Shortest Remaining File First Scheduling On Redhat Linux 9

Submitted to the
Department of Computer Science
College of Computing Sciences
New Jersey Institute of Technology

in partial fulfillment of
the requirements of the degree of
Master of Science
By
Mihir C Patel

APPROVALS

Proposal Number: ____________________________

Approved By : ____________________________

(Dr. Teunis J. Ott)

Date Submitted : ____________________________

Abstract

In today's world a client accessing a busy web server can expect a large response time. In this project response time is defined as the time from when the client sends out the SYN-packet requesting to open a connection until the client receives the last byte of the file requested. This project proposes a method for improving the performance of web servers. The idea is to give preference to requests for small files. The implementation is done in the linux kernel (redhat 9) and involves changing the way in which packets are enqueued in a network device's queue. The shortest remaining file first (SRFF) scheduling technique, defined later, gives priority to requests for short files, or those requests with short remaining file size. Now it is clear that this kind of scheduling reduces the waiting time in the queue, in particular the requests for small files. Preempting long requests for short request is desirable because forcing long requests to wait behind shorter ones results in a much lower mean response time than the situation where short requests wait behind longer ones. The key to this project is placing packets in the device queue. In this project we assign priority to a packet depending on the number of bytes sent by that flow (the current packet's flow). Thus a small request behind a large one does not waste time waiting in the queue.

This project uses network sniffing software, Tcpdump, for testing purpose.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PROBLEM STATEMENT

The project is an attempt to improve the response time offered (in certain cases) by a server. Since there are multiple clients requesting data from a server, the server has a predefined mechanism for the order in which it serves these requests. The response to these requests is sent in fixed size packets, so depending on the request, it may be served in a single packet or in multiple packets. Once the packets are ready to be sent out they are given to the device driver. The device driver then transmits the packet on the wire. Since multiple requests are served simultaneously at a server the speed at which packets are generated and given to the device may be faster than the speed at which a device can actually send these packets on the wire. So a queue is associated with a device where these packets are put in before they hit the wire. The device then grabs packets from this queue at its own pace and sends it out on the wire.

Linux in its default state has three queues for every network device. The three queues, namely q0, q1 and q2 have priorities between them. q0 has the highest priority followed by q1 and then q2. Packets are always drained from q0 first, if q0 is empty packets are removed from q1 and if both q0 and q1 are empty packets are removed from q2. All the queues have the same maximum permissible length (100 packets in case of Ethernet). Each queue internally has FIFO within it. In this project we assign a value to the priority field of a packet depending on which queue the packet needs to be inserted. The first few packets of a flow are inserted in q0, the queue with the highest priority. When the number of packets of a flow cross a set limit, say x1, they are inserted into q1, the queue with lesser priority than q0. When the number of packets of a flow crosses another limit, say x2 where x2 > x1, further packets of this flow are inserted in q2, the queue with the least priority. By doing so we guarantee that a small flow behind a big one does not suffer a long wait time in the queue. An example is as described below.

Consider that a server has two requests. Request one came first and is a request for a large file. Request two comes after and is a request for a small file. By default packets are filled as follows

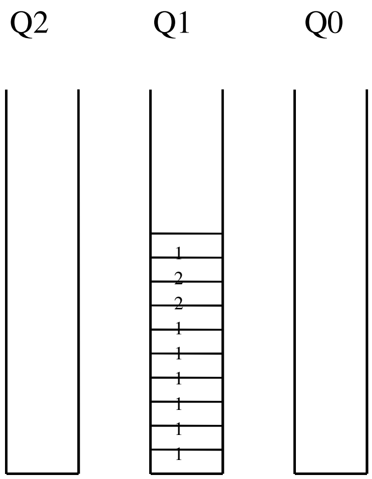


Fig. 1 Device queue with packets in it.

1 – packets due to request 1

2 – packets due to request 2

By filling the device queue as shown above, packets due to request 1 start filling the queue first and by the time packets due to request two are to be inserted, the queue is filled to a good extent and as the queues follow FIFO discipline the packets of request 2 are inserted at the tail end of the queue. This increases the waiting time for packets of request 2 in the queue. Numerically analyzing the situation, let request 1 generate 50 packets and request 2 generate 2 packets. Now, request 1 as it came first has generated packets and starts filling the queue. By the time request 2 generates packets and has them ready to be inserted in the queue, the queue already has about 35 packets due to request 1. Now the two packets due to request two are inserted in location 36 and 37 in the queue. Thus a request that can be served in 2 packets has to wait for the 35 packets in front of it to be served. This adds to the response time of request 2. So instead of filling the queue as shown above if we give priorities to flows depending on the number of packets it generates, we can reduce this waiting time.

The same situation above would be handled by this project as follows

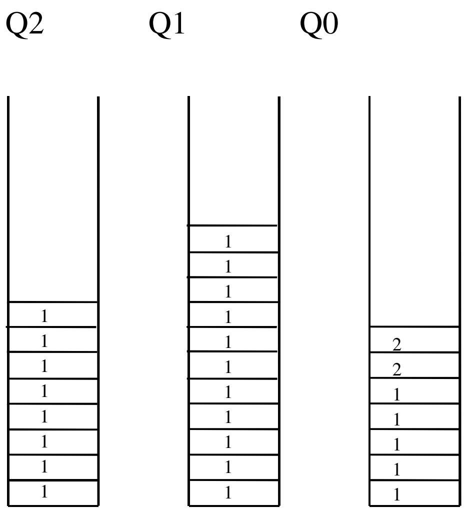


Fig. 2 Device queue with packets in it.

1 – packets due to request 1

2 – packets due to request 2

This project keeps track of all active flows and maintains a counter for the number of bytes sent by each flow. For example, the first few packets (say up to 1500 bytes) of a flow are inserted into q0, the next few packets (say up to 5000 bytes) of a flow are inserted into q1 and all other packets are inserted into q2. The numbers given above are flexible and we can set them depending up on how we want to classify a small, medium or big flow. In the example in section 6, we classify a file with less than 5,000,000 bytes as a small file. A file with more than 5,000,000 bytes but less than 30,000,000 bytes as a medium size file and a file with more than 30,000,000 bytes as a large file. So in the case as above(with limits 1500 and 5000 bytes for small and medium files respectively) the first 5 packets due to request 1 are inserted into q0 and the subsequent packets due to request 1 are inserted and into q1. After 15 packets have been inserted in q1 the rest of the packets due to request 1 are inserted into q2. Thus when packets due to request 2 have their turn to be inserted in the queue, they are inserted in q0. By doing this the wait time for packets in the queue reduces, thus improving the response time.

Analyzing the response time of the requests in both cases (with and without module) we have the following:

In the default linux mode request 1 would have been completely served in the time it takes the system to serve 52 packets (50 packets of request 1 and 2 packets of request 2). This is because the two packets of request 2 are in between the 50 packets of request 1. The response time for request 2 will be the time it takes to serve 37 packets, as packets of request 2 are inserted in location 36 and 37 in the queue.

In the case that this project handles the situation, the response time for request 1 would be the same, (the time it takes the system to serve 52 packets). The response time for request 2 would greatly be improved as now it is the time it takes the system to serve 7 packets (5 packet of request 1 and then the 2 packets of request 2). Thus we see a significant improvement in the response time of certain request(s) at no or negligible expense on other requests.

1.2 PREVIOUS WORK

Many papers have dealt with reducing the response time for requests at a web server. A lot of these papers deal with SRPT scheduling, which involves knowing the size of the response before hand. The most relevant of the papers is one by Mor Harchol Balter and Nikhil Bansal. Their work was to bring srff behavior in web servers. The length of the transfer in this case was available as it was a http based system. A paper by Bender, Chakrabarti and S. Muthukrishnan reject this idea of using SRPT scheduling because large files have an arbitrarily high maximum slowdown. The idea by Crovella et al is about connection scheduling at the application level only. This controls the order in which read and write calls are made. This does not incorporate any low level scheduling. This improves the mean response time but reduces the server throughput.

1.3 Glossary

IP	:	Internet Protocol
TCP	:	Transmission Control Protocol
UDP	:	User Datagram Protocol
ICMP	:	Internet Control Message Protocol
ARP	:	Address Resolution Protocol
Pfifo	:	Priority FIFO
netdev	:	Network Device
dev	:	Device
dflt	:	Default
ops	:	Operations
mtu	:	Maximum Transfer Unit
init	:	Initialize

srff	:	Shortest Remaining File First
srpt	:	Shortest Remaining Processing Time
q	:	Queue
lilo	:	Linux Loader
Grub	:	Grand Unified Bootloader
B	:	Bytes (8 bits)
MB	:	Mega Bytes (8 * 10^6 bits)
Mbps	:	Mega bits per second

2 Design

There are two approaches for implementing this scheme

1. Embed the software in the kernel, involves kernel recompilation.

2. Insert the software as a loadable kernel module.

A detailed description of the possible approaches is as follows

2.1 Kernel Recompilation Approach

In the kernel recompilation method we have to add/change the file(s) in the linux kernel. This involves finding a place where inserting these changes would be plausible. We have to make sure that the function(s) we insert are called in all cases. Thus we also need to change the sequence of function calls made, while preserving the calls that linux would make otherwise. Once the right place to insert the function is figured out, the function is inserted and the sequence of function calls are set we need to recompile the kernel. Recompiling the kernel involves a series of steps (described below). A successful recompilation forms a new image of the kernel. In order for the effects of this new kernel to be shown we need to reboot the system, selecting the new image we created in the list shown during start up.

