Contents

List of Tables

List of Figures

Chapter 3

Technique for Multiple Path Routing

In this chapter we present the techniques for setting up the Linux kernel to utilize multi-path routing, where the kernel can send a packet over the direct route, or a set of alternate indirect routes. For indirect routes, the original packet will be first put in an IP over IP packet payload and sent to a proxy, which relays the packet to its final destination. For direct routes, the packet is left in its original state and is sent to the destination without going to a proxy.

Within the Linux kernel, the multi-path enhancements are able to send alternate the packets to different proxies. For example in Fig. 3.1, ten packets are for a simple web request: six packets send from the client and four return packets are from the server. If there are three proxies, the first proxy could receive two packets (packets one and seven); the second proxy receives two packets (packets three and eight) and; the third receives the remaining two packets (packets four and ten). The ten packets transmitted are spread over the three proxies. Fig. 3.2 is an illustration of Multi-path routing.

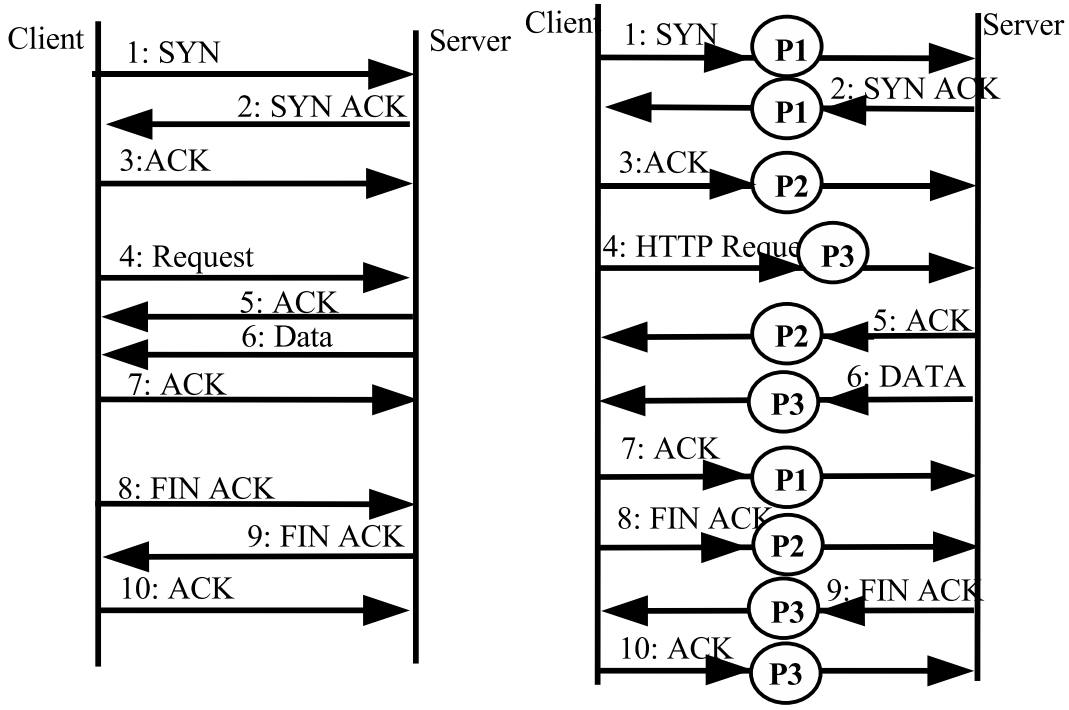

Fig. 3.1. Simple web request (left) and using multi-path routing (right)

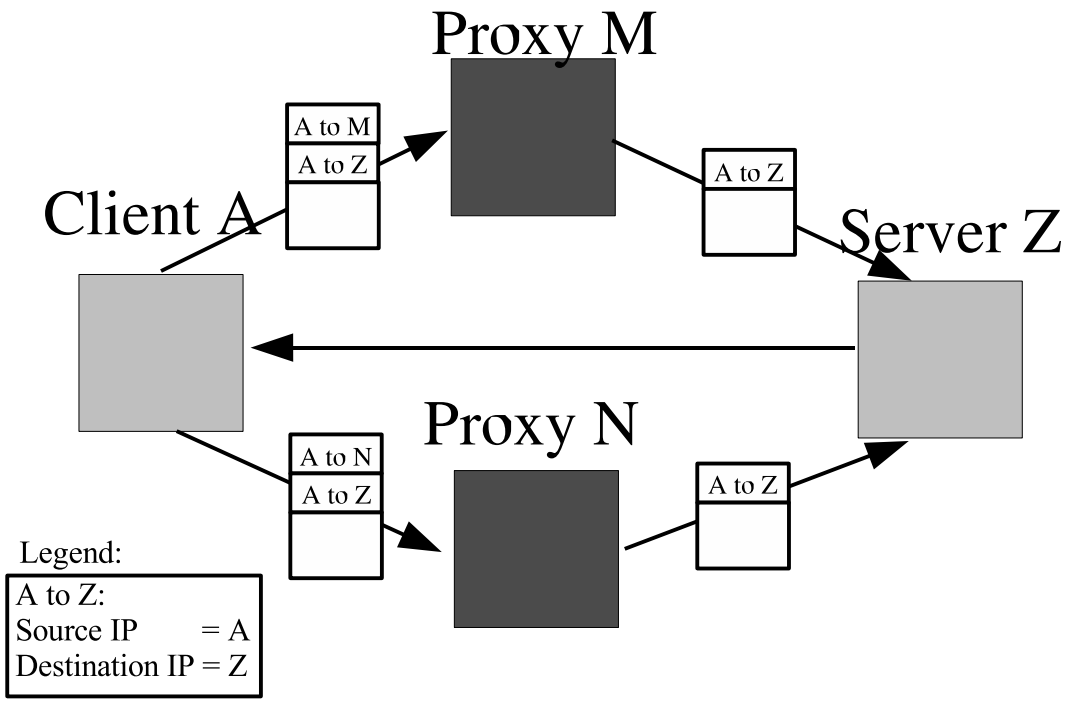

Fig. 3.2. Two two way multi-paths

The most common technique for establishing an indirect route is to apply a technique called IP tunnels where the original packet is put in the payload of an “outer” IP packet for routing and the outer destination IP address routes to the proxy. Fig. 3.3 shows an IP in IP encapsulation header used to route the packets[15]. In the multi-path routing network, there are two types of paths: one-way multi-path and two-way multi-path. In one way multi-path, one end node spreads the packets over multiple routes, the return packets are all sent through the direct routes. The other end nodes may not have knowledge that the incoming packets are relayed by a set of proxies. The proxies supporting the one-way multi-path will strip-off the outer IP header and forward the inside IP packet.

In two way multi-path, both end nodes spread packets over multiple routes. The proxies support the two-way multi-paths will also strip-off the outer IP header and forward the inside IP packet, but for both directions.

