A Linux networking study

René Serral, Marisa Gil
{rserral,marisa}@ac.upc.es
CCABA/UPC

February 27, 2004

Abstract

The fast evolution and increase in use of nowadays networks forces researchers to look for efficient ways of managing all the information that travels through those networks. Added to that, the growing use of Linux as main operating system for servers and big grid computing farms, has developed many research lines for increasing the networking capabilities and efficiency of the kernel code.

Having that in mind, this document shows the basic internals of the networking code of Linux kernel and discusses the efficiency of the existing implementation in the 2.6 version. Besides, various tests are done for detecting existing bottlenecks in the networking core. Finally are also exposed guidelines for improving certain aspects of the kernel networking behaviour.

1 Introduction

The constant grow of network bandwidth and usage forces system coders to look for better ways of managing the system resources and kernel behaviour, this permits that basic issues such as packet forwarding or packet processing can be as efficient as possible.

Nowadays most of the Internet's core networks are based on Gigabyte Ethernet technology (up to 10Gb Ethernet interconnections), this kind of connection is the one offered also by many ISPs (Internet Service Providers) to their customers. So, the servers, firewalls and in general all the services offered have to be fast, reliable and capable of managing all the potential flow of information that the network can achieve. Same problem is found when the network is dedicated to grid computing farms, where the problem to deal with is the need of low latencies in the network communication for increasing the overall computation power.

With all the above picture, Linux fits in due to the huge amount of servers/firewalls running this operating system. So, kernel developers have to find ways of issuing all the packet matching/processing as fast as possible for assuring a good service to their customers.

This document is divided as follows. First we present the **Linux Networking Internals**, this section is dedicated to explain the basis of the networking kernel internals. Next section discusses the **Networking Environment**, where is explained the working environment built for testing the kernel, later are detailed the **Experiments and results** with a precise description of the whole test suite and the obtained results, finally the report ends with the **Conclusion and Further work**.

2 Linux Networking internals

The main goal of this section will be the basic explanation of the Linux Networking code inside the kernel. This section only tries to be a simple introduction to packet managing/processing that will ease the comprehension of the rest of the document. On later sections efficiency will be discused starting from this basic explanation. For doing the packet's journey explanation easier, this is done with an example.

The chosen protocol for explaining the kernel behaviour has been UDP. This protocol has been chosen because is less complicated and does the whole explanation easier than TCP. Besides, is enough representative for having an idea on how the kernel works.

This section has been split in two different parts, the first explaining how a packet is received from the network card, the second discusses the issues of packet transmission.

2.1 Packet reception

The main goal of all the kernel operations is efficiency and reliability. Those are the main reasons for which all the network kernel operations are based on interruptions, softirqs and bottom halves (see [1] Chapter 4 for more information on this subject). Having the above in mind, the networking code is divided into layers, those layers try to adapt to the protocol IP stack [2]. But for the sake of efficiency there are points where is better to mix up some information between layers.

Figure 1 shows an overview of all the important parts of the packet journey, each part will be detailed in the following sections.

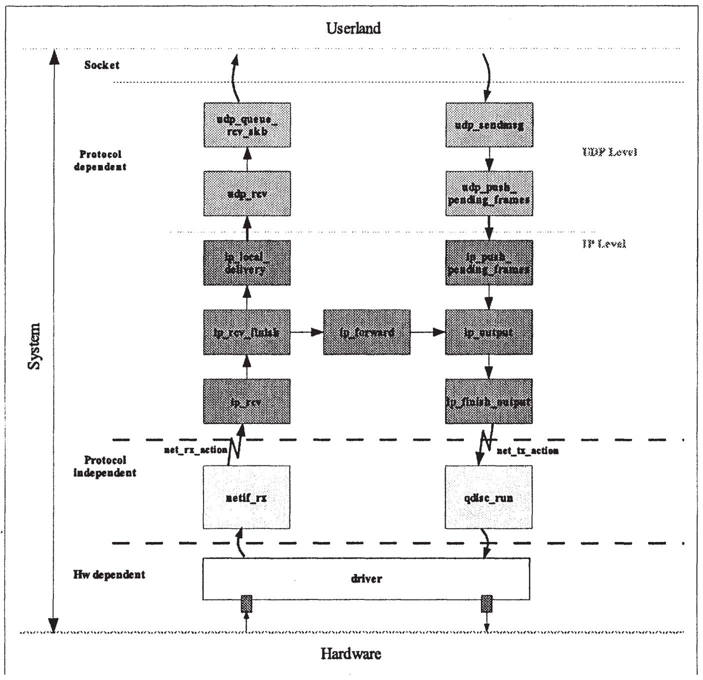

Figure 1: Kernel travel of an UDP packet

2.1.1 Hardware Dependent

When the Network card receives a packet from the wire saves it into a buffer and raises a reception interruption. This interrupt depends on the network card, but in general the function that the kernel maps to this interruption has the form: `{driver}_interrupt()` and can be found at the *drivers/net/{driver}.c* file. The point of this interruption is to be as fast as possible because blocks the access to the network card to avoid concurrency problems. Finally the function `netif_rx(...)` of the next level is called.