2.1.1 Steps for Recompiling the Kernel

First make copies of all files that we plan to modify. Make a log file where all the work is documented (i.e, the files that are being modified and where copies of the original files are saved and other important things, if any). Now make necessary changes in the files (‘.c’ and/or ‘.h’) and save them. Also document the changes that have been made. The next step before recompiling the kernel is to check if there is enough space to have a new image of the kernel. The space utilized can be got using the ‘df’ command. It is a safe idea to have a new image of the kernel after recompilation and not overwrite the default linux image because if for some reason the new kernel fails we have the option to boot from the default linux image. A new image can be formed by changing the Makefile in / usr/src/linux-2.4/. Change the line starting with

‘#extraversion’. Save the Makefile and do the following steps to get a new image of the kernel.

```
# make mrproper
# make menuconfig
 (This will give you a GUI with parameters to be
set.)
# make dep
# make bzImage
# make modules
  (This one takes long!)
# make modules_install
# make install
# cd /etc
  (Moves you to the directory /etc )
# more lilo.conf
  (Check the new kernel is listed there!)
# reboot
```

(This reboots the system. When given the choice, choose the new kernel image).

2.2 Loadable Kernel Module Approach

This approach makes use of the netfilter hooks in the linux kernel. Hooks are places in the linux kernel where user defined functions can be inserted. There are multiple hooks in the linux kernel, so first we have to decide which hook we are interested in. We then write our function and store it in a '.c' file, eg. My_module.c. The file in addition to our function has other functions that help in registering it at the hook. Now together we call this file a module. We then need to compile the module to make it useable. Compiling is done by either writing a command at the prompt or by a 'Makefile' (explained in section 5.2). This project uses the makefile option. A makefile is written that has the command to compile the file 'My_module.c' to give the executable 'My_module.o'. This makes compilation easy, as now it is as simple as executing the makefile. This is done by the command 'make'. Now after using the 'make' command we have the executable for our module, and now we need to insert it into the kernel. Installing this module is done using the command 'insmod My_module.o'. Now that the module is inserted we can see the changes in the behavior of the system. To stop the effects of the module we need to remove it from the hook. This is done using the command ‘rmmod My_module’. This unregisters the module from the hook and thus the effects of this module are no more seen in the system. Any print messages used in the module can be seen in the log file at /var/log/messages. Thus when a module is inserted, it registers itself at the hook and when the functions at the hook are called the effect of our function is seen.

2.3 Deciding Factor for an Approach

In the kernel recompilation method, for every change we make we have to recompile the kernel to see its effect. Kernel recompilation as described earlier is a lengthy process. After the recompilation we also need a reboot of the system as we have to be in the newly created image to observe the changes. The new image formed by the kernel recompilation is an image of the whole linux kernel, thus is pretty huge in terms of disk space. It is not efficient to have a new image in order to observe a small change in the system when alternatives are available. Thus this approach is expensive in terms of time and resource (disk space).

In the option using netfilter hooks we see the effect of the changes in matter of seconds. Compiling only one file (in this case My_module.c) and registering it with the hook. It does not take a long time like kernel recompilation, does not require a reboot of the system and also takes up less disk space. Using the module approach also makes the software portable as we can copy the executable and insert it in a different machine, with out any reboot of the new machine. Another advantage of the module approach is that the module can be unregistered from the hook at any time. The kernel recompilation method would need a reboot to a different image. Thus this project uses the netfilter approach.

2.4 Description of the various NetFilter Hooks

Netfilter is a subsystem in the Linux Kernel. Netfilter makes such network tricks as packet filtering, network address translation (NAT) and connection tracking possible through the use of various hooks in the kernel’s network code. These hooks are places that kernel code, either statically built or in the form of loadable module, can register functions to be called for specific network events. An example of such an event is the transmission of a packet.

Netfilter defines five hooks for IPv4. The declaration of the symbols for these can be found in linux/netfilter_ipv4.h. The name of these hooks and a brief description is given below:

S.No	Hook	Called
1	NF_IP_PRE_ROUTING	After sanity checks, before routing decisions.
2	NF_IP_LOCAL_IN	After routing decisions if packet is for this host.
3	NF_IP_FORWARD	If the packet is destined for another interface.
4	NF_IP_LOCAL_OUT	For packets coming from local processes on their way out.
5	NF_IP_POST_ROUTING	Just before outbound packets "hit the wire".

Table 1: Names and location of the 5 IPv4 hooks

After the hook functions have done whatever processing they need to do with a packet they must return one of the predefined Netfilter return codes. The codes are as follows:

S.No	Return Code	Meaning
1	NF_DROP	Discard the packet.
2	NF_ACCEPT	Keep the packet.
3	NF_STOLEN	Forget about the packet.
4	NF_QUEUE	Queue packet for user space.
5	NF_REPEAT	Call this hook function again.

Table 2: Netfilter Return Codes

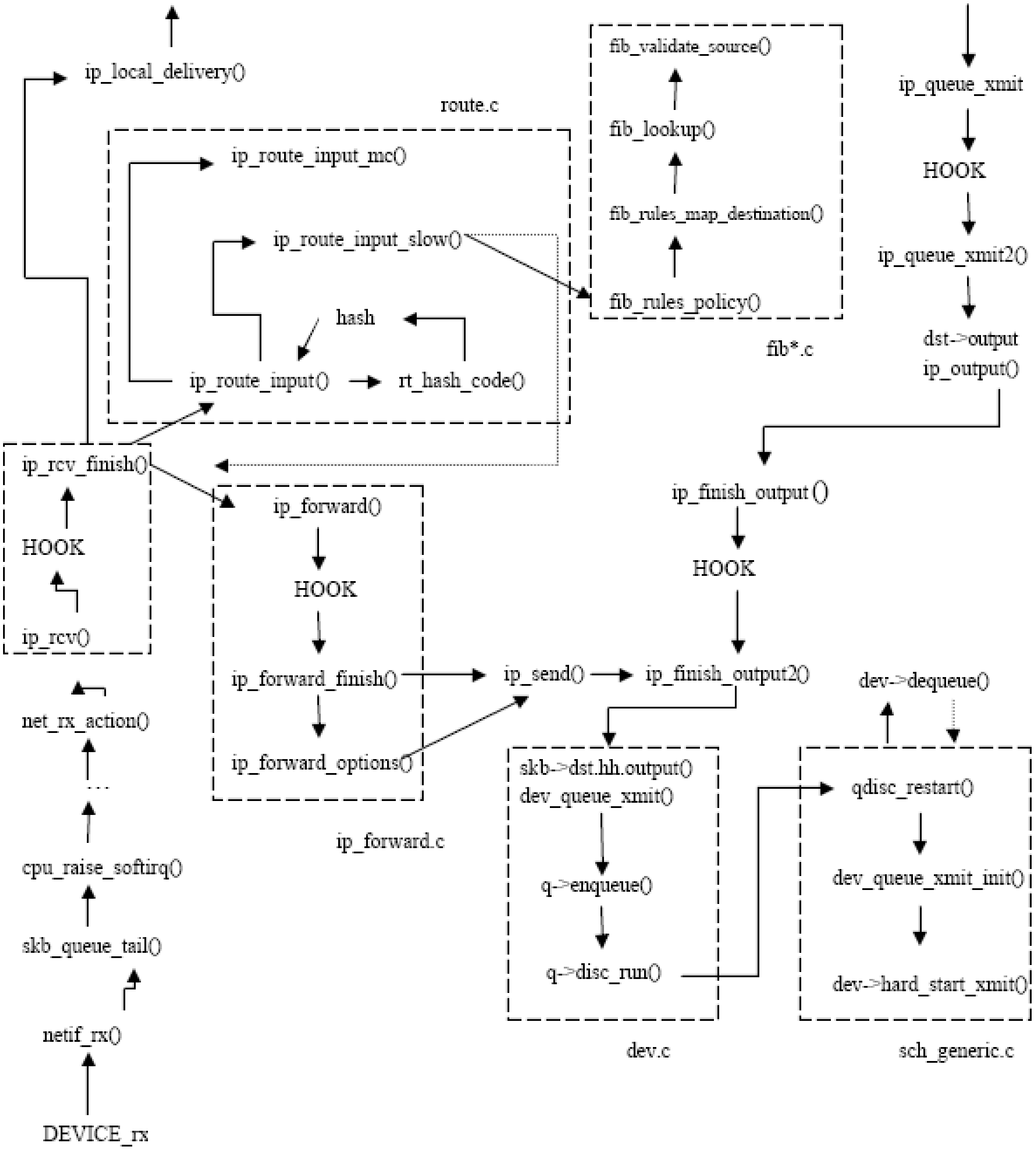


Fig. 3 Various Functions and Hooks in the Linux Network Layer data path.

