

THESIS
1
2004
56979797

**LIBRARY
Michigan State
University**

This is to certify that the
thesis entitled

**CHANNEL BALANCING STRATEGIES TO OPTIMIZE
UPLINK UTILIZATION**

presented by

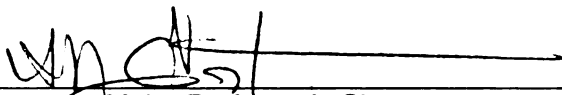
ASHOK NALKUND

has been accepted towards fulfillment
of the requirements for the

**Master of
Science**

degree in

Computer Science


Major Professor's Signature

12/9/03

Date

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE
FEB 04 2011		

CHANNEL BALANCING STRATEGIES TO OPTIMIZE UPLINK UTILIZATION

By

Ashok Nalkund

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2003

ABSTRACT

CHANNEL BALANCING STRATEGIES TO OPTIMIZE UPLINK UTILIZATION

By

Ashok Nalkund

Linux®¹ provides a rich set of features for networking including routing, firewalling, traffic-shaping and support for LAN and WAN interfaces. But the support for balancing generic outgoing traffic across multiple uplinks is very basic. This support is limited to balancing routes over the multiple equal cost links. In this thesis, we study, implement and compare different strategies to distribute outgoing traffic across multiple links. The strategies presented in this thesis are: per-packet distribution, per-route distribution, per-session distribution and bandwidth-usage-based distribution. The networking code in the Linux operating system was modified to implement these features. Experimental results indicate that per-session and per-packet based distribution strategies perform better than the other two strategies with respect to efficient utilization of bandwidth on the multiple links.

¹Linux® is a registered trademark of Linus Torvalds

COPYRIGHT

Copyright by ASHOK NALKUND (nalkunda@cse.msu.edu), 2003

DEDICATION

This work is dedicated to my brother, Sriharsha.

ACKNOWLEDGMENTS

First and foremost, I would like to express my heart-felt thanks to my adviser Prof. Ni. None of this would have been possible if not for his guidance and help . I would like to take this opportunity to thank all the wonderful people who have contributed in making Linux such a unique operating system. Without the efforts of the open-source community, Linux might not have been the robust operating system it is. I would also like to thank my friends and fellow laboratory members. I would like to particularly mention my friends (in no order) Abhishek Patil, Pradeep Padala, Ravi Parimi and Smitha Kommareddi. My thanks to the faculty and staff of the Computer Science department for their support. Last but not the least, I would like to thank my parents, my brother and my sisters for their support and confidence in me.

Contents

	Page
List of Figures	viii
Chapters:	
1. Introduction	1
2. Related Work	3
2.1 Linux Virtual Server Project	3
2.2 Round-Robin DNS	5
2.3 Policy Based Routing	6
2.3.1 Source Policy Routing	8
2.3.2 Routing with multiple uplinks	9
2.3.3 Load Balancing	10
2.4 EQL Driver: Serial IP Load Balancing	10
2.5 TEQL: “True” (or “trivial”) link equalizer	10
2.6 Linux Ethernet Bonding Driver	11
2.7 Queuing Disciplines For Linux	12
2.7.1 Classless Queues	12
2.7.2 Classful Queues	13
2.7.3 Hierarchical Token Bucket (HTB)	15
2.8 Limitations of Related Work	16
3. Linux Networking Internals	17
3.1 Data structures	20
3.1.1 sk_buff	20
3.1.2 socket	21
3.1.3 sock	21
3.1.4 net_device	22
3.1.5 Qdisc	22
3.1.6 dst_entry	23
3.1.7 neighbour	23
3.1.8 rtable	24
3.1.9 fib_table	24
3.1.10 fn_hash	25
3.1.11 fn_zone	26
3.1.12 fib_node	27
3.1.13 fib_info	27
3.1.14 fib_nh	27
3.1.15 rt_key	27

3.1.16	fib_result	28
3.1.17	nf_conntrack	28
3.1.18	nf_ct_info	28
3.1.19	ip_conntrack_tuple_hash	28
3.1.20	ip_conntrack	28
3.1.21	ip_conntrack_info	29
3.1.22	ip_conntrack_tuple	29
3.1.23	ip_conntrack_protocol	29
3.2	Packet Reception	30
3.3	Packet Transmission	36
3.4	Packet Forwarding	42
3.5	Netfilter Framework	43
4.	Channel Balancing Algorithms	48
4.1	Strategies	48
4.1.1	Route Based Channel Balancing	48
4.1.2	Per-packet Based Channel Balancing	49
4.1.3	Session Based Channel Balancing	50
4.1.4	Bandwidth Usage Based Channel Balancing	51
4.2	Implementation	52
4.2.1	Route Based Channel Balancing	53
4.2.2	Per-packet Based Channel Balancing	54
4.2.3	Session Based Channel Balancing	56
4.2.4	Bandwidth Usage Based Channel Balancing	58
4.2.5	Discussion of the strategies	61
5.	Experimentation Details	62
6.	Results	65
7.	Conclusions and Future Work	73
	Bibliography	75

List of Figures

Figure	Page
2.1 Linux Virtual Server	4
2.2 Hierarchy of qdiscs	14
3.1 Neighbor Table Structure	23
3.2 FIB Table concepts	25
3.3 FIB Table Details	26
3.4 Netfilter Hooks	44
4.1 Multiple Routes between hosts	49
4.2 Nexthop Selection Logic in Multipath Route	52
4.3 Route Based Channel Balancing	53
4.4 Per-Packet Based Channel Balancing	55
4.5 Session Based Channel Balancing	56
4.6 Bandwidth Usage Based Channel Balancing	59
5.1 Testbed Setup	63
6.1 Bandwidth usage for File Size 8KB	66
6.2 Bandwidth usage for File Size 16KB	67
6.3 Bandwidth usage for File Size 64KB	68
6.4 Bandwidth usage for File Size 128KB	68
6.5 Bandwidth usage for File Size 512KB	69
6.6 Bandwidth usage for File Size 1M	69

6.7	Bandwidth usage for File Size 2M	70
6.8	Bandwidth usage for File Size 4M	70
6.9	Bandwidth usage for File Size 8M	71
6.10	Bandwidth usage for File Size 16M	71
6.11	Bandwidth usage for Various (mixed) File Sizes	72
6.12	Transfer Times	72

Chapter 1

Introduction

Internet has revolutionized the way people live, work, entertain and communicate. A large number of homes have network connectivity these days. Network connectivity has become cheap and affordable and also critical to the performance of many small and medium businesses. Having redundant network connections is one way to ensure increased connectivity and hence increased network reachability. One would like to use the redundant network connections efficiently under normal circumstances and only in emergency situations like one of them going down, fall back on the other connections. In these normal circumstances, efficient balancing of the traffic across the redundant network connections is important to keep the utilization high and costs low¹. In this thesis, we study different strategies to balance the outgoing traffic across multiple network connections.

There are various commercial networking equipment manufacturers who provide advanced routing features in their equipment. However such equipment are very expensive for organizations with small budgets. Linux is a free operating system which provides a rich set of networking features. But the features available for balancing outgoing traffic are not sufficiently powerful and hence very basic outgoing traffic balancing can be achieved. There are several projects related to load-balancing im-

¹Network connections can be billed differently: bandwidth allocated, bytes transfered, etc

plemented under Linux. But these projects have certain limitations which restrict their application (see chapter 2).

In this research, we intend to study, implement and test different strategies for balancing outgoing traffic across multiple network connections. The goal is to effectively utilize the available network connections without too much of a performance degradation of applications depending on the network.

In chapter 2 we describe some of the related works. In chapter 3 we delve into the internal details of the networking code in Linux. The design and implementation of different strategies is discussed in chapter 4. Chapter 5 discusses the experimentation setup, configuration and testing strategies. Results of the experimentation are discussed in chapter 6. Finally we present the conclusions and some ideas for future work related to this research in chapter 7.

Chapter 2

Related Work

One often confuses load-balancing of incoming traffic with load-balancing of outgoing traffic. While there are many technologies for implementing incoming load-balancing, techniques to implement outgoing load-balancing are very few. In this chapter we describe some of the load-balancing techniques for both incoming as well outgoing traffic. This will give the reader a better understanding of the differences between the two scenarios and also an understanding of how it is achieved.

2.1 Linux Virtual Server Project

Quoting the official website of the Linux Virtual Server Project(LVS)[2]:

The Linux Virtual Server is a highly scalable and highly available server built on a cluster of real servers, with the load balancer running on the Linux operating system. The architecture of the cluster is transparent to end users. End users only see a single virtual server.

This is an incoming-load-balancer. As mentioned above, the LVS project provides a load-balancer which distributes the incoming requests among many internal servers which are invisible to the end users. Figure 2.1¹ illustrates a typical Linux Virtual Server environment. The load-balancer running the LVS accepts incoming requests

¹Source: Linux Virtual Server Project. <http://www.linuxvirtualserver.org>.

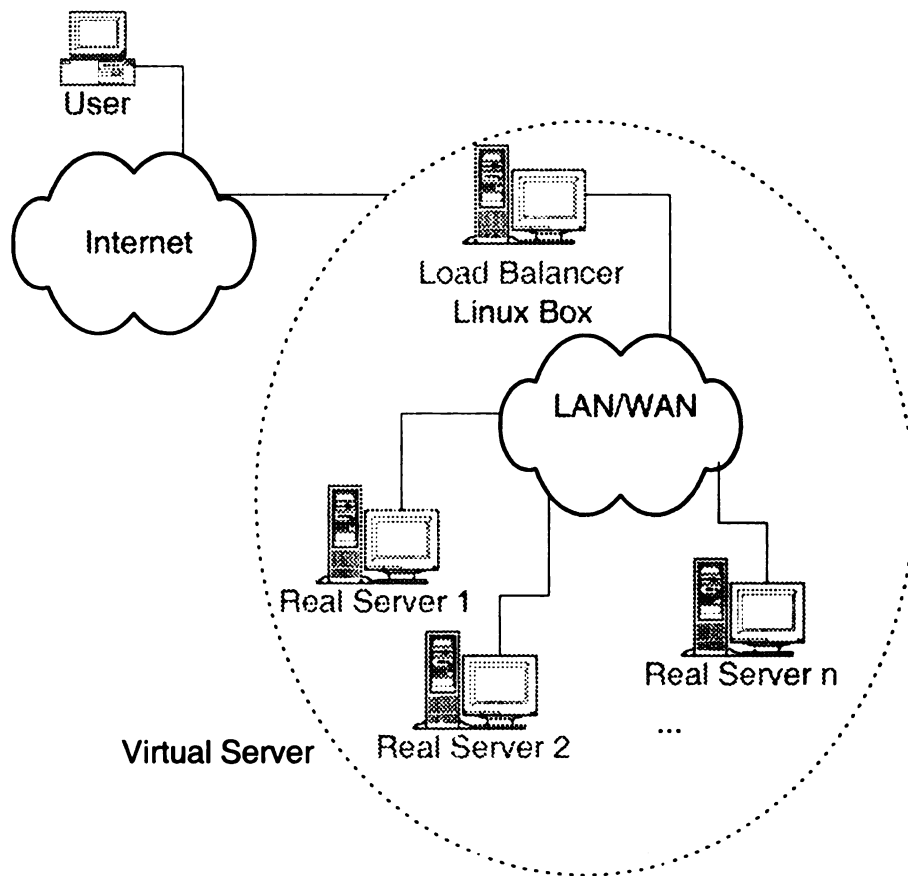


Figure 2.1: Linux Virtual Server

and then routes them to the real servers according to different configured schedulers: round-robin, weighted-round-robin, least connection, weighted-least-connection and persistent client connection. Load-balancing can be configured for individual protocol and port combination. The real servers can be added or removed transparently making the architecture very scalable. The Linux Virtual Server is implemented in three ways: NAT², IP Tunneling and Direct Routing. In NAT, the load-balancer has a public IP address and the real-servers have private IP addresses. The clients connect to the load-balancer which then performs DNAT³ on the packets and routes

²Network Address Translation

³Destination Network Address Translation

them to the internal servers. The disadvantage of this implementation is that every request and response packet has to be rewritten by the load-balancer which becomes a bottleneck. With IP tunneling, the load-balancer schedules requests to the different real servers, and the real servers return replies directly to the clients. This allows the load-balancer to handle huge amounts of requests, scaling to over 100 real servers. Due to the very low overhead involved at the load-balancer, IP Tunneling allows a load-balancer to serve as a very high-performance virtual server. The third technique is to use Direct Routing. Here also the load-balancer only processes the client-to-server half of the connection⁴ and the server responses can follow any network path to the clients. This method does not have the overhead of tunneling and hence can scale very well.

2.2 Round-Robin DNS

Round-Robin DNS [12] is another incoming-load-balancing technique used for balancing the incoming requests among a set of servers. Here a single name is mapped to different IP addresses in a round-robin fashion, thereby controlling which server a client connects to. For every client that requests a FQDN⁵-to-IP mapping, in other words a DNS lookup, a different server IP address will be chosen from the list in a round-robin manner. However, one must remember that these mappings are cached on the client system as well as on the intermediate DNS servers which might have run the query at the clients' request. Thus clients which happen to send the queries to the same intermediate DNS servers will be routed to the same IP address and hence the load-balancing is not perfect. Also the Time To Live (TTL) for the DNS record

⁴Often the requests are very short followed by long responses from the server.

⁵Fully Qualified Domain Name

is difficult to choose in these settings. If a large value is chosen, then the same IP address will be provided for a longer time which skews the load-balancing towards this IP address. On the other hand, if the TTL value is small, the load-balancing will be more efficient but the DNS queries themselves will be a bottleneck. Also depending on the amount of traffic caused by different clients, the load-balancing will be different. Thus even with the IP address rotation, the servers might be loaded differently due to the difference in the clients' requests. Further, once a server fails, the client which receives the IP address of the failed server will be unable to connect to the server even if it tries to reconnect. Only after the TTL expires and the entry for the dead server is removed from the list of IP address at the DNS server⁶ will the client be able to connect to the server.

2.3 Policy Based Routing

Policy Based Routing [4, 3] uses policies or 'rules' to make a routing decision for a packet. This feature is available in Linux by compiling the kernel with the "IP: advanced router" (*CONFIG_IP_ADVANCED_ROUTER* = yes) and "IP: policy routing" (*CONFIG_IP_MULTIPLE_TABLES* = yes) features. The routing policy database allows multiple routing tables on a Linux router. The appropriate table is looked up as specified by the 'rules'. The *iproute2*⁷ package is available for manipulating the routing policy database and the routing tables. By default, the kernel has three tables: *local*, *main* and *default* (table ID 253 denotes the *default* table). Following is the output produced by running the 'ip rule' command on a Linux system:

```
# ip rule
```

⁶It might so happen that by the time the TTL expires, other client requests have caused the dead server's IP addresses to be the one to be given to the next request.

⁷<ftp://ftp.inr.ac.ru/ip-routing/>

```
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup 253
#
```

As shown above, by default all the rules apply to all the packets. We can generate new rules and override the default routing of packets. Following are the contents of the three tables on a machine which has two interfaces (*eth0* is down, *eth1* is up)⁸:

```
# ip route show table local
broadcast 67.167.130.128 dev eth1  proto kernel  \
          scope link  src 67.167.130.153
broadcast 127.255.255.255 dev lo   proto kernel  \
          scope link  src 127.0.0.1
local 67.167.130.153 dev eth1  proto kernel  \
          scope host   src 67.167.130.153
broadcast 67.167.130.255 dev eth1  proto kernel  \
          scope link  src 67.167.130.153
broadcast 127.0.0.0 dev lo   proto kernel  \
          scope link  src 127.0.0.1
local 127.0.0.1 dev lo   proto kernel  \
          scope host   src 127.0.0.1
local 127.0.0.0/8 dev lo   proto kernel  \
          scope host   src 127.0.0.1
#
# ip route show table main
```

⁸The lines have been broken for readability

```

67.167.130.128/25 dev eth1 proto kernel \
    scope link src 67.167.130.153
127.0.0.0/8 dev lo scope link
default via 67.167.130.129 dev eth1
#
# ip route show table 253
#

```

2.3.1 Source Policy Routing

Source Policy Routing [3] balances the traffic based on the source IP address. We create new tables and add routes to the tables. Policy rules are created that specify the table to lookup for particular source IP address.

```

# echo 201 Test >> /etc/iproute2/rt_tables
# ip rule add from 192.168.1.100 table Test
# ip rule
0:      from all lookup local
32765:  from 192.168.1.100 lookup Test
32766:  from all lookup main
32767:  from all lookup 253
#

```

Now we add the appropriate routing entries in the new table:

```

# ip route add default via 192.168.1.1 dev eth0 table Test
# ip route show table Test
default via 192.168.1.1 dev eth0
#

```


2.3.2 Routing with multiple uplinks

If multiple uplinks are available, then one would like to make sure that packets arriving on an interface are answered on the same interface. This can be achieved by setting up *split access* [3] on these multiple interfaces. We create routing tables for each interface and create routes for network and default route in each table. Then we add rules specifying that packets arriving from any of the connected networks should be answered on the same interface by causing the corresponding table to be looked up.

