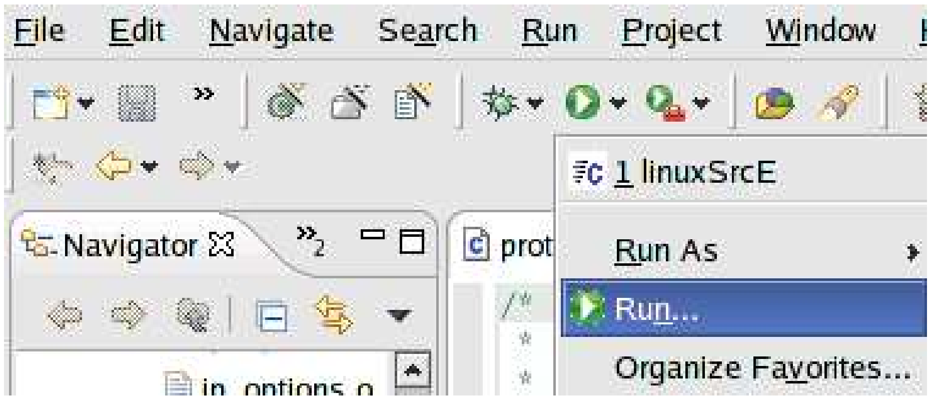


Fig. A.24. Run icon

Click the "new" button at the bottom (make sure "C/C++ local" is highlighted otherwise the new button will be disabled), as shown in Fig. A.25.

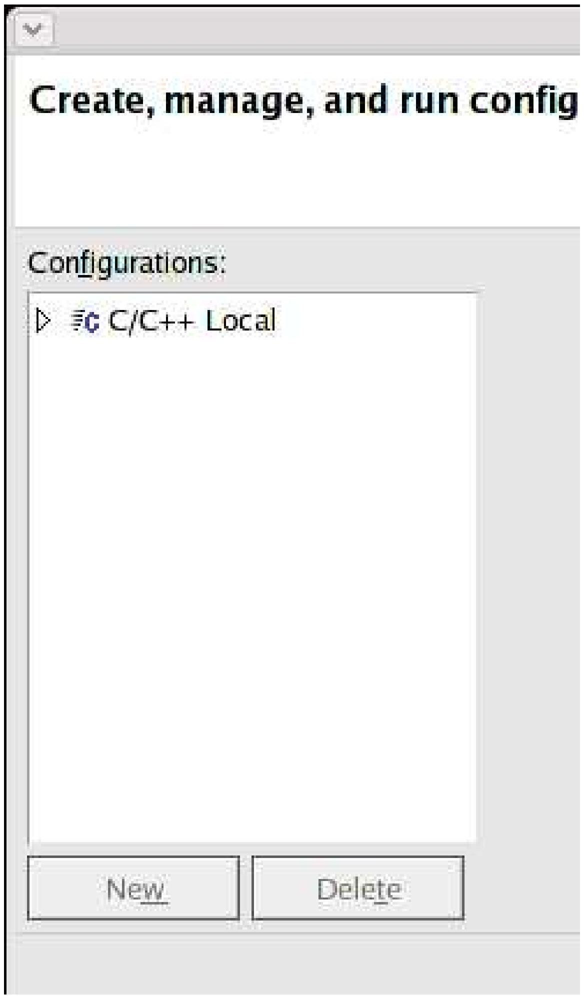
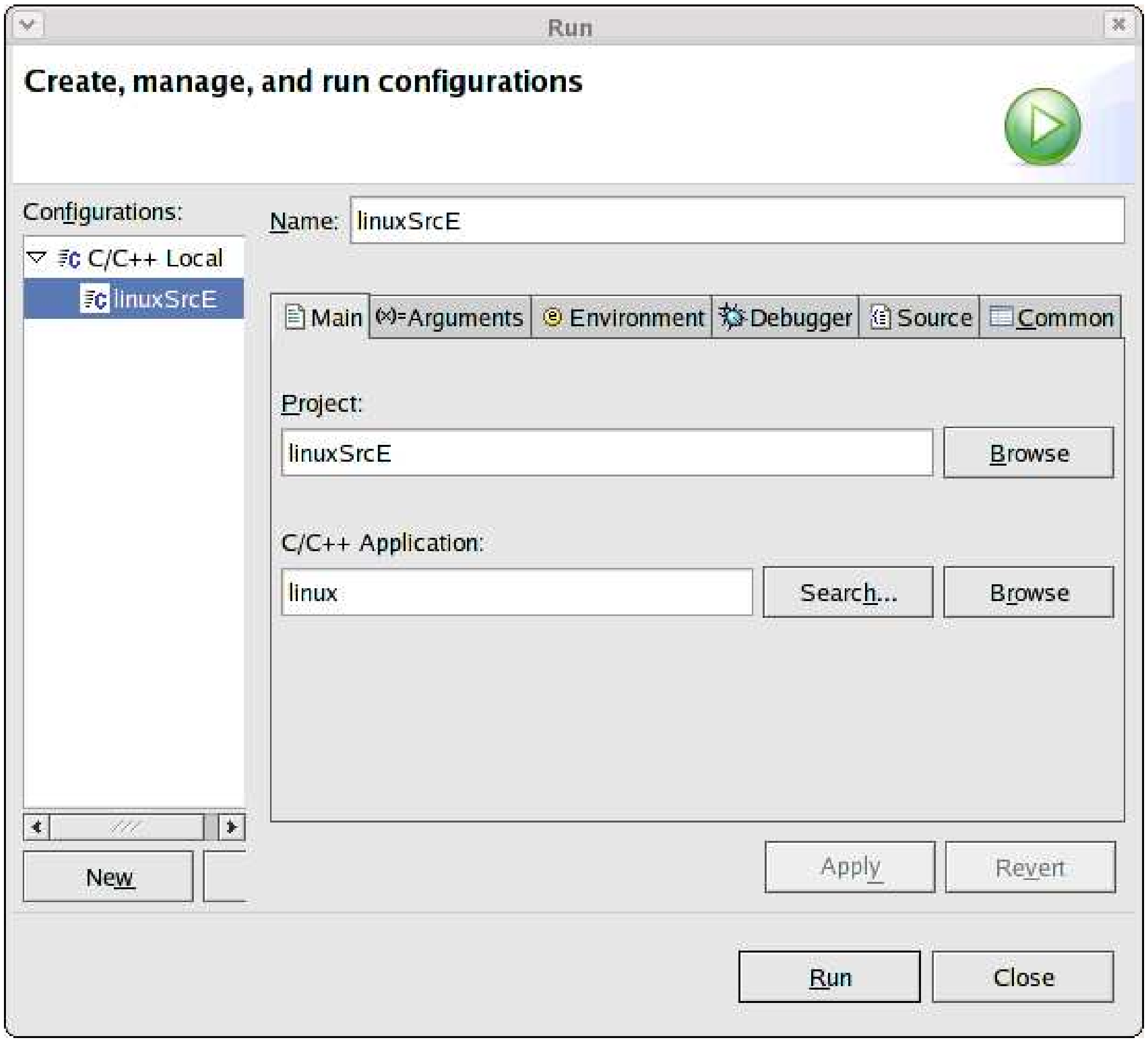


Fig. A.25. Creating a new run

Under the "main" tab as shown in Fig. A.26. In the name field, call the project anything. The example uses "linuxSrcE," same as the project name.



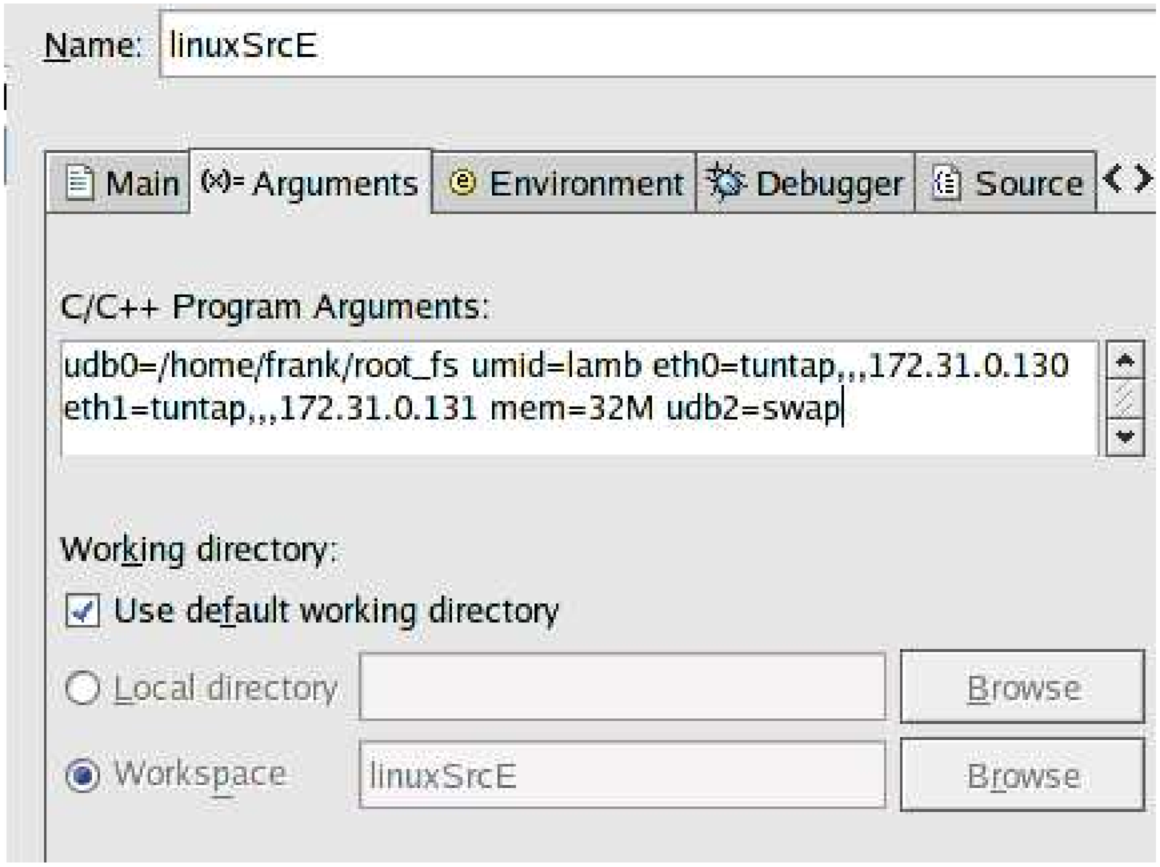
Name: linuxSrcE

Project: linuxSrcE

C/C++ Application: linux

Fig. A.26. Run main window

Under the "argument" tab (Fig. A.27) put the arguments used to run uml (i.e. linux <argument>). The arguments the example UML session uses are: "udb0=/home/frank/root_fs umid=lamb eth0=tuntap,,,172.31.0.130 eth1=tuntap,,,172.31.0.131 mem=32M udb2=swap".