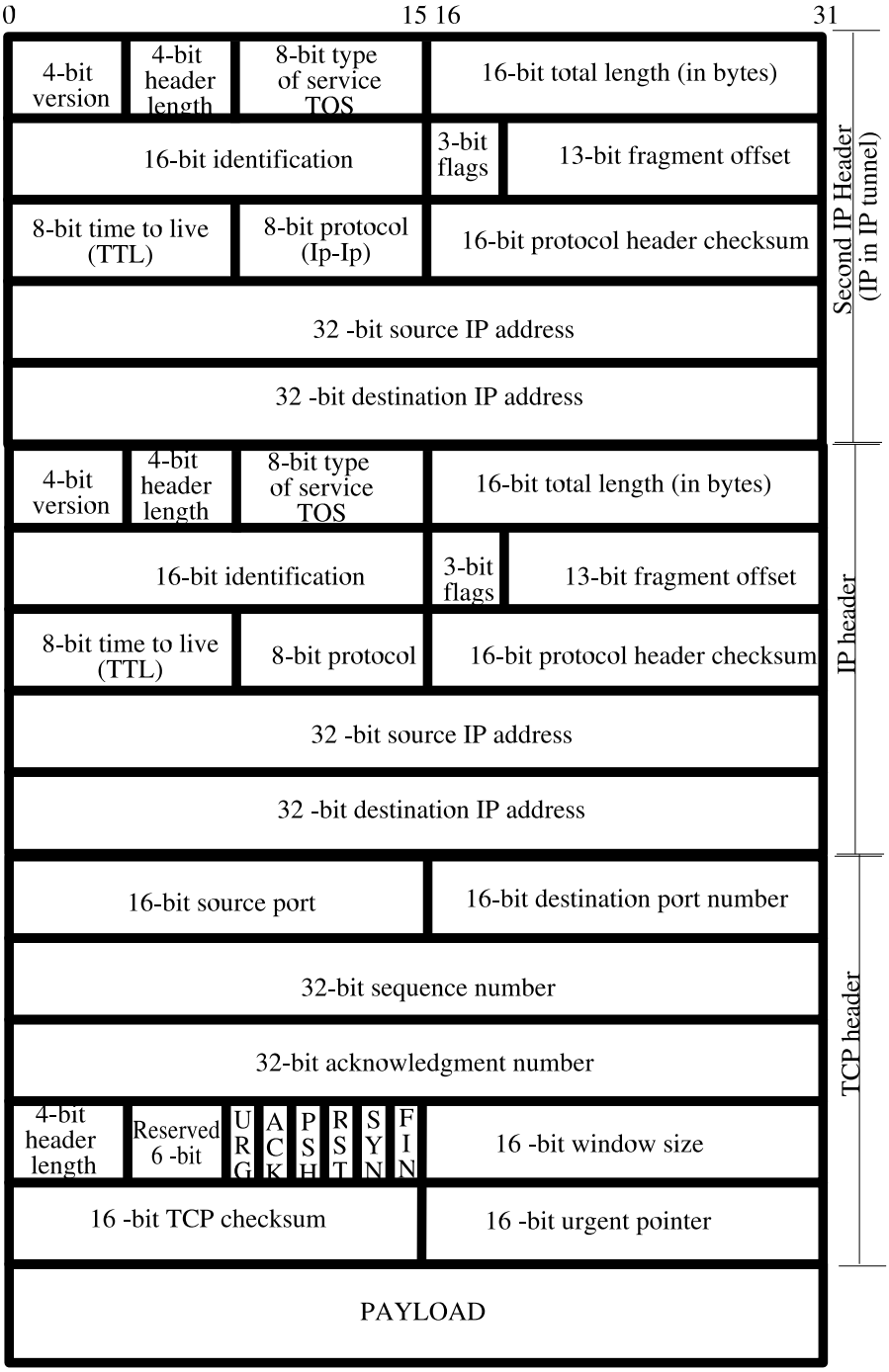

Fig. 3.3. IP in IP encapsulation[15]

Three modifications were made to the individual Linux machines and to the actual kernel to achieve multi-path routing. The changes include: creating an IP tunnel to interface with the kernel changes, modifying the destination entry's device driver in the Linux kernel to change the routing

path, 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 destination IP address, the other IP header, specifies to the proxy destination; the destination IP address of the inner or encapsulated header specifies the final destination.

The main advantage of IP tunneling is enabling of the reroute to the proxy. For this project when the packet is received by a proxy, the outer IP header is removed. A proxy removes the outer IP header and forwards the inner IP packet to the destination according to the remaining IP header. The remaining packet is repackaged with a new Ethernet frame and sent. In case of one-way multi-path routing the return packet is sent without an encapsulated IP header and bypasses the proxies. In the case of two-way multi-path routing, a new outer IP header is created with the IP address of the final destination as the destination IP address and the IP address of the proxy as the source IP address. This provides the final destination information about the proxy and can utilize this information for measuring the round trip time of the indirect path or use the proxy for the return route.

Creating an IP tunnel is not trivial. 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. 3.4 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 and running 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. When IP tunnel is not turned on

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 the adaption of the new "ip route" is is going along quickly. The "route" command has the routing information laid out differently therefore using "ip route" takes some adjustments for the user.

```
# client side
> 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
> ip route add 172.31.0.197 dev tunl1

# server side
> ip tunnel add tunl1 mode ipip remote 172.31.0.172 dev eth0
```

Fig. 3.5. IP tunnel commands

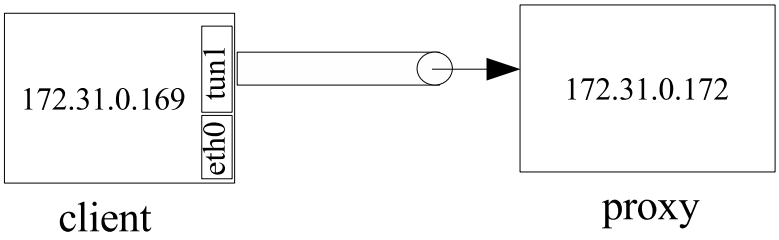

Fig. 3.6. Two computers establishing a tunnel

The commands in Fig. 3.5 are the commands for the client with (172.31.0.169 IP address) needed to setup an IP Tunnel between it and the proxy with IP address 172.31.0.172. This establishes the tunnel and creates a device driver. The commands in the bottom of Fig. 3.5 are the commands for the server with 172.31.0.172 needed to create an IP tunnel pointing to the client to be able to receive an IP in IP encapsulated packet.

On the client side, packets are sent to tunl1 and encapsulated with an additional IP header, as shown in Fig. 3.6. The routing table is updated to guide the kernel where packets are heading and if the packets need to be routed through a tunnel.