2.5 Choosing the Hook

When the packet reaches the host from the network, it goes through the network layer functions and when it reaches net_rx_action(), it is passed to ip_rcv() i.e. it reaches ip layer. After passing the first netfilter hook the packet reaches ip_rcv_finish(), which verifies whether the packet is for local delivery. If it is addressed to this host, the packet is given to ip_local_delivery(), which in turn will give it to the appropriate transport layer function. A packet can also reach the IP layer coming from the upper layers (e.g., delivered by TCP, or UDP, or coming directly to the IP layer from some applications).The first function to process the packet is then ip_queue_xmit(), which passes the packet to the output part through ip_output(). In the output part, the last changes to the packet are made in ip_finish_output() and the function dev_queue_transmit() is called; the latter enqueues the packet in the output queue. It also tries to run the network scheduler mechanism by calling qdisc_run (). This pointer will point to different functions, depending on the scheduler installed. A FIFO scheduler is installed by default. The scheduling functions (qdisc_restart() and dev_queue_xmit_init ()) are independent of the rest of the IP code. When the output queue is full, q->enqueue returns an error which is propagated upward on the IP stack. This error is further propagated to the transport layer (TCP or UDP). If an incoming packet has a destination IP address other than that of the host, the latter acts as a router (a frequent scenario in small networks). If the host is configured to execute forwarding (this can be seen and set via / proc/sys/net/ipv4/ip_forward), it then has to be processed by a set of complex but very efficient functions. If the ip_forward variable is set to zero, it is not forwarded. The route is calculated by calling ip_route_input(), which (if a fast hash does not exist) calls ip_route_input_slow(). The ip_route_input_slow() function calls the FIB (Forward Information Base) set of functions in the fib*.c files. The FIB structure is quite complex. If the packet is a multicast packet, the function that calculates the set of devices to transmit the packet to is ip_route_input_mc(). In this case, the IP destination is unchanged. After the route is calculated, ip_rcv_finished() inserts the new IP destination in the IP packet and the output device in the sk_buff structure. The packet is then passed to the forwarding functions (ip_forward() and ip_forward_finish()) which send it to the output components.

With the flow of a packet in the network layer being as described above, the place suitable for inserting our module is NF_IP_POST_ROUTING hook. This is the last hook in the network layer, and it is only by this time that the packet has in it all the header information necessary to calculate the priority of the packet. If the system is a router instead of a server, then our module needs to be inserted at NF_IP_FORWARD.

3 Background

3.1 Description of device activation and fucntions used

When a device is initialized in linux, queue(s) are created and associated with the device. These queue (s) behave according to a certain discipline. The linux kernel has in it many such disciplines that a queue can take. So, when a device is activated a decision is made as to which discipline would be assigned to the queue(s) of this device. Assigning a discipline to a queue is done by giving meaning to the function pointers that do operations on the queue. There are a handful of such pointers that operate on a queue, for example ‘init()’, ‘enqueue ()’, ‘dequeue()’ and a few more. These pointers are called to do operations on the queue. So by having ‘init()’ point at the ‘init’ function of the desired discipline, we can have the queue for a device initialized according to that discipline. Similarly by pointing ‘enqueue()’, ‘dequeue()’ and the rest of the function pointers to the appropriate functions of a discipline we have the queue behave according to the discipline. A series of functions are called during the activation of a device. The first function to be called is ‘dev_open()’. This function calls the device’s private ‘open()’ function. It also loads the multicast list and calls functions that creates a device queue and give it a discipline. It notifies the system about the change in the state of the device. Calling the ‘dev_open()’ function on an active device is a noop and the function on failure returns a negative error number. A function call of interest here is the call to ‘dev_activate()’. This call is made after the device is up, to create the device queue. By default the functions assigned to the function pointers, ‘enqueue()’, ‘dequeue()’ and the others are

Enqueue	-	‘pfifo_fast_enqueue’
Dequeue	-	‘pfifo_fast_dequeue’
Requeue	-	‘pfifo_fast_requeue’
Reset	-	‘pfifo_fast_reset’
Init	-	‘pfifo_fast_init’

The default mechanism in linux, pfifo, creates 3 queues and each queue intern has FIFO mechanism within it. So the default mechanism is a combination of priority and FIFO.

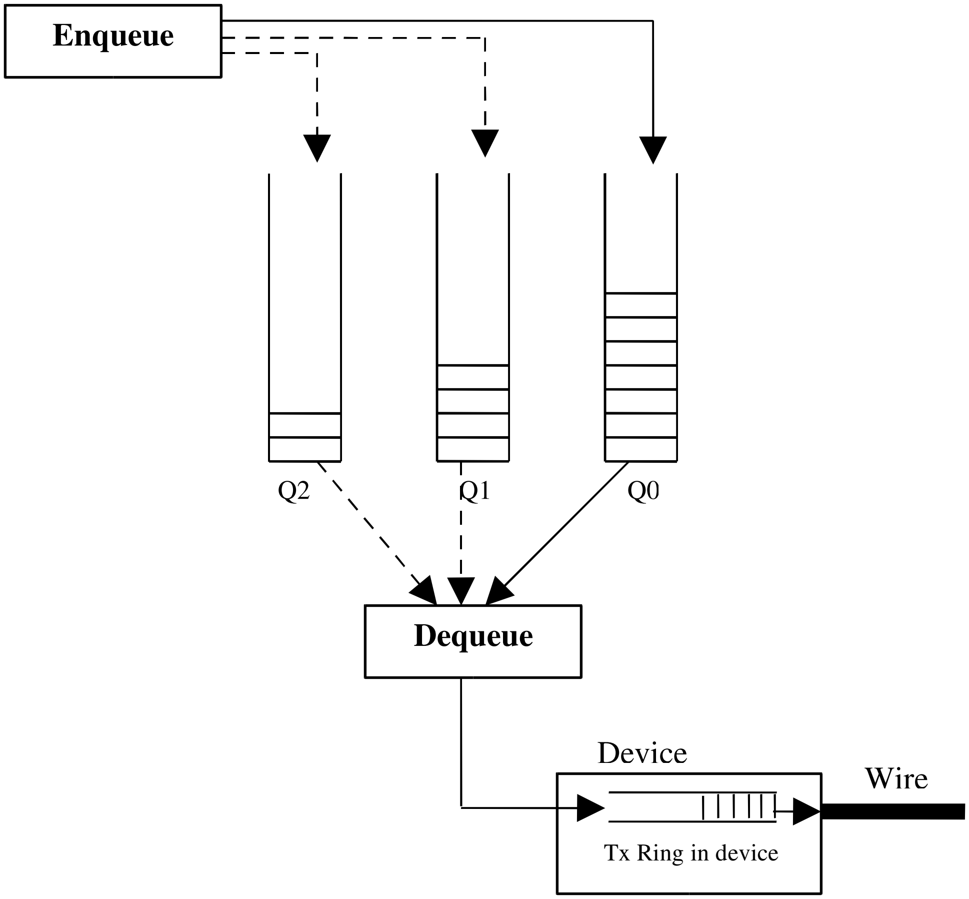


Fig.4 – shows the three queues and the relation of the enqueue, dequeue functions with these queues. Also shows the path for a packet to the device and its internal buffer/ring to the wire.

‘enqueue()’ inserts a packet into one of the three queues depending on the priority field in the packet. ‘dequeue()’ tries to remove a packets from queue ‘q0’. If ‘q0’ is empty it tries to remove a packet from ‘q1’ and when both ‘q0’ and ‘q1’ are empty it tries to remove a packet from ‘q2’. The ‘dequeue()’ function may be interrupted, and when it returns the ‘dequeue()’ function starts again from ‘q0’, thus preference is given to ‘q0’. Packets after being dequeued are given to the driver which places them in the device’s internal transmit ring and sends them out on the wire.

4 Algorithm

4.1 Algorithm for this Project

This project assigns a priority for every packet passing through the network layer. This priority field determines into which queue the packet should be inserted at the device layer (3 queues per device). An outgoing packet flows through the various network layers and it reaches the network layer (layer 3) and the hook (NF_IP_POST_ROUTING) where we have registered our module. All out going packets that flow through the network layer pass through this hook. By the time the packet reaches the hook it has its network header ready with all the variables given appropriate values. This module does three major functions

1. keeps a record of the packet
2. creates and maintains a linked list of active flows (flows that have at least one packet in the device queue)
3. cleans the linked list at regular intervals

First the module checks for the transport layer protocol of the packet. If the packet is a TCP packet, then the source and destination port numbers are got by looking into the transport layer header. One complication that arises here is that, there is no direct access to the transport layer headers from the network layer. So in order to get the port number we do pointer arithmetic (explained in section 5.2) on the packet and reach out for the port numbers. The rest of the details are obtained form the network header itself and since our module is at the network layer we have direct and complete access to all the fields of the network layer. A node is created that stores the source and destination IP addresses, source and destination port numbers, the transport layer protocol, the time at which the packet arrived and number of data bytes in the packet. These details are stored and a linked list is formed for all the active flows in the system. If a packet is the first packet of a flow then a new node is created and inserted into the linked list. Otherwise the node corresponding to the packet details is searched and updated. The only updates done are the number of bytes seen of that flow and the last time a packet of that flow was seen. Now that we have the number of bytes seen from that flow, we can decide as to which queue the packet should go into. For example, if the total number of bytes seen of this flow is less than 1500 bytes then it is considered a high priority flow and the packet should be inserted in to q0. If the number of bytes seen of this flow is greater than 1500 bytes but less than 5000 bytes then the flow is considered a moderate flow and the packets at this time go into q1. If there are more than 5000 bytes seen from this flow then it is a big flow and all the rest of the packets of this flow go into q2. Once this change is made in the packet it is sent forward as it would go normally. One more task that the module does is, the clean up of the linked list created. This is done once every 2500 milliseconds. A sequential traversal is done through the nodes in the linked list and if a node has not had any update in the last 2500 milliseconds then the node is deleted assuming the flow has finished.