Name: linuxSrcE

C/C++ Program Arguments:

```
udb0=/home/frank/root_fs umid=lamb eth0=tuntap,,,172.31.0.130
eth1=tuntap,,,172.31.0.131 mem=32M udb2=swap
```

Working directory:

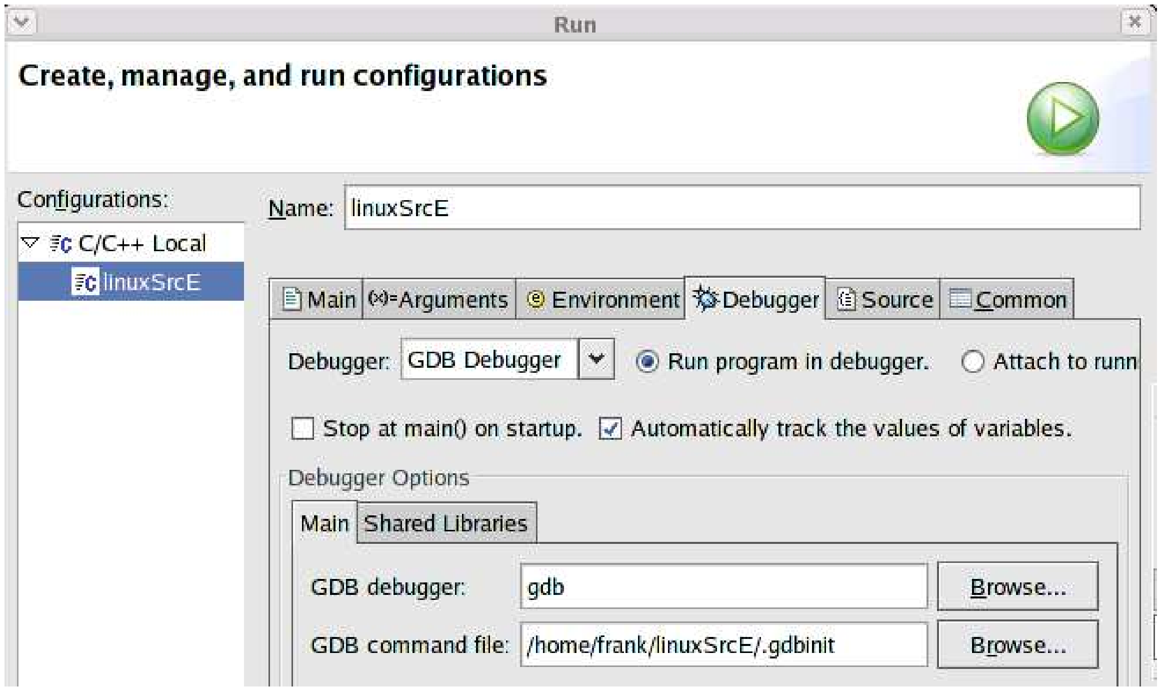
[✓] Use default working directory

Local directory

Workspace: linuxSrcE

Fig. A.27. Run argument window

Under the "debugger" tab(Fig. A.28). Set the drop down combo box to "GDB debugger" and click the radio button "run program in debugger". De-select "Stop at main() on startup"GDB debugger. Gdb debugger should be gdb. If you have built a newer version, put in the absolute directory where the newer gdb is located. In "GDB command file", put the location of your ".gdbinit" file.


Run

Create, manage, and run configurations

Configurations:

- C/C++ Local
  - linuxSrcE

Name: linuxSrcE

Main | Arguments | Environment | Debugger | Source | Common

Debugger: GDB Debugger [✓ Run program in debugger.] Attach to runn

Stop at main() on startup. [✓ Automatically track the values of variables.]

Debugger Options

Main | Shared Libraries

GDB debugger: gdb Browse...

GDB command file: /home/frank/linuxSrcE/.gdbinit Browse...

Fig. A.28. Debug Window

Add a make file. Window->Show view->Make target. The "make target" window on the side should come up (Fig. A.29). Right click on the root directory (in the example linuxSrcE) and click add make target.

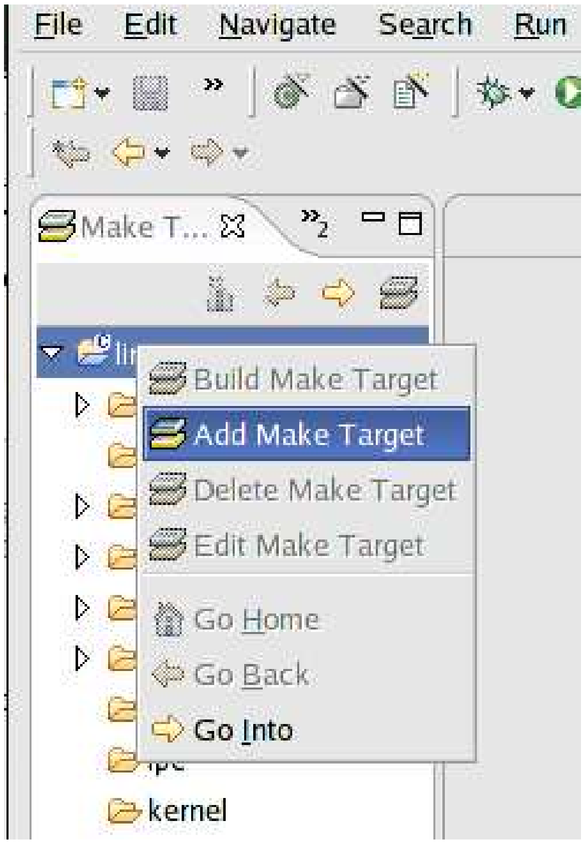
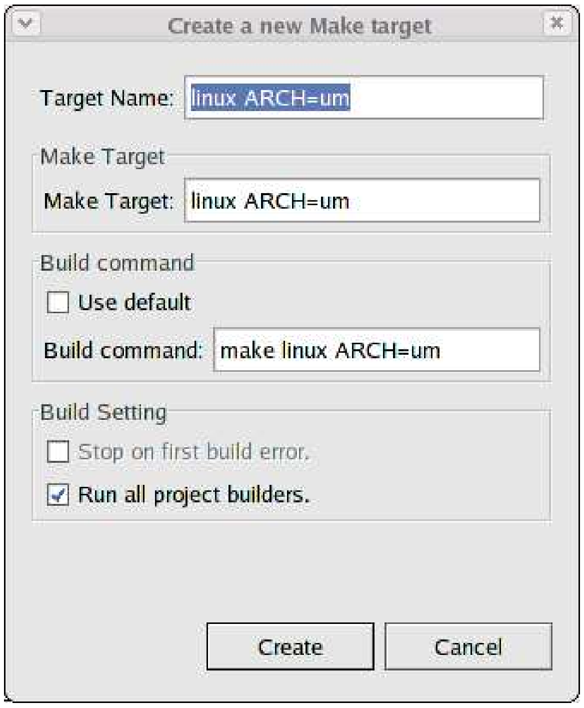


Fig. A.29. Adding a make target

A window "Create a new make target" should come up. Fill in the target name with a label for the make command; the example uses “linux ARCH=um”. In the "make target" field fill in “linux ARCH=um”, then de-select “use default”. In "build command," type "make linux ARCH=um". Un-select "stop on first build error" and select "run all project builders." Hit the "create" button. Fig. A.30 is a filled in window. Setup part is done.



Target Name: linux ARCH=um

Make Target

Make Target: linux ARCH=um

Build command

[ ] Use default

Build command: make linux ARCH=um

Build Setting

[ ] Stop on first build error.

[✓] Run all project builders.

Fig. A.30. Setting up make target

To build the source, right click the newly created make target and select "run make" as shown in Fig. A.31.

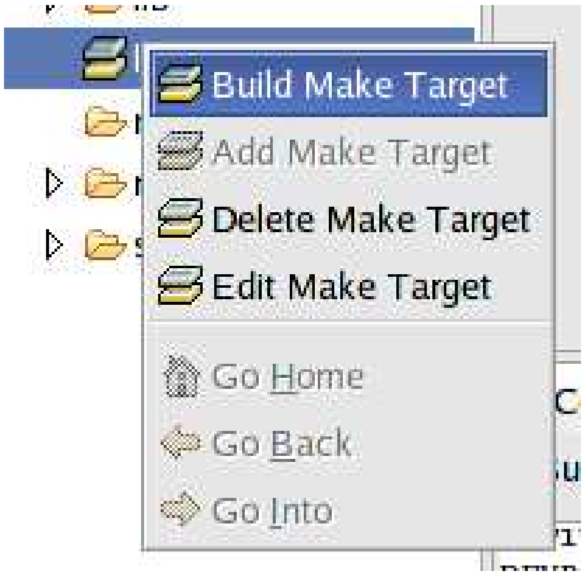


Fig. A.31. Building make target

To run there should be a icon on the tool bar (green circle with a white play triangle) as shown in Fig. A.32. It should drop down and you should see the name of the run session.

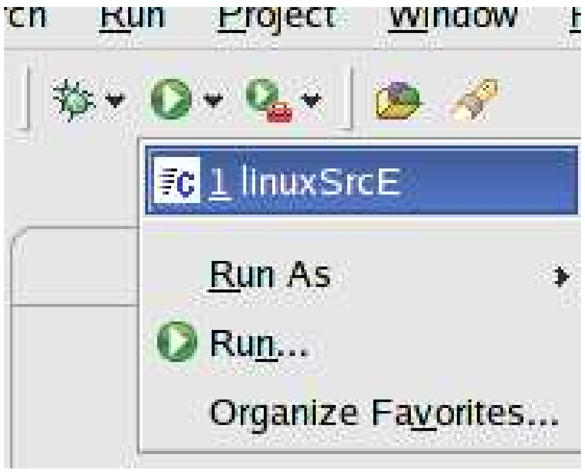


Fig. A.32. Running the program

To debug there should be a cockroach looking icon next to the run as shown in Fig. A.33. Click it and you should see the name of the run session. Once you start debugging, there are two windows you should be looking for the "console" and the "debug".

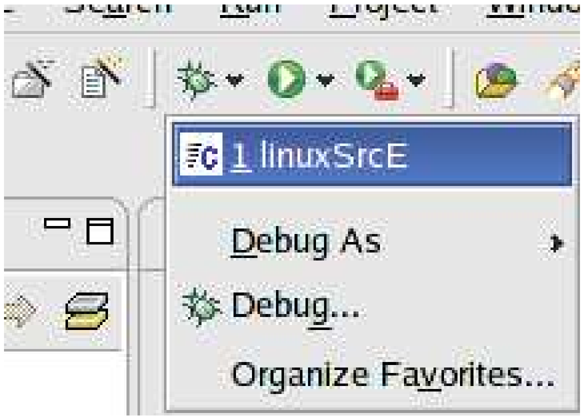


Fig. A.33. Start the debugger

A.9 Perl script used for testing

There are two perl scripts used for testing, client.pl and server.pl (both included with the media). The client.pl reads read in an input file called input.txt and setsup the proc file system by sending commands to the system prompt. The commands setup the kernel by changing the values in the proc file system (/proc) and also bandwidth limitation are for each of the connections.

A TCP socket connection is established with the server.pl. The server.pl is listening for the client.pl. When client.pl starts the connection with the machine running server.pl, it sends the commands read in from the input.txt. The server.pl reads only the commands needed to setup the server.

When the client.pl script waits one second and then start a web benchmark[24] to measure the speed of the connection. When the web benchmark is completed the results are parsed from the output and stored in a output file.

Appendix B

Instruction manual

The last section described the procedures needed to run User Mode Linux. This section will give command by command instructions on how to setup a network enabled User Mode Linux with skas debugging capabilities. Appendix A is a great resource if more details are needed. There are three major installations which need to be done: creation of the UML kernel, rebuild the host machine's kernel, installation of the UML tools. The configuration consists: setting up the tun/tap driver.

Currently, Fedora Core Two has issues with detaching from threaded processes under a debugger. Fedora Core One which is currently in Fedora Legacy support at the time of this writing is recommended for installing this UML kernel setup.

B.1 Building User Mode Linux

There are three major files that are needed. The root_fs, a kernel source with the correct version of the UML patch, and the UML tools. The kernel patch and the latter two can be downloaded entirely from the sourceforge website[2].

B.1.1 UML Building a UML Kernel

Building the UML kernel is a tricky process, because the initial build may not compile. When this happen, choose another kernel version with matching UML kernel patch and try again. The steps are fairly basic.