3.2 Changes in the Linux network device driver

Fig. 3.2 shows how the packets are sent to a proxy. The standard direct connections go through the device such as eth0; the packets destined for a proxy through an IP tunnel are sent to a device such as tunl1. The multiple path routing takes advantage of using the INET's underlying device drivers. The modifications to the INET code alternates the IP tunnel addresses. Each indirect route is mapped to a tunnel device. The corresponding proxy will be set up with two tunnel devices one pointed to the client; the other to the server. 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 implemented in the Linux kernel in with a scheduler handling the IP header[7].

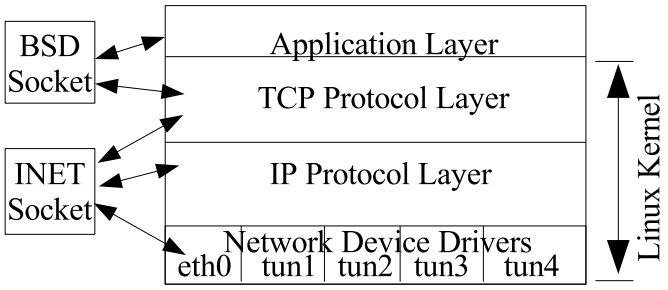

Fig. 3.7. Linux kernel with multiple network devices

Fig. 3.7 shows the Linux kernel structure with four tunnel devices. When the Linux kernel receives a packet, it is saved in a structure called sk_buff. Pointers are passed to reference the headers and payload. There is a socket for each connection called an INET socket. Another socket, called a BSD socket is used to communicate with the application process. The differences between the INET and BSD socket are detailed 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 3.8. 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.

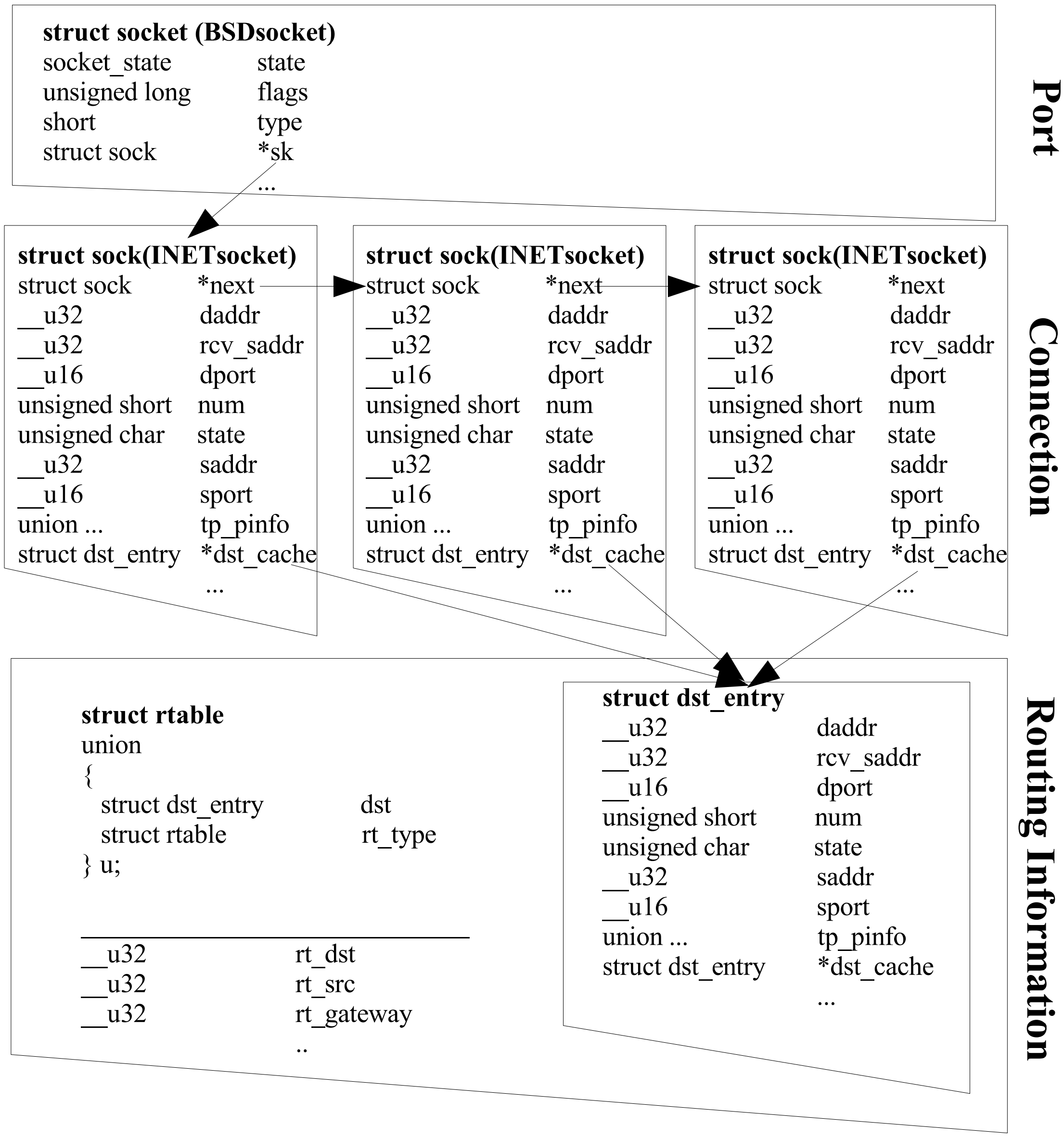

Fig. 3.8. TCP routing and socket structure

3.3 Weighted scheduler

When the dst_entry is being processed, the kernel modifications looks up all the network device drivers and creates an array of network device drivers. This array allows the scheduler to choose the path without looking up the network device driver. For this simple weight scheduler, the first tunnel driver, labeled tunl1 is considered the path with the lower ratio.

The weight for the connection establishes how many packets are allowed through a connection.

For example, there are two paths. The kernel receives a weight of 10 from the application layer; this means 10 packets would go through the network device tunl2 and one packet would go through the network device tun1.

This weight scheduler is simple and was a idea to test the validity of using a scheduler and reporting any benefits for further work.

3.4 Packet buffering

The Linux kernel stores outbound packets in a queue. The packets in the queue are sent when the device driver fires an interrupt and sends the packets. Multi-path routing uses multiple network devices and causes packets to arrive out of order. If one or multiple packets get delayed and the packet arrives out of order, the TCP protocol sends duplicate acknowledgment (dup ack) requesting a retransmission of the delayed packet. If the sender receives three dup acks, it will retransmit immediately the missing segment without waiting for the retransmission timer to expire. This is called (RTT) Fast Retransmit[16]. This poses a problem and is a major hit on performance.