5 Implementaion

5.1 Code

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <net/ip.h>

// user defined data structure to store the active flow details
struct flows
{
__u16 sport,dport;      // source and destination port  numbers
__u32 saddr,daddr;      // source and destination IP address
__u8 protocol;                   // Transport layer protocol name
__u32 num_of_bytes; // Total number of bytes sent by this flow
__u32 tolp;             // Arrival time of the last packet of this flow
struct flows *next;     // Pointer to the next node in the linked list
};

static struct flows *head = NULL;
static struct nf_hook_ops nfho;

// This is the hook function itself
unsigned int hook_func(unsigned int hooknum,
struct sk_buff **skb,
const struct net_device *in,
const struct net_device *out,
int (*okfn)(struct sk_buff *))
{
struct flows *cur_pack, *temp, *temp1;
cur_pack = (struct flows *) kmalloc(sizeof(struct flows), GFP_KERNEL);
struct sk_buff *sb = *skb;
static __u32 last_clear_table=0;
__u8 ip_hlen;
__u16 ip_tot_len;
cur_pack->tolp = jiffies;

ip_hlen = (sb->nh.iph->ihl)*4;
ip_tot_len = ntohs(sb->nh.iph->tot_len);

if(sb->nh.iph->protocol == IPPROTO_TCP)
{
cur_pack->saddr = sb->nh.iph->saddr;
cur_pack->daddr = sb->nh.iph->daddr;
cur_pack->sport = *(unsigned int *) (sb->data + (sb->nh.iph->ihl * 4));
cur_pack->dport = *(unsigned int *) ((sb->data + (sb->nh.iph->ihl * 4)) + 2);
cur_pack->protocol = sb->nh.iph->protocol;
cur_pack->num_of_bytes = (ip_tot_len-ip_hlen);
cur_pack->next = NULL;

// Checking to see if the last time the linked list was cleared is more than 2500
milliseconds ago, if so clear the linked list.
if((jiffies-last_clear_table)>250)
{
last_clear_table=jiffies;
printk("Time to clean the table \n");
temp=head;
```

```
temp1=head;
while(temp!=NULL)
{
if((last_clear_table-temp->tolp)>250)
{
if(temp==head)
{
printk("Deleting the first node \n");
head=head->next;
kfree(temp);
temp=head;
temp1=head;
}
else
{
printk("Deleting inbetween node \n");
temp1->next=temp->next;
kfree(temp);
temp=temp1->next;
}
}
else
{
if(temp==head)
{
temp=temp->next;
}
else
{
```

```
temp1=temp;
temp=temp->next;
}
}
}
last_clear_table=jiffies;
}

if(head==NULL)
{
head=cur_pack;
}
else
{
temp = head;
while(temp != NULL)
{
if(temp->saddr==cur_pack->saddr  &&  temp->daddr==cur_pack->daddr  &&
temp->sport==cur_pack->sport && temp->dport==cur_pack->dport)
{
temp->num_of_bytes = temp->num_of_bytes+cur_pack->num_of_bytes;
temp->tolp=cur_pack->tolp;
if(temp->num_of_bytes>5000)
sb->priority=1;                    // queue 2
else if(temp->num_of_bytes>1500)
sb->priority=0;                    // queue 1
else
sb->priority=6;                    // queue 0
temp=NULL;
```

```
kfree(cur_pack);
}
else
{
if(temp->next==NULL)
{
temp->next=cur_pack;
temp=NULL;
}
else
temp=temp->next;
}
}
}
}
return NF_ACCEPT;
}

//Initialisation routine
int init_module()
{
nfho.hook     = hook_func;
nfho.hooknum  = NF_IP_POST_ROUTING;
nfho.pf       = PF_INET;
nfho.priority = NF_IP_PRI_FIRST;
nf_register_hook(&nfho);
return 0;
}

//Cleanup routine
```

```
void cleanup_module()
{
nf_unregister_hook(&nfho);
}
```

5.2 Explanation of the Code

The code can be divided into three distinct parts

1. Registering the hook
2. Un registering the hook
3. The hook function (functionality routine)

Registration of a hook function is a very simple process that revolves around the nf_hook_ops structure, defined in linux/netfilter.h. The definition of this structure is as follows:

```
struct nf_hook_ops
{
struct list_head list;
// user fills in from here down
nf_hookfn * hook;
int pf;
int hooknum;
int priority;     // hooks are ordered in ascending priority
};
```

The list member of this structure is used to maintain the lists of Netfilter hooks and has no importance for hook registration as far as users are concerned. Hook is a pointer to a nf_hookfn function. This is the function that will be called for the hook. nf_hookfn is defined in linux/netfilter.h. The pf field specifies a protocol family. Valid protocol families are available from linux/socket.h but for IPv4 we will use PF_INET. The hooknum field specifies the particular hook to install this function for and is one of the values listed in table 1. Finally, the priority field specifies where in the order of execution this hook function should be placed. For IPv4, acceptable values are defined in linux/netfilter_ipv4.h in the nf_ip_hook_priorities enumeration. We use NF_IP_PRI_FIRST in our module. As this is the only module we have, it makes no difference as to what we use here, but in the case where we have more than one module we would like to give a proper order of execution.

Registration of a Netfilter hook requires using a nf_hook_ops structure with the nf_register_hook() function. nf_register_hook() takes the address of an nf_hook_ops structure and returns an integer value.

However, if we actually look at the code for the nf_register_hook() function in net/core/netfilter.c, we will notice that it only ever returns a value of zero.

unregistering a Netfilter hook is a lot simpler than registering it. It is done by calling the function nf_unregister_hook() with the address of the same structure we used to register the hook.

To get a better understanding of the functionality of the code we need to have an understanding of the data structures used. The module has only one user defined data structure, flows. The definition of the structure is given below

```
struct flows
{
__u16 sport,dport; // source and destination port  numbers
__u32 saddr,daddr;            // source and destination IP address
__u8 protocol;                // Transport layer protocol name
__u32 num_of_bytes;           // Total number of bytes sent by this flow
__u32 tolp;                   // Arrival time of the last packet of this flow
struct flows *next; // Pointer to the next node in the linked list
};
```

This structure is used to keep track of all the active flows in the system. It has fields to store the socket details, thereby identifying a flow. ‘saddr’ is a 32 bit field that stores the source IP address of the flow. ‘daddr’ is another 32 bit field that stores the destination IP address of the flow. ‘sport’ and ‘dport’ are 16 bits each and store the source and destination port numbers respectively. These four fields together uniquely identify a flow. The protocol field is a 8 bit field that stores the transport layer protocol of the flow. This is necessary as there is no point in keeping track of UDP packets that are connection less and also ICMP packets where all the four details uniquely describing a socket are unavailable. ‘tolp’ is a 32 bit field that stores the time when the last packet of this flow was seen. The time is stored in units of 10 milliseconds. The last field in this structure is a pointer to the next node of this type in the linked list.

Now that the data structure is clear, we can understand the code better. When the module is called, it first creates a node of the type ‘flows’ and fills in all the values from the packet at hand. ‘tolp’, time when the packet arrived is got by ‘jiffies’. This returns the current time in linux in 10 milliseconds accurate. The source and destination IP address are got by looking into the network header of the packet, and as the module is at the network layer we have access to this information. The source and destination port numbers can be got form the transport layer header, if the packet is a TCP packet. So first a check is done to see if the packet is a TCP packet, and if it is the port numbers are got from the transport layer header. Since the module is at the network layer, we do not have direct access to this information, so we do pointer arithmetic to gain access to this information. We have a pointer to the start of the network header and we know the layout of the packet. We also have access to the length of the network header, so now with all the above information we can reach the transport layer header.

Source port number and destination port number are the first two fields in the case of TCP packets.

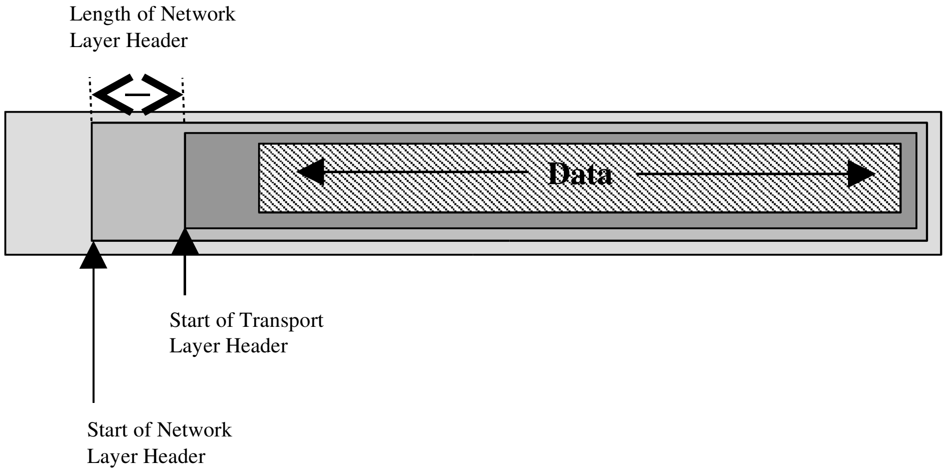


Fig.5 Layout of a Packet showing the various headers.

Source port number = *(ip_header_pointer + (ip_header_length * 4))

Destination port number = *(ip_header_pointer + (ip_header_length *4) + 2)

As the ip_header_length got from the IP header is in words, we multiply by 4 to get the length of the IP header in bytes. The '+2' in getting the destination port number is because it is the second field in the transport layer header and the source port number, the first one, is 2 bytes long. The next information we need is the number of data bytes (with respect to the IP part) in the packet. The IP header gives us in addition to the IP header length also the total length of the packet. So the difference in the values of the two fields is the length of the IP packet. The next pointer is made null, as we are not sure if a node corresponding to this flow already exists in the linked list. Now with all the details collected we check if it is time to clean the linked list. If it more than 2500 milliseconds since we cleaned the linked list, we do a linear traversal through all the nodes of the linked list. A check is done every node to see when was the last time a packet corresponding to that flow was seen by the system. If this interval happens to be more than 2500 milliseconds, the node is deleted from the linked list. If no packet of a flow is seen in a time as high as 2500 milliseconds then the software considers the flow to have finished. After the clean up, we now need to update the linked list with respect to the packet at hand. So a search is done in the linked list to see if the there already is a node corresponding to this flow (the flow from which this packet came). Then the number of bytes seen of this flow is updated and the time when the last packet of this flow was seen is also updated. Now if for example, the total number of bytes seen of this flow is more than 5000 bytes the priority field in the packet is made ‘1’, implies that the packet will be inserted into q2. if on the other hand the number of bytes seen of this flow is less than 5000 but more than 1500 then the priority field in the packet is made ‘0’, implies that the packet will be inserted in q1. If the number of bytes of this flow are less than 1500 bytes then the flow is considered to be a small flow, so the priority field in the packet is made ‘6’, implies that the packet will be inserted into q0. After all these updates are done the new node created for the packet is deleted. If on the other hand no node (corresponding to the packet at hand) is found in the linked list it implies that this is the first packet of the flow. So the new node created for this packet is inserted at the end of the linked list, thereby increasing the number of active flows in the system by one.

Now that the whole module is clear, the next step we need is to compile the module. As said earlier in this project we write a makefile.

A makefile is used with the Unix make utility to determine which portions of a program to compile. A makefile is basically a script that guides the make utility to choose the appropriate program files that are to be compiled and linked together.

The make utility keeps track of the last time files were updated so that it only updates the files containing changes. However, all of the files that are dependent on the updated files must be compiled as well, which can be very time-consuming. With the help of makefile, the make utility automates this compilation to ensure that all files that have been updated - and only those - are compiled and that the most recent versions of files are the ones linked to the main program, without requiring the user to perform the tasks separately.

A makefile contains three types of information for the make program: a target (the name of what the user is trying to construct); the rules (commands that tell how to construct the target from the sources) and a dependency (the reason that the target should be constructed, which is usually because it is out of date in respect to its components). To create a makefile, the user makes a file containing shell commands and names it "makefile." The commands are executed according to the rules in the makefile when the user types "make" while in the directory containing the file.

The makefile used in this project is as follows