Routing entries for table T1:

```
# ip route add $P1_NET dev $IF1 src $IP1 table T1
```

```
# ip route add default via $P1 table T1
```

Routing entries for table T2:

```
# ip route add $P2_NET dev $IF2 src $IP2 table T2
```

```
# ip route add default via $P2 table T2
```

Reply on the same interface:

```
# ip route add $P1_NET dev $IF1 src $IP1
```

Reply on the same interface:

```
# ip route add $P2_NET dev $IF2 src $IP2
```

Default route:

```
# ip route add default via $P1
```

2.3.3 Load Balancing

With the above mentioned split access in place, one can setup crude load-balancing by specifying a multipath route as the default route. With a multipath route, the kernel will balance the route lookup over all the nexthops specified in the multipath route in accordance with the weights assigned to each.

```
# ip route add default nexthop via 10.0.1.2 dev eth0 weight 1\  
    nexthop via 192.168.1.2 dev eth1 weight 1
```

The above multipath route will equalize the routes over *eth0* and *eth1* equally.

2.4 EQL Driver: Serial IP Load Balancing

EQL is a device driver available in Linux which provides a software device to load-balance IP serial links (SLIP or uncompressed PPP) to increase the bandwidth. This is useful in cases where we have two or more modems and want to increase the bandwidth by binding the modems together. We compile the kernel with the *eql* patch⁹ and configure an *eql* interface with an IP address. Then the default route is set to point to the *eql* device. The devices are then “enslaved” with the *eql_enslave* command. Devices are freed using the *eql_emancipate* command.

2.5 TEQL: “True” (or “trivial”) link equalizer

TEQL [3] is a new virtual device available in Linux which equalizes the traffic going out on the physical slave interfaces. For this virtual device to be used, the slave devices must be active, i.e be able to raise the busy signal. This device is capable of equalizing physical interfaces which have varying bandwidths but it is advisable

⁹[ftp://slaughter.ncm.com/pub/Linux/LOAD_BALANCING/eql-1.1.tar.gz](http://slaughter.ncm.com/pub/Linux/LOAD_BALANCING/eql-1.1.tar.gz)

that the bandwidths should be comparable to avoid reordering problems. To use this feature, first we insert the *sch_teql* module which creates a new device *teqlN* and a new qdisc (see section 2.7) with the same name. The physical interfaces can now be enslaved to this device and the desired routing setup.

```
# tc qdisc add dev eth0 root teql0
# tc qdisc add dev eth1 root teql0
# ip link set dev teql0 up
```

The slave interfaces are configured as they would normally be and the default route is set to point to the *teql* interface. The most restricting requirement in this technique is that both ends of the links are required to participate for it to work properly. This is sometimes not possible if the ISP is not very forthcoming with accommodating customers' requests.

2.6 Linux Ethernet Bonding Driver

The Ethernet bonding driver provides for bonding Ethernet channels together. Its called '*Etherchannel*' by Cisco, '*Trunking*' by Sun and '*Bonding*' [5] in Linux. Multiple Ethernet connections can be '*bonded*' together to act as a single channel with increased bandwidth. This requires the other end of the connection also to support bonding. Although this seems similar to the EQL driver (section 2.4), this driver manages Ethernet segments while the EQL driver manages serial lines. The kernel needs to be compiled with the '*Bonding Driver Support*' (*CONFIG_BONDING*). A bonding network interface is defined in the system configuration files similar to the other network interfaces¹⁰. The IP information is provided for the bonding interface. The physical Ethernet devices are not configured with any IP information and are

¹⁰For RedHat Linux systems, this can be in */etc/sysconfig/network-scripts/ifcfg-bond0* file.

made slaves of the bonding device. The MAC address for the bonding interface is chosen from the first enslaved Ethernet interface. The bonding driver can be configured to take care of failing interfaces by monitoring their MII link status.

2.7 Queuing Disciplines For Linux

Linux provides a very rich set of features for managing bandwidth by means of *queues*. With queuing we can control and shape the outgoing traffic from a machine. There are different queuing disciplines available in Linux. Some of them are classless which just accept data and only reschedule, delay or drop it. Classful queues allow different kinds of traffic to be treated differently. The term *qdisc* is used to refer to a queuing discipline.

2.7.1 Classless Queues

These queues are used to shape traffic on the entire interface, without any subdivisions. All packets are treated the same. Following are some of the classless queues available on Linux.

pfifo_fast

pfifo_fast [3] is a queue which performs a ‘First In, First Out’ scheduling of the arriving packets. There are three bands which are prioritized, band 0 receiving higher priority than band 1, which receives more priority than band 2. Band 0 contains packets which have the ‘minimum delay’ TOS¹¹ flag set. As long as there are packets in band 0, band 1 is not processed (similarly for band 1 and band 2). The TOS value in the packet is used to map the packet to one of the bands. The mapping can itself be specified with *priomap* which is set by the kernel. The queue length can be

¹¹Type of Service

configured with *ifconfig* or *tc* command. This is the default qdisc for any interface. Adding any other qdisc to the interface causes this qdisc to simply return without any action.

Token Bucket Filter (TBF)

The *TBF* [3] qdisc ensures that packets are sent out at some administratively set rate while allowing for short bursts of traffic which may exceed the set rate. TBF consists of a buffer (bucket) which is filled with *tokens* at a specific rate called the *token rate*. The token rate is the administratively set rate mentioned above. Each token dequeues a packet from the data queue and sends it out. If the token rate is greater than the data rate, tokens simply accumulate up to the bucket size. Excess tokens are then discarded. If the data rate is greater than the token rate, then some packets will be dropped until tokens are available. The accumulation of the tokens when the data rate is smaller than the token rate allows for short bursts of data exceeding the token rate. The bucket size (burst size), token rate and various other parameters are configurable. Note that in actual implementation, the token corresponds to bytes and not to packets. This qdisc is very convenient to slow down an interface to match the actual available bandwidth as in case of cable modems and DSL lines.

2.7.2 Classful Queues

In classful queues [3], classes and filters are used to manage the bandwidth. Filters added to a classful queue look at the packet and decide what to do with the packet and return this decision to the classful queue. The classful queue then enqueues the packet in the appropriate class. The filters and classes can be nested to achieve better control over the traffic as shown in Figure 2.2. The class at the bottom of this nesting

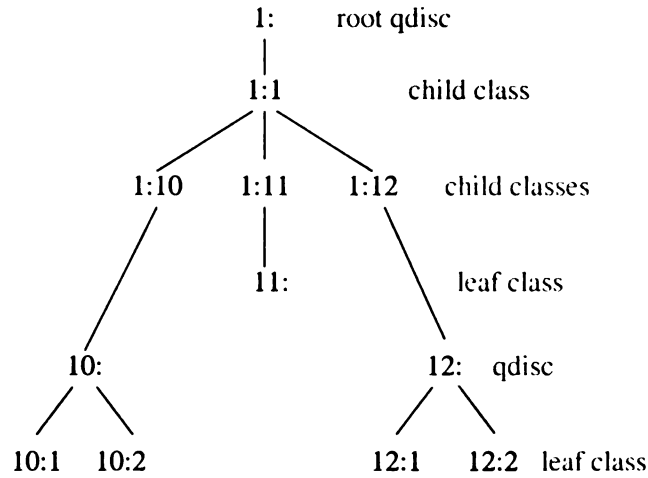


Figure 2.2: Hierarchy of qdiscs

enqueues the packet to the qdisc it contains. When a packet arrives for transmission, the *root qdisc* enqueues it to the appropriate child class by filtering the packet. The child class can in turn apply filtering to enqueue the packet to one of its child classes. When a packet needs to be dequeued by the kernel to send to the interface, the root qdisc gets a dequeue request which is passed on to the child class. The class with the lowest handle containing a packet returns the packet to the kernel. The handle for a class is specified when the class or qdisc is added to the interface. Due to this nesting, the child class cannot dequeue a packet faster than its parent will allow.

Class Based Queuing (CBQ)

CBQ [3] is a very widely used and complicated classful qdisc which also works as a shaper. However the shaping algorithm used in CBQ is not very precise and hence causes CBQ to behave unexpectedly in some situations like not achieving the desired traffic rate. CBQ uses the idle time between requests for packets by the hardware layer to shape the traffic. Determining the idle time is a very tricky issue due to various reasons like the driver implementation for the hardware, hardware details

(bus speed), etc. Even with these limitations, CBQ works well in many circumstances. The parameters which configure the shaping include the average packet size, physical bandwidth, required rate of traffic, etc. The queuing properties of CBQ are configured by specifying the *weight*, *priority* and *allocated amount* for each inner class. The *priority* property makes CBQ a priority queue where packets are dequeued from the highest priority inner queue before checking lower priority inner queues. The *amount* specifies how much data an inner queue can send out when requested.

2.7.3 Hierarchical Token Bucket (HTB)

Hierarchical Token Bucket (HTB) [3] is similar to CBQ but instead of working with idle time calculations to shape traffic, it works as a classful Token Bucket Filter. HTB can be used to specify how to divide the available bandwidth for different purposes and also how to share the bandwidth among them, by lending and borrowing, if required. This is a scalable queuing discipline unlike CBQ which gets complex even for a simple setup. If a class requests bandwidth less than the amount assigned, the remaining bandwidth is distributed to the other classes that request for bandwidth. Thus excess bandwidth can be lent to other classes. First we add a HTB qdisc as root qdisc to the interface. Then we add a *root class*¹². This is required because bandwidth cannot be shared among the root classes. Hence we create this root class and then make the actual classes children of this root class. Then we add the classes specifying the bandwidth guaranteed for the class. Further we add filters to classify the packets into these classes.

¹²A root class is a class with the root qdisc as its parents

2.8 Limitations of Related Work

The projects discussed in this chapter are unsuitable for achieving generic outgoing traffic balancing. Some of these works are applicable only to incoming traffic, for example Linux Virtual Server Project and Round-Robin DNS (see sections 2.1 and 2.2 respectively). Others like policy routing (see section 2.3) have restrictions over what traffic they balance. The rules in policy routing specify what kind of traffic will be balanced, for example packets with a specific source IP address. To balance the generic outgoing traffic, enough rules would have to be created to cover all kinds of packets in the generic traffic. EQL and Ethernet bonding drivers (see sections 2.4 and 2.6 respectively) apply only to specific types of interfaces (IP serial links and Ethernet interfaces respectively). Further, EQL, TEQL and Ethernet bonding, also require the cooperation of the ISP providing the connections which may not be possible in some case. Finally queuing disciplines (section 2.7) can only be used to shape the traffic which is already queued for an interface. This does not provide for balancing the traffic across multiple interfaces.

Thus we see that balancing the generic outgoing traffic is very difficult if not impossible with the available tools.

Chapter 3

Linux Networking Internals

The networking code in Linux is one of the most complex and largest piece of code in the kernel, in fact, the networking code makes up for around 20% of the entire Linux kernel code. In this chapter we discuss the details of the networking code in the Linux kernel and trace the path of a packet received, sent out and forwarded by the Linux kernel.

The Linux kernel supports different network architectures including IPv4 Internet protocols (*PF_INET*), IPv6 Internet Protocols (*PF_INET6*), Appletalk (*PF_APPLETALK*), AX.25 (*PF_X25*), IPX - Novell Protocols (*PF_IPX*), etc. It also allows for different scheduling algorithms like Packet First-In-First-Out (pfifo), Byte First-In-First-Out (bfifo), Token Buffer Filter (TBF), Stochastic Fairness Queuing (SFQ), etc. The networking code in Linux is based upon the Swansea University Computer Society's NET3.039 code. The current version of the networking code in Linux 2.4 kernel is NET4.0. The code has been divided into family of protocols and further into layers. Each layer has a well-defined interface with the adjacent layers. To make the code efficient, copying of data is avoided between the layers, instead, enough space is reserved for every packet for the header/trailer for the different layers. In this chapter, we concentrate on the IPv4 Internet Protocols family.

When an application generates traffic, it sends it to the transport layer (TCP or UDP) through the socket interface. The transport layer then passes it on to the network layer (IP). This layer looks at the destination of the packet and decides whether the destination is the local host or a different host. If the packet is for the same host, it passes it back to the upper layer for passing it on to the local destination application. If the packet is for another host, the kernel searches the route cache, and if required the Forwarding Information Base (FIB), for a route to the destination. It prepares the packet for transmission to the destination and hands the prepared packet to the interface to be sent out on the physical medium.

On the other hand, when a packet arrives at an interface and if the hardware address belongs to the host or is a broadcast, it queues the packet for processing by the kernel. The IP layer looks at the destination and if the packet is for the host, it passes it on to the transport layer for further processing. The transport layer then de-multiplexes and sends the data to the appropriate program. If the packet is for another host and forwarding is enabled on the host, the routing cache and FIB tables are consulted to route the packet correctly. If forwarding is disabled, the packet is dropped.

In Linux, the work to be done when an interrupt is received, is divided into two parts: top-half and bottom-half. Different tasks in the system have different bottom-half handlers, like networking bottom-half handler, timer bottom-half handler, SCSI bottom-half handler etc. In the top-half, only the most essential work, like moving data from the hardware buffer to kernel buffer, is performed. The top-half sets a flag to indicate that the kernel needs to complete the work for the interrupt by running the bottom-half handler. When the kernel scheduler runs, it checks these flags and runs the corresponding bottom-half handlers. These bottom-half handlers complete the

work related to the interrupt. For example, the networking bottom-half handler gets a packet from the queue, checks for the protocol type and hands it to the protocol-specific receive function.

In 2.4 versions of the Linux kernel, the bottom-half handler for networking is implemented as a *softirq*. Bottom-halves are very demanding on the system: only one bottom-half can run on a system irrespective of the number of CPUs the system has. Softirqs can run on multiple CPUs at the same time and are used for very very high frequency threaded jobs. The bottom-half handler for networking *net_bh()* has now been replaced with the *net_rx_action()* softirq in the 2.4 versions.

Now that you have a basic understanding of networking in Linux, you may skip to the next chapter if you do not wish to get entangled in the intricacies of the Linux networking code. However for those who decide to continue, having the Linux kernel code handy is strongly advised. This will let you browse the actual code as you read about various functions and data structures. The source and header filenames in this document are relative to the kernel source location, usually */usr/src/linux*.

The following sections are based on Linux kernel version 2.4.18 which can be obtained from the Linux Kernel website: <http://www.kernel.org>. We should note that in the following discussions, a number of issues like fast-routing, multiprocess systems, etc are not discussed. The discussion of these issues would complicate the discussion and the actual control path of the packet would not be clearly understood. Hence, only the control paths which would be taken in the most general cases are discussed. We also assume that all the interfaces involved are Ethernet interfaces.

3.1 Data structures

At the highest level, BSD sockets are used to represent a network connection. These structures hide the implementation details of the different protocol families. INET sockets are specific to IPv4 protocol family. The generic operations on the BSD sockets invoke functions specific to the protocol family such as the INET socket operations. In the following sections we discuss some of the important data structures in the Linux networking code.