2.1.2 Protocol independent

The function `netif_rx(...)` has the only mission of copying the packet on the actual CPU queue (for being nice with the CPU cache and for concurrency limitations on Bottom Halves) this function then schedules the packet reception issuing a `cpu_raise_softirq` softirq that is mapped to `netif_rx_action`. This way the Network card is released without blocking meanwhile the packet is processed. The other important work done in this level, by the softirq, is to determine the protocol that identifies the packet, IPv4 will be supposed.

Besides all the above work done at this level, another important task issued is to fill the timestamp field of the socket buffer associated to the packet. This is done as fast as possible for having the maximum timestamp precission as possible. The kernel at this point also permits special hardware to fill the timestamp field instead of the kernel, in the case of external timing hardware like GPS. This part will be important in subsequent sections where delays and time values are measured.

2.1.3 Protocol Dependent

Assuming that the packet is IPv4 the protocol dependent part at this level will handle the package, the function in charge of issuing the packet handling is `ip_rcv(...)`, this function does all the sanity checks, such as, verifying the packet's CRC, packet's length... Then the control of the packet is passed to Netfilter [3] (firewalling/security package), which, if the packages complies with the security restrictions is passed to `ip_rcv_finish(...)` function if the packet's destination is the local box, or `ip_forward(...)` otherwise. At this point the system will enquire the IP packet to look for fragmentation. The case with fragmentation is left for further study.

For the sake of this discussion let's assume that the packet's destination is local, so, the kernel processes it through the `ip_local_delivery(...)` function. This function detects the next layer that corresponds to the packet and sends it up there, this example supposes the case of an UDP packet. So the next called function will be `udp_rcv(...)` which verifies that the packet is correct, CRC check, payload length... If all is correct the next step is done by looking for the associated socket for that packet and queue it, this is done by the `udp_queue_rcv_skb(...)` function, finally the packet is delivered to upper layers, which explanation goes beyond the scope of this document.

2.2 Packet transmission

Understanding packet transmission after packet receiving is very easy. For doing an easier explanation, this time, the packet journey will be discused from UDP level, and descending until physical level.

2.2.1 Protocol Dependent

As before, this explanation doesn't take into account the upper layers of the protocol, so, is supposed that the socket associated to the connection exists, and the sent packet is UDP.

The function in charge of receiving the packet to be forwarded is `udp_sendmsg(...)`, this function verifies the socket and prepares all the possible UDP options, also detects if the packet is part of a flow for optimising the packet construction. Eventually the control reaches the next function, that is `udp_push_pending_frames(...)` that fills all the necessary fields of the UDP header, such as source port, destination port... If necessary also fills the CRC that needs the IP address, that is fetched from the flow of the connection tracking algorithm. Next step is to go to the next level, that is IP. The function involved in that is `ip_push_pending_frames(...)` that takes a packet from the socket queue, constructs the IP header, issues the mandatory sanity checks and gives the control to the netfilter package. If the packet succeeds the security policy then is sent through the `ip_output(...)` function, that only verifies if the packet needs fragmentation, if not, then the cached route of the packet is taken, and the package is sent through the `ip_finish_output(...)`, that issues the final security check through netfilter and sends the packet with `ip_finish_output2(...)`. Eventually the functions will activate the protocol independent part with a softirq `net_tx_action(...)`.

2.2.2 Protocol independent

Once the kernel thread awakes from the softirq executes the function `qdisc_run(...)`, for the sake of simplicity not all the options will be explained, only say that here is issued all the traffic shaping, traffic scheduling and policing. In other words, this part configures all the device queues for changing the default FIFO queue of the network cards. This is very useful for assuring Quality of Service to certain flows. Finally, the actual output device takes the packet.

An important point is that `qdisc_run(...)` is local for each hardware device for configuring different behaviours.