1. Go to the User Mode Linux website and see the patches available for download. On the down-

load website for User Mode Linux (http://user-mode-linux-sourceforge.net/dl-sf.html)[2] and under the label “Build From Source,” there is a listing for both 2.4 and 2.6 kernels. The latest 2.4 UML kernel patch is recommended.

2. Go to the Linux Kernel Repository located at http://www.kernel.org[13] or one of the associated mirrors (for faster downloads)[1] and download the correct kernel version. There should be two different compressions and patch files. Avoid the patch files and download a kernel compressed with either bzip2 or gzip (ending with extension bz2 or gz). The file should like: linux-2.4.27.tar.bz2.

3. Uncompress the kernel and apply the patch. If kernel file is *.bz2, type âtar xvjf kernelFileâ. If the kernel file is *.gz, type âtar xvzf kernelFileâ. Where kernelFile is the name of the kernel downloaded in step 2.

4. Change directory into the top level of the kernel source and clean out the kernel source by typing ”make mrproper.”

5. Uncompress and copy the UML patch to the top level of the kernel source tree . Type “bunzip2 UMLPatchFile” to uncompress the patch file. Where UMLPatchFile is the name of the UML patch file downloaded in step one.

6. With the patch file at the top of the kernel source tree, apply the patch by typing, “patch -p1 < UMLPatchFile”.

7. Run make configuration menu and save the default settings. Type “make xconfig ARCH=um” (case sensitive), and hit “save and exit” button located on the top right of the window. The default settings for the make file will work fine.

8. Build the kernel by type “make linux ARCH=um.” A “linux” executable file should be in the top of the Linux code source tree.

[1] The 2.4 kernel is also located at <ftp://ftp.us.kernel.org/pub/linux/kernel/v2.4/>and the 2.6 kernel at <ftp://ftp.us.kernel.org/pub/linux/kernel/v2.6/>

B.1.2 Root file system

Go back to the UML download page and under the header âThe root filesystem,â choose one of the root file systems. root_fs_slack8.1.bz2 is recommended and was used for the real servers. The client and servers used an expanded and updated version Slackware as a root file system1. Fans of RedHat are warned that RedHat root filesystems are custom to specific hardware (i386 hardware) and do not work as well with a virtual devices.

Rename the root_fs file, downloaded above, to “root_fs” which is the default root file system file.

B.1.3 Installing UML tools

This next step will set up monitoring and networking software on the host machine so the UML session can communicate with the host and other UML sessions.

1. On the UML download page and under the header âUML Utilities,â download the latest UML Utilities.
2. Uncompress the compressed tar package by typing. “tar xvjf UMLUtilityFile.” Where UMLUtilityFile is the file downloaded in step one. A directory called, “tools” should be created.
3. Go into the newly created “tools” directory and install the utility by typing, “make all; make install.” This need to be done with root privileges.

B.2 Running UML

To run an UML session with networking capabilities, two IP addresses are needed for each virtual network device. Choose two IP addresses in the network. Execute the Linux kernel with the command in Fig. B.1.

```
linux mem=128M udb=root_fs_slackware_7.0_big udb2=swap debug=go //
eth0=tuntap,,,<IP address # 1>
```

Fig. B.1. Running user mode linux

An x terminal should come up as shown below. For most Slackware root file systems, just typing in root should automatically login without a password.

Once inside the UML session has started type the following in Fig. B.2 to setup network the network connection.

```
ifconfig eth0 <IP address #2>
route del -net <Subnet> dev eth0 netmask 255.255.0.0
route add <IP of host machine> -host dev eth0
route add default gw <IP of host machine>
```

Fig. B.2. Ifconfig commands for inside the virtual file system

B.3 Running the perl test files

Running the test files (located on the media) are straight forward. Create an input file with command simiilar to those in Fig. B.3 and listed in the perl scripts. Setup the intermediate proxies with an IP Tunnel from the host machine and the server (Section 3.1). Also setup the host machine and server with tunnels to the intermediate proxies.

```
# no buffer 5 nodes first node limited by 250kB/s rest 100kB/s

numOfNode=5 multipath=1 destAddr=663cc680 srcAddr=653cc680 bufferOn=1 bufferSize=1
file="indexc2.html" numOfReq=5 label="5 node 1:5 250/100" rateLimitFirstNode=250
rateLimit=100

numOfNode=5 multipath=1 destAddr=663cc680 srcAddr=653cc680 bufferOn=1 bufferSize=5
file="indexc2.html" numOfReq=5 label="5 node 1:10 250/100" rateLimitFirstNode=250
rateLimit=100
numOfNode=5 multipath=1 destAddr=663cc680 srcAddr=653cc680 bufferOn=1 bufferSize=10
file="indexc2.html" numOfReq=5 label="5 node 1:15 250/100" rateLimitFirstNode=250
rateLimit=100
numOfNode=5 multipath=1 destAddr=663cc680 srcAddr=653cc680 bufferOn=1 bufferSize=15
file="indexc2.html" numOfReq=5 label="5 node 1:20 250/100" rateLimitFirstNode=250
rateLimit=100
numOfNode=5 multipath=1 destAddr=663cc680 srcAddr=653cc680 bufferOn=1 bufferSize=20
file="indexc2.html" numOfReq=5 label="5 node 1:25 250/100" rateLimitFirstNode=250
rateLimit=100
numOfNode=5 multipath=1 destAddr=663cc680 srcAddr=653cc680 bufferOn=1 bufferSize=25
file="indexc2.html" numOfReq=5 label="5 node 1:50 250/100" rateLimitFirstNode=250
rateLimit=100
```

Fig. B.3. Example input.txt file

Put the server.pl on the server and the client.pl on the client machine. Start the server.pl first. Make sure all ip_forward's are turned on for all the machines (/proc/sys/net/ipv4/ip_forward). Start the client.pl and debug output should start flying on the screen. The debug output is processed before and after the testing so there is no performance hit on the tests.

Appendix C

Assessment on how INET processes TCP/IP packets

The last section was about the UML tool. This section is about how Linux's TCP suite, INET, Handles TCP packets. This section will give a slight overview of the TCP/IP layers and then in the next section go into more detail over some of the major structures in the INET. These next two sections are helpful to understanding the kernel modifications, and how the Linux kernel handles the TCP protocol. The INET (Linux implementation of the TCP IP suite) is purely the same as the RFC's and some explanation of the differences are needed.

C.1 Overview of TCP

The TCP is the transport communication protocol. In the protocol stack it is four (transport) right above the Internet Protocol (IP). The TCP guarantees the arrival of the packets. The way the Linux kernel uses TCP is the packet arrives at the network card and the information (digital signals) is transferred to memory. The Linux kernel creates a structure in memory (called a sk_buff) which has pointers to the newly received digital transmission now in memory. Each transmission is a packet â a part of the original transmitted data with the headers attached. Each TCP packet consist of three different types of headers (Ethernet, IP, TCP) and the data . The need for TCP and other transport layer protocols is to allow the partitioning of data.

If a user is downloads a modestly size file, say the latest UML root_fs from sourceforge (the file would be roughly 16 megs). Transferring all this file in one big packet (one set of headers) would be cumbersome. If another user else wants to use the bandwidth, they would have to wait until the transmission of 16 megabytes is complete. With a transport layer protocol, like TCP, the users share bandwidth.

The data is partition into small packets (roughly 1500 bits a piece) each having their own set of headers. The TCP protocol was developed for the department of defense. If a network connection was broke and other routes were available, the department of defense wanted a way to ensure the data's arrival. These headers tell the packet where to go and ensures the packet gets there. This is what the Ethernet and IP headers do. The Ethernet and IP headers contain sets of address (kind of like street address) to tell the packet where to go.

he data packets are divided into pieces, and the Internet is not all the same bandwidth and reliability. Some packets will arrived mixed up, corrupted, twice or not at all. The TCP headers contain information to manage the arrival of the packets and makes sure the application receives the data in order.

C.2 TCP Header and Sk_buff

Fig. C.1 shows the TCP header and Fig. illistrates the TCP header structure in the Linux code. The total length of the TCP header is 20 bytes; four bytes for each row. Eight bits equal one byte; the TCP header is 160 bits. Of those 160 bits, 64 bits are used for the sequence and acknowledgment numbers (32 bits for the sequence and 32 bits for the acknowledgment number). The Seq and Ack numbers are how the TCP keeps packets in order.

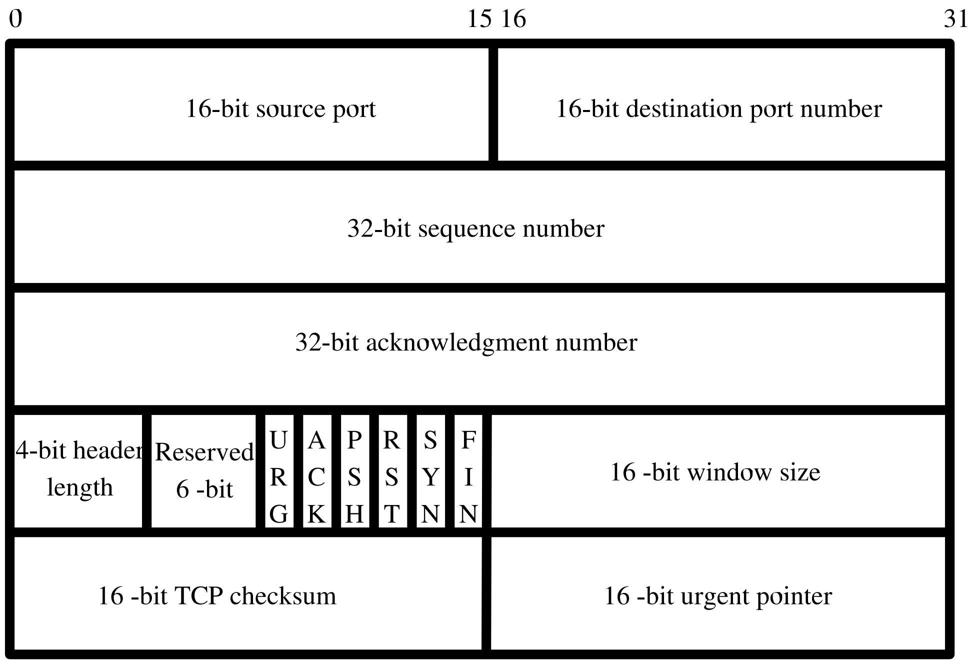


Fig. C.1. Tcp header

The header has all sorts of information. In the kernel, TCP header is accessed by a pointer in the sk_buff (stands for socket buffer). The sk_buff is the most important structure in the networking of the kernel because it references the incoming network packet and a cental hub for routing information. In Fig. C.2 is a code sniped of the sk_buff:

```
struct sk_buff {
 /* These two members must be first. */
 struct sk_buff * next; /* Next buffer in list */
 struct sk_buff * prev; /* Previous buffer in list */
 struct sk_buff_head * list; /* List we are on */
 struct sock *sk; /* Socket we are owned by */
 struct timeval stamp; /* Time we arrived */
 struct net_device *dev; /* Device we arrived on/are leaving by */
 /* Transport layer header */
 union {
   struct tcphdr *th;
   struct udphdr *uh;
   struct icmphdr *icmph;
   struct igmphdr *igmph;
   struct iphdr *ipiph;
   struct spxhdr *spxh;
   unsigned char *raw;
 } h;
 /* Network layer header */
 union {
   struct iphdr *iph;
   struct ipv6hdr *ipv6h;
   struct arphdr *arph;
   struct ipxhdr *ipxh;
   unsigned char *raw;
 } nh;
 /* Link layer header */
 union {
   struct ethhdr *ethernet;
   unsigned char *raw;
 } mac;
 struct dst_entry *dst;
```

Fig. C.2. header section of the sk_buff

```
struct tcphdr {
  __u16 source;
  __u16 dest;
  __u32 seq;
  __u32 ack_seq;
  #if defined(__LITTLE_ENDIAN_BITFIELD)
  __u16 res1:4,
  doff:4,
  fin:1,
  syn:1,
  rst:1,
  psh:1,
  ack:1,
  urg:1,
  ece:1,
  cwr:1;
  #elif defined(__BIG_ENDIAN_BITFIELD)
  __u16 doff:4,
  res1:4,
  cwr:1,
  ece:1,
  urg:1,
  ack:1,
  psh:1,
  rst:1,
  syn:1,
  fin:1;
  #else
  #error "Adjust your <asm/byteorder.h> defines"
  #endif
  __u16 window;
  __u16 check;
  __u16 urg_ptr;
};
```

Fig. C.3. IP tunnel error messages

C.3 Establishing a TCP connection

The sequence and acknowledgment numbers mentioned earlier manages the arrival and order of the packets. The Linux kernel labels each packet that is sent with a beginning sequence number on each side. This sequence number is incremented by a known value. If the packets become out of sequence, the receiver can tell by the sequence number.