3.1.1 `sk_buff`

`sk_buff`¹ is one of the most important data structure in the Linux networking code. It represents a buffer which provides control information and data to be transmitted or received. It reserves enough storage so that at each layer of the networking stack, the headers or trailers can be added without any additional overhead. The data is copied only twice through its entire journey through the networking layers: once from user-space to the kernel-space and once from the kernel-space to the output medium. Important members of this data structure include pointers to the head of the buffer list, next and previous pointers, pointer to the `sock` (see section 3.1.3) structure which owns this buffer, `net_device` (see section 3.1.4) pointer to the device from which this packet arrived or on which this packet will be sent out, pointers to the different network layer headers and trailers, checksum, length of the data currently in the buffer, pointer to the connection tracking structure `nf_ct_info` (see section 3.1.18) and control information for the upper layer protocol.

¹`include/linux/skbuff.h`

3.1.2 socket

*socket*² represents a BSD socket in the networking code. Some of its members include the state of the socket, flags, pointer to the inode associated with the socket, pointer to the file associated with the socket, a pointer to the low-level *sock* associated with the socket and a pointer to the *proto_ops* structure which contains function pointers to handle different operations on the socket. For a TCP socket, it points to the *inet_stream_ops*³ which contains functions to handle a TCP socket bind, connect, listen and other operations. Most often, the *sock* is an INET socket.

3.1.3 sock

The *sock*⁴ structure represents a messed up collection of network layer and bits of transport layer information for a socket. Some of the most important members of this structure are pointers to the next and previous *sock* structures (*next*, *prev*), local and foreign addresses (*rcv_saddr*, *daddr*), source and destination port numbers (*sport*, *dport*), address family (*family*), state of the socket (*state*), pointer to the destination cache entry (*dst_cache*), receive and send/write queues (*receive_queue*, *write_queue*), pointer to a *struct proto* structure (*prot*), which has pointers to the transport layer functions to operate on the socket, transport layer options structure (*tp_pinfo*), pointer to the parent BSD socket (*socket*). The *prot* points to *tcp_prot*⁵ for TCP sockets and *udp_port*⁶ for UDP sockets.

²include/linux/net.h

³net/ipv4/af_inet.c

⁴include/net/sock.h

⁵net/ipv4/tcp.c

⁶net/ipv4/udp.c

3.1

dat

rela

net

ter

int

ter

IP

en

(se

3.

q

pu

op

(c

d

B

3.1.4 net_device

*net_device*⁷ represents a network device in the Linux networking code. It includes data related to higher levels of networking as well as basic I/O. The important fields related to networking include the name of the interface (*name*), pointer to the next *net_device* (*next*), unique device identifier (*ifindex*), pointer to functions to get interface statistics (*get_status* for wired and *get_wireless_stats* for wireless interfaces), interface address (*dev_addr*), pointers to the queuing disciplines attached to the interface (*qdisc*), pointers to specific higher level protocols including AppleTalk, IPv4, IPv6 etc, transmit queue length (*tx_queue_len*), pointers to functions to perform different actions on the interface like start transmit (*hard_start_xmit()*), set MAC address (*set_mac_addr()*).

3.1.5 Qdisc

*Qdisc*⁸ represents a queuing discipline. It has pointers to the queuing and dequeuing functions (*enqueue*, *dequeue*), pointer to the next *Qdisc* structure (*next*), pointer to a *Qdisc_ops* structure (*ops*), which has pointers to functions for different operations of specific disciplines, pointer to the interface to which this qdisc is bound (*dev*), a statistics structure (*struct tc_stats stats*)⁹, which tracks the bytes, packets, drops and other information of the qdisc. *ops* points to *tbqdisc_ops*¹⁰ for a Token Bucket Filter (TBF) qdisc and *pfifo_qdisc_ops*¹¹ for a Packet First-In-First-Out qdisc.

⁷include/linux/netdevice.h

⁸include/net/pkt_sched.h

⁹include/net/pkt_sched.h

¹⁰net/sched/sch_tbf.c

¹¹net/sched/sch_fifo.c

3.1

and

pol

imp

use

wh

the

3.

3.1.6 dst_entry

This represents a destination cache entry. *dst_entry*¹² contains pointers to input and output functions and information about a route. Its members include device pointer (*dev*), pointer to the neighbor (*neighbour*), hardware header pointer (*hh*), input and output functions (*input*, *output*), various other information like flags, last used time, path MTU, etc. It also holds a pointer to a *neigh_ops*¹³ structure (*ops*) which contains family, protocol and pointers to functions for rerouting, destroying the route, etc.

3.1.7 neighbour

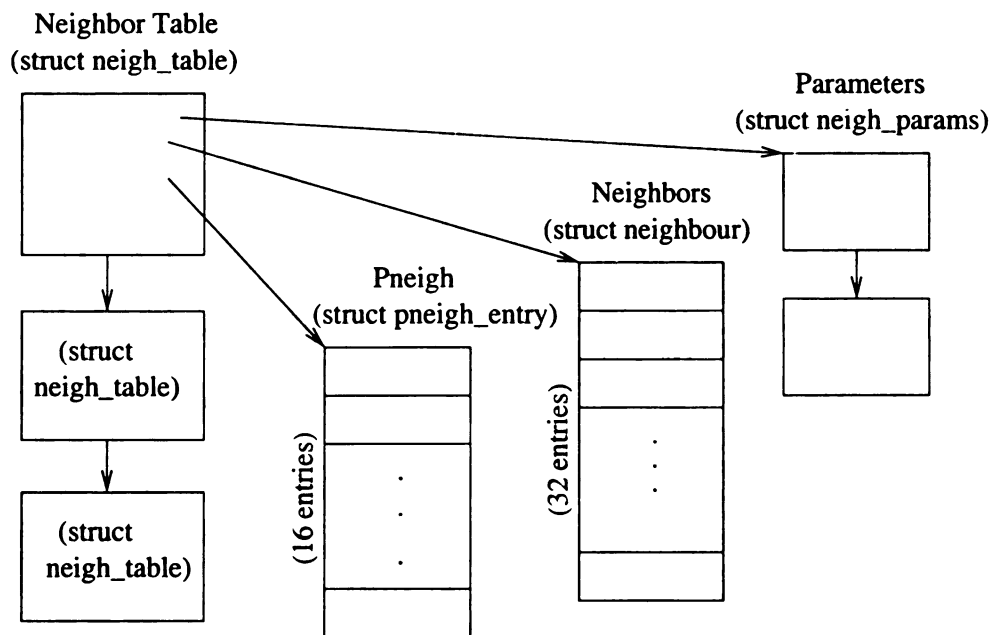


Figure 3.1: Neighbor Table Structure

¹²include/net/dst.h

¹³include/net/dst.h

This represents information about the neighbour connected to the system one hop away. Like other objects in the Linux kernel, it has pointers to the next *neighbour* structure (*next*). Other information in the structure include pointer to the *neigh_table*¹⁴, pointer to the device (*dev*), flags, hardware address (*ha*), pointer to the output function (*output*), pointer to the *neigh_ops*¹⁵ structure (*ops*) which has pointers to the functions for different operations like destroying the *neighbour* structure, transmitting a packet from the queue (*queue_xmit*).

3.1.8 rtable

An entry in the routing table is represented by a *rtable*¹⁶ structure. This contains a union (*u*) of destination cache entry (*dst*) and the next *rtable* structure (*next*). Thus the union allows the same member to be used as a pointer to the next *rtable* or as a pointer to the destination cache entry for the route. Other information in *rtable* are the flags and type of the route (*rt_flags*, *rt_type*), source and destination addresses (*rt_src*, *rt_dst*), interface index (*rt_iif*), gateway address (*rt_gateway*) and key for route lookup (*key*).

3.1.9 fib_table

The use of multiple routing tables in Linux is made possible by the use of *fib_table*¹⁷ structures (see Figures 3.1.9¹⁸ and 3.1.10¹⁹). This structure contains a table identifier (*tb_id*), table manipulation functions like table lookup (*tb_lookup*),

¹⁴include/net/neighbour.h

¹⁵include/net/neighbour.h

¹⁶include/net/route.h

¹⁷include/net/ip_fib.h

¹⁸Source: Linux IP Networking, <http://www.cs.unh.edu/cnrg/gherrin>

¹⁹Source: Linux IP Networking, <http://www.cs.unh.edu/cnrg/gherrin>

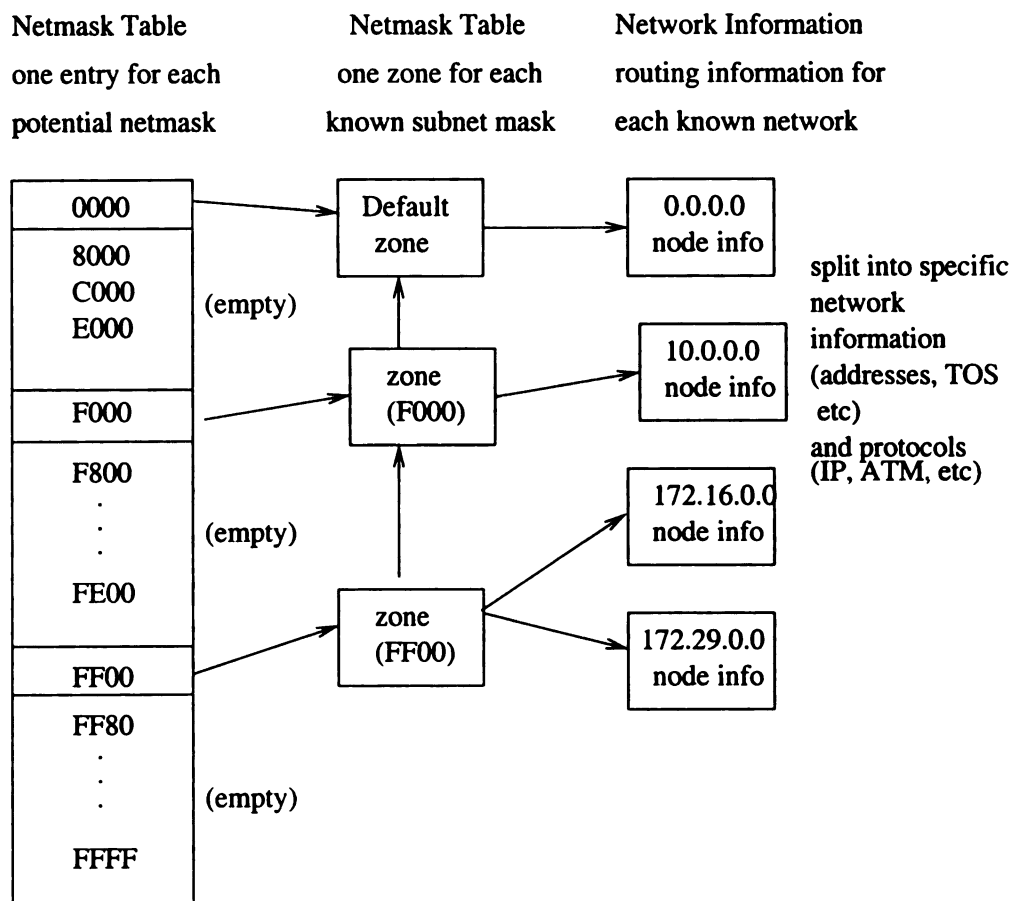


Figure 3.2: FIB Table concepts

insertion (*tb_insert*), deletion

(*tb_delete*) and pointer to the associated FIB hash table (*tb_data*).

3.1.10 *fn_hash*

*fn_hash*²⁰ is the FIB hash table which contains pointers (*fn_zones*) to the individual zones representing netmask. The number of zones is 33 which represents one zone per bit in the mask. The *fn_zone_list* points to the first non-empty zone in the table. The

²⁰net/ipv4/fib_hash.c

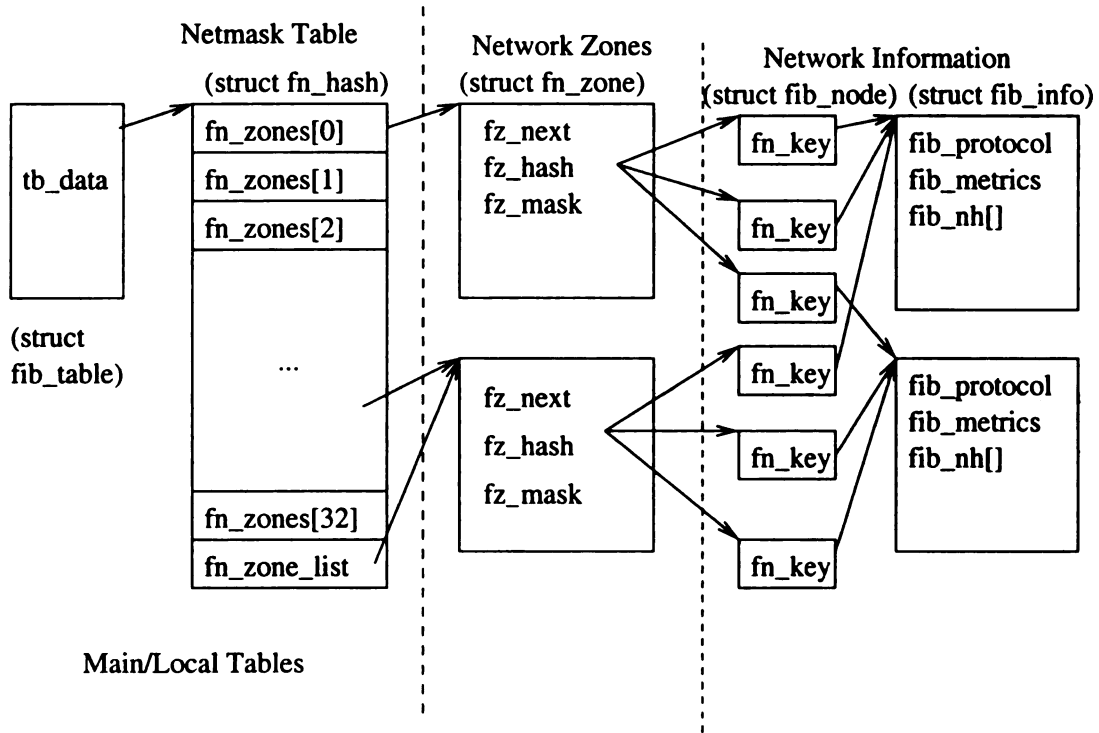


Figure 3.3: FIB Table Details

zones are organized from most specific (longest prefix) to the most general (shortest prefix), thus the `fn_zone_list` points to the most specific zone in the table.

3.1.11 `fn_zone`

Each `fn_zone`²¹ represents routing data for one netmask. Thus for the 32-bit IPv4 addresses, there are 33 such zones. It contains a pointer to the next non-empty zone (`fz_next`), pointer to the `fib_node` hash table (`fz_hash`), number of entries (`fz_ent`), number of buckets in the hash table (`fz_divisor`), mask to generate the hash (`fz_hashmask`), mask for the zone (`fz_mask`) and order of the zone which is the index of the zone in `fn_hash`.

²¹`net/ipv4/fib_hash.c`

3.1.12 `fib_node`

`fib_node`²² denotes information specific to a route. Its fields include pointer to the next node (`fn_next`), pointer to `fib_info` structure (`fn_info`), the search key (`fn_key`), TOS (`fn_tos`), type of route (`fn_type`), scope of the route (`fn_scope`) and state (`fn_state`). The key, a 32 bit unsigned number, is formed by ANDing the destination address with the netmask of the zone. Thus it encodes both the destination address and the mask of the route.

3.1.13 `fib_info`

`fib_info`²³ contains information that can be shared between many routes. This includes the route metrics (`fib_metrics`), number of nexthops available for the route (`fib_nhs`), array of `fib_nh` structures (`fib_nh`), network protocol (`fib_protocol`), status information (`dead`) and pointers to the next and previous `fib_info` structures (`fib_next`, `fib_prev`).