With multi-path routing, we experience more fast retransmits due to a persistent unnecessary reordering of arrival packet. A solution to reduce the fast retransmits is to queue the arriving packets, wait for the delayed packet, and delay the triggering of fast transmit. Packet buffering is proposed and uses 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 with the earliest sequence number 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 structure. 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 packets. 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.

3.5 Utilizing the Proc File System

The Linux proc file system is a memory resident file system used by the Linux Kernel, and its one purpose is to exchange 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 resides entirely in memory. At startup the kernel populates the proc file system and mounts it to a directory for the user and applications to access. For example, the IP routing table uses the proc to communicate the gateway and device information needed to route incoming and outgoing packets properly. When the command "route" is executed, the "route" program displays a file in the proc file system, /proc/net/route, to the screen.

This project uses the proc file system[17] to turn off and on the multiple path routing, specify the weights of the paths for spreading packets, and control other variables used for logging and buffering. Within the Linux code, a proc directory can be created in the /proc using the command "proc_mkdir." Fig. 3.9 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);
  // bufferOn is a static int. It is initialially copied into bufferOnVal
  sprintf(bufferOnVal, "%d\n", bufferOn);
  // 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. 3.9. 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. 3.10 shows the two functions passed 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.

There is global static int variable called bufferOn. The bufferOn variable is written to the bufferOnProc->data during initialization. When the proc value changes, procWriteBuffer is called and writes the value from the user into the variable bufferOn. bufferOn is synchronized with the proc value.

```
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
  // bufferOn is a static variable and used by the kernel code
  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. 3.10. 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. 3.11

```
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. 3.11. Setting proc values

3.6 Testing using HTB bandwidth limiting

Rate limiting is a condition needed to test the multi-path routing. HTB (Hierarchical Token Buffer) is a routing and traffic control tool; [18] is an interface to the IP chains in the kernel. Of the different types of command line interfaces to IP chains, and HTB seems to be the easiest to set up. The HTB utility can be downloaded from the maintainers' website[19] and is included in newer version of Iproute2[20]. Iproute2 is the software used to setup the IP Tunnels.

The HTB 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. Qdiscs specifies tc queueing discipline realized by the queue. 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. 3.12. Setting up tc qdisc to limit bandwidth

The Fig. 3.12 shows the commands to set up the rate limiting using tc qdiscs. There are three sets of commands: set up the root for the eth0, set up each class, and setup each filter.

The tc filter command is used to classify packets from certain sources and map them into a specific queue or flow. For example, the last command in Fig. 3.12 maps packet from 128.198.60.173 to flowid 1=11 which has 683kbps bandwidth.