The initial sequence number is determined by the sender. It creates a packet (sk_buff) and sends it down the protocol stack and out to the Internet. The TCP header has a section of one bit FLAGS. There are six flags: URG, ACK, PSH, RST, SYN, and FIN. The initial packet send by the sender not only has the initial sequence number, but also has the SYN flag set. When the receiver gets the initial packet. It processes the packet and sends its own sequence number and places the just received sequence number in the acknowledgment field. The sender then sets two flags in the TCP header: SYN and ACK.

How the INET starts the connection is discussed in Section D.2 and a back trace is done in Tables D.5 and D.7.

When the receiver receives the SYN and ACK packet. It sends a packet with the ACK flag marked to acknowledge the SYN/ACK packet. Fig. C.4 is a finite state diagram of the TCP establishing a connection.

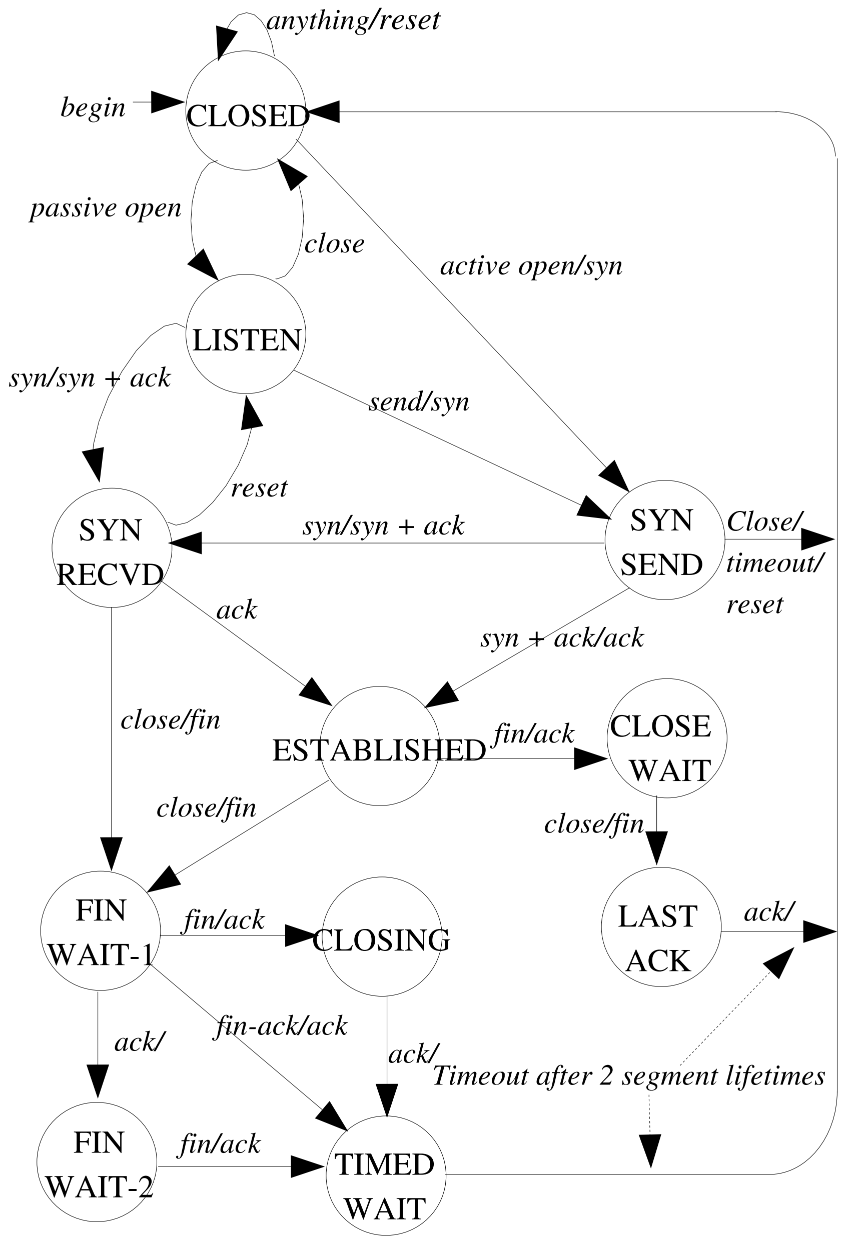


Fig. C.4. Tcp state diagram

This complex flowchart is intimating at first. This diagram is from the RFC for the TCP protocol[25]. The next section goes through this diagram and talks about the first three packets used to establish a connection, called the three-way handshake. The example will be a simple web request.

The three-way hand shake is used to establish the connect between two computers. The finite state diagram start at the âbeginâ label. The finite state diagram is for each side of the connection;

one state diagram represents the sender and the other represents the receiver. For a connection there are two starting positions on the finite state diagram for each side of the connection shown in Fig. C.5.

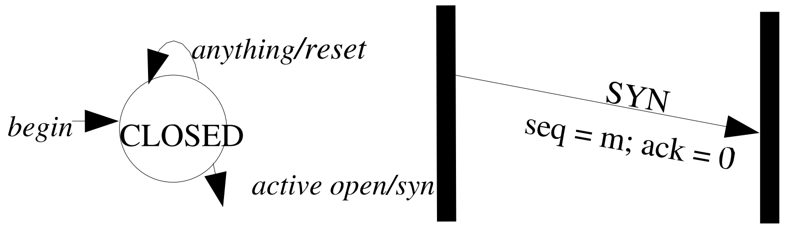


Fig. C.5. Sending the initial syn message

Starting with the sending side, the "begin" label at the upper left. The TCP socket is in closed state. The user (at the computer) decides to open a web page, send some email, telnet, or something requiring a TCP connection. The socket receives a system call from the user level[26]. The socket then creates a packet and sets the SYN flag. From the âCLOSEDâ state we move to the âSYN SENTâ state since a packet with a SYN flag was sent. The flags in the TCP header correlate the the SYN, ACK, FIN, and RST shown on the diagram. When a packet is sent with one of these flags. Traverse the state diagram from one state to another based on the flags on the packets send. Also populate the 32-bit sequence number field with an 'm' value (any value between 0-2^32).

A SYN packet is sent and the connection is in the SYN SENT state. From the line connecting the CLOSED and the SYN SENT state there is a label with two words "active open / syn". The right side of the front slash is what the sender is sending. The left side is either a state, in this case active open, or what was received from the other machine. Now the sender is sitting at the SYN SEND state. It receives a packet with the SYN ACK flags set in the TCP header from the receiver. The sequence number we sent is now in the 32-bit acknowledge field with 1 added to 'm' (m +1). The socket is following our state diagram too (or programmed to) and then sends a packet with an ACK message. Now on the finite state diagram follows the line labeled "SYN + ACK/ ACK" is at ESTABLISHED as shown in Fig. C.6. The connection is established.

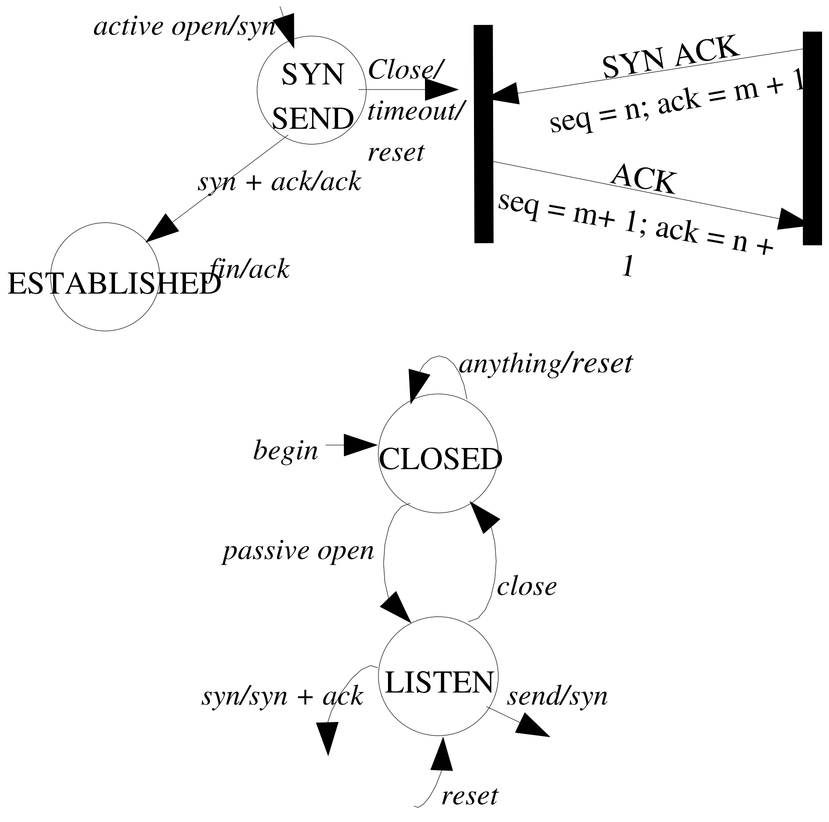


Fig. C.6. SynAck message

The receiver starts the same place, the beginning label. Now the receiver needs to be in a listening state. The services (Apache, email, or telnet) are communicating to the kernel space through a socket. This socket needs to bound (this is where BIND comes in) to a port number. So when a packet with the correct port number comes in knows where to go. If the TCP socket is not listening for this packet, it just zooms right by into the bit bucket. When the TCP socket is listening for the packet, it receives the packet and sends it up to the application (or service).

Main point is the socket needs to be in a listening state to receive packets going to a specific application. So from the CLOSED state, the socket does a âpassive openâ by order of a system call originating from the applications. So the TCP state moves from CLOSED state to LISTEN state by a “passive open”. The socket is now listening for any packet that comes in with the correct port number.

When the receiver gets the initial packet with the SYN flag set. It generates its own sequence number and places the received sequence number in the acknowledge field. The socket then sets the

SYN ACK flags and sends the packets. Now the TCP socket moves to the SYN RECVD state as shown in Fig. C.7.