3.1.14 `fib_nh`

`fib_nh`²⁴ represents the nexthop information in a route. It contains a pointer to the device (`nh_dev`), flags, scope, weight of the nexthop (`nh_weight`), outgoing interface index (`nh_oif`) and the address of the gateway (`nh_gw`)

3.1.15 `rt_key`

`rt_key`²⁵ contains information which form the key for route lookup. This includes the source and destination addresses (`src`, `dst`), incoming and outgoing interface in-

²²`net/ipv4/fib_hash.c`

²³`include/net/ip_fib.h`

²⁴`include/net/ip_fib.h`

²⁵`include/net/route.h`

dices (*iif*, *oif*), firewall mark if configured (*fwmark*), TOS (*tos*) and scope of the route (*scope*).

3.1.16 **fib_result**

A *fib_result*²⁶ structure is returned when a route is found in the routing table. This structure contains the prefix length (*prefixlen*), the index of the next hop selected (*nh_sel*), type and scope of the route (*type*, *scope*) and the routing rule (*fib_rule*) if multiple tables support is configured.

3.1.17 **nf_contrack**

*nf_contrack*²⁷ represents a general netfilter connection. This is used only for usage count (*use*) and deletion through the *destroy* function pointer. Other specific connection structures, for example *ip_contrack* contain this structure within them.

3.1.18 **nf_ct_info**

nf_ct_info contains a pointer to a *nf_contrack* structure (*master*).

3.1.19 **ip_contrack_tuple_hash**

ip_contrack_tuple_hash is an entry in the connection hash table. It contains the *ip_contrack_tuple* (*tuple*) and the pointer to the IP connection (*ctrack*).

3.1.20 **ip_contrack**

*ip_contrack*²⁸ represents an IP connection in the connection tracking mechanism. It contains an *nf_contrack* structure (*ct_general*) for usage count, a couple of

²⁶include/net/ip_fib.h

²⁷include/linux/skbuff.h

²⁸include/linux/skbuff.h

ip_conntrack_tuple_hash structures (one each for original and reply direction, *tuple_hash*), status to indicate whether we have seen packets in both the directions (*status*), pointer to a *ip_conntrack_helper* if registered. If this connection is expected by another connection, that connection is stored in a *nf_ct_info* structure (*master*). An array of *nf_ct_info* structures which indicate the relation of the packet with the connection is available as *infos*.

3.1.21 *ip_conntrack_info*

*ip_conntrack_info*²⁹ is an enumeration type which indicates the status of the connection. The different values of this type include *IP_CT_ESTABLISHED* to indicate an established connection, *IP_CT_NEW* to indicate a new connection etc.

3.1.22 *ip_conntrack_tuple*

*ip_conntrack_tuple*³⁰ contains the information related to an IP connection. The fields in this structure hold the destination information (IP address, port for TCP and UDP, and type and code for ICMP), source information (IP address, port for TCP and UDP, and type and code for ICMP) and transport layer protocol identifier. The source related information is manipulable and is encapsulated in an *ip_conntrack_manip*³¹ structure.

3.1.23 *ip_conntrack_protocol*

ip_conntrack_protocol holds information and function pointers for specific protocols. Its fields include the protocol identifier (*proto*), protocol name (*name*), function pointer to convert packet to tuple (*pkt_to_tuple*), function pointer to invert the tuple

²⁹include/linux/netfilter_ipv4/ip_conntrack.h

³⁰include/linux/netfilter_ipv4/ip_conntrack_tuple.h

³¹include/linux/netfilter_ipv4/ip_conntrack_tuple.h

for the packet (*invert_tuple*), function pointer to handle the packet (*packet*), function pointer to create a new connection of this protocol (*new*), etc. This is represented by *ip_conntrack_protocol_tcp*³² for TCP connections, *ip_conntrack_protocol_udp*³³ for UDP connections and by *ip_conntrack_protocol_icmp*³⁴ for ICMP connections.

3.2 Packet Reception

The journey of a packet in the kernel begins with the hardware interrupt of the interface receiving the packet from the physical medium. The interrupt service routine (ISR) for the interface checks for the length of the queue (ring buffer overflow), any transmission errors like checksum, frame errors, etc. Then it allocates space (an *sk_buff* structure, usually called *skb*). It checks for the packet's protocol ID and whether the packet is a hardware broadcast, multicast or destined for other host³⁵. It then calls the *netif_rx()*³⁶. *netif_rx()* enqueues the passed *skb* at the end of the receive queue for the CPU and raises the networking receive softirq (*NET_RX_SOFTIRQ*) flag.

The *NET_RX_SOFTIRQ* is mapped to the *net_rx_action()*³⁷. When the kernel finds that the *NET_RX_SOFTIRQ* has been raised, it calls *net_rx_action()*. In this function, packets (*skb*) are de-queued from the CPU's receive queue. For every packet type, i.e. protocol family, that has been registered, the interface registering the protocol family is checked to match the incoming interface. If the interfaces match or

³²`net/ipv4/netfilter/ip_conntrack_proto_tcp.c`

³³`net/ipv4/netfilter/ip_conntrack_proto_udp.c`

³⁴`net/ipv4/netfilter/ip_conntrack_proto_icmp.c`

³⁵When an Ethernet interface is in promiscuous mode, packets destined for other MAC addresses also will be processed.

³⁶`net/core/dev.c`

³⁷`net/core/dev.c`

if the protocol applies to all the interfaces, the function (*packet_type.func*) to handle packet of the particular protocol family is called. For INET family, this function is *ip_rcv()*³⁸. As long as the queue has packets and there is time, packets are de-queued and processed.

ip_rcv() is the main IP receive function. This function gets a pointer to the IP header of the packet and performs checks like IP version, checksum, minimum packet size, etc. If these checks pass, it invokes the *NF_IP_PRE_ROUTING* hook in the netfilter framework. If the hook returns a success, then *ip_rcv_finish()*³⁹ is called.

ip_rcv_finish() invokes *ip_route_input()*⁴⁰ if the *skb->dst*, where *skb* is a *sk_buff* structure discussed in 3.1.1, is not already set correctly. The *skb->dst* will be set correctly if the source address of the packet is a loopback interface, otherwise it is not set. If *ip_route_input()* returns an error, the packet is dropped, i.e the buffer (*skb*) is freed and an error code *NET_RX_DROP* is returned. If *ip_route_input()* returns success, the packet is processed further. If the packet contains any IP options, they are processed. Finally, *skb->dst->input()* is called to process the packet. In INET family, this function maps to *ip_forward()*⁴¹ for packets destined to other host and to *ip_local_deliver()*⁴² for local packets. The journey of the packet along *ip_forward()* is described in 3.4.

ip_route_input() is the function where the route lookup starts. First a hash is generated from the source address, destination address, incoming interface index and Type of Service (TOS) of the packet. This is used for lookup in the route cache hash

³⁸net/ipv4/ip_output.c

³⁹net/ipv4/ip_input.c

⁴⁰net/ipv4/route.c

⁴¹net/ipv4/ip_forward.c

⁴²net/ipv4/ip_input.c

table. The corresponding bucket in the route cache hash table is traversed searching for a route that matches the destination address, source address, incoming interface, outgoing interface index set to 0 and the TOS. If firewall marking is enabled in the kernel (*CONFIG_IP_ROUTE_FWMARK* = yes), then firewall mark is also matched with the route cache entry. If an entry is found, then the entry is updated with the last used time, number of times used and a pointer to the entry is made available in the packet's *dst*, which is a *dst_entry* structure discussed in 3.1.6, and the function returns. The route cache entry contains *dst_entry* as its member. This structure, as discussed in 3.1.6, holds all the information required for the packet to be sent along the selected route which includes the neighbour information, the device, hardware header cache, the output function to be used to send the packet along the route, etc. If a route cache entry is not found, then *ip_route_input_mc()*⁴³ is invoked if it is a multicast packet, otherwise *ip_route_input_slow()*⁴⁴ is invoked. In this work, we do not discuss *ip_route_input_mc()* and only concentrate on *ip_route_input_slow()*.

ip_route_input_slow() constructs a key (*rt_key*) based on the destination address, source address, TOS, firewall mark if configured, incoming interface index, route scope set to *RT_SCOPE_UNIVERSE*⁴⁵. If the source address is multicast address or the source address has a bad address class or the source address is a loopback address⁴⁶, then an error is returned which causes the packet to be dropped. Other checks like source and destination addresses being 0, source address having a zero network address are also checked and the packet is dropped if any such test succeeds.

⁴³net/ipv4/route.c

⁴⁴net/ipv4/route.c

⁴⁵Route scopes are used to indicate the type of route: site, link, host, universe.

⁴⁶A packet with a loopback source address will have the *skb->dst* already set by the output routine which would have already been checked in *ip_rcv_finish()* before invoking *ip_route_input()*. Hence a packet with loopback source address will never reach this function.

If the packet has a valid source and destination address, then the actual route lookup is performed. If multiple routing tables support (`CONFIG_IP_MULTIPLE_TABLES = no`) is not enabled in the kernel, `fib_lookup()`⁴⁷ is called to lookup the route in the Forwarding Information Base (FIB) tables. Otherwise, if the multiple routing tables support is enabled, then the table lookup routine `fib_lookup()`⁴⁸ is invoked. If the route lookup results in a local route, then the packet is destined for the local machine. In this case, a route cache entry is created and its fields including the search key, function to process the input packet (which is `ip_local_deliver()`), incoming interface index, etc are set. If it is not a local packet and if IP forwarding is not enabled on the interface, then the packet is dropped. At this point, if multipath is configured in the kernel and if there are more than one nexthops available for the route, `fib_select_multipath()`⁴⁹ is called which selects one of the multiple nexthops based on the route configuration. After further checks like no outgoing interface available or invalid source (should not be a broadcast or local address), a route cache entry is created and filled in with the information about the route selected. The input and output functions for the route cache entry are set to `ip_forward()` and `ip_output()`⁵⁰ respectively. Finally the route cache entry created above is inserted into the route cache by calling `rt_intern_hash()`⁵¹. One should note that only the most important functionalities of `ip_route_input_slow()` have been discussed here.

⁴⁷`include/net/ip_fib.h`

⁴⁸`net/ipv4/fib_rules.c`

⁴⁹`net/ipv4/fib_semantics.c`

⁵⁰`net/ipv4/ip_output.c`

⁵¹`net/ipv4/route.c`

*fib_lookup()*⁵² calls the table lookup routine, *tb_lookup()*, on the *local_table* and *main_table*. If they fail an error is returned else success is returned. The structure of these tables is discussed in 3.1.9.

In *fib_lookup()*⁵³, the table of routing rules is searched to find the correct routing table to use for lookup. The search is based on the source address, destination address, TOS, firewall mark and incoming interface index. Once a matching rule is found, the action specified for the rule specifies whether to proceed with the route lookup or to return an error. The lookup proceeds if the action is *RTN_UNICAST* or *RTN_NAT*. The matching rule specifies the table to use for the lookup and the *tb_lookup()* is invoked on that table. If *tb_lookup()* returns an error, the error is propagated to the caller and the packet is finally dropped.

tb_lookup() maps to *fn_hash_lookup()*⁵⁴. In this function, the FIB tables are searched for longest matching route entry. As discussed in 3.1.11, each routing table has 33 zones (one for each bit in the network mask). The search starts with the zone pointed to by *fn_zone_list* which is the first non-empty zone in the table. The zones are linked together with the most restrictive zone (32 bit netmask) being at the head and relaxing the restriction with each zone. In each zone, *fz_key()*⁵⁵ is called to generate the search key by ANDing the destination address and the zone netmask. Then *fz_chain()*⁵⁶ is called which invokes *fn_hash()*⁵⁷ to generate a hash to quickly locate the potential *fib_node* in the hash table of *fib_node* for the zone, which might

⁵²include/net/ip_fib.h

⁵³net/ipv4/fib_rules.c

⁵⁴net/ipv4/fib_hash.c

⁵⁵net/ipv4/fib_hash.c

⁵⁶net/ipv4/fib_hash.c

⁵⁷net/ipv4/fib_hash.c

have an entry for the destination. The search key is compared with the *fib_node* key. If the search key is less than that of the node key, then the search moves to the next zone. If the search key is greater than the node key, the search continues with the next *fib_node* entry. If there is a perfect match, then we have a route which is the longest match for the destination. Now, the *fib_node* entry is checked for its scope and state. If the node is in a zombie state or if the scope of the *fib_node* is less than the scope of the search key, then the search continues with the next *fib_node*. Else *fib_semantic_match()*⁵⁸ is invoked to setup the *fib_result* structure with the results of the lookup. If *fib_semantic_match()* returns success, then the *type*, *scope* and *prefixlen* of the *fib_result* is set to the values in the *fib_node*.

In *fib_semantic_match()*, one of the parameters passed is the *fib_info* of the *fib_node* which was selected during the search earlier. As discussed in 3.1.13, *fib_info* contains a list of nexthops available for this route. If the multipath routing is not configured in the kernel (*CONFIG_IP_ROUTE_MULTIPATH* = no), then the number of nexthops is at the most 1, which is selected if it is not dead. Otherwise, the first nexthop which is not dead is selected to be the nexthop for the packet. This is indicated by setting *res->nh_sel* = *nh_sel* where *res* is a *fib_result* structure and *nh_sel* and *nhsel* are indices into the array of nexthops.

In *fib_select_multipath()* which is invoked if multipath is enabled in the kernel and if the number of nexthops for the selected route is greater than 1, a weighted-round-robin algorithm is used to select the nexthop for the packet. Each nexthop specified in the route is given a weight. The algorithm selects the nexthop according to the weights.

⁵⁸net/ipv4/fib_semantics.c

ip_local_deliver() is the function which finally delivers the packet destined for the local host up to the higher layers. If the packet is fragment, *ip_defrag()*⁵⁹ is invoked which reassembles the packet. Then the *NF_IP_LOCAL_IN* netfilter hook is invoked which if successful calls *ip_local_deliver_finish()*⁶⁰.

ip_local_deliver_finish() gets the higher layer protocol information from the IP header of the packet. If the protocol is *IPPROTO_RAW* indicating that the packet is destined for a raw socket, *raw_v4_input()*⁶¹ is invoked for further processing. If the protocol is not *IPPROTO_RAW*, then corresponding protocol handler is invoked as *ipproto->handler()*. For TCP, this resolves to *tcp_v4_rcv()*⁶², for UDP, this resolves to *udp_rcv()*⁶³ and for ICMP, it resolves to *icmp_rcv()*⁶⁴

3.3 Packet Transmission

A packet transmission starts with a call to the *sock_write()* which first gets the *socket* associated the file descriptor being written to and then sets up the message header and calls *sock_sendmsg()*⁶⁵. The *socket* is discussed in 3.1.2.

sock_sendmsg() calls *sock->ops->sendmsg()* of the *socket* which maps to different functions for different types of sockets. For INET sockets, it maps to *inet_sendmsg()*⁶⁶.

⁵⁹*net/ipv4/ip_fragment.c*

⁶⁰*net/ipv4/ip_input.c*

⁶¹*net/ipv4/raw.c*

⁶²*net/ipv4/tcp_ipv4.c*

⁶³*net/ipv4/udp.c*

⁶⁴*net/ipv4/icmp.c*

⁶⁵*net/socket.c*

⁶⁶*net/ipv4/af_inet.c*

inet_sendmsg() first binds an unbound socket and then invokes *sk->prot->sendmsg()* where *sk* is the INET socket associated with the BSD socket. INET sockets are discussed in 3.1.3.

sk->prot->sendmsg() maps to *tcp_sendmsg()*⁶⁷ for TCP sockets and *udp_sendmsg()*⁶⁸ for UDP sockets. We take up *tcp_sendmsg()* and trace the packet as it travels down the network stack.

tcp_sendmsg() first waits for the connection setup to complete. Then it forms packets from the message. If there is space in the last awaiting packet (content length < MSS⁶⁹), the data is added to it. Finally *tcp_send_skb()*⁷⁰ is invoked.

tcp_send_skb() first enqueues the packet to the TCP write queue which holds the packets to be transmitted. Then *tcp_transmit_skb()*⁷¹ is invoked to send packets immediately if required, for example for packets which have the FIN bit set. Otherwise, the function is invoked later by other functions to transmit the packets.