2.2.3 Hardware Dependent

As in reception an interruption is raised, that is `{driver}_interrupt()` that looks first for packets to be sent (`{driver}_tx_interrupt()`) or as seen before to receive. And the packet is finally copied to the network card and later to the network.

The above discussion hasn't detailed very much the sending procedure. So, parts such as ARP, or route selections have been avoided for the sake of simplicity.

3 Working environment

The best solution for an accurate network study is to use a dedicated network for doing the tests. The goal of such tests is to highlight the proposed problem and guide to a possible solution. This testbed has to be reliable enough to have at least:

- A linux router for measuring the forwarding capabilities
- 2 linux machines, for sending and receiving traffic.
- A synchronisation network, this way the computation of delays is possible.
- All the components of this testbed have to be dual stack, for testing either the IPv4 and the IPv6 configuration.

Figure 2 shows the simplified scheme of the used testbed. This scenario is composed by 3 sender machines, another machine for receiving the traffic, the interconnection backbone and the synchronisation network. All the test and measurements will be done at the backbone.

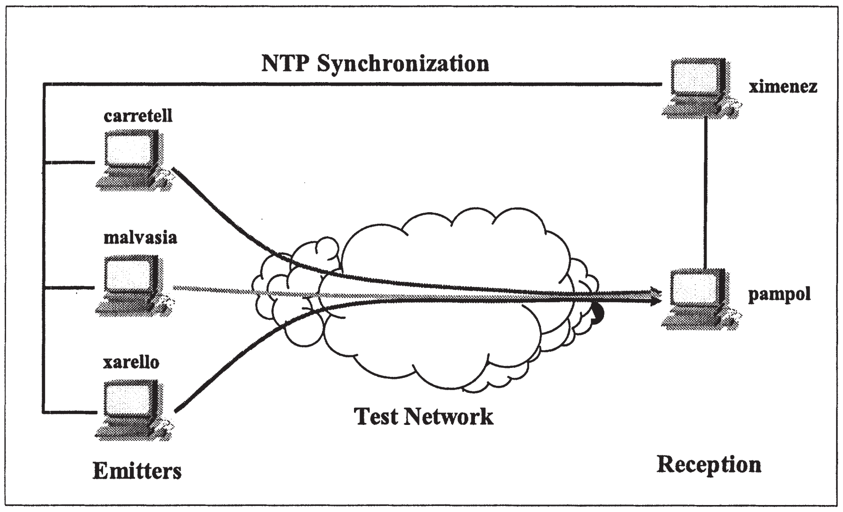

Figure 2: General scenario's scheme

3.1 Tests Methodology

There are several things that are important for assuring good results on networking experiments.

- *Time synchronised machines:* For calculating delays is mandatory to have source and destination synchronisation. The taken timestamps have to be from the same source of time, so they can be compared.
- *Controlled environment:* For being as much accurate with the results as possible all the networking and operating system events should be controlled, so all the results will be due only to known events.
- *Structured methodology:* before even starting the measurements, is very important to prepare the whole tests' schedule to be done, is also important to have a good naming convention for labelling the results for later identification.

Taking into account all the above conditions the whole used scenario is inside a private network, all the external traffic is discarded, and only the important packets to analyse are actually travelling through the network.

The part of the synchronisation will be done with the Network Time Protocol, which ensures the time synchronisation of all the machines involved on the NTP network. The basis of the NTP protocol is to adjust the internal clock ticks progressively of every machine to assure, after some time, the same clock frequency on all of them [4].

3.1.1 User level analysis

The used methodology for doing all the tests is very simple. Once the machines are synchronised is necessary to use an application capable of keeping track of all the tests while are being done. For this purpose there is an application called *NetMeter* [5] (a Network Metering tool). With the help of this application is possible to save the state of all the test. Then, if some tests have to be repeated, is an easy task to recover the configuration file and repeating them, even if necessary, changing some parameters of the experiment.

NetMeter also permits to generate a graphical representation of the network parameters under study.

3.1.2 Kernel level analysis

The kernel analysis has been done with code coverage tools, GCOV kernel module has been used. So, the router machine has been compiled with code coverage options, and after every test the coverage statistics have been collected for later comparison. The point of examining the source code is to look for possible misleading behaviours, forced by the fact that the machine is overloaded and spends most of the time forwarding packets.

One of the most important comparisons done is the coverage differences between big packet size and small packet size. The goal of this comparison is to infere the bottleneck points when the client machines are sending lots of small packets. This study is very important because the actual traffic of Internet is composed basically of such small packets, for example in TCP Acknowledgment or small pictures loaded from web pages. So improved small packet handling is mandatory for an efficient kernel technology.