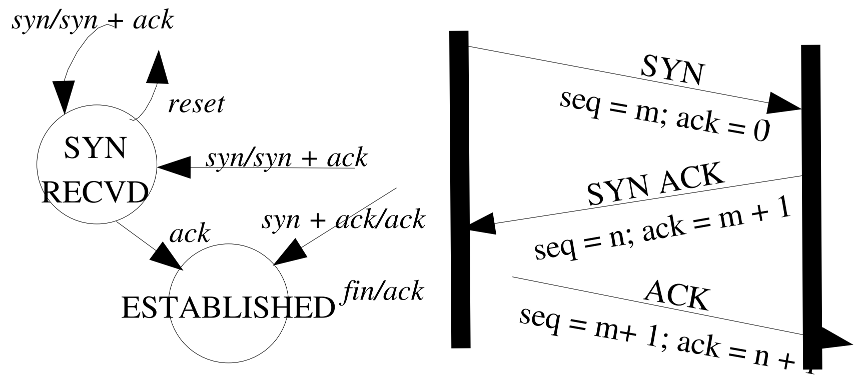


Fig. C.7. Three way handshake moving to an established connection state

The sender will then send the receiver an ACK packet saying it has received the SYN ACK packet. The sender's socket then moves to the ESTABLISHED state. These three packets are used for establishing a connection, for the sender's and receiver's TCP sockets to exchange sequence numbers and prepare for the actual data transfer.

C.3.1 Inet code processes SYN and SYN ACK

In the INET code, the INET socket on the server side is already created and the socket is in a listen state. When the socket is put into connection another INET socket is forked and placed in listening mode.

Function	Description
tcp_v4_rcv	all packets are received at this function
tcp_v4_do_rcv	starts the processing
tcp_rcv_state_process	NET socket state (sock->state) is set to TCP_LISTEN. processes the state, checks to see if the socket is in a listening state, then checks the header flag (done in a case statement).
tcp_v4_conn_request	requests a connection. Populates the dst_entry with the header
tcp_v4_send_synack()	sends the synack packet. Calls the correct functions to build a TCP/IP packet (i.e., ip_build_and_send_pkt). Goes through the correct channels to send a packet, calls ip_output –> ip_finish_output2 –> eventually dev_queue_xmit

Table C.4. Backtrace of a syn packet arriving

Looking in Fig. C.4, the packet comes into tcp_v4_rcv and eventually tcp_rcv_state_proces where the packet is sorted to the correction function based on the socket in the TCP_LISTEN state and the SYN flag being set. tcp_ve_send_synack( ) is called and the synack is sent to the client.

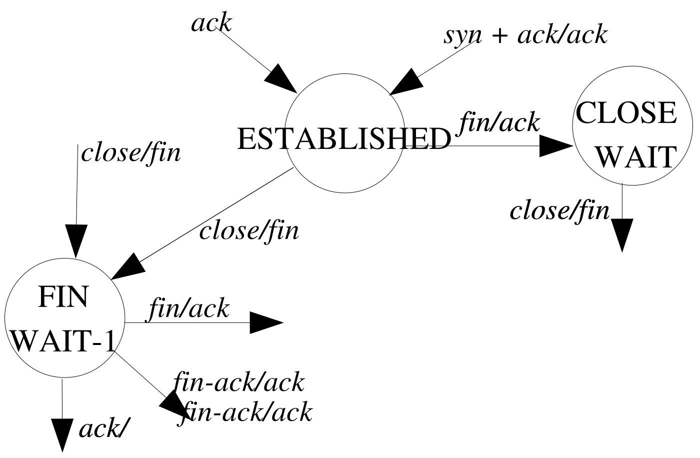
When the client receives the synack, as shown in Fig. C.6, tcp_rcv_state_process again sorts the packet based on the the state, TCP_LISTEN. An acknowledgment packet is sent via tcp_send_ack () and the state is moved from TCP_LISTEN to TCP_ESTABLISHED.

Function	Description
tcp_v4_do_rcv	Calls tcp_rcv_state_process
tcp_rcv_state_process	The state on the INET socket (sock->state) is set to TCP_SYN_SENT and calls tcp_rcv_state_process. The timer is reset (tcp_reset_xmit_timer), sequence numbers are set, the state is moved to Established in the state diagram (tcp_set_state(sk, TCP_ESTABLISHED)), the window is set(tp->snd_wnd = ntohs(th->window)), an Ack message is sent back (tcp_send_ack)

Table C.6. Backtrace of a syn ack packet arriving

C.4 Connection established

The transfer of requests and data between the server and client occurs after the connection is established as shown in Fig. C.8.



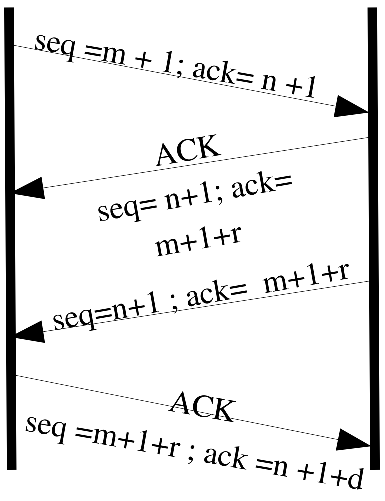


Fig. C.8. Packet transmission

Now both the sender and receiver are on the ESTABLISHED state. When the sender sends a request and the receiver responds. The state does not change until the connection is being closed.

This example will use web requests for the packet transfer. The request and the web data is not more than 1500 bytes each (roughly the maximum number of bits a packet can send at any one time). So the webpage requested does not have any packets and for this example, and the packet will not be fragmented and reassembled. The sender (web client) sends one request and the receiver (web server) sends just one packet of web information back.

When the sender sends the request. It uses its sequence number (which is returned in the acknowledge, incremented by one). Our sequence number is (m + 1), m being the original number sequence number.

With the exception of sending data, as a general rule, add one to the sequence number to packets with the SYN, both [SYN ACK], FIN, and [FIN ACK] flags set. This means the only time not to add one to the incoming sequence number is when the ACK flag is set. Fairly simple. Side note, almost half the packets have only the ACK field set, the other half add one too the incoming sequence number. Generally adding one to the sequence number and then placing it in the acknowledge field tells the other side, a packet has arrive. The adding one to the incoming sequence (outgoing acknowledge) is used by the socket to keep track that the packet has been received.

Here is another example up in the diagrams. Sender sends a packet with sequence number 1000 and acknowledge of 3000. Receiver receives the packets and takes the sequence number 1000 and places it in the acknowledge field and adds one, becomes 1001. The acknowledge field is now the sequence field, so the receiver is sending the sender his sequence number of 3000 (untouched) and an acknowledge of 1001. This is how the acknowledge and sequence numbers work for setting up and closing a connection. For sending and receiving data requests, the TCP does something a little different to change the incoming sequence number.

Sending and receiving data the TCP uses the size of the data (or the size of the payload) to increment the sequence number. My guess is this is to save a spot on payload size and just put it in the sequence number, considering there is not such field for payload size. The incoming sequence number is updated based on the size of the data payload. Below, there is a more extensive picture of the TCP header with where the payload would be located. Generally, TCP packets are setup as: Ethernet header, IP header, TCP header, then data payload as shown in Fig. C.9.

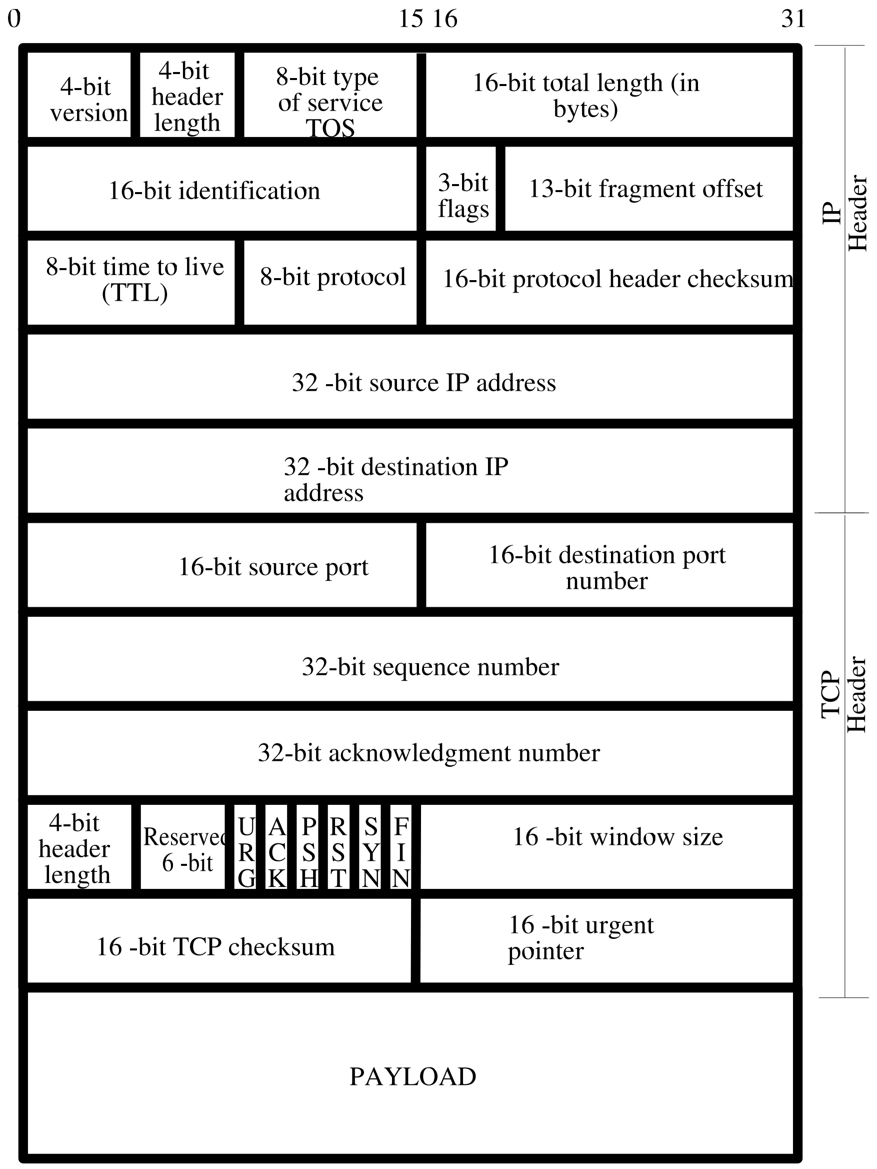


Fig. C.9. More extensive header of the ip, tcp, and payload

The data payload is where the server to client data request and data responses are kept. Again, the three way handshake (three packets opening a connection) and four way close do not contain payloads; these packets are used merely for establishing and closing a connection.

A quick example of how the incoming sequence number is updated. The sender sends a web request. A typical command would be, “GET / HTTP/1.0\r\n” to request a webpage. The size of the request is 190 byes (actual size of a request taken from an ethereal capture of the network driver). Again, this web request is going to be in the payload. Let us continue with the example above and start with a sequence number of 1001 and an acknowledge number of 3001. When the web request is received by the sender. The receiver looks at the size of the payload (web request). The payload is 190 bytes. The receiver adds 190 to the incoming sequence number, sets the ACK flag, and places the incoming sequence number in the outgoing acknowledgment making it 1192 with an outgoing sequence number of 3001 (untouched).