When a packet needs to be transmitted *tcp_transmit_skb()* is invoked. *tcp_transmit_skb()* first builds the TCP header for the packet. Then it invokes *tp->af_specific->send_check()* where *tp* is a *tcp_opt* structure and *af_specific* is the pointer to the INET family specific functions. *af_specific* is set to *ipv4_specific*⁷². *send_check()* maps to *tcp_v4_send_check()*⁷³ which computes the checksum if required. Finally *tp->af_specific->queue_xmit()* is invoked which maps to *ip_queue_xmit()*⁷⁴.

⁶⁷net/ipv4/tcp.c

⁶⁸net/ipv4/udp.c

⁶⁹Maximum Segment Size

⁷⁰net/ipv4/tcp_output.c

⁷¹net/ipv4/tcp_output.c

⁷²net/ipv4/tcp_ipv4.c

⁷³net/ipv4/tcp_ipv4.c

⁷⁴net/ipv4/ip_output.c

In *ip_queue_xmit()*, route lookup is performed if the packet is not already routed, by calling *ip_route_output()*⁷⁵. If the route lookup fails, the packet is freed and an error is returned. The transport layer mechanism will take care of the retransmission in this case. If a route is found, then a pointer to the *dst* structure in the route is set up in the packet. Now the IP header is built and added to the packet. The *NF_IP_LOCAL_OUT* netfilter hook is called which if successful invokes *ip_queue_xmit2()*⁷⁶.

In *ip_route_output()* a key is formed from the destination address, source address, outgoing interface index and the TOS. The outgoing interface index is set if the socket is bound to a particular interface. Then *ip_route_output_key()*⁷⁷ is invoked with the key.

In *ip_route_output_key()* the route cache hash table is searched for a route. First a hash is generated from the source address, destination address, outgoing interface index and TOS of the packet. This is used for lookup in the route cache hash table. The corresponding bucket in the route cache hash table is traversed looking for a route that matches the destination address, source address, outgoing interface, incoming interface index being 0 and type of service flag. If firewall marking is enabled in the kernel (*CONFIG_IP_ROUTE_FWMARK* = yes), then firewall mark is also matched with the route cache entry. If an entry is found, then the entry is updated with the last used time, number of times used and a pointer to the entry is made available in the *struct rtable **rp* passed as argument to the function and the function returns. If a route cache entry is not found, then *ip_route_output_slow()*⁷⁸ is invoked.

⁷⁵include/net/route.h

⁷⁶net/ipv4/ip_output.c

⁷⁷net/ipv4/route.c

⁷⁸net/ipv4/route.c

ip_route_output_slow() first checks if the packet has a source address bound to it, in which case it finds the device having that source address. Else it checks if the outgoing interface is specified for the packet, in this case, it selects an address bound to the interface. Then a check is made to see if the packet has a destination address. If it does not have a destination address, the source and destination addresses are set to the loopback address and the loopback device is chosen as the device to send out the packet on. If the previous tests fail, then *fib_lookup()* is invoked to find the route for the destination. This function has been discussed earlier in 3.2. If *fib_lookup()* returns an error but the outgoing interface is specified for the packet, the destination is assumed to be on the link and a link-scope source address is selected. If the route lookup fails and the outgoing interface is not specified for the packet, then the destination is unreachable and an error is returned. If the route lookup yields a local route, then the loopback device is selected for the outgoing interface. Else, if multipath is configured in the kernel and there are multiple nexthops available for the route, *fib_select_multipath()* is called to select a nexthop from the multiple nexthops available. If multipath is not enabled or if there is only one nexthop available for the route, the default nexthop is selected. At this point, the outgoing interface and the nexthop have been selected for the packet. A route cache entry is created and filled in with the destination address, source address, TOS, outgoing interface index, etc. The *rth->u.dst.output* which specifies the function used for outgoing packets on this route is set to *ip_output()*. If the route is a local route, then *rth->u.dst.input*, which specifies the function used for incoming packets on this route, is set to *ip_local_deliver()*. *ip_output()* and *ip_local_deliver()* have been discussed in 3.2. Multicast packets and multicast routing are handled here which might change

the input and output functions setup above. Finally the route cache entry is inserted into the route cache with a call to *rt_intern_hash()*, which has been discussed in 3.2.

In *ip_queue_xmit2()*, if the packet is larger than the PMTU⁷⁹ of the route, then the *ip_fragment()*⁸⁰ is called. Else, the IP checksum is computed with a call to *ip_send_check()*⁸¹ and then *skb->dst->output(skb)* is called which maps to *ip_output()*.

In *ip_output()*, *ip_do_nat()*⁸² is invoked if NAT is configured in the kernel, else *ip_finish_output()*⁸³ is invoked.

ip_finish_output() invokes the *NF_IP_POST_ROUTING* netfilter hook which if successful calls *ip_finish_output2()*⁸⁴.

In *ip_finish_output2()*, if a hardware header cache is available for the packet's outgoing interface, the hardware header is added to the packet and then *hh_output()* function of the hardware header cache is invoked. If no hardware header cache is available for the packet and if the *neighbour* data is available for the route, then the *output()* of the *neighbour* data is invoked. If neither is available, the packet is dropped and an error is returned.

In general, *hh_output()* function maps to *dev_queue_xmit()*⁸⁵. When a neighbour is deleted, *hh_output()* maps to *neigh_blackhole()*⁸⁶ which just frees the packet buffer and returns an error.

⁷⁹Path Maximum Transmission Unit

⁸⁰*net/ipv4/ip_output.c*

⁸¹*net/ipv4/ip_output.c*

⁸²*net/ipv4/ip_nat_dumb.c*

⁸³*net/ipv4/ip_output.c*

⁸⁴*net/ipv4/ip_output.c*

⁸⁵*net/core/dev.c*

⁸⁶*net/core/dev.c*

output() of the *neighbour* maps to different functions depending on the state of the neighbour. These functions provide address resolution functionality. If the neighbour is dead, it maps to *neigh_blackhole()*. If the neighbour is in a good, then it maps to *neigh->ops->connected_output*. For Ethernet devices, the *connected_output()* maps to *neigh_resolve_output()*⁸⁷.

dev_queue_xmit() checks for the device queue and if the device has a queue, enqueues the packet for transmission by the device. It then runs the qdisc (*qdisc_run()*)⁸⁸ attached to the device and returns either success or failure depending on whether the queuing was successful or not. Some types of devices, for example loopback, tunnels etc, do not have an associated queue. In such cases, the *hard_start_xmit()* associated with the device is invoked. This function maps to the routine which takes care of the actual transmission of the packet from the interface and hence depends on the device driver. It is interesting to note that in the case of a loopback interface, the *hard_start_xmit()* invokes the *netif_rx()* simulating a packet reception by the loopback interface.

It is in *qdisc_run()* that the queuing disciplines come into picture. This calls the *qdisc_restart()*⁸⁹ which de-queues a packet from the device's queue and calls the *hard_start_xmit()* to transmit the de-queued packet. The packet to be de-queued is decided by the qdisc installed on the interface and any traffic shaping setup on the interface.

⁸⁷`net/core/neighbour.c`

⁸⁸`include/net/pkt_sched.h`

⁸⁹`net/sched/sch_generic.c`

3.4 Packet Forwarding

All packets which are being forwarded by the system follow the path taken by the incoming packets till they are passed to *ip_forward()*. This is the point where the incoming packets destined for the localhost and those being forwarded go their separate paths.

At this point, the outgoing interface for the packet has been selected. The routing table entry for the route chosen is available in *(struct rtable*)skb->dst* and the corresponding interface is available in *rt->u.dst.dev*.

In *ip_forward()*, the packet is checked to make sure it is destined for a host and not a broadcast or other type of packet which should not be forwarded. The packet is dropped if it is not destined for a host. The IP Time-To-Live (TTL) is checked and if it is less than or equal to 1, the packet is dropped and an ICMP time exceeded error message is sent to the origin of the packet. Other options like strict source routing (SR), don't fragment (DF) etc are also checked. Appropriate ICMP error messages are sent back and the packet is dropped if the packet fails any of the checks.

If network address translation (NAT) is enabled on the outgoing interface, *ip_do_nat()*⁹⁰ is invoked on the packet. If *ip_do_nat()* returns an error, the packet is dropped.

Finally the *NF_IP_FORWARD* netfilter hook is invoked on the packet. On success, this hook invokes *ip_forward_finish()*⁹¹ on the packet.

⁹⁰net/ipv4/ip_nat_dumb.c

⁹¹net/ipv4/ip_forward.c

If there are no IP options, *ip_forward_finish()* invokes *ip_send()*⁹² on the packet. If there are options, then *ip_forward_options()*⁹³ is invoked to process the options and then *ip_send()* is invoked to send off the packet.

ip_send() calls *ip_fragment()*⁹⁴ if the packet is larger than the Path Maximum Transmission Unit (PMTU) of the outgoing route which in turn fragments the packet and then calls *ip_finish_output()*⁹⁵ else calls *ip_finish_output()*.

ip_finish_output() calls the *NF_IP_POST_ROUTING* netfilter hook and if the hook is successful it calls *ip_finish_output2()*⁹⁶.

From here, the packet follows the same path as taken by a packet being transmitted from the system, which has been discussed in 3.3.

3.5 Netfilter Framework

From the Linux netfilter Hacking HOWTO⁹⁷:

Netfilter is merely a series of hooks in various points in a protocol stack...

The framework provides a number of hooks which are called at different times during a packet's journey through the protocol stack on the system as depicted in Figure 3.4⁹⁸.

After checking the incoming packets for IP version, checksum, etc, *ip_rcv()* passes them through the *NF_IP_PRE_ROUTING* hook ([1] in Figure 3.4). After success-

⁹²include/net/ip.h

⁹³net/ipv4/ip_options.c

⁹⁴net/ipv4/ip_output.c

⁹⁵net/ipv4/ip_output.c

⁹⁶net/ipv4/ip_output.c

⁹⁷<http://netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html>

⁹⁸Source: Linux netfilter Hacking HOWTO, <http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html>

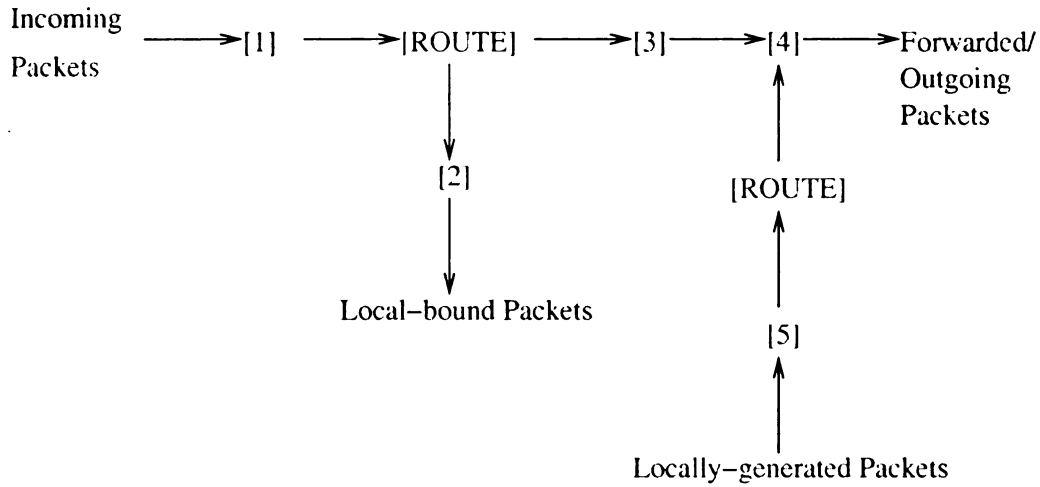


Figure 3.4: Netfilter Hooks

fully passing through the hook, the routing code decides whether the packet is destined for the local system or it has to be forwarded to another host. If the packet is for the local system, the *NF_IP_LOCAL_IN* ([2] in the above Figure 3.4) is invoked from *ip_local_deliver()*. If on the other hand the packet is for a different system, *ip_forward()* calls the *NF_IP_FORWARD* ([3] in Figure 3.4). Packets which are generated by the local system first pass through *NF_IP_LOCAL_OUT* ([5] in Figure 3.4) netfilter hook called from *ip_queue_xmit()*. Locally generated packets destined for other system as well as forwarded packets finally pass through the *NF_IP_POST_ROUTING* hook ([4] in Figure 3.4).

The hooks can return one of the following results: accept, drop, stolen, queue or repeat. If accepted, the packet continues its journey through the protocol stack. If the verdict returned is drop, then the packet is dropped and the traversal is aborted. Stolen indicates that the hook has taken control over the packet and that the protocol stack should not continue processing the packet. Queue provides a mechanism to

queue the packet for user-level handling of the packet. Repeat causes the hook to be called again on the packet.

Any module can register itself at any of the hooks. During the registration, a function is specified which will be invoked when the hook is called. This function will be given a pointer to the packet and should return one of accept, drop or queue verdicts. During the registration, the priority of the function for that hook is specified. When the hook is invoked, the registered functions are called in the order of their priority. For example, the following modules register their functions with the *NF_IP_LOCAL_OUT* (in the order of priority/call): Conntrack, Mangle, NAT (destination NAT), Filter⁹⁹. Conntrack module handles the connection tracking mechanism in Linux while the Filter module is responsible for filtering packets based on different criteria like source and destination address, source and destination port number, ICMP type, etc. The filter module is used to setup rules to configure firewall on the system.

Now we discuss the connection tracking mechanism in the netfilter framework. This mechanism is later used in our work to identify the connection to which a packet might belong.

*ip_conntrack_in()*¹⁰⁰ is registered at *NF_IP_PRE_ROUTING* hook. A packet after passing the basic IP checks comes here where it is checked if a connection is already associated with this packet (probably coming on a loopback device), in which case *NF_ACCEPT* is returned. If the packet is a fragment, then *ip_ct_gather_frags()*¹⁰¹ is invoked to gather the fragments. Then the protocol of the transport layer is identified from the packet. If the packet is an ICMP error message, it is dealt with

⁹⁹include/linux/netfilter_ipv4.h

¹⁰⁰net/ipv4/netfilter/ip_conntrack_core.c

¹⁰¹net/ipv4/netfilter/ip_conntrack_core.c

here and *NF_ACCEPT* is returned. Then *resolve_normal_ct()*¹⁰² is invoked to get a connection associated with the packet. If *resolve_normal_ct()* fails, *NF_ACCEPT* is returned. The *proto->packet()*, where *proto* is a *ip_conntrack_protocol* (discussed in 3.1.23) structure, is invoked. This function maps to different functions for different protocols, for example, for TCP it maps to *tcp_packet()*¹⁰³ and for UDP it maps to *udp_packet()*¹⁰⁴. After some more checks, if there is a helper function registered for the packet's protocol, it is invoked. Finally, if *resolve_normal_ct()* had indicated that the packets have been seen in both the directions for the connection, the *IPS_SEEN_REPLY_BIT*¹⁰⁵ bit is set in the connection status and the function returns the result of the call to *proto->packet()*.

resolve_normal_ct() first forms the IP connection tracking tuple (*ip_conntrack_tuple* as discussed in 3.1.22) for the packet. If the tuple cannot be formed an error in the form of NULL is returned. Else *ip_conntrack_find_get()*¹⁰⁶ is invoked to find any existing connection corresponding to the tuple. If a connection is not found for the tuple, then *init_conntrack()*¹⁰⁷ is called to create a new connection for the tuple. If there is any problem creating a new connection, then an error is returned. Once we have a connection corresponding to the tuple (either a new connection or an existing connection), various checks are performed to set the status of the connection, for example, is this packet in the original direction or is it a reply

¹⁰²net/ipv4/netfilter/ip_conntrack_core.c

¹⁰³net/ipv4/netfilter/ip_conntrack_proto_tcp.c

¹⁰⁴net/ipv4/netfilter/ip_conntrack_proto_udp.c

¹⁰⁵include/linux/netfilter_ipv4/ip_conntrack.h

¹⁰⁶net/ipv4/netfilter/ip_conntrack_core.c

¹⁰⁷net/ipv4/netfilter/ip_conntrack_core.c

packet, if a reply has been seen then the connection is set to established status, etc.