Finally there are additional kernel requirements when setting up the HTB utility. According the HTB utilities' website[19], the following QoS (Quality of Service) kernel options need to be set. These QoS kernel options are listed in Fig. 3.13

```
#
# 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. 3.13. Kernel options needed for tc to run

Chapter 4

Performance Analysis

Hundreds of tests were done using a Perl script written by the author. 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[21] as a web benchmark. Apache bench contains statistical information. For many tests, the webpage was retrieved up to five 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 testbed consists of 10 machines with the same configuration running Linux Fedora Core One with 600 MHz Pentium III processors, 256 megabytes of memory, and 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. These proc values were added in the /proc/sys/net/ipv4/multipath and /proc/sys/net/ipv4/multipath2 and are listed in Table A.1.

4.1 Baseline results

These are the baseline results with no bandwidth limiting, weighting different paths, or buffering incoming packets. The remainder tests deals with these different scenarios. Below in Fig 4.1 shows the four different types of connections: Direct Connect, IP Tunnel, Multiple Path Routing, and Multiple Path Routing with Packet Buffering.

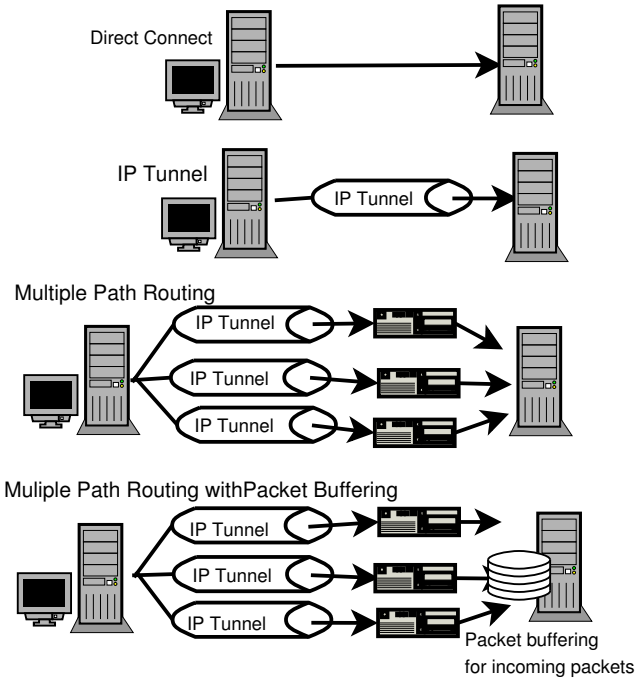

Fig. 4.1. Types of connections

Direct Connection is a connection without any modifications. IP Tunnel is a connection using packet with IP in IP encapsulation header. The first header specifies the proxy. The second header contains the destination's IP address, and the packet is routed to the destination by the proxy using the second header.

Multiple path routing uses IP tunneling and kernel modifications to spread packets over multiple paths. Multiple Path Routing with Packet Buffering has a buffer before the normal TCP code in the kernel. This buffer sorts out of sequence packets.

The results in Table 4.2 and Fig. 4.2 show the control bandwidth for the different types of connections with retrieving a 2.4 MB web document.

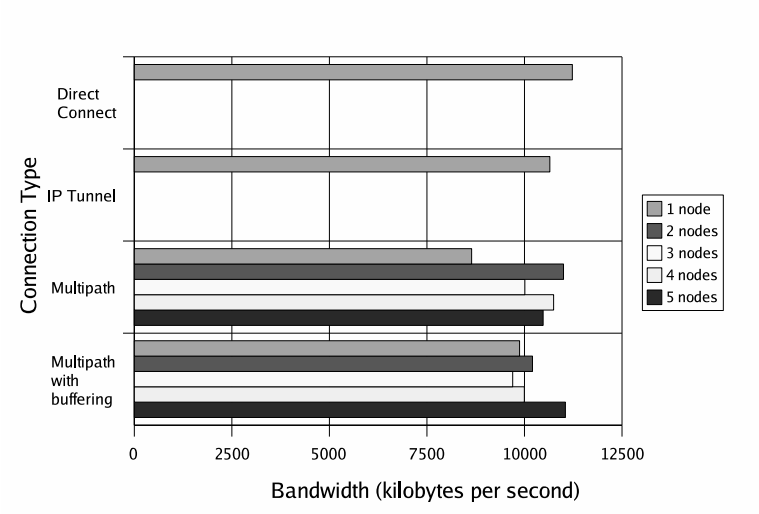

Fig. 4.2. Baseline results

The multiple path tests are done over five proxies. Direct Connection and IP Tunnel connections do not include any proxies.

	Number of proxies				
	0 or 1	2	3	4	5
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 4.1. Baseline results

The Direct Connection and the IP Tunnel were also bench-marked on other machines to verify the tests. These two additional results were added to the table but are not in Fig. 4.2. These additional baseline tests were done on Pentium III's with a faster 1 Gigahertz processors.

4.2 Impact of the number of proxies on aggregated bandwidth

The first scenario is to restrict the overall connection speed to a small bandwidth and see how the resulting 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 real world scenario because long distance Internet connections generally do not have 100 mbps bandwidth, at least not at the time of the writing of this thesis.

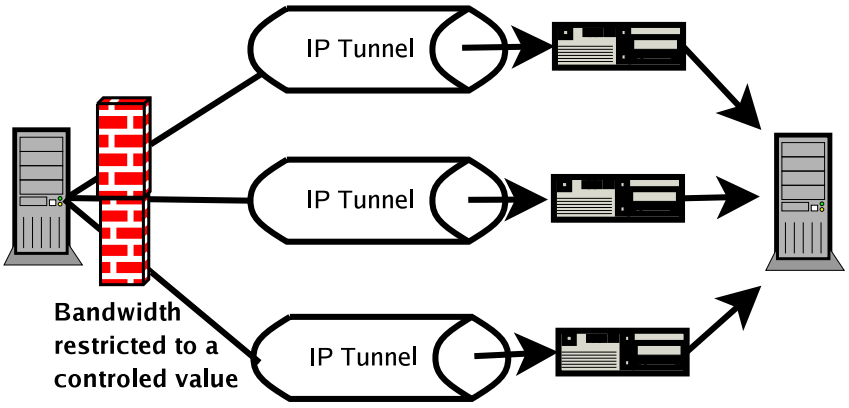

Fig. 4.3. Bandwidth limited scenario

The next set of graphs present the results based on connection types. The first two tests do not have any intermediate nodes. There are two sets of bandwidth, the variable bandwidth and the unrestricted bandwidth, in addition to the number of nodes. The bandwidth for each bar (located in the key) is rate limited bandwidth. The x-axis is the resulting bandwidth between the client and the server.

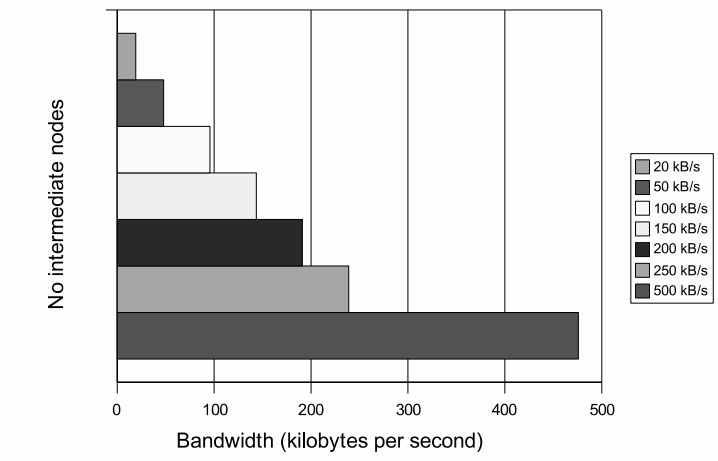

Fig. 4.4. Direct connection with bandwidth rate limiting

	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 4.2. Direct connection with bandwidth rate limiting

For the Direct Connection baseline, the resulting bandwidth shows a correlation to the bandwidth restrictions. For example, 19.16 kBps is the resulting bandwidth for a restricted connection at 20 kBps. This verifies that the HTB works with a small deviation of less than 9 percents.

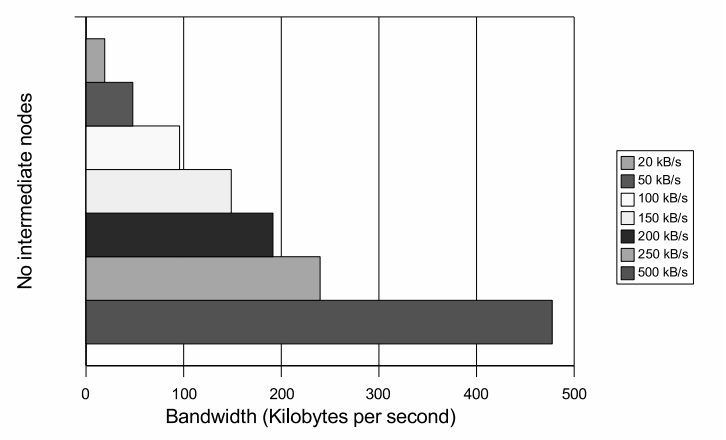

Fig. 4.5. IP Tunnel with bandwidth rate limiting