```
#Makefile
CC= gcc -I/usr/src/linux-2.4/include
CFLAGS = -O2 -D__KERNEL__ -Wall

My_module.o:My_module.c
```

6 Tests and Results

The test case considered for this project is as follows

A FTP connection between Chekov and Marconi (inside the internet laboratory) is made. Three files f1, f2 and f3 of sizes 5.7MB, 31.89MB, 79.53MB (approximate) respectively are sent from Chekov to Marconi. The three transfers are initiated almost simultaneously, in reality f3 initiated before f2 and f2 initiated before f1. Thus bringing the situation where a request for a small file comes after the request for a big file. The tests were done thrice without the module inserted and thrice with the module inserted. The expected result of the test was to see a decrease in the response time for the small file when the module was inserted in the Linux kernel. The response time of the big file was not predicted as it may have no significant change as described in section 1.1 or it may have an acceptable increase as packets of other transfers are given priority or the response time may decrease if in the case without modules there was a overflow in the queue (software queue) and now with the module inserted the packets are distributed better among the queues and no loss of packets is seen. The results were seen using the TCPDUMP utility.

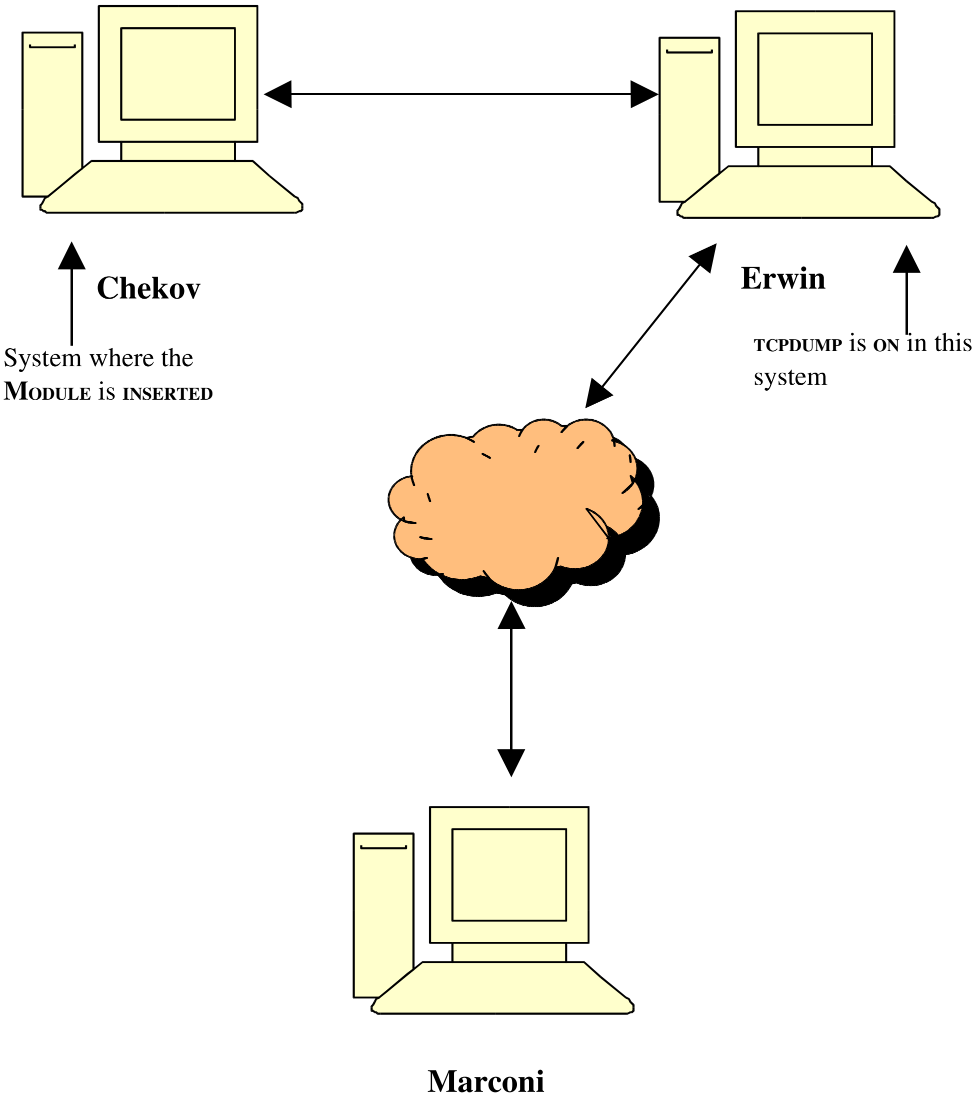


Fig. 6 Topology of the Internet Laboratory (Machines we are concerned with)

To make testing possible (as the LAN speed in the LAB is very high) the module is changed as follows

Packets are inserted into q0 as long as the total number of bytes seen from a flow is less than 5,000,000 bytes. Packets are inserted in q1 if more than 5,000,000 bytes but less than 30,000,000 bytes of a flow are seen. If more than 30,000,000 bytes of a flow are seen, packets of that flow are inserted in q2. With the module changed as above the file sizes were carefully chosen for the test, almost all packets of file f1 will be in q0, almost all packets of file f2 will be in q0 and q1 (close to the maximum permissible in each queue) and file f3 is the only one that will have the maximum permissible number of packets (bytes) in q0, q1 and a lot of packets q2.

TCPDUMP is started at Marconi, it is a router in this connection and does not have any other load than forwarding packets so it can handle the extra work of recording all the packets without much delay.

The results of the transfer of files f1, f2 and f3 without the module inserted is as follows

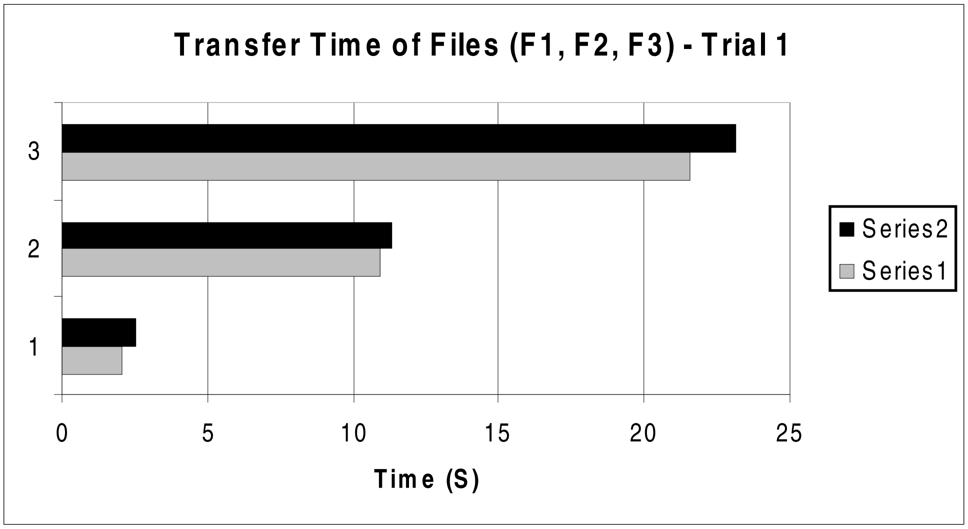
S.No	File F1	File F2	File F3
1	2.554343	11.311439	23.113718