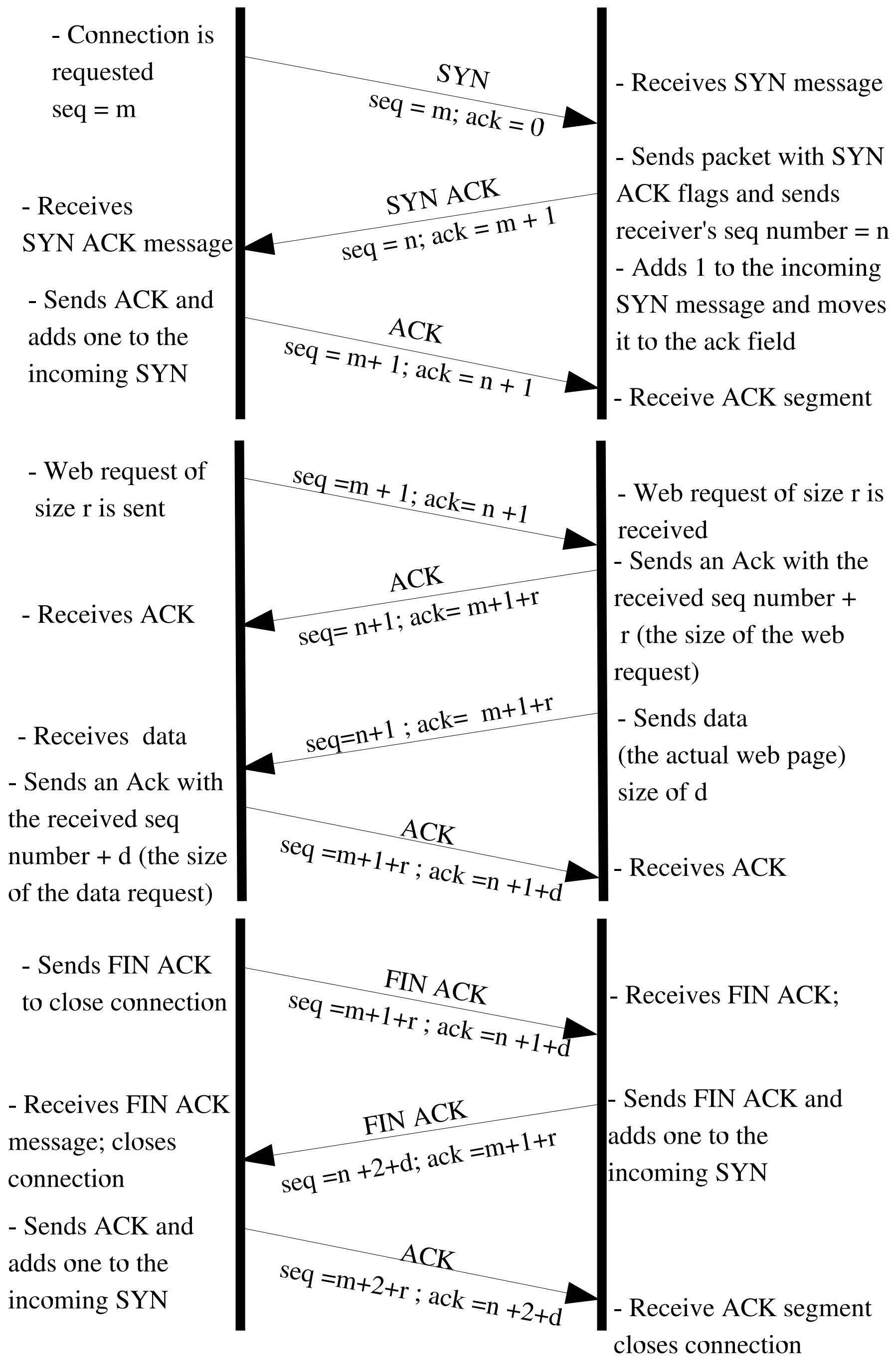

- Connection is requested seq = m
SYN
seq = m; ack = 0
- Receives SYN message
- Sends packet with SYN ACK flags and sends receiver's seq number = n
- Adds 1 to the incoming SYN message and moves it to the ack field
SYN ACK
seq = n; ack = m + 1
- Receives SYN ACK message
- Sends ACK and adds one to the incoming SYN
ACK
seq = m+ 1; ack = n + 1
- Receive ACK segment
- Web request of size r is sent
seq =m + 1; ack= n +1
- Web request of size r is received
- Sends an Ack with the received seq number + r (the size of the web request)
ACK
seq= n+1; ack= m+1+r
- Receives ACK
seq=n+1 ; ack= m+1+r
- Sends data (the actual web page) size of d
- Receives data
- Sends an Ack with the received seq number + d (the size of the data request)
ACK
seq =m+1+r ; ack =n +1+d
- Receives ACK
- Sends FIN ACK to close connection
FIN ACK
seq =m+1+r ; ack =n +1+d
- Receives FIN ACK;
- Sends FIN ACK and adds one to the incoming SYN
FIN ACK
seq =n +2+d; ack =m+1+r
- Receives FIN ACK message; closes connection
- Sends ACK and adds one to the incoming SYN
ACK
seq =m+2+r ; ack =n +2+d
- Receive ACK segment closes connection

Fig. C.10. Transfer of packets showing the syn and ack numbers changing

The sender will then send the request up to userland and the application (in our case a web server) will process the request. Make a system call to send data out as a packet. The sender then sends another packet (two in a row) with the data requested (webpage). For this example, the webpage is only 598 byes (from an ethereal network driver capture) and is only one packet since the entire size of the packet containing the data requested is under 1500 bytes. The sender sends the webpage data. The PSH and ACK flags are set.

The requested data is received by the receiver. The receiver then looks at the size of the payload (size of the webpage data) which is 598 bytes. The incoming sequence number is 3001. The sender takes the incoming sequence number and adds 598 for a total of 3599 and sets the ACK flag with an acknowledge field of 3599. The example can be illistrated in Fig. C.10.

There are a minimum of four packets transferred when requesting a webpage, and a minimum of ten packets including the TCP overhead packets of opening a closing a connection in Linux. Sending the actual request requires four packets: two ACK's, one request, and one data packets. A good rule of thumb is: the only time the packet are updated (in a simple transfer) is when something other than an ACK (only ACK) packet is received. If the packet is received to setup a connection or close a connection. Only one is added the the incoming sequence number than transferred to the outgoing ACK packet in the acknowledgment field. For data, the size of the payload (either the data request or the actual data) is increment to the incoming sequence number and then placed in the acknowledgment field of the outgoing ACK packet.

C.4.1 Inet transferring data

Sending a request or data is talked about in Sections D.3.2 and D.2.

When the request or data is received on the other side the packet is retrieved with a soft interrupt is fire (do_softirq). The packet is placed on the backlog and processed (process_backlog). In the netif_receive_skb, the packet is tagged for the correct protocol and ip_rcv is called (since the packet is tcp_ip). Ip_rcv assigns the correct IP version and points the iphdr (IP header pointer) to the correct spot in memory and validates the checksum. ip_rcv_finish creates the correct dst_entry, and the ip_option is populated. ip_local_deliver_finish determines the transport and calls the TCP protocol. tcp_v4_rcv sets the cache buffer on the skb, validates the checksum, figures out which INET socket, and sends up to tcp_v4_do_rcv for processing. tcp_v4_do_rcv mainly calls tcp_rcv_establish (sends acks) and tcp_rcv_state_process (tcp engine manager and sends packets up to the user). This is also talked about in Section D.3.3.

Aftering being processed by the tcp, tcp_data_queue sends packets up to the user by placing them in a queue. An interrupt is called and wakes up sock_recvmsg. The data is sent up to the application layer through (assumming) sock_read and system_read.

C.5 Closing the Connection

Closing a connection entails three packets being sent. This part the INET code did not follow standards, because it does not match what the incoming packets suggest. Below is a modified finite state diagram with how the INET does this process. This section is going to talk about what the INET does oppose to the actual published way of closing a connection.

There are thee packets involved (published way there are 4). The sender sends a packet with the FIN and ACK flags set. Data transfer is done so there is no data in these packets. Once the receiver receives the FIN ACK packet, the receiver adds one to the incoming sequence number and sends and with the incremented incoming sequence number in the outgoing acknowledgment field, also sets the FIN ACK flags on the outgoing packet.

Once the receiver's FIN ACK packet is received by the sender. The receiver increments the incoming sequence number by one and sets the outgoing packet with the just the ACK flag. The sequence and acknowledgment numbers are switched and the packet is sent. The connection on the sender's side is then closed to the receiver.

Once the receiver gets the last ACK packet. The connection is closed on both sides. So the transfer of data through sockets can be partitioned into three parts: opening the connection, transferring the data, and closing the connection.

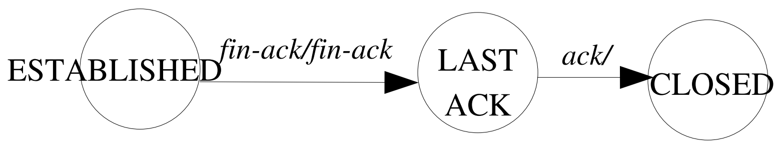


Fig. C.11. Inet's modified close

C.5.1 Inet closing a connection

The backlog suggests, the application receives all the data it needs and does a system call (sys_close) to close the socket. The sys_close function calls sock_close, sock_release, and inet_release (releases the INET socket). tcp_close is called and sends the FIN packet (tcp_send_fin).

When the server receives the FIN packet and processes the packet in tcp_rcv_state_process where the the application on the server side closes the connection.

C.6 Window

Up to this point, there has been a minimum of ten packets fired between the sender and receiver for one packet of information. About half the packets are acknowledges stating the packet has been received. The connection stays open for a set time called “Keep alive”. The INET code has timers. The finite state diagram indicates the state can timeout and reset. Once a packet is fired (such as a SYN packet or a web request) a timer inside the INET code starts which schedules for the packet to be resent in the future. If the packet arrives the socket cancels the scheduled event.

The timers and the sequence in which packets arrive are all part of a bigger concept called “flow control”. Flow control makes sure the packets arrive at a given rate but also in the correct order, removing any duplicates and removing any erroneous packets. The sequence and acknowledge numbers are used for keeping the packets in order. Another field called the window. The window tells the other side how many packets that can be sent before an ACK packet is sent. For example, our small webpage request only had one packet. Assume another download of a bigger webpage. When the connection is established (three-way handshake), the window is sent in the SYN ACK. The window is the amount of data that are allowed to be sent before receiving an acknowledgment. The INET has mechnisms built into it to figure out how many packets and amount of information the computer can handle before problems occur. The sender or client is the one who sets up the initial window. The window is used for flow control. It controls the amount of data that can be sent by both sides. Fig. C.12 is a small diagram to illustrate.

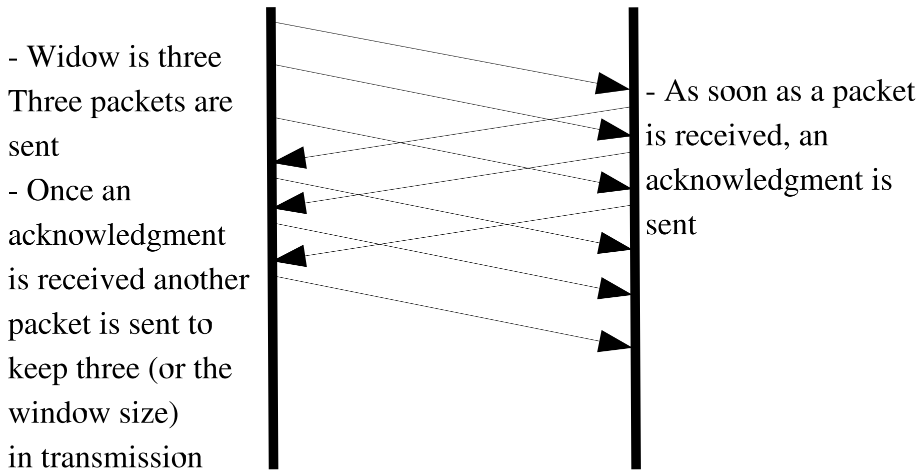

- Widow is three
Three packets are sent
- Once an acknowledgment is received another packet is sent to keep three (or the window size) in transmission
- As soon as a packet is received, an acknowledgment is sent

Fig. C.12. Window flow contro

In Fig. C.12, the window is three packets. Only three packets can be sent before receiving an acknowledgment. When the acknowledgment for one of the packets is received another packet can be sent. If the window is three, only three packets can be sent without having an acknowledgment.