	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 4.3. IP Tunnel with bandwidth rate limiting

One observation of the results from the Direction Connection and IP Tunnel tests is that IP Tunnel does not impose a noticeable performance cost.

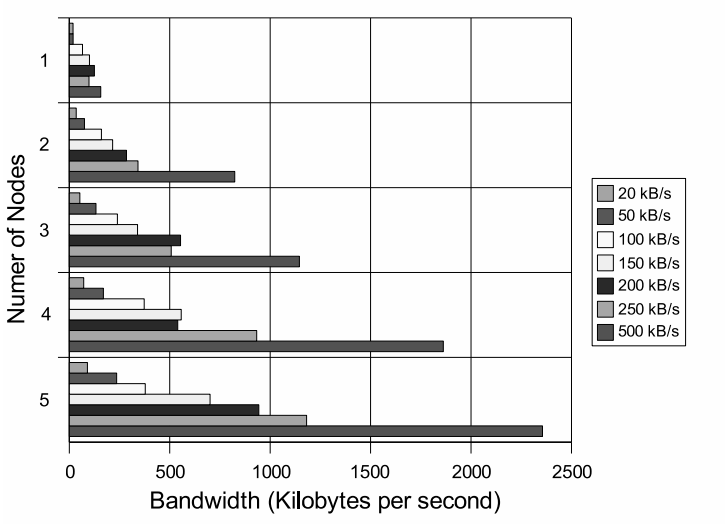

Fig. 4.6. Multiple path routing with bandwidth rate limiting

# 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 4.4. Multiple path routing with bandwidth rate limiting

One of the most important observations is with the aggregated multiple path Linux enhancement utilizing the aggregated bandwidth of the available multiple paths. For example, the aggregated bandwidth increases as additional proxies are added. A multi-path with one proxy is 18.59 kBps and that with three proxies is 52.62 kBps, almost three times the bandwidth.

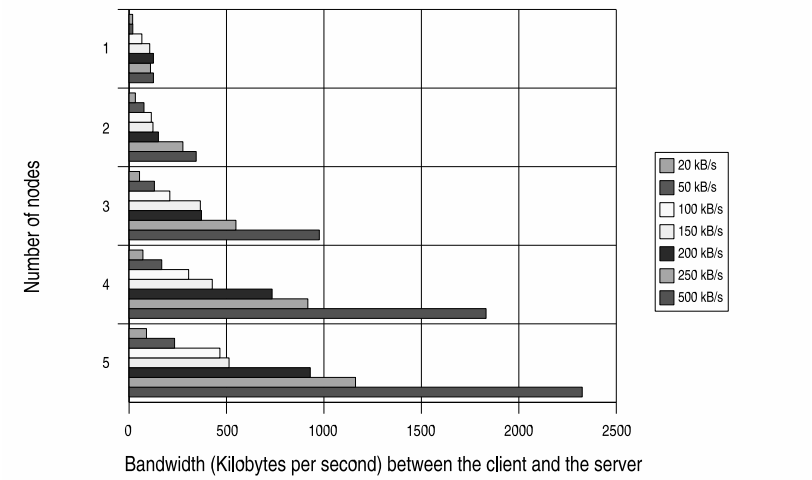

Fig. 4.7. Multiple path routing with buffer and bandwidth rate limiting

# 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 4.5. Multiple path routing with buffer and bandwidth rate limiting

The data from the multiple path with a buffer was a baffle. It is believed if a buffer was applied to the enhanced kernel, the bandwidth would increase, since multi-path routing without a buffer suffers a performance degradation caused by 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 however shows that buffering causes an additional overhead penalty and lowers the bandwidth.

4.3 Impact of Weighted scheduler on Bandwidth

These tests were done for two reasons: to see the effect of uneven bandwidth among connections and how weighted scheduling packets impacts the aggregate bandwidth. The enhanced kernel allows indirect routing and can spread the traffic according to a set of weights associated with specific connections via a proxy. A weight of 1:6 means the first proxy receives one packet while the other

proxies receive six packets of the seven packets sent. The weighted scheduling of 1:3:3 is shown in Fig. 4.8.

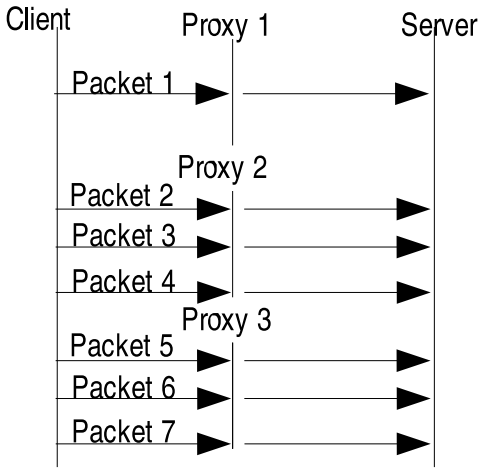

Fig. 4.8. Weighted packet distribution for a 1:6 ratio

The goal for the weighted packet scheduling is to adapt the sending of packets to the actual available bandwidth of a connection. 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

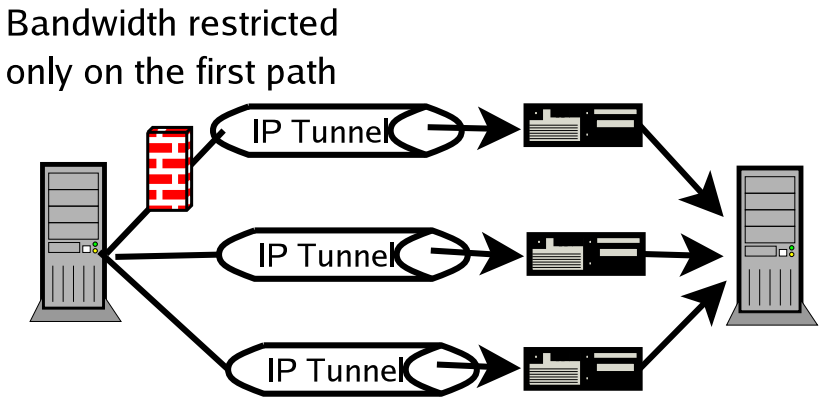

Fig. 4.9. One path with bandwidth lower then the other paths

4.3.1 Weighted paths and one path with small bandwidth

This scenario restricts the first path by a controlled bandwidth and applies a weight to the number of packets allowed for 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 proxy. The first ratio value goes to the first proxy; remaining ratio is equally given to the remaining paths. For these tests, only two and five proxies were tested. A one proxy test would be a repeat from Tables 4.4 and 4.5. Since packet buffering is also being studied, the last two tests are with packet buffering.

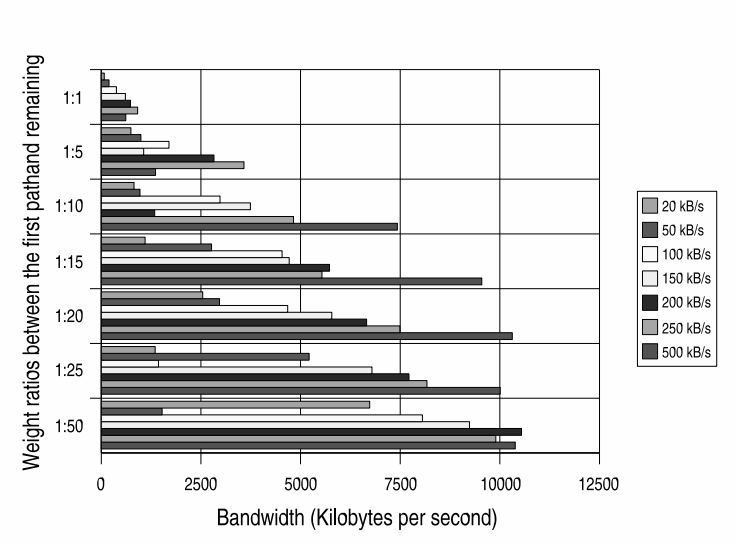

Fig. 4.10. Two nodes with no buffer and weighted packet distribution

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 4.6. Two nodes with no buffer and weighted packet distribution

Fig. 4.10 shows bandwidth increases with assigning more of packets towards a paths with more bandwidth. Fig. 4.10 has two paths. The first path has a controlled bandwidth changing from 20 kBps to 500 kBps. The remaining path is restricted by the 100 mbps router. Fig. 4.10 shows additional bandwidth gains by placing proper weight on each of the connections. These results show the potential of weighted scheduling. The scheduler could change weights based on measurement of available bandwidth of a path[1] to further increase aggregated bandwidth.

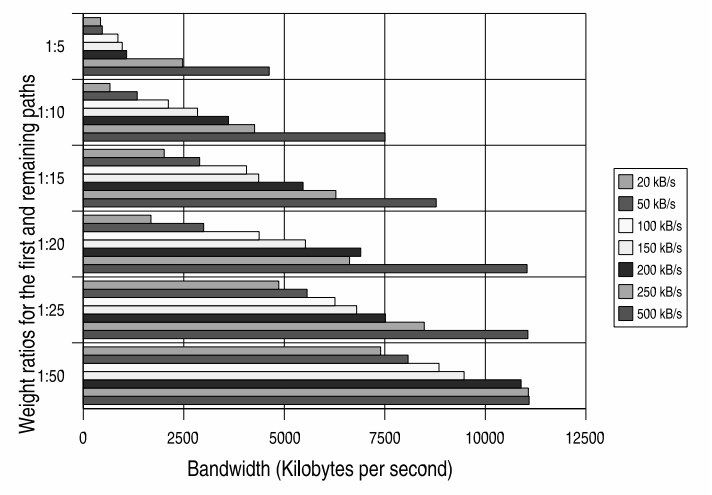

Fig. 4.11. Five proxies without buffer and weighted packet distribution

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 4.7. Five proxies without buffer and weighted packet distribution

Fig. 4.11 shows the benefits of applying a weighted packet scheduling and shows the benefits of multiple path routing where increasing the number of proxies increases the overall bandwidth.