2	2.216865	11.931338	23.002878
3	3.134354	12.892016	22.586394

Table 3: Time in seconds (micro seconds accurate) for the transfer of files F1, F2 and F3 from Chekov to Marconi. Three trials are taken for better precision.

The results of the transfer with the module inserted at Chekov are as shown in the table below

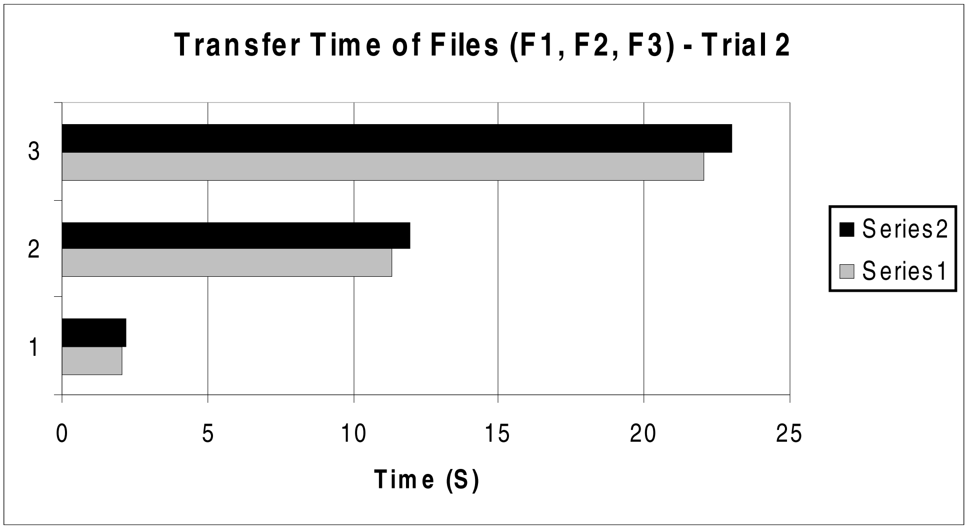
S.No	File F1	File F2	File F3
1	2.056640	11.345540	22.049724
2	2.041893	10.944714	21.556246
3	2.359610	12.110914	22.199290

Table 4: Time in seconds (micro seconds accurate) for the transfer of files F1, F2 and F3 from Chekov to Marconi (with Module inserted). Three trials are taken for better precision.



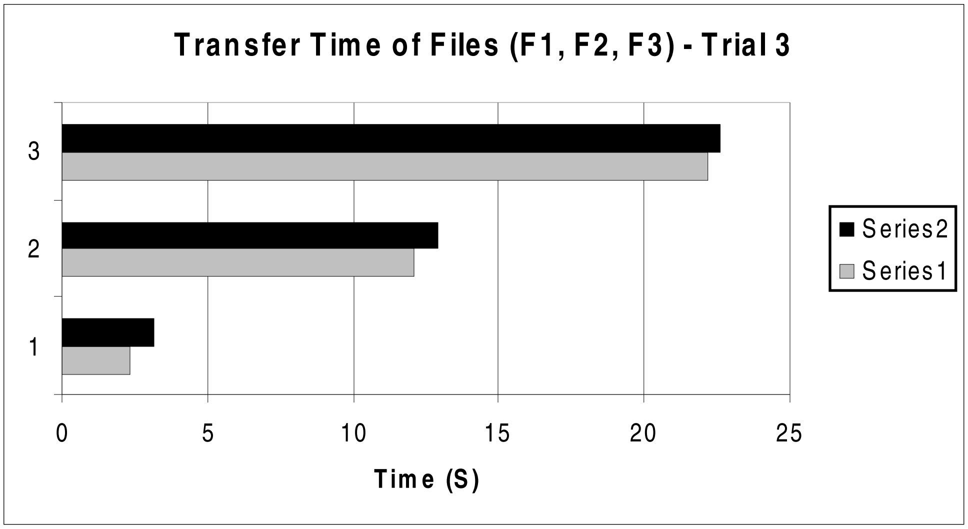
Series 1: Time Taken to Transfer the File with Module Inserted.

Series 2: Time Taken to Transfer the File without the Module Inserted.



Series 1: Time Taken to Transfer the File with Module Inserted.

Series 2: Time Taken to Transfer the File without the Module Inserted.



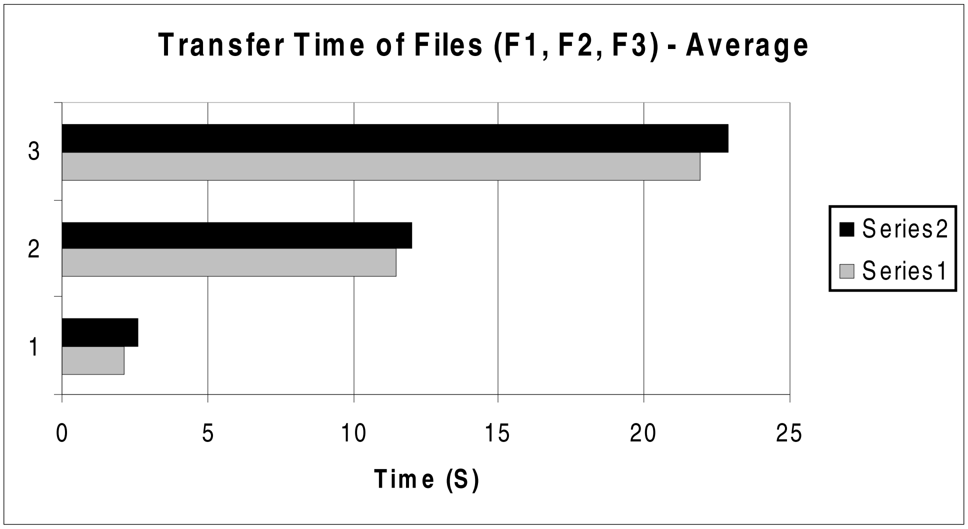
Series 1: Time Taken to Transfer the File with Module Inserted.

Series 2: Time Taken to Transfer the File without the Module Inserted.

The following table gives the average time taken for the transfer of the files F1, F2 and F3

S.No	File F1	File F2	File F3
Without Module			
1	2.635187	12.044931	22.900997
With Module			
2	2.152714	11.467056	21.935087

Table 5: Average Time for the file transfers



Series 1: Time Taken to Transfer the File with Module Inserted.

Series 2: Time Taken to Transfer the File without the Module Inserted.

With the average times as shown above it is clear that we have an improvement in the response time for the small file with also the response time for the other requests improving by a small amount. The percentage upgrade in the response time for the three files are as follows

$$\text{\% Upgrade in the Response time for F1} = \frac{\text{Difference in the time taken for the transfer (with \& without the module)}}{\text{Time taken for the transfer without the module}} * 100$$

$$= \frac{(2.635187 - 2.152714)}{(2.635187)} * 100$$

$$= 18.31\ \%$$

The calculation above proves shows that the response time for the file F1 has improved by 18.31 %.

Similarly the improvement in the response time for files F2 and F3 can be calculated

% Upgrade in the Response time for F2	=	4.80 %
% Upgrade in the Response time for F3	=	4.22 %

The total bandwidth utilization in both the cases is described below

Bandwidth utilization without the module

F1 is of size 5756923 Bytes => 46055384 bits

It was transferred in 2.635187 Seconds

Thus the transfer speed in this case is