The function then returns the connection associated with the tuple.

init_conntrack() is the function where a new connection is created for a packet. The tuple (*ip_conntrack_tuple* discussed in 3.1.22) is passed to this function which is used to generate a hash for placing it in the connection hash table. If the number of connections is more than the maximum number of the connections possible, a connection is dropped from table randomly or from the chain into which the new connection will be put into. Next the tuple for the reverse direction of the packet is generated by calling *invert_tuple()*¹⁰⁸. If the inverse tuple generation fails, then an error is returned without putting the new connection into the connection table. A new connection is allocated and its members filled in. The tuple and the connection information is filled in for both the original as well as the reverse direction packets. The packet's transport layer protocol's *new()* function is invoked to setup information specific to the protocol in the connection structure. After filling in other members of the connection like timeout information, the connection's *tuplehash* (discussed in 3.1.19) is returned.

As remarked earlier, the above discussion might be overwhelming to some readers. It is expected to be :). However, section 4.1 discusses the strategies at a higher level and should be more easy to follow than the above discussion.

¹⁰⁸`net/ipv4/netfilter/ip_conntrack_core.c`

Chapter 4

Channel Balancing Algorithms

In this chapter, we look at the four different strategies to balance the outgoing traffic on an equal cost multipath route. First we discuss the strategies and then their implementation details. All the strategies require the kernel to have mulitpath feature (*CONFIG_IP_ROUTE_MULTIPATH* = yes) enabled.

4.1 Strategies

For the discussion of the strategies, we assume that some source hosts (*srcA*, *srcB*, *srcC*, etc) have more than one route to some destination hosts (*dst1*, *dst2*, *dst3*, etc) using multiple interfaces (*eth0*, *eth1*, *eth2*, etc) on the gateway/router. We assume that these routes have equal metrics, i.e. equal cost. If the cost of the routes were unequal, then the best route would be used and the others discarded¹. Therefore, for following discussions, we assume more than one equal cost routes to the destination hosts (see Figure 4.1).

4.1.1 Route Based Channel Balancing

In this strategy, load balancing is considered when the route for the destination is looked up. For example, when *srcA* causes a route lookup on the router/load balancer for *dst1*, the first route, via *eth0* might be used. As long as the route cache entry for

¹Some router vendors provide feature to balance traffic across unequal routes also.

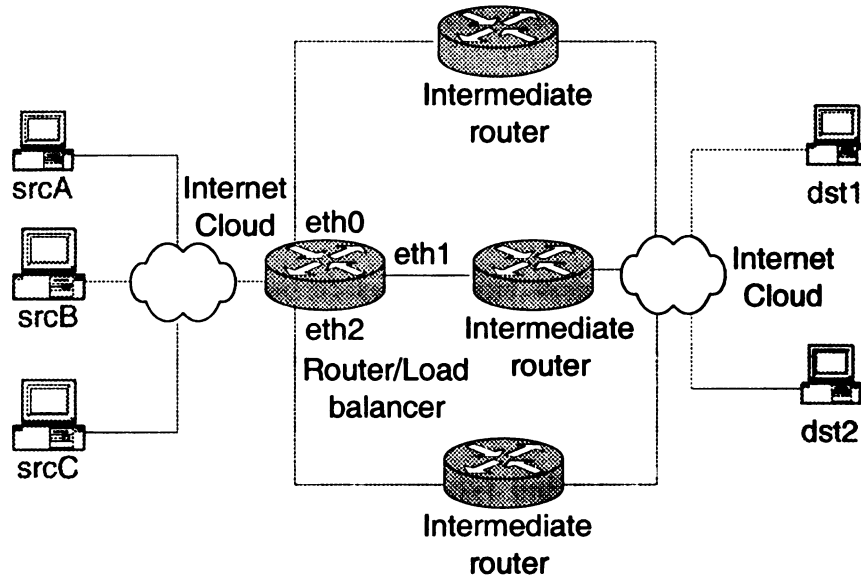


Figure 4.1: Multiple Routes between hosts

dst1 is valid, all data for *dst1* will be routed over the same path. When a new route is looked up, say for *dst2*, the load balancing algorithm will select the next equal cost path, via *eth1*. Now all the data for *dst2* will be routed along *eth1* as long as its route cache entry is valid. The multipath route can be configured to have different weights for the different equal cost routes. For example, the route via *eth0* might be given a weight of 2, while the rest are given a weight of 1. This will cause *eth0* to be used twice as many times as the others during the route lookup. Further, if there are connections to *dst1*, *dst2*, *dst3* and *dst4*, the connections to *dst1* and *dst4* will use *eth0* while *dst2* and *dst3* will use *eth1* and *eth2* respectively.

4.1.2 Per-packet Based Channel Balancing

In this strategy, load balancing is applied for every packet. When a packet needs to be transmitted or forwarded to a destination for which there are multiple equal cost routes available, the load balancer will select the appropriate route. For example

if we have 3 packets to be forwarded to *dst1* and we have 3 equal cost routes to *dst1*, then one packet will be sent over each route, thus balancing the traffic. The balancing is not restricted to a specific destination. If there were 3 packets, one packet destined for *dst1* and two packets destined for *dst2*, then the first packet will be sent over first route, second packet over the second route, third packet over the third route. Thus one packet will be sent over each route. Thus, per-packet based channel balancing does not consider the destination or the source information when balancing the traffic. It considers each packet individually and picks a route for it. This may cause packets belonging to the same connection (say a web upload session) to be sent out over different routes.

In this strategy also the administrator has the choice to assign different weights to each participating route and control how much traffic is sent over each route.

4.1.3 Session Based Channel Balancing

We define a session as collection of related packets transferred between two hosts, the *source* initiating the connection to the *destination*. In other words, a TCP connection and a UDP connection are examples of a session. In IPv4 family, the source and destination IP addresses, the source and destination ports for TCP and UDP or the type and code for ICMP, define a connection.

In this strategy, the load balancing is done based on sessions. When a session is initiated (either on the load balancer or on a host which sends data to the load balancer for forwarding) and comes in for routing, the load balancer selects a route from the multiple equal cost routes. In the future, all arriving packets which belong to that session are routed along the same route. When another session comes in for routing,

the load balancer selects another route. We use the *netfilter*² framework's connection tracking mechanism to determine if a packet belongs to an existing connection or is a new connection.

For example, consider a TCP session initiated from *srcA* to *dst1*. The port numbers involved are 8011 (source port) and 22 (destination port). The connection can be specified as *<srcA:8011, dst1:22>*. When this session is initiated, the load balancer will select a route, say via *eth0*. All packets which belong to this session will then be routed along *eth0*. If another session defined by *<srcB:7291, dst2:80>* comes in for routing, the load balancer will now select another route, say via *eth1* and all packets of this session will then be routed along *eth1*.

4.1.4 Bandwidth Usage Based Channel Balancing

In this strategy, the current bandwidth usage on the routes is used to decide the route for the current packet. The bandwidth usage on each route (interface) is measured periodically and weight-averaged over a period of time. When a packet comes in for routing, the route (interface) that has the maximum unused bandwidth available is used. Similar to the strategy in section 4.1.2, this strategy also considers each packet individually. This may cause the packets belonging to the same connection to be routed over different routes which may result in out-of-order delivery at the destination.

For example, if there are two 10Mbps routes to a destination say through *eth0* and *eth1* and their bandwidth usage is 3Mbps and 4Mbps respectively, then the next incoming packet will be sent out on *eth0* because it has more of unused bandwidth, 7Mbps.

²<http://www.netfilter.org/>

```

ip_route_input_slow() {
    search FIB tables
    if(multipath route found) {
        /* Now we select one of the multipath
        routes according to the strategy being used */
        if(session_debug == true) { /* Using session based strategy */
            if (connection associated with packet is found) {
                if(outgoing interface is set) {
                    if(fib_select_multipath_previous() == false) {
                        /* Dead interface ? */
                        fib_select_multipath_session()
                    }
                } else { /* New connection */
                    fib_select_multipath_session()
                }
            } else { /* Setup the session */
                fib_select_multipath_session()
            }
        } else if (bandwidth_debug == true) { /* Using bandwidth based strategy */
            fib_select_multipath_bandwidth()
        } else {
            /* This will be route based or per-packet depending on the */
            /* EQUALIZE flag in the multipath route */
            fib_select_multipath()
        }
    }
}

```

Figure 4.2: Nexthop Selection Logic in Multipath Route

4.2 Implementation

In this section we discuss the implementation details of the different strategies discussed above. We discuss the changes made to the stock³ Linux kernel, any configuration requirements for the kernel build, the method to enable a particular strategy and some sample commands to illustrate the setup process. Figure 4.2 illustrates the control flow for selecting the nexthop for various strategies.

³Any default installation of Linux operating system distributed by vendors.

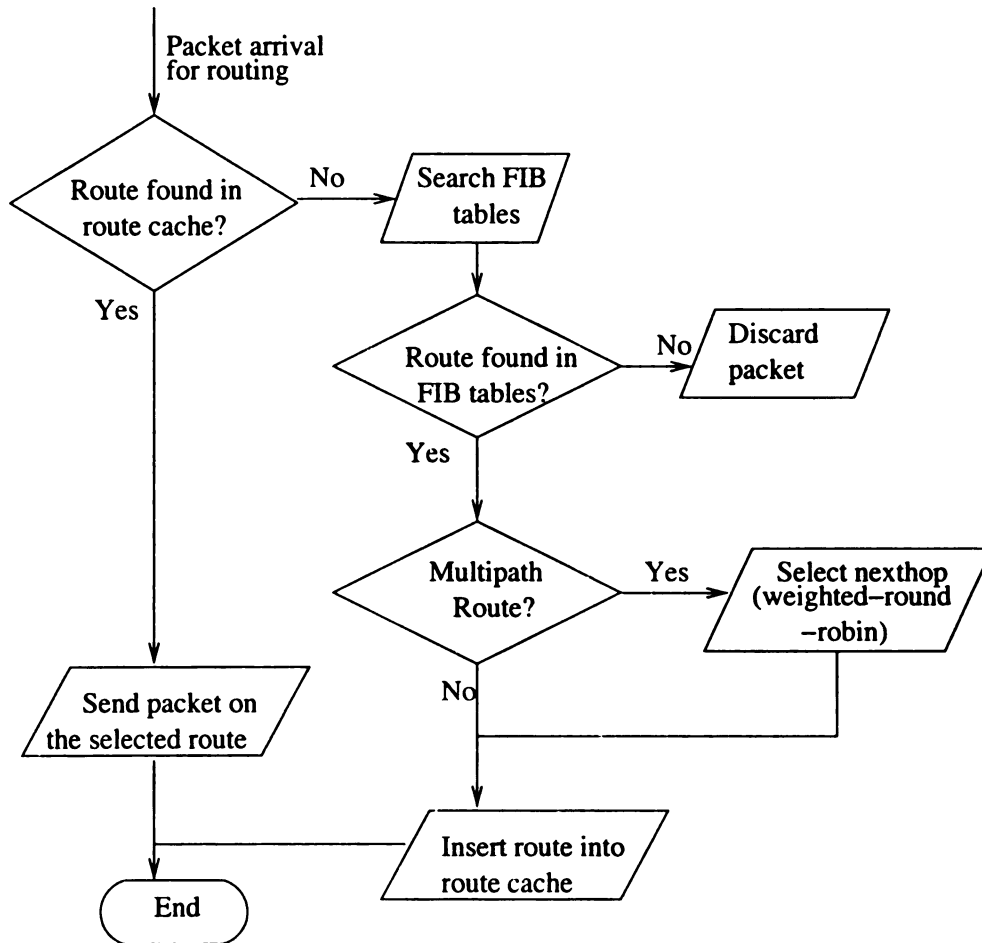


Figure 4.3: Route Based Channel Balancing

4.2.1 Route Based Channel Balancing

This load balancing technique is available in the stock Linux kernel. Before the load balancing feature can be used, the administrator needs to configure the multipath route.

The following command will setup the equal cost multipath with equal weights to each participating route⁴:

```
# ip route add 20.0.0.0/24 \
```

⁴The command has been split into multiple lines for readability.

```
nexthop via 10.0.1.2 dev eth0 weight 1 \  
nexthop via 10.0.2.2 dev eth1 weight 1 \  
nexthop via 10.0.3.2 dev eth2 weight 1 \
```

The route lookup starts with call to *ip_route_input()* which searches the route cache for a matching route. If it fails, *ip_route_input_slow()* is invoked to search the FIB tables for a suitable route. If the result of the search is a multipath route, then the *fib_select_multipath()* is invoked which selects the nexthop based on the weights assigned to the nexthops. The chosen nexthop is added to the route cache so that packets destined to the same destination can now be routed along the chosen route without causing a route lookup.

4.2.2 Per-packet Based Channel Balancing

This strategy is not available in the stock Linux kernel and the *nano*⁵ patch should be applied to the kernel and the kernel rebuilt. When rebuilding the kernel, the multipath feature (*CONFIG_IP_ROUTE_MULTIPATH* = yes) should be enabled. The following command will setup per-packet based load balancing with equal weights to each participating route (notice the use of the **equalize** keyword in the command):

```
# ip route add equalize 20.0.0.0/24 \  
    nexthop via 10.0.1.2 dev eth0 weight 1 \  
    nexthop via 10.0.2.2 dev eth1 weight 1 \  
    nexthop via 10.0.3.2 dev eth2 weight 1
```

The *nano* patch changes the route lookup code. When a packet arrives destined for a host for which there is no route in the route cache, the FIB tables are searched as in section 4.2.2. If a multipath route is found, then before adding the route entry into the

⁵<http://trash.net/~kaber/equalize/equalize.2.4.18.patch>

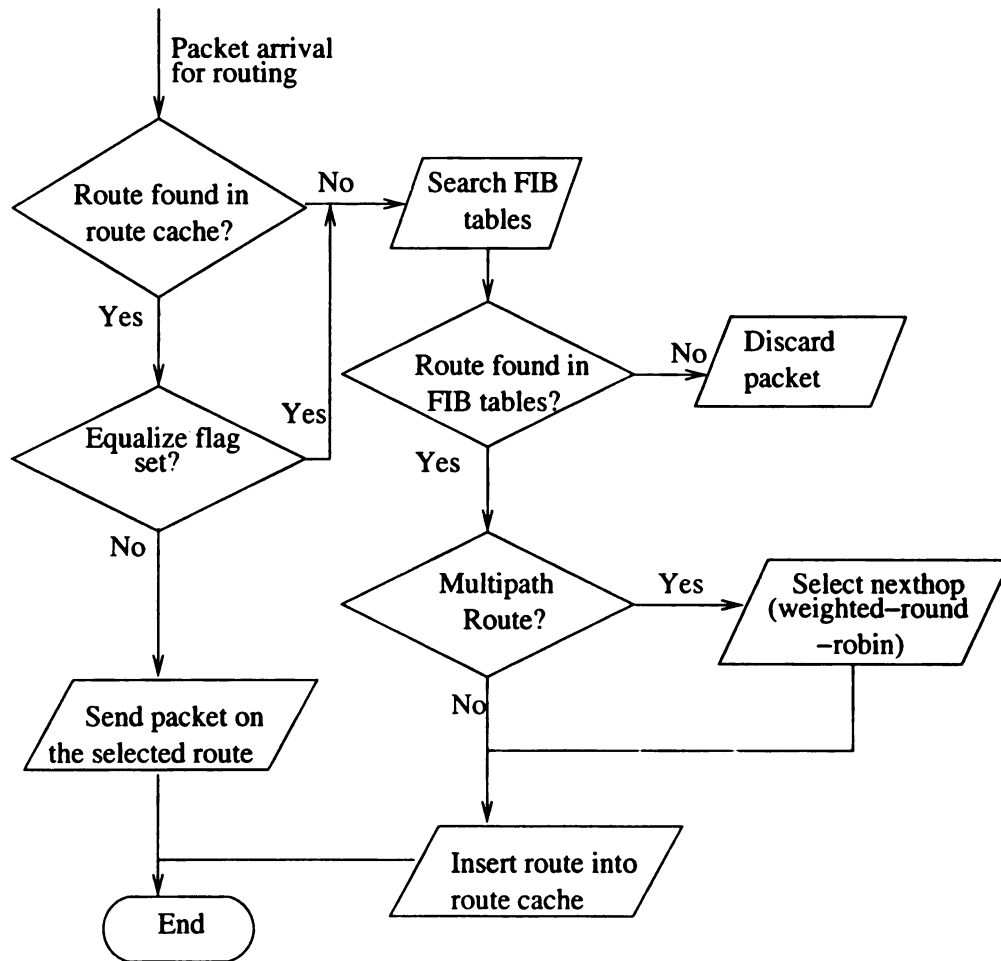


Figure 4.4: Per-Packet Based Channel Balancing

route cache, it is marked with *RTCF_EQUALIZE* flag. Later when this route cache entry is selected to route another packet, the route cache entry is deleted and the FIB tables are searched again. This makes sure that all packets using the multipath route cause a search of the FIB tables and the packets are distributed over the multiple nexthops according to the weights. The route cache lookup routine for forwarded packets *ip_route_input()*, *ip_route_input_slow()* which sets up the route cache entry for forwarded packets, *ip_route_output_slow()* which sets up the route cache entry for