The next two results are from tests done with a connection based buffer with a maximum buffer

[1] The Linux kernel has timers and the infrastructure to monitor 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.

size of 100 packets. The buffer stores packets until the correct and expected packet arrives and sends the sorted packets for processing.

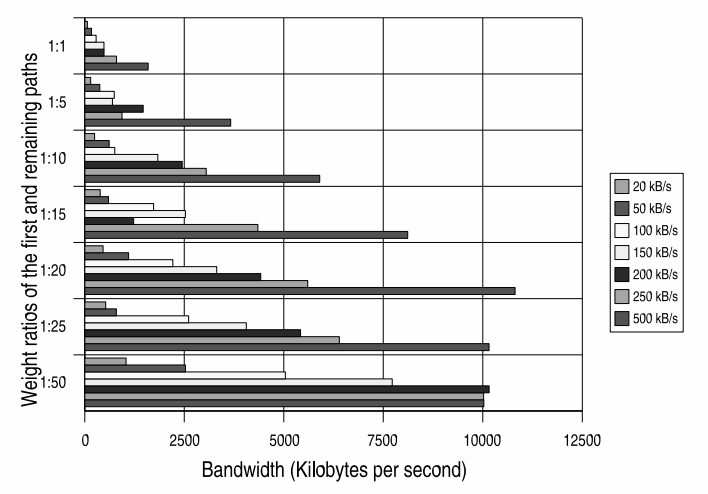

Fig. 4.12. Two proxies with buffer and weighted packet distribution

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 4.8. Two proxies with buffer and weighted packet distribution

The results in Fig. 4.12 proves that buffering the incoming packets does not necessarily yield additional bandwidth improvements. Fig. 4.12 does further solidifies the efficiency of placing a weight on the path.

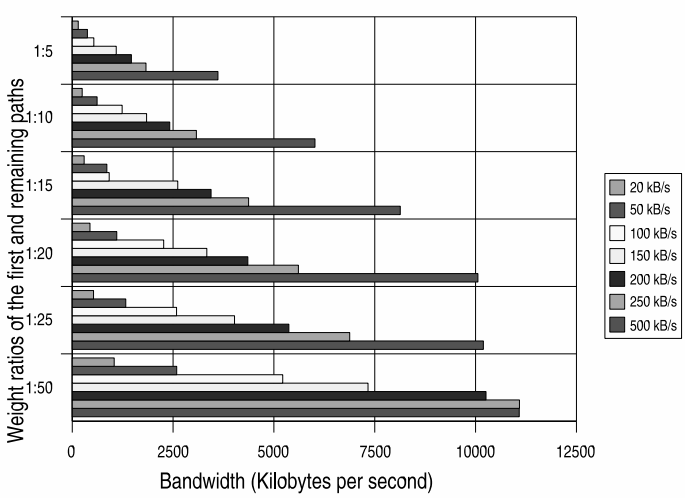

Fig. 4.13. Five proxies with buffer and weighted packet distribution

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 4.9. Five proxies with buffer and weighted packet distribution

Fig. 4.13 further demonstrates buffering the incoming packets causing a bandwidth degradation greater than the duplicate acknowledgements. Ethereal packet sniffing has shown the buffering does actually significantly reduce the duplicate acknowledgements caused by the multiple path technique. Even though it does not improve aggregated bandwidth, it will reduce overall network traffic slightly.

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 band-

width limited and weights applied to each path. One sets of tests taken with: 100 kilobytes per second overall bandwidth.

With an overall bandwidth of 100 kBps on the other node and the first path bandwidth is restricted based on the bandwidth in the key. These results are for tests with two and five proxies. Y-axis represents the weight ratio applied to the first path and the remaining paths. The X-axis is the resulting bandwidth between the client and server.

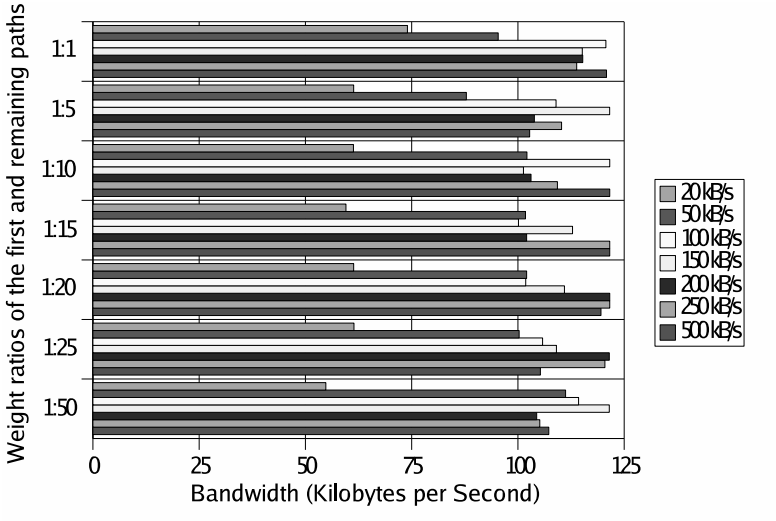

Fig. 4.14. Two proxies, without buffer, first proxy variable bandwidth, other nodes 100 kBps

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 4.10. Two proxies, without buffer, first proxy variable bandwidth, other nodes 100 kBps

Fig. 4.14 shows no major impact with rate limiting on the remaining connections. With virtual machines where the overall bandwidth is restricted by the virtual network device, even slower bandwidths (4-5 kB/s), connection based buffering showed a overall bandwidth increase. These results

are proved inconclusive; Fig. 4.14 shows overall bandwidth lowered with buffering.

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

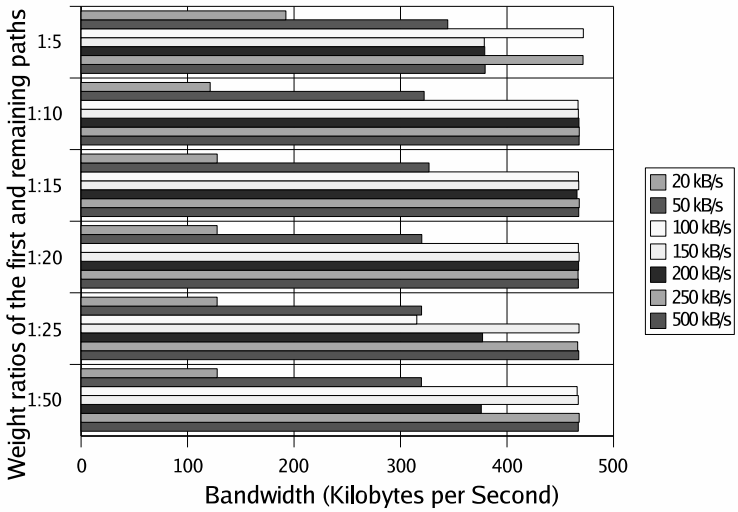

Fig. 4.15. Five proxies, with buffer, first proxies variable bandwidth, other node 100 kBps

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 4.11. Five proxies, with buffer, first proxy bandwidth variable, other proxies 100 kB/s

The author debated on the necessity of the results in Fig. 4.15. It shows under further constraints placing a weight on the intermediate packets increases bandwidth.

4.4 Impact of buffer size on multipath aggregated bandwidth

These last set of results answer the question if the overall bandwidth is affected by available buffer size. This is done by having the end nodes change their buffer size. Only one node is used and it is restricted by a controlled bandwidth listed in the key in 4.16. Buffer size is a threshold defining the maximum number of out of sequence packets allowed. In the experiments with debug turned on, the number of out of sequence packets in the buffer never grew above 20. These tests restrict the overall bandwidth on one of the proxy node connection and changed the buffer size.

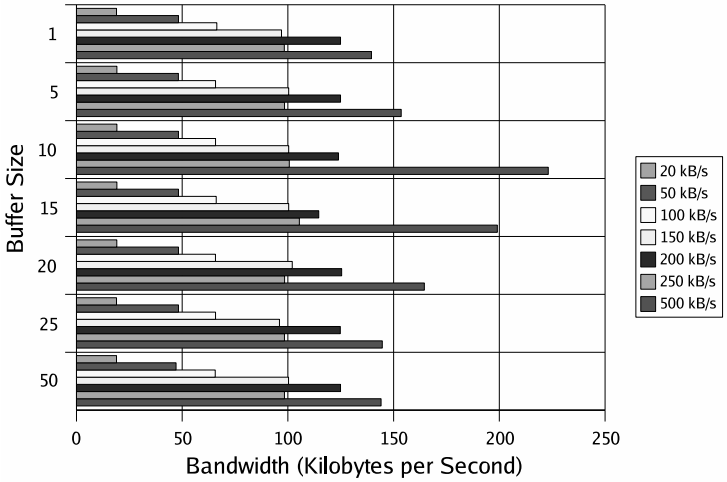

Fig. 4.16. Buffer size changing with controlled bandwidth

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 4.12. Buffer size changing with controlled bandwidth

Fig. 4.16 shows the aggregate bandwidth is mainly affected by the restriction placed on the connection and not the buffer size. The tests done with the bandwidth limited to 250kB/s is an