```
                    46055384  (bits)
          =         --------  -------

                    2.635187 (Seconds)
```

= **17.48 Mbps**

F2 is of size 31898736 Bytes => 255189888 bits

It was transferred in 12.044931 Seconds

Thus the transfer speed in this case is

$$= \frac{255189888 \quad \text{(bits)}}{12.044931 \quad \text{(seconds)}}$$

$$= \mathbf{21.19\ Mbps}$$

F3 is of size 79532032 Bytes => 636256256 bits

It was transferred in 22.900997 Seconds

Thus the transfer speed in this case is

$$= \frac{636256256 \quad \text{(bits)}}{22.900997 \quad \text{(seconds)}}$$

$$= \mathbf{27.78\ Mbps}$$

Bandwidth utilization with the module inserted in the Linux kernel is as follows

F1 was transferred in 2.152714 seconds (with module)

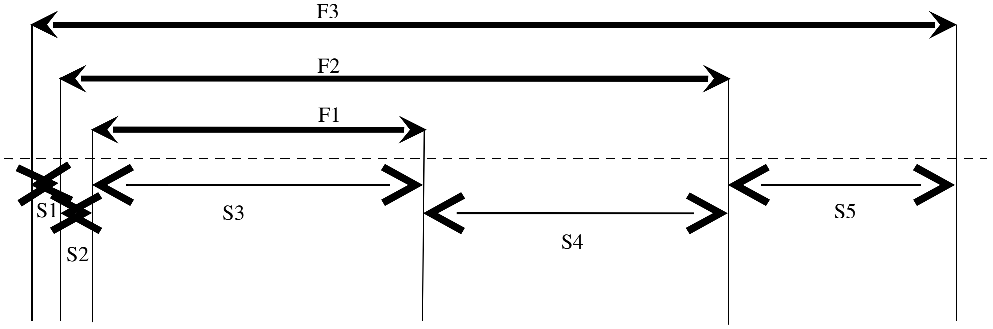
Therefore the transfer speed when calculated as above is **21.39 Mbps**

F2 was transferred in 11.467056 seconds (with module). Therefore the transfer speed when calculated as above is **22.25 Mbps**

F3 was transferred in 21.935087 seconds (with module). Therefore the transfer speed when calculated as above is **29.01 Mbps**

Bandwidth Calculation (in Detail)

Trial 1



Number of Bytes Sent	Time Elapsed	Bandwidth Utilized
S1 – 1601248 Bytes	S1 – 0.308665 S	S1 – 41.50 Mbps
S2 – 2600576 Bytes	S2 – 0.310337 S	S2 – 67.04 Mbps
S3 – 17544840 Bytes	S3 – 2.552042 S	S3 – 55.00 Mbps
S4 – 45918976 Bytes	S4 – 8.445657 S	S4 – 43.50 Mbps
S5 – 51060624 Bytes	S5 – 11.494316 S	S5 – 35.54 Mbps

Total number of Bytes Sent = 118726264 Bytes => (949810112 bits)

Total Elapsed Time in Seconds = 23.111017 Seconds

Average Bandwidth Utilized = 41.10 Mbps

Fig. 7 Start and Stop Times of the File Transfers.
Number of bytes transferred in each interval.

During **S1** only **F3** is sent on the Link so Bandwidth utilized is **41.50 Mbps**

During **S2, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 34.00 Mbps

F2 – 33.04 Mbps

During **S3, F3, F2** and **F1** are sent on the Link so Bandwidth utilized is as follows

F3 – 13.71 Mbps

F2 – 22.81 Mbps

F1 – 18.47 Mbps

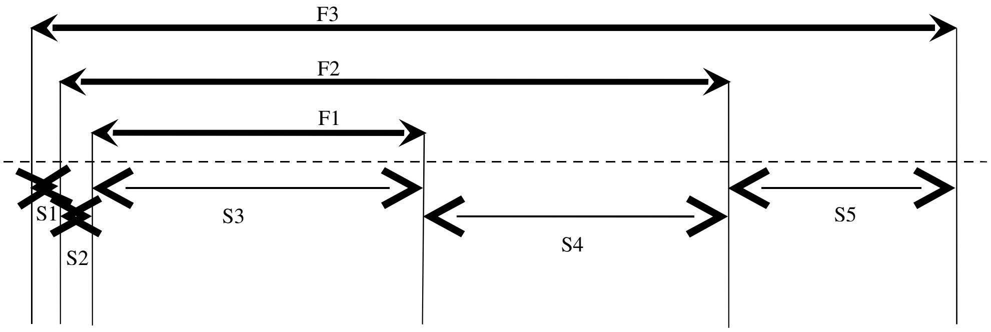
During **S4, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 20.87 Mbps

F2 – 22.62 Mbps

During **S5** only **F3** is sent on the Link so Bandwidth utilized is **35.54 Mbps**

Trial 2



Number of Bytes Sent	Time Elapsed	Bandwidth Utilised
S1 – 949496 Bytes	S1 – 0.285565 S	S1 – 26.50 Mbps
S2 – 2473072 Bytes	S2 – 0.320414 S	S2 – 61.77 Mbps
S3 – 14505248 Bytes	S3 – 2.214523 S	S3 – 52.40 Mbps
S4 – 54013336 Bytes	S4 – 9.393935 S	S4 – 46.00 Mbps
S5 – 46761408 Bytes	S5 – 10.786093 S	S5 – 34.68 Mbps

Total number of Bytes Sent = 118702560 Bytes => (949620480 bits)

Total Elapsed Time in Seconds = 23.000530 Seconds

Average Bandwidth Utilized = 41.29 Mbps

Fig. 9 Start and Stop Times of the File Transfers.
Number of bytes transferred in each interval.

During **S1** only **F3** is sent on the Link so Bandwidth utilized is **26.60 Mbps**

During **S2, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 23.98 Mbps

F2 – 37.79 Mbps

During **S3, F3, F2** and **F1** are sent on the Link so Bandwidth utilized is as follows

F3 – 20.53 Mbps

F2 – 10.73 Mbps

F1 – 21.14 Mbps

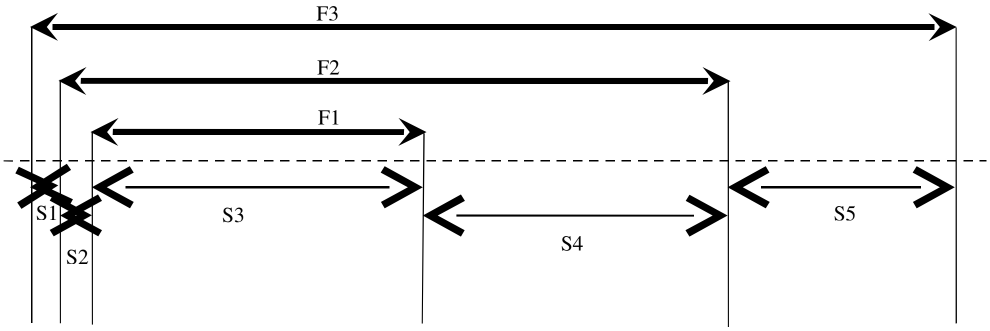
During **S4, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 22.19 Mbps

F2 – 23.81 Mbps

During **S5** only **F3** is sent on the Link so Bandwidth utilized is **34.68 Mbps**

TRIAL 3



Number of Bytes Sent	Time Elapsed	Bandwidth Utilised
S1 – 2191184 Bytes	S1 – 0.342334 S	S1 – 51.21 Mbps
S2 – 1520416 Bytes	S2 – 0.287178 S	S2 – 42.35 Mbps
S3 – 19068928 Bytes	S3 – 3.132024 S	S3 – 48.71 Mbps
S4 – 53607856 Bytes	S4 – 9.470204 S	S4 – 45.29 Mbps
S5 – 42452232 Bytes	S5 – 9.352405 S	S5 – 36.31 Mbps

Total number of Bytes Sent = 118840616 Bytes => (950724928 bits)

Total Elapsed Time in Seconds = 22.584149 Seconds

Average Bandwidth Utilized = 42.10 Mbps

Fig. 10 Start and Stop Times of the File Transfers.
Number of bytes transferred in each interval.

During **S1** only **F3** is sent on the Link so Bandwidth utilized is **51.21 Mbps**

During **S2, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 21.09 Mbps

F2 – 21.26 Mbps

During **S3, F3, F2** and **F1** are sent on the Link so Bandwidth utilized is as follows

F3 – 20.15 Mbps

F2 – 13.50 Mbps

F1 – 15.06 Mbps

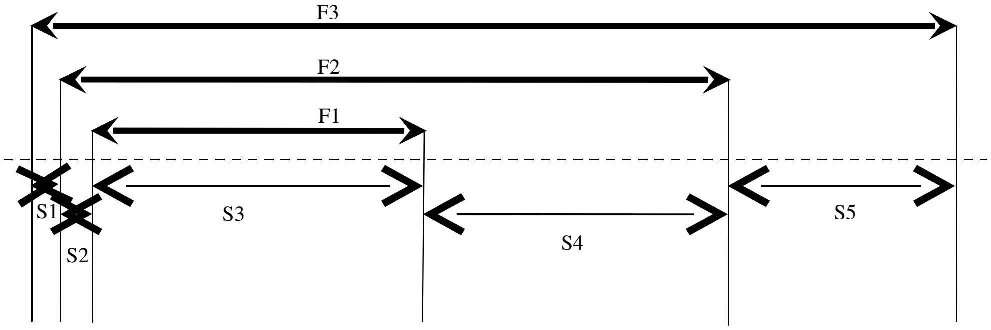
During **S4, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 22.94 Mbps

F2 – 22.35 Mbps

During **S5** only **F3** is sent on the Link so Bandwidth utilized is **36.31 Mbps**

TRIAL 4



Number of Bytes Sent	Time Elapsed	Bandwidth Utilised
S1 – 1950760 Bytes	S1 – 0.266991 S	S1 – 58.45 Mbps
S2 – 2646248 Bytes	S2 – 0.244770 S	S2 – 66.88 Mbps
S3 – 15620280 Bytes	S3 – 2.055328 S	S3 – 60.80 Mbps
S4 – 53019968 Bytes	S4 – 9.043214 S	S4 – 46.90 Mbps
S5 – 46057984 Bytes	S5 – 10.437152 S	S5 – 35.30 Mbps

Total number of Bytes Sent = 118695240 Bytes => (949561920 bits)

Total Elapsed Time in Seconds = 22.047455 Seconds

Average Bandwidth Utilized = 43.07 Mbps

Fig. 11 Start and Stop Times of the File Transfers.
Number of bytes transferred in each interval.

During **S1** only **F3** is sent on the Link so Bandwidth utilized is **58.45 Mbps**

During **S2, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 28.63 Mbps

F2 – 38.25 Mbps

During **S3, F3, F2** and **F1** are sent on the Link so Bandwidth utilized is as follows

F3 – 17.66 Mbps

F2 – 20.34 Mbps

F1 – 22.80 Mbps

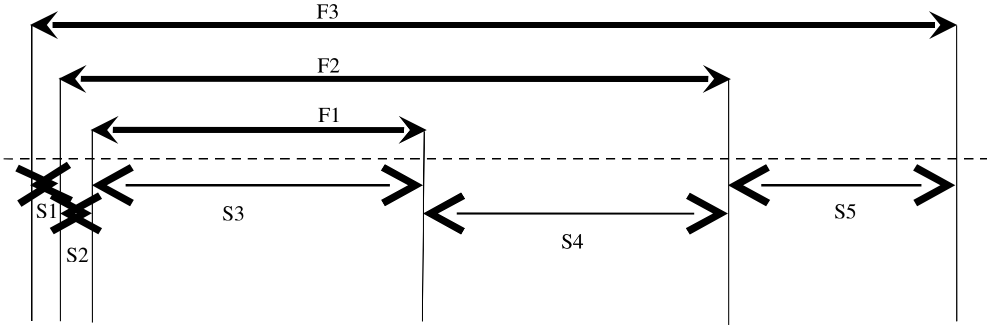
During **S4, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 23.87 Mbps

F2 – 23.04 Mbps

During **S5** only **F3** is sent on the Link so Bandwidth utilized is **35.30 Mbps**

Trial 5



Number of Bytes Sent	Time Elapsed	Bandwidth Utilised
S1 – 1662744 Bytes	S1 – 0.341395 S	S1 – 38.96 Mbps
S2 – 1718568 Bytes	S2 – 0.283206 S	S2 – 48.55 Mbps
S3 – 15452032 Bytes	S3 – 2.039847 S	S3 – 60.60 Mbps
S4 – 53421064 Bytes	S4 – 8.618819 S	S4 – 49.59 Mbps
S5 – 46411528 Bytes	S5 – 10.270640 S	S5 – 36.15 Mbps

Total number of Bytes Sent = 118665936 Bytes => (949327488 bits)

Total Elapsed Time in Seconds = 21.553907 Seconds

Average Bandwidth Utilized = 44.04565 Mbps

Fig. 12 Start and Stop Times of the File Transfers.
Number of bytes transferred in each interval.

During **S1** only **F3** is sent on the Link so Bandwidth utilized is **38.96 Mbps**