C.7 Checksum

The previous sections talked about how the connections are opened and closed and how flow control is done. This section talks about assuring the accuracy of the TCP header. There is another field called the checksum. In Fig. C.13 is a diagram of the one's complement addition of the fields in the TCP header[27].

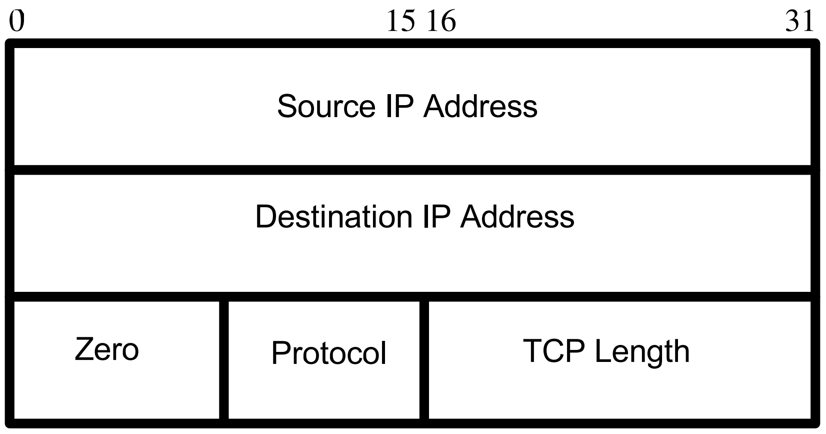


Fig. C.13. Checksum fields

The checksum is calculated with one's complement sum of the source IP, destination, zero, protocol, and the TCP length. When the packet arrives the same computation verifies the packet arrived in tact.

Appendix D

Comments on TCP structures

This next section is about the different structures that make up the Linux kernel's networking and will be touching on the most important structures. Major points about the networking section in the Linux code can be covered with the sk_buff, and sockets. This next section is mainly a reference describing the different structures. This information would assist in creating a new protocol or changing the TCP/IP (INET implementation) for research needs.

D.1 sk_buff

The sk_buff contains all the information the packet needs to get to the destination. ip_output.c:ip_queue_xmit() is the last place in the call stack the inet socket is used to give extensive information about the packet then the sk_buff is the only parameter (and maybe the routing tables) passed through the call stack to the network device[28, 29, 30].

For the TCP, the sk_buff is created for outbound packets in tcp_output.c:tcp_v4_connect() and released at net/core/dev.c:dev_queue_xmit(). For the incoming packet the sk_buff is created in the process_backlog, one reference was not entirely sure where the incoming packet is created and believed the packet was created in dev_alloc_skb ( )[31].

The sk_buff partitions into six different parts: the link list, packet header, routing, control buffer, stats, and markers.

D.1.1 Link list

The sk_buff is a structure attached to other sk_buff's in a linked list structure and has pointers to control it location, having pointers to the head, next, and prev. This link list contains other sk_buff structures heading to the same destination generally going to the common router gateway. This linked list is a little different.

n each sk_buff in the link list, there are two additional pointers to a network device (*dev) and to the inet socket (* sk). In addition to being double linked to each other with pointers to the head and tail, there are also pointers pointing to the device driver and the socket. Now, when the sk_buff is being populated with packet information it is just pointing to the socket. As the routing information is calculated, the device pointer is populated. At one point, every sk_buff in the link list is pointing to both a device and socket. And, as the packets are about to be setup to hardware, the socket pointer is set to null and only pointing to the device. This section of pointers groups the sk_buff based on connection and then on outbound device[32, 33].

D.1.2 Packet header

This section talks about the packet header information. The sk_buff and most of the kernel is designed for abstraction using pointer arithmetic. With many different standards, each of the protocols have to correspond by having a similar interface, so the same network mechanism works with different protocols (e.g., UDP and TCP protocols).

The sk_buff allows different protocols by using C Unions. There are three different headers (same as theOSI network stack): Transport, Network, and Link (or hardware). In the sk_buff, these headers are all pointers. A kmalloc (oppose to malloc) is called using sk_put( ). When receiving a packet, the header type is determined and the sk_buff maps a poitner to the correct packet header structure. The sk_put reserves the memory for each header. Since memory is allocated going from a top down structure, the timing of when sk_put is called is important. The IP header cannot be populate before populating the TCP header. The timing of when everything is done is just as or more important as what is populated into the sk_buff.

Networks have different type of computers on them. These computers read information differently in memory. Most of the higher end systems (newer Apples, Suns, and HP Unix) use big endian. Intel uses little endian (for performance). The big and little endians needs to be accounted for when storing data. The difference between big and little endian is four bytes: little endian has bytes in reverse (not the bits, the bytes). An IP address: 172.31.0.173 (hex: ac 1f 00 1d) would be stored as (hex: 1d 00 f1 ac). The network send data as big endian does not reverse the bytes. When creating a iphdr (IP header), there are preprocessor statements which line the flags and other fields in the right order. One important and simple concept is he difference between the actual packet in memory and the sk_buff, an independent structure pointing to memory.

sk_put ( ) reserves or expands the memory space of the packet. The sk_buff structure consists of pointers referencing the data area memory. In the Linux source code (net/ipv4/ip_output.c:ip_queue_xmit()), a call to skb_push reserves memory for the entire socket structure.

As information passes through each of the layers, additional headers are added using the skb_push function. In addition to the actual header, routing information is also stored in the dst_entry.

D.1.3 Routing

When sending a packet, the dst_entry actually dictates the header information. The dst_entry stores the destination and mac addresses for transmission. The dst_entry stands for “destination entry”. In the 2.4 kernel the dst_entry is assigned for a new connection in the tcp_ipv4.c:tcp_v4_connect()[34].

Going through each of the parameters in the call above, tt is a routing table structure, and nexthop is the gateway address. RT_CONN_FLAGS(sk) sets the connection flag, and sk->bound_dev_if is the outbound device. Generally, the packet goes to a network device (e.g., eth0). If the packet is heading towards an ip tunnel, the packet heads towards tunnel network driver (tunl1). The only thing populated from the ip_route_connect() is the routing table. The routing table is populated based off the actual IP routing table listed in the /proc/route and arp cache entry.

The dst_entry is part of the routing table with other routing information, like the gateway, routing destination, and routing source. The routing table is placed inside of the INET sock, and the sk_buff has a pointer to dst_entry set by the INET socket.

One of the entries on the dst_entry the hh (cached hardware header) determines in the ip_output.c:ip_finish_ouptp
if the packet is destined for an IP tunnel or regular transmission. The source follows a different path in ip_finish_output2 based on hh value. If the hh is set to null, the sk_buff will be directed down another source path which eventually leads to the ipip.c functions (functions for IP tunneling), otherwise the sk_buff continues neigh_resolve_output() uninterrupted.

The dst_entry also contains a pointer to the dev. The dev pointer points to the network device the packet is sent and received. The network device also contains pointers to a lot of abstract functions. These functions pointers allow for the correct function to be called providing a layer of abstraction.

Occasionally, information needs to be able to cross the OSI layers and be available for the developer to change. The next section of code, called the control buffer, deal with a buffer available for miscellaneous information storage.

D.1.4 Control buffer

The control buffer is 48 bytes and is static stored. In this statically declared memory there is the tcp_option and other values which is the engine managing the TCP layer.

The control buffer has preprocessor defined macros which allow easier assignment of the individual fields. For example, in the tcp_ipv4.c:tcp_v4_rcv there are statements like: TCP_SKB_CB(skb)->seq = ntohl(th->seq); where the TCP_SKB_CB assigns the seq in the host bit order to the control buffer. The macro TCP_SKB_CB hides all the type casting to tcp_skb_cb structure which is stored in the tcp.h. TCP uses the control buffer but not UDP. UDP not using the control buffer make sense because of the less overhead for UDP. IP uses the the control buffer by the inet_sk_param struct for the first field of the tcp_skb_cb struct. The TCPSKB_CB is used for the Ipv4 and has a separate processor condition for Ipv6 (IP version 6), if called.

Inside the inet_sk_param, the ip_option structure is stored. The ip_option structure is used in addition to the other statically values in the sk_buff to record things like the first IP hop (generally a gateway) and various flags such as: if a time stamp is needed or if strict source route is used. The inet_sk_param also contains a flag stating what type of route: masquerade, translated, or IP forwarding. The remainder of the tcp_skb_cb structure consists of the TCP related information.

There are four remaining fields for the TCP: sequence, end sequence, time stamp to help compute round trip time, and TCP header flags. The sequence number is actually populated by adding the previous sequence number and the size of the data payload.

D.1.5 Stats and Markers

The last section of the sk_buff consist of a few static variables used to store measure ment and classification information.

The remaining pointers (towards the end of the sk_buff) are pointers to different position of the outgoing packet in memory. There are four different pointers to the data: head, data, tail, and end. The head points to the beginning of the buffer, and the end points to the end of the buffer. The data points to the beginning of the data, and the tail points to the end of the data. Sometimes the tail and end pointers are different because Ethernet frames have two sections, a front and a back. The packet header is stored in the area between the head and the data pointers.

The next section will describe the socket layers. The socket layers consist of the INET socket and the BSD socket and interfaces with the user and kernel processes.

D.2 BSD and INET Sockets

One of the major differences between the INET implementation of the TCP/IP for Linux oppose to other UXIX operating systems is the use of an additional structure called the INET Socket. In this sub-section, I am going to give a quick over view of the BSD socket's use and then go into detail about the INET socket's. For INET implementations, the INET socket does most of the connection oriented work. And for opening a connection, the whole process still starts at the application layer or with the user.

D.2.1 TCP connection using sockets

The application (at the user's discretion) starts the whole process by calling the operating system through a system call. There is a whole set of system calls that are standard for Unix favor operating systems called, POSIX. The system call for starting a network connection is sys_socketcall( ) and sys_connect( ) both located in net/socket.c[35, 36]. The sys_socketcall( ) transfers information from the user and calls sys_connect ( ). sys_connect ( ) looks up the bsd_socket and sets up the bsd_socket.

The bsd_socket (or BSD socket) is common in all Unix favor operating systems. Microsoft has their own favor of the BSD sockets called winsock. The difference between the Linux socket and most other Unix favor operating systems is Linux has an additional socket, which is part of the BSD socket, called the INET socket. Again, the main difference between the BSD and INET sockets is BSD sockets interface with the user or application while the INET sockets are connection based. So within a BSD socket there is one for port 80 (default web server port) but there are multiple INET sockets handling all the connections going into port 80.

Once the BSD socket is setup. The sys_connect ( ) calls the inet_stream_connect ( ) which sets the INET socket's state, and from this point on, the INET socket is passed into the code separately from the BSD socket. The BSD looks up the INET socket using a hash table with destination address and port. Then the inet_stream_connect ( ) calls many other methods which populate the INET socket's connection information (i.e., routing information). The Fig. D.1 gives a visual representation of the different layers. For a back trace of the code, please reference Table D.7.

There are two sockets, INET and BSD. They are activated by systems calls initiated by the user or the application. Of these two sockets, INET socket has the most control over the connection.