anomaly because the result bandwidth is higher for 200kB/s and appears to match 150 kB/s. The test script might have a typo and 150 was used oppose to 250, but after check the test scripts and retesting the numbers they seem accurate. The author is not sure what is causing this drop in performance at 250 kB/s bandwidth restriction. If this drop in performance appeared in more than one place a further investigation might be warranted.

Additional bandwidth limiters were applied in addition to HTB[18, 20], queueing to restrict bandwidth described in Section 3.6. Tests with a 10Mbit hub were also performed. These results were very poor and inconsistent because of the number of collisions.

4.5 Summary of results

The experiments show multiple path routing increases bandwidth, and weighted scheduler improves bandwidth when the available bandwidths of the paths are different.

The results from Sections 4.2 and 4.3 show the bandwidth increases as multiple path routing is activated. The bandwidth increases almost proportionally with the number of paths. Further tests were done to make sure that TCP was not compensating with excessive bandwidth results are shown in 4.14, and Fig. 4.11. These results restricted the overall bandwidth between proxy nodes and show TCP compensates for out-of-order sequences.

Tests in Section 4.3 also show the potential to further increase bandwidth by applying a proper weight for spreading packets among paths. 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.

Additional buffering seems to degrade the performance. Only under unreliable and low bandwidth situations, it increases helps the overall bandwidth. This can be observed 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 initial results done on virtual machines, a couple of hypothesis arose:

- 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 disproved 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 packets. Other tests done with the overall bandwidth capped, showed the TCP protocol still recovered from out of sequence packets without excessive bandwidth.

The results in Section 4.4 also show the size of the buffer does not affect bandwidth. However it also shows that buffering the incoming packets can remove duplicate packets and is a good area for future studies.

Bibliography

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

[2] W. Wang, "Design and implemenatation of a linux lvs based content switch," Master's thesis, University of colorado at colorado springs, 2001.

[3] C. Prakash, "Enhance features and performance of conetnet switches," Master's thesis, University of colorado at colorado springs, 2001.

[4] M. Gerla, S. Lee, and G. Pau, "Tcp westwood simulation studies in multiple path cases," *Proc. international symposium on performance evaluation of computer and telecommunication systems (spects)*, 2002 2002.

[5] 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.

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

[7] Y. Cai, "Linux kernel enhancements for multipath collection," September 2004, uccs network research seminar.

[8] Emc company. Vmware worstation 4.5. [Online]. Available: http://www.vmware.com

[9] Microsoft corporation. Microsoft virtual server 2005. [Online]. Available: http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx

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

[11] J. W. Ross and G. Westerman, "Preparing for utility computing: the role of it architecture and relationship management." *IBM Systems Journal*, vol. 43, no. 1, pp. 5–19, 2004.

[12] L. Spitzner, *Honeypots: tracking hackers*. Addison-Wesley Professional, 2002.

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

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

[15] W. Simpson. (1995, October) Rfc 1853: ip in ip tunneling. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1853.txt

[16] W. Stevens. (1997, January) Rfc 2001, tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

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

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

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

[20] 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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