During **S2, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 10.30 Mbps

F2 – 38.25 Mbps

During **S3, F3, F2** and **F1** are sent on the Link so Bandwidth utilized is as follows

F3 – 16.24 Mbps

F2 – 21.30 Mbps

F1 – 23.07 Mbps

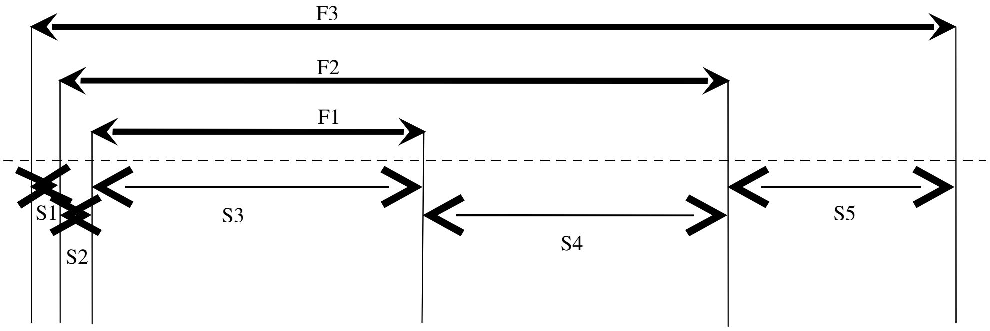
During **S4, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 25.79 Mbps

F2 – 23.80 Mbps

During **S5** only **F3** is sent on the Link so Bandwidth utilized is **36.15 Mbps**

TRIAL 6



Number of Bytes Sent	**Time Elapsed**	**Bandwidth Utilised**
S1 – 2223240 Bytes	S1 – 0.305535 S	S1 – 58.21 Mbps
S2 – 974080 Bytes	S2 – 0.281557 S	S2 – 27.68 Mbps
S3 – 15207592 Bytes	S3 – 2.357509 S	S3 – 51.61 Mbps
S4 – 55814420 Bytes	S4 – 9.469518 S	S4 – 47.15 Mbps
S5 – 44373424 Bytes	S5 – 10.369299 S	S5 – 34.23 Mbps

Total number of Bytes Sent = 118591656 Bytes => (948733248 bits)

Total Elapsed Time in Seconds = 22.783418 Seconds

Average Bandwidth Utilized = 41.64 Mbps

Fig. 14 Start and Stop Times of the File Transfers.
Number of bytes transferred in each interval.

During **S1** only **F3** is sent on the Link so Bandwidth utilized is **58.21 Mbps**

During **S2, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 25.41 Mbps

F2 – 2.27 Mbps

During **S3, F3, F2** and **F1** are sent on the Link so Bandwidth utilized is as follows

F3 – 16.34 Mbps

F2 – 15.27 Mbps

F1 – 20.00 Mbps

During **S4, F3** and **F2** are sent on the Link so Bandwidth utilized is as follows

F3 – 23.62 Mbps

F2 – 23.53 Mbps

During **S5** only **F3** is sent on the Link so Bandwidth utilized is **34.23 Mbps**

The results show that the link is not overloaded and to observe better results the system needs to be overloaded.

A surprising finding was that the time to transport a file improved (with the module inserted in the kernel) for all the three files (small, medium and large). It was expected that the time would decrease considerably for the small files, and if the time increased in case of the large file would be by a small amount. The reason for the counter intuitive result may be related to the fact that even while the three file transfers were in progress simultaneously, the link utilization was only about 55 – 60%. This, together with the TCP feed back may have caused the uniform improvement.

Another reason for an improved transfer time in all the three files may be attributed to the fact that there are fewer drops (with the module inserted) at the software queue associated with a network device when compared to the case when the module is not inserted. This is because packets are inserted in all the three queues (software queues) instead of being inserted only in one queue.

This point can be investigated in future projects.

7 FUTURE IMPROVEMENTS

The future prospects of this project are to take into consideration the type of flows. If there are well known port numbers like telnet (23), FTP control channel (21) we may want to create nodes in the front end of the linked list. These ports create a lot of packets but each with very little data. So instead of wasting time traversing the linked list at the hook, if we had nodes corresponding to these port numbers at the front of the linked list it would save time. Another alternative could be to always put the packets from these ports (FTP control or telnet) in q0 as they may have important information. It also saves the time at the hook, as the computation is reduced. We can also change the scheduling mechanism and increase the number of queues (9 instead of 3), thus we can get really fine in segregating the response, but at the cost of additional computation involved.

One more feature that can be added is to put all UDP packets in q0, the queue with the highest priority, as real time traffic may be sent using UDP packets and giving high priority to these packets is desirable.

8 Problems Encountered

The system crashes in the scenario described below:

A ping from system 1 to system 2 (with the module inserted in system 1) using options like ‘-i’, ‘-f’, ‘-s’ made the system vulnerable.

The syntax of the ping command is given below:

ping –s 14720 –i 0.01 marconi (from hawking) or

ping –s 14729 –f marconi (from hawking)

Observation: The system worked normally for some time and then crashed. The amount of time for which the system worked was not the same in all trials.

9 References

1. Bansal, N. and Harchol-Balter, M. 2001. Analysis of SRPT Scheduling: Investigating unfairness. In Proceedings of ACM SIGMETRICS ’01.
2. D0 Code – comprehensively cross – referenced and searchable code
http://www-d0.fnal.gov/D0Code/source
3. Phrack ...a Hacker magazine by the community, for the community...
http://www.phrack.org
4. Tutorial on Kernel Recompilation
http://web.njit.edu/~ott
5. Behrouz A. Forouzan, 2003, TCP/IP Protocol Suite, Mc Graw Hill.
6. The “Networking” code in Linux, Teunis J. ott and Rahul Jain July 29, 2004.
7. A Map of the Networking Code in Linux Kernel 2.4.20 by M.Rio et al. 31 March 2004.

I APPENDIX

Some of the commands and their options learnt during the project are:

1. Ping and some of its options are described below

Ping Marconi

This sends ping packets with an interval of 1 second between successive packets. The size of the ping packet is 64 bytes (56 + 8 due to ICMP).

Ping –s 14720 Marconi

This option '-s', is used to specify the size (in bytes) of the ping packet. 8 bytes due to ICMP are added to the specified size. If the size specified is less than 8 bytes then the time option is not included in the ping packet.

Ping –i 0.01 Marconi

This option '-i', is used to specify the interval between successive ping packets. The interval specified is in seconds.

Ping –f Marconi

This option is used to flood the link with ping packets. When this option is used ping places a dot '.' On the screen for every packet transmitted and takes of a dot for every packet received.

2. mii-tool and the option used with it is described below:

This utility is used to set the status of a network device.

```
mii-tool
```

This command prints all the devices and their status.

The speed of the device can be changed with the following command

```
mii-tool –F 10baseT-FD
```

The above command forces the device to work at **10Mbps**.

```
mii-tool –F 100baseTx-FD
```

The above command forces the device to work at **100Mbps**.