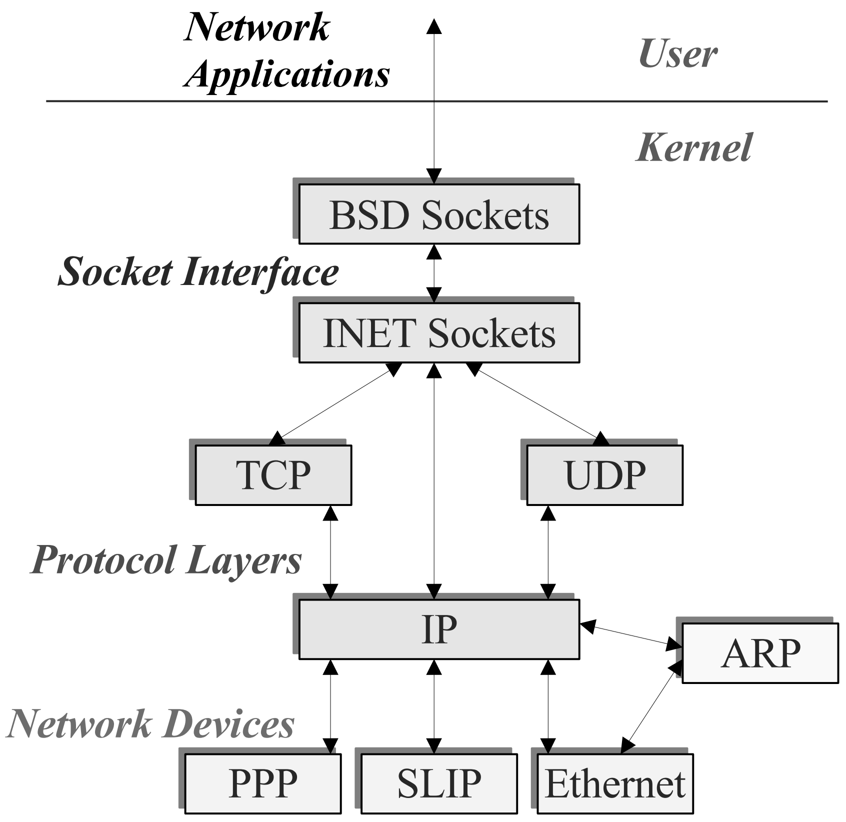


Fig. D.1. Sections of the linux networking code

The INET socket is what maintains and manages the connection. This includes keeping track of the sequence numbers in the TCP headers. There are dozens of entries in the INET socket structure and they do not partition off nicely like the sk_buff. This next section is going to cover work done with the INET source code.

The INET socket is created in sys_socketcall ( ) when it calls the sys_socket( ) and sock_create(family, type, protocol, &sock) functions.

D.2.2 INET Socket: IP routing

In Fig. D.2 is a code snip of the beginning of the INET socket structure. Four variable declarations tell the code which connection the socket belongs. This include multiple applications using the same port. For example, if there are five web browsers on one page all downloading information from a website. If the destination addresses and ports are the same, TCP is smart enough to transfer all the traffic through one connection. These connections are defined by the four variables below.

```
struct sock {
 /* Socket demultiplex comparisons on incoming packets. */
 __u32 daddr; /* Foreign IPv4 addr */
 __u32 rcv_saddr; /* Bound local IPv4 addr */
 __u16 dport; /* Destination port */
 unsigned short num; /* Local port */
 ...
 unsigned short family; /* Address family */
 ...
 struct dst_entry *dst_cache; /* Destination cache */
 ...
 __u32 saddr; /* Sending source */
```

Fig. D.2. Socket structure

The __u32 means the variable is atomic. Atomic is an operating system terms which mean the value cannot be modified unless the associated mask is checked and modified first[37]. The mask is a way to prevent deadlock. Another worthy variable to mention is family. There is a list of all the different families in include/linux/socket.h. The family variable tells the code what type of socket. AF_INET (which equals 2) is the family for the TCP/IP protocols. Having a family variable shows the pains taking detail to make the socket conform to abstract interfaces allowing for multiple of types of protocols.

The dst_cache is really a pointer to the dst_entry. This was talked about in detail in the sk_buff section above. The dst_entry in the sk_buff comes and goes with the packet. This variable in the socket is where the dst_entry (routing information) is kept for the connection. In the net/ipv4/ip_queue_xmit (where the IP header is added to the sk_buff), the sk_buff copies the dst_cache and stores it. Finally, there is the saddr (source address). This is probably a constant stored somewhere in the operating system and just assigned to the at the creation of the socket.

D.2.3 INET Socket: TCP management engine

One of the most interesting parts of the socket is how it manages the sequence numbers. Since the socket uses the object-based programming technique abstraction, different protocols can have the socket manage connection information. Under the union structure tcp-pinfo, the values which control the connection are kept. These connection values are stored in the tcp_opt called af_tcp, shown in Fig. D.3.

```
union {
 struct tcp_opt af_tcp;
 #if defined(CONFIG_IP_SCTP) || defined (CONFIG_IP_SCTP_MODULE)
 struct sctp_opt af_sctp;
 #endif
 #if defined(CONFIG_INET) || defined (CONFIG_INET_MODULE)
 struct raw_opt tp_raw4;
 #endif
 #if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
 struct raw6_opt tp_raw;
 #endif /* CONFIG_IPV6 */
 #if defined(CONFIG_SPX) || defined (CONFIG_SPX_MODULE)
 struct spx_opt af_spx;
 #endif /* CONFIG_SPX */
} tp_pinfo;struct sock {
```

Fig. D.3. Tp_pinfo union structure

The tcp_opt stands for tcp options and manages the sequence numbers and acknowledgment numbers used by the TCP protocol. In the previous appendix, I talked about these values in detail and how TCP uses them for creating a reliable connection.

The tcp options structure shown in Fig. D.4, stores many management variables which serve as the engine of the TCP connection. One variable of interest is the rcv_nxt. The rcv_nxt hold the next expected TCP sequence number. This value is used in deciding if a packet goes onto the buffer, talked about in Section 3.3. If the incoming packet's sequence number does not match what is expect. The packet is out of order. Using a buffer hides the latency of the correct packet and corrects any out of order sequence.

```
struct tcp_opt {
int tcp_header_len; /* Bytes of tcp header to send */
  __u32 pred_flags;
  __u32 rcv_nxt; /* What we want to receive next */
  __u32 snd_nxt; /* Next sequence we send */
  __u32 snd_una; /* First byte we want an ack for */
  __u32 snd_sml; /* Last byte of the most recently transmitted small packet */
  __u32 rcv_tstamp; /* timestamp of last received ACK (for keepalives) */
  __u32 lsndtime; /* timestamp of last sent data packet (for restart window) */
  ...
  /* Data for direct copy to user */
struct {
  struct sk_buff_head prequeue;
  struct task_struct *task;
  struct iovec *iov;
  int memory;
  int len;
} ucopy;

__u32 snd_wl1; /* Sequence for window update */
__u32 snd_wnd; /* The window we expect to receive */
__u32 max_window; /* Maximal window ever seen from peer */
...
__u8 reordering; /* Packet reordering metric. */
__u8 queue_shrunk; /* Write queue has been shrunk recently.*/
...
__u8 keepalive_probes; /* num of allowed keep alive probes */
unsigned int keepalive_time; /* time before keep alive takes place */
unsigned int keepalive_intvl; /* time interval between keep alive probes */
...
}
```

Fig. D.4. Abreviated version of the tcp option structure

The final two parts touchs on is the timers and the abstract function pointers. Timers are events scheduled on the Linux kernel. This is similar to simulations. When we want to determine the round trip time, the Linux kernel grabs the time stores it in a time stamp variable which a structure timeval which has second and microsecnods encapsulated. There is also a link list of timeval used for determining when the TCP connection times out. Final part of the sock structure talks about is the abstract function.

Located at the end of the sock structure, six function pointers allow interchangeable functions to be assigned to the socket. These function pointers allow functions for the UDP to be used when the INET socket is used for a UDP connection or allows for TCP functions for a TCP connection; this is all done using abstraction. The next section will talk about about a couple of kernel tricks with the Linux Kernel.

D.3 Kernel network tips and tricks

D.3.1 Changing a packets sending address

To change the address of an outing packet from the destination the user originally intended, change the nexthop value in the ip_output.c:ip_queue_xmit(struct sk_buff *skb, int ipfragok) function.

D.3.2 Sending a packet

The easiest way to send a packet is using the tcp_v4_send_ack(struct sk_buff *skb, u32 seq, u32 ack, u32 win, u32 ts) located in the tcp_ipv4.c. This function outlines a blueprint of sending a packet. If the packet starts a new connection, creating and populating the INET socket is a gigantic task. For extra reference, below is a stack trace of establishing a TCP connection from the client's side.

Function	description
dev_queue_xmit	Skb is freed.
nf_hook_slow	
arp_send	
arp_solicit	
__neigh_event_send	
neigh_resolve_output	
ip_finish_output2	hh = null determines if tunl1 or eth0 device used.
nf_hook_slow	
ip_output	Increments IP stat.
ip_queue_xmit2	Adds IP checksum; sets the sk peer and IP ID field.
nf_hook_slow	
ip_queue_xmit	Rt is copied to skb's dest_entry; IP header is built
tcp_transmit_skb	TCP header is built, tcp_otions are built/updated;
	adds TCP checksum; sets sk in skb.
tcp_connect	INET socket's destination IP/port are set.
	DST entry is created and set in socket.
inet_stream_connect	Marks the INET socket state.
	At this point the INET socket is sent apart from the BSD socket.
sys_connect	System call. Looks up and sets up the BSD socket.
sys_socketcall	Copies info from the user.

Table D.5. Backtrace of sending a packet

D.3.3 Receiving a packet

Every packet that is received through a TCP connection goes through the function tcp_v4_rcv from there the packet is processed by the TCP sections of code and is sorted based on the packet's flags and the state of the TCP connection[38].

Function	description
tcp_v4_rcv	Sets up the TCP information before sending it to be processed.
ip_local_delivery_finish	Determines the transport protocol, TCP.
nf_hook_slow	
ip_local_delivery	Reassembles IP fragments
ip_rcv_finish	Creates the correct dst_entry and ip_option structure is populated.
	Calls ip_local_delivery using an abstract function pointer.
nf_hook_slow	
ip_rcv	Assigns the correct IP version and points the iphdr to the
	correct spot in memory and validates the checksum
netif_rcveive_skb	Packet is tagged for the correct protocol, IP/TCP.
process_back_log	Retrives packet from backlog activated by soft interupt.
net_rx_action	
do_soft_irq	Main difference between 2.2 and 2.4 is
	the inclusion of a soft interupt.

Table D.7. Backtrace of receiving a packet

Appendix E

Media

E.1 DVD

The attached DVD has the following:

- Source code used for User Mode LInux (kernel source), UML patch file , and slackware root file systems.
- Copy of the template used for Lyx (Linux version) and style sheets for IEEE transactions.
- Patch file for the 2.4.26 kernel for creating a kernel capable of doing multipath routing
- Copy of the perl scripts used for testing along with the input files.

E.2 Writing tools

This thesis was written using the instructions from Dave Havatin's PhD disertation using LATEX and the frontend Lyx. Dave has excellent instructions on how to use Lyx on windows for a thesis or disertation in IEEE for UCCS. This thesis has also used NatBib to generate the biography[1] there are "official" style sheets on the Internet interfacing with a bibtex file. Bibtex files can be generated using graphical front ends like JabRef found at the sourcefore repository. The author used JabRef

[1] The library's "Guide to writing theses and diserations" says to write a computer science thesis should be in Chicago sytle. Check with your advisor. The advisor of this thesis and the graduate commitee chair (at the time of this thesis writing) both said IEEE. The graduate committee chair signs the style sheet complience form, so what the chair says is law regardless of what is in writing. This thesis was orginally written using Chicago. The thesis writer got the book and did extra steps for Chicago style to have to redo the style for IEEE.

for generating the bibtex file then grabbed the “official” IEEEtran style sheet to lower the possiblities of comments on the citation style. The figures were done in dia and open office’s diagram program (diagrams were saved in encaslated postscript).

In the option of the author, writing a thesis in Lyx and $\LaTeX$ is much easier and looks better than writing one in word or openoffice. There are better citiation tools, auto-geenration for figures, and less chances of a corrupted document.