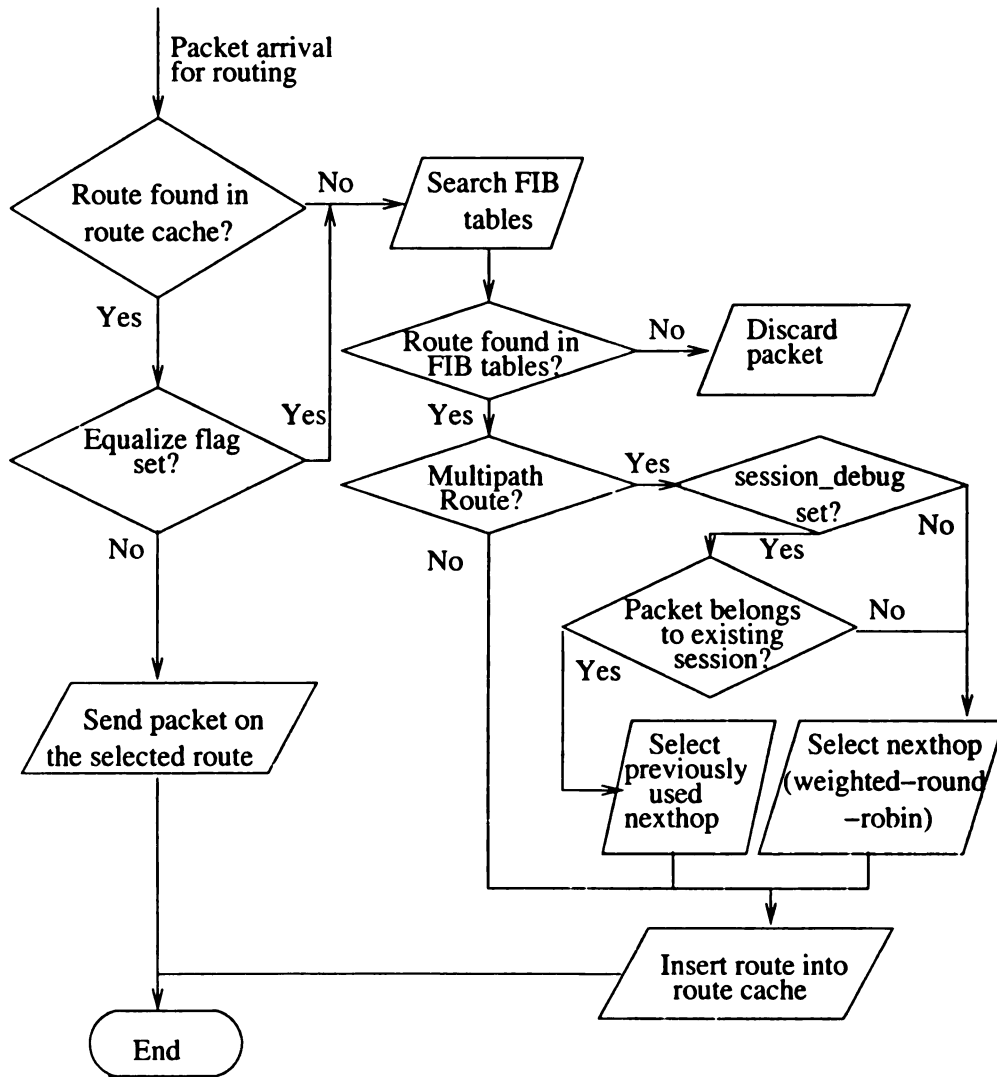


Figure 4.5: Session Based Channel Balancing

outgoing packets, `ip_route_output_key()` which searches the route cache for outgoing packets are modified to implement this strategy.

4.2.3 Session Based Channel Balancing

This implementation requires the application of the *nano* patch to the stock Linux kernel. This strategy requires the connection tracking mechanism (CON-

FIG_NETFILTER, *CONFIG_IP_NF_CONNTRACK* and *CONFIG_IP_NF_FTP*) to be configured in the kernel and not as separate modules.

The *ip_conntrack_tuple* structure was modified to record the last chosen outgoing interface for the connection (*oif*). *oif* is initialized to -1 in *resolve_normal_ct()* before calling *init_conntrack()* for a new session. When the new connection is created in *init_conntrack()*, *oif* of the reply packet's tuple is initialized to -1.

A flag, *session_debug*, is added to the kernel to indicate that session based channel balancing is configured. This flag can be set by inserting the module *session_debug.o* which was written separately.

In *ip_route_input_slow()*, if the FIB table search for the packet's destination yields a multipath route, then we check if *session_debug* is set. If set, we check if the packet is associated with a connection. If a connection is found for the packet, we get the IP connection structure (*ip_conntrack*) of the packet. We now have information about the packet's direction as well as the outgoing interface used by the packet's connection. If the outgoing interface is not set (*oif* = -1), then this indicates that the packet is part of a new connection. Here we call *fib_select_multipath_session()*⁶. If the outgoing interface is set indicating that the packet is part of an existing connection, we call *fib_select_multipath_previous()*. If *fib_select_multipath_previous()* returns an error due to a dead interface or other reasons, we call *fib_select_multipath_session()* similar to a packet belonging to a new connection.

If the packet did not have a connection associated with it, we call *fib_select_multipath_session()* to select the nexthop. *fib_select_multipath_session()* invokes *fib_select_multipath()* to select the nexthop according to the weights of the nexthops.

⁶net/ipv4/fib_semantics.c

If the *session_debug* was not set, we call *fib_select_multipath()* to select the nexthop, similar to section 4.1.2 strategy.

We implement the new function *fib_select_multipath_previous*⁷ which searches the multipath route for the nexthop last used by the connection. It takes the outgoing interface index of the connection as one of its parameters. If the interface of the nexthop last used is still alive, then the *nh_sel* of the route cache entry is set to indicate the selection of the nexthop and a success is returned. If the nexthop corresponding to the last used interface is not found or if it is dead, then an error is returned.

Finally in *ip_route_input_slow()*, just before inserting the route cache entry into the route cache, the outgoing interface for the packet's connection is set to the interface on which the packet is being sent out.

To use this strategy, the administrator should insert the module *session_debug.o* to set the *session_debug* flag and then set up the multipath route:

```
# insmod session_debug.o
# ip route add equalize 20.0.0.0/24 \
    nexthop via 10.0.1.2 dev eth0 weight 1 \
    nexthop via 10.0.2.2 dev eth1 weight 1 \
    nexthop via 10.0.3.2 dev eth2 weight 1
```

4.2.4 Bandwidth Usage Based Channel Balancing

To implement this strategy, in addition to the kernel modifications, a kernel module was implemented to measure and estimate the current bandwidth usage on all the

⁷`net/ipv4/fib_semantics.c`

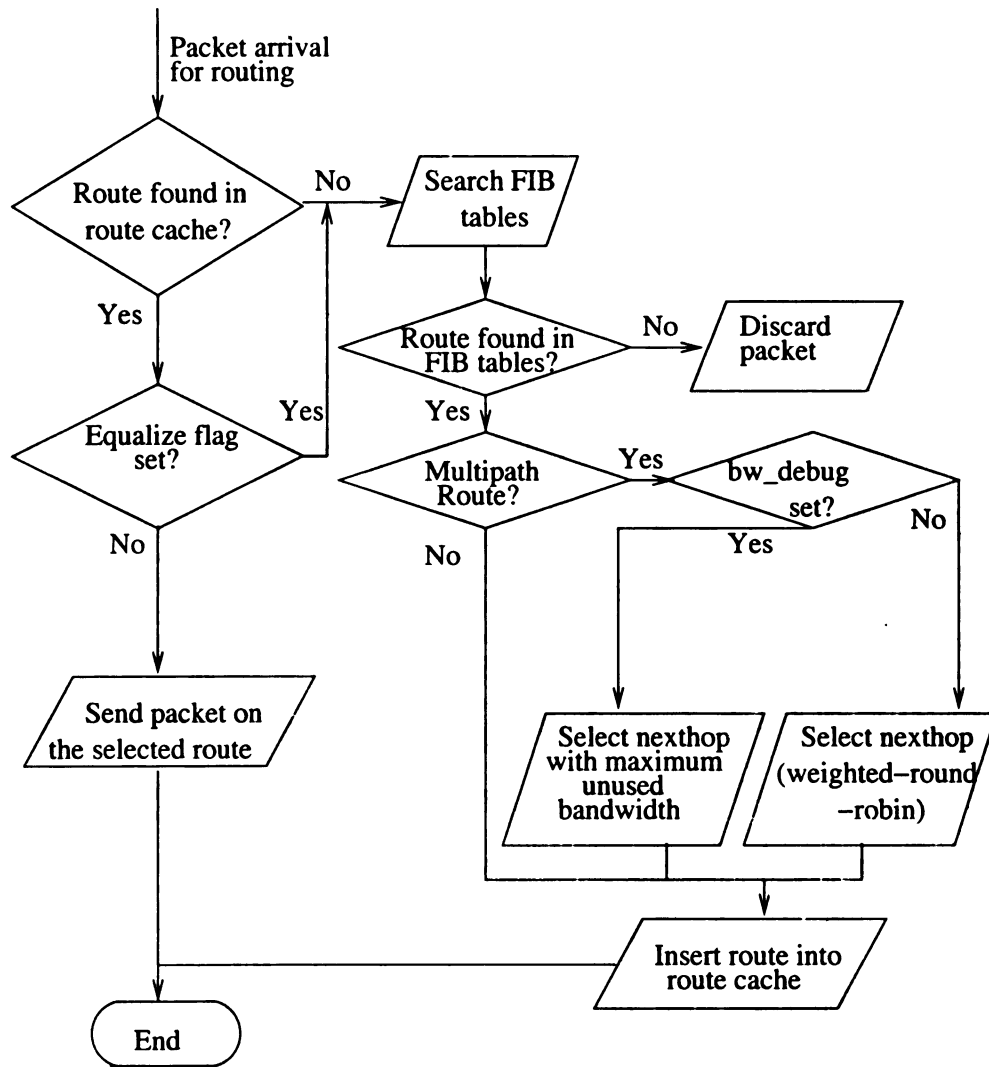


Figure 4.6: Bandwidth Usage Based Channel Balancing

interfaces. This strategy also requires the application of the *nano* patch to the stock Linux kernel.

The *net_device* structure was modified to implement the bandwidth estimator. A *bwusage*⁸ structure (*bwusage*) was added to *net_device*. *bwusage* holds various statistical information about the interface: its name (*name*), transmission rate in bits/sec and packets/sec (*tx_bps* and *tx_pps* respectively), average transmission rate

⁸`include/net/bwestimator.h`

in bits/sec and packets/sec (*tx_avbps* and *tx_avpps* respectively), total bytes and packets sent out (*tx_total_bytes* and *tx_total_packets* respectively) and bytes and packets sent out during the last measurement interval⁹ (*tx_bytes* and *tx_packets* respectively).

In *ip_route_input_slow()*, if the FIB table search for the packet's destination yields a multipath route, then we check if *bw_debug* is set. This flag indicates the selection of the bandwidth usage based channel balancing strategy and is set by inserting *bw_debug.o* module into the kernel. *bw_debug.o* is a kernel module written for this thesis. If set, we call *fib_select_multipath_bandwidth()*¹⁰ to select the nexthop for the packet. If *bw_debug* is not set, we call *fib_select_multipath()* to select the nexthop, similar to section 4.1.2 strategy.

In *fib_select_multipath_bandwidth()*, we iterate over all the available nexthops and identify the one with the maximum unused bandwidth. The bandwidth information is available from the *bwusage* structure of the interface associated with the nexthop. The nexthop with the maximum unused bandwidth is then selected to send the packet. *nh_sel* of the *fib_result* is set to indicate the selection.

To use this strategy, the administrator should insert the module *bw_debug.o* to set the *bw_debug* flag and then set up the multipath route:

```
# insmod bw_debug.o
# ip route add equalize 20.0.0.0/24 \
    nexthop via 10.0.1.2 dev eth0 weight 1 \
    nexthop via 10.0.2.2 dev eth1 weight 1 \
    nexthop via 10.0.3.2 dev eth2 weight 1
```

⁹This is set to 1 sec which can be configured in the bandwidth estimation module.

¹⁰`net/ipv4/fib_semantics.c`

4.2.5 Discussion of the strategies

Each of the strategies discussed above have certain points of interest. The route based strategy and session based strategy ensure that the packets of a connection passing through the route will be routed in-order while the per-packet based and bandwidth based strategies do not ensure it. The packets of a connection might be routed over different routes in these strategies. While this might seem to affect the TCP's timing mechanism which controls the TCP window size (according to the Sliding Window Protocol), it should be noted that the IP protocol in itself does not guarantee any in-order delivery of packets. Thus, even though the packets passing through the load balancer might be routed in-order in the route based and session based strategies, there is no guarantee that they will reach the destination host in-order.

Another issue to note is that except for the route based strategy, all the other strategies invoke a FIB table search for every packet. This might cause scalability issues when routing large amounts of data. But experimental results on a LAN show that these strategies can achieve the same routing rate as the route based strategy for small sized data transfers. We should note here that this implementation is not intended for large networks where the traffic is flowing at gigabit-per-second (Gbps) rates. In such situations, one must consider the affects of the FIB lookups for every packet in per packet based, session based and bandwidth based strategies.

Chapter 5

Experimentation Details

In this chapter we describe the testbed setup, the traffic pattern and type, and the statistics collected.

The testbed setup is shown in Figure 5.1. As shown in the figure, two servers (*srcA* and *srcB*) serve as the source of data and two hosts (*dst1* and *dst2*) serve as the sink. These hosts (*dst1* and *dst2*) have been configured with two additional loopback addresses (*lo:1* and *lo:2*)¹ each, to which the data is sent. This gives us 4 different destination addresses. The data is sent from *srcA* and *srcB* to *dst1:lo1*, *dst1:lo2*, *dst2:lo1* and *dst2:lo2*. The default route on *srcA* and *srcB* is configured via the channel balancer router (*balancer*). On *balancer*, the default route is a multipath route consisting of routes via the three intermediate routers (*testa*, *testb* and *testc*). Depending on the channel balancing strategy being tested, we either set or unset the *EQUALIZE* flag on the multipath route. On the intermediate routers (*testa*, *testb* and *testc*), routes are setup to route packets destined for the destination addresses *dst1:lo1* and *dst1:lo2* via *dst1:eth0* of *dst1*. Similarly, packets destined for *dst2:lo1* and *dst2:lo2* are routed on the intermediate routers via *dst2:eth0* of *dst2*. To be able to route any acknowledgment packets or other error packets, intermediate routers are configured to route data destined for *srcA* and *srcB* over the link connecting them to

¹We use the name of the interface configured with an IP address to denote the IP address for readability purposes.

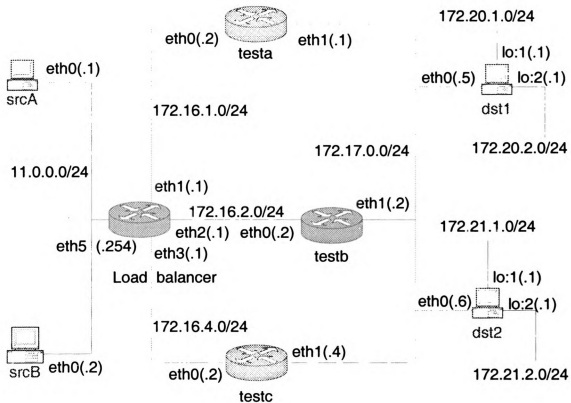


Figure 5.1: Testbed Setup

balancer. Finally, *dst1* is configured to send data destined for the source servers via *testa* and *dst2* is configured to send data destined for the source servers via *testb*. This setup enables the routing of data between the source and destination hosts.

We use the Secure Copy program (*scp*) to transfer the data. The transfer is initiated from the source servers. Session based strategy requires that the connections be outgoing on the multipath route. If the connection had been initiated from the destination hosts, then connection tracking mechanism would remember the incoming interface and mark that as the interface to use for packets going in the reverse directions. Hence to balance the sessions on the multipath route, we should have the connection initiated from *inside* the network and going *out* on the multipath

route. This is achieved by using the *scp* command on the source servers similar to the following:

```
ashoknn-src1:~>scp file1.dat nnashok@ashoknn-dst1:data/
```

The above command will copy the *file1.dat* from the *ashoknn-src1* machine to the *ashoknn-dst1* machine.

Files of different sizes: 8KB, 16KB, 64KB, 128KB, 512KB, 1MB, 2MB, 4MB, 8MB and 16MB, were transferred. The tests were conducted by sending 20 files each of the sizes 8KB, 16KB, 64KB, 128KB and 512KB. For each of sizes 1MB, 2MB, 4MB, 8M and 16MB, 5 files were transferred. Each source would transmit the files of a given size (say 8KB) to each of the destination hosts (*dst1:lo1*, *dst1:lo2*, *dst2:lo1* and *dst2:lo2*). Thus during the test for 8KB file size, *srcA* and *srcB* each would send 20 files of 8KB size to *dst1:lo1*, *dst1:lo2*, *dst2:lo1* and *dst2:lo2* each.

We used *iptables* rules to determine how much data was transferred on each interface for each source-destination combination. In addition, for the bandwidth usage based strategy, a bandwidth usage estimator module was inserted into the kernel which measured the bandwidth usage on each interface every second and averaged it over a period of 8 seconds.

Chapter 6

Results

Figures 6.1 through 6.11 show the bandwidth utilization on each interface during file transfer of different sizes under different strategies.

In route based channel balancing, we observe that two of the routes participating in the multipath route are used almost equally. However we see an anomaly during the transfer of files of 2MB size. This might be due to the timing of the arriving connections which caused the same route to be taken for more connections.

We observe that the bandwidth utilization on each interface is approximately the same when per-packet based channel balancing strategy is used. This verifies the design of the strategy. Although we expected performance degradation due to possible out-of-order delivery of packets at the destination, it is not reflected in the experimental results. It could be due to the small number of hops involved (3 hops) in reaching the destination. If there were more hops, we might have been able to notice the affects of the out-of-order delivery of packets.

Session based strategy also shows equal utilization of each route. But this would be because during each test, the same amount of data was transfered by each connection thus equalizing the bandwidth utilization on the routes. The difference in the bandwidth utilization of the routes is visible in Figure 6.11 where the connections transfered different amounts of data. In this case, the timing of the incoming connec-

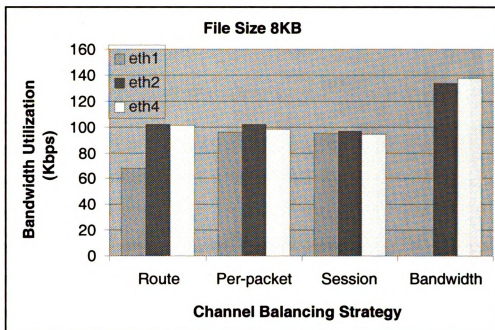


Figure 6.1: Bandwidth usage for File Size 8KB

tions would also matter in the utilization of the routes. For example, consider three transfers of 1MB, 1KB and 16KB respectively. Balancing these sessions, lets say that 1MB session is routed over *eth1*, 1KB session routed over *eth2* and 16KB session routed over *eth4*. Now if an 8MB session came in, it would be routed over the first route *eth1* thereby raising the bandwidth utilization of *eth1* drastically. Therefore, in this strategy, the bandwidth utilization of each session will have an impact on the bandwidth utilization of each route.

We notice that in all the transfers with bandwidth based strategy, *eth1* is not utilized. This is because the simulated bandwidth available for *eth1* was only 1Mbps whereas the other interfaces were configured to simulate 10Mbps. Thus the available bandwidth on *eth1* was lesser than the available bandwidth on the other interfaces.

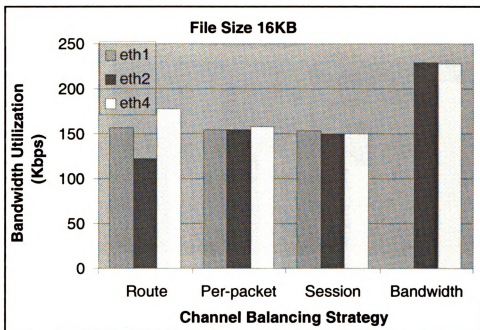


Figure 6.2: Bandwidth usage for File Size 16KB

If the bandwidth of the routes was comparable, then we can expect behavior similar to the per-packet based channel balancing strategy.

The transfer times involved in these tests (see Figure 6.12) indicate that the different strategies perform equally well when the size of the files transferred is small ($\leq 512\text{KB}$). As the size of the file increases, the per-packet based and session based strategies show better performance. We should remember that in all the case except in the mixed file size case, the file size in each connection was same. This would cause the data transferred by every connection to be the same and hence the bandwidth utilization of the routes in the session based tests would be approximately the same.

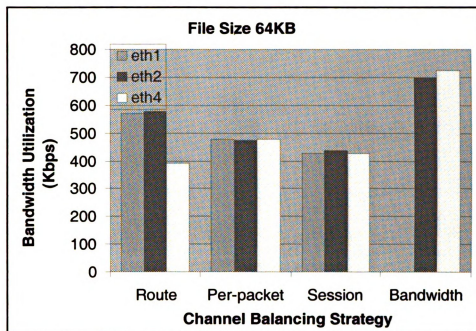


Figure 6.3: Bandwidth usage for File Size 64KB

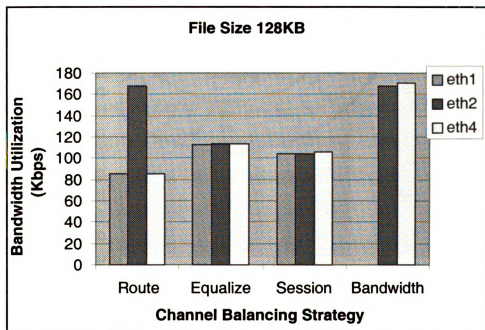


Figure 6.4: Bandwidth usage for File Size 128KB

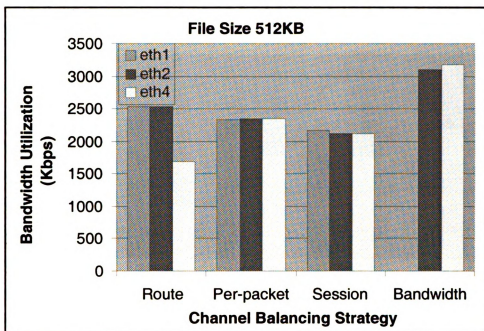


Figure 6.5: Bandwidth usage for File Size 512KB

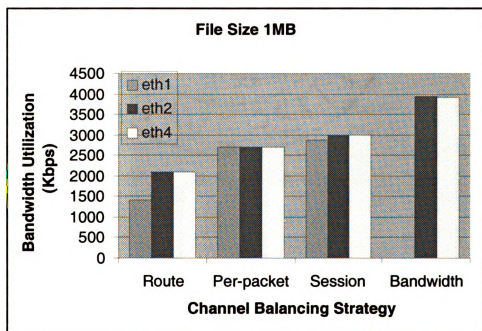


Figure 6.6: Bandwidth usage for File Size 1M

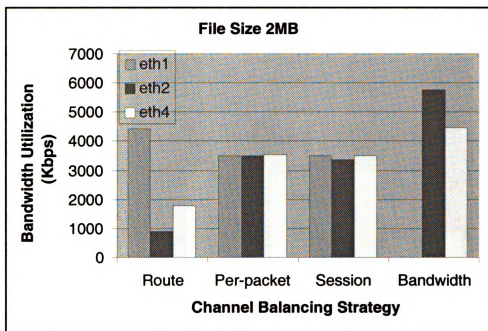


Figure 6.7: Bandwidth usage for File Size 2M

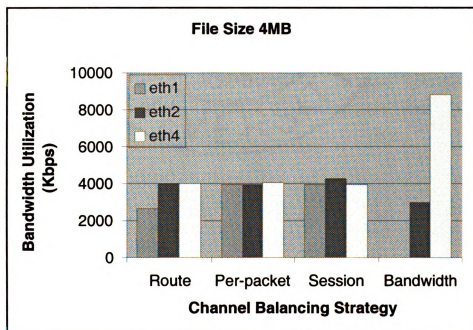


Figure 6.8: Bandwidth usage for File Size 4M

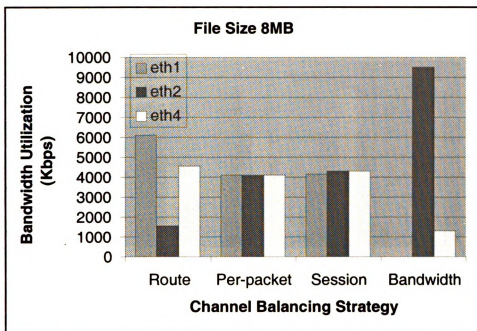


Figure 6.9: Bandwidth usage for File Size 8M

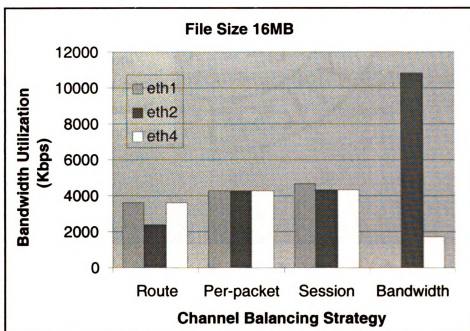


Figure 6.10: Bandwidth usage for File Size 16M

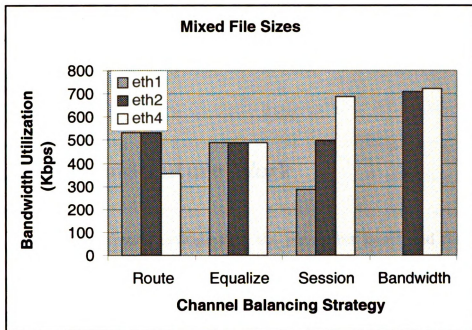


Figure 6.11: Bandwidth usage for Various (mixed) File Sizes

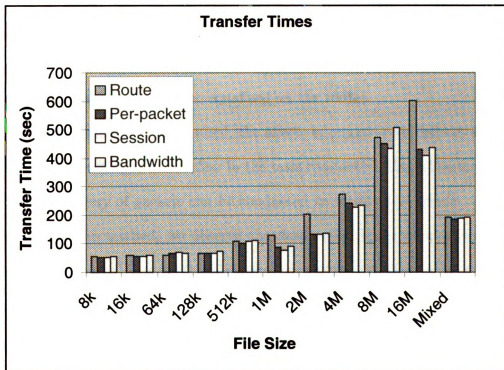


Figure 6.12: Transfer Times

Chapter 7

Conclusions and Future Work

The experimental results indicate that the per-packet based and session-based channel balancing perform better than the route-based and bandwidth-based channel balancing in the experimental setup. Bandwidth based channel balancing strategy tends to push more data on higher capacity links than try to push data on all the available links. Balancing traffic based only on the unused bandwidth might cause data to be pushed on an interface at a rate above the threshold value when the queuing delays on the interface will decrease the performance. This in itself can be studied further as another channel balancing strategy where the percentage of the unused bandwidth is sought to be equalized on the routes.

Due to the topology of the testbed, the effects of out-of-order delivery on the performance of the strategies is not clear in the tests conducted. Experiments simulating out-of-order delivery of packets can be conducted to study these effects.

Looking at the testbed, we observe the destination hosts were only three hops away from the source. The effects of varying the number of hops between the source and destination can also be taken up for future research.

Ideally we would like to see the performance of the different strategies in a real-world scenario. It is in the real-world scenario (or a well simulated environment) that the effectiveness of the strategies can be truly evaluated. The variations in the

connection times, lengths, data transferred, destination addresses, number of hops, latency, out-of-delivery of packets, etc, may cause the strategies to behave in a manner not brought out by the laboratory experiments. Also, experiments can be conducted to study the effects of the packet size on the performance of the strategies.

Other strategies can also be looked into as part of future work. One such strategy would be to look at the retransmission rates on the routes and pick the one with the least retransmission rate. History of the past connections can also be used to choose the route, if connections to a particular destination perform well on one of the routes, then future connections to that destination should be routed over the same route. As mentioned above, the bandwidth usage based strategy can be modified to use the percentage of unused bandwidth as an equalizing measure.

Bibliography

- [1] Glenn Herrin. *Linux IP Networking A Guide to the Implementation and Modification of the Linux Protocol Stack* <http://www.cs.unh.edu/cnrg/gherrin/linux-net.html>, 2000
- [2] The Linux Virtual Server Project. <http://www.linuxvirtualserver.org>
- [3] Bert Hubert. *Linux Advanced Routing & Traffic Control HOWTO*. <http://lartc.org/lartc.html>, 2002
- [4] Alexey Kuznetov. *Policy Routing*. `/usr/src/linux/Documentation/networking/policy-routing.txt`, 1999
- [5] Thomas Davis. *Linux Ethernet Bonding Driver mini-howto*. `/usr/src/linux/Documentation/networking/bonding.txt`
- [6] Alavoor Vasudevan. *The Linux Kernel HOWTO*. <http://www.tldp.org/HOWTO/Kernel-HOWTO.html>, 2003
- [7] Rusty Russell and Harald Welte. *Linux netfilter Hacking HOWTO*. <http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html>, 2002
- [8] Christoph Simon. *Nano-Howto to use more than one independent Internet connection*. <http://www.ssi.bg/ja/nano.txt>, 2001
- [9] Gianluca Insolvibile. *Inside the Linux Packet Filter*. <http://www.linuxjournal.com/article.php?sid=4852>, 2002
- [10] Gianluca Insolvibile. *Inside the Linux Packet Filter, Part II*. <http://www.linuxjournal.com/article.php?sid=5617>, 2002
- [11] Wensong Zhang. *Linux Virtual Server for Scalable Network Services*. <http://www.linuxvirtualserver.org/ols/lvs.ps.gz>, 2000
- [12] T. Brisco. *RFC 1794 - DNS Support for Load Balancing*. <http://www.faqs.org/rfcs/rfc1794.html>, 1995
- [13] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*, O'Reilly and Associates, Inc. 2001
- [14] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, O'Reilly and Associates, Inc. 2002

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 02504 8111