Usually GCov module is used for detecting and isolating dead code, this report gives a twist at GCov usage by detecting the path through the kernel of an IP packet, giving the possibility of detecting possible malfunctions or inefficiencies on the default packet journey. As can be seen later on this paper such analysis is done on small packets through the kernel.

3.2 Scenario's description

The main goal of the scenario is to be flexible, capable of doing in the same environment tests with different protocols (IPv4/IPv6) seamlessly and with enough components for having reasonable delays, comparable with the used hardware's internal clock.

The Figure 3 shows the detailed version of the scenario. There can be seen the physical connection between the components. For the sake of simplicity, the NTP circuit is not represented.

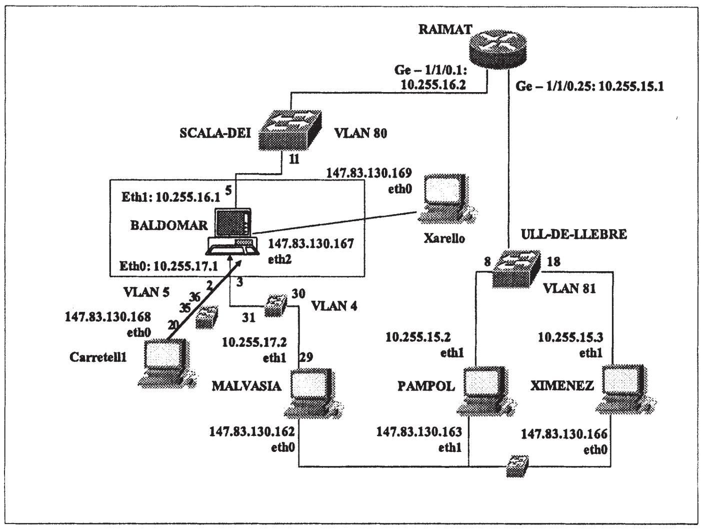

Figure 3: Detailed version of the scenario

The most important part of the scenario is the central router (*baldomar*). This machine is a Pentium III with the RedHat 9 distribution (kernel 2.6.0test11), that will be the main piece of the study. The kernel version installed is compiled with GCov patches for analysing the coverage of the executed kernel code.

All the machines on the left part of the scenario will be the sender machines, where *carretell* and *malvasia* (RedHat Linux) are connected to *baldomar* through an Ethernet link (10Mbps), meanwhile *xarello* (Sparc Station - Debian Linux) is connected through a Fast Ethernet link (100Mbps). All the output is connected (*pampol* - Debian Linux) on a Fast Ethernet link.

The rest of the components are either Ethernet switches or IP routers for increasing the round-trip time of the scenario and for interconnecting all the important parts. So, given the the low interest of this components for our study won't be discussed here.

4 Experiments and Results

For accomplishing the above goals some tests have been run. The main test suite is based on the study of packet forwarding, so, different traffic flows will be sent through the router, the obtained results will hold the packet transmission delay, here either kernel analysis and user level analysis will be done. To have more diversity of results, this tests will run with different packet sizes, this way is also possible to compare the impact of the number of packets in the network. Added to that, some of the tests will be done with the Next Generation Internet Protocol (IPv6) [6], this time only the user level study will be done, the results will be compared with IPv4's.

The Linux kernel version under test will be the 2.6.0test11, last available version during the experiments.

The tests will stress the router's network connection permitting end-to-end delay calculation. These results will be used for the analysis of the source code, helping to identify the possible bottlenecks (GCOV [7] has been used).

4.1 Packet sizes and bandwidth allocation

All the packet sizes specified in this section are payload sizes, which means that, for an UDP payload size the real physical size will be: $S_{phy} = P + H_{UDP} + H_{IPv?} + H_{MAC}$.

Where:

$P = UDP\ Payload$

$H_{UDP} = UDP\ Header = 8bytes$

$H_{IPv4} = IPv4\ Header = 20bytes$

$H_{IPv6} = IPv6\ Header = 40bytes$

$H_{MAC} = MAC\ Header = 18bytes$

The Table 1 lists the relation between all the tests, all the tests suppose a link capacity of 100Mbps:

Packet Size	IPv4	IPv6
Small	100 bytes	1.400 bytes
Big	80 bytes	1.380 bytes

Table 1: Tests

So at physical level the packet size is $1.446b$ for big size packets and $126b$ for small ones (IPv4).

As you may have noticed, the packet's payload size between IPv4 and IPv6 experiments is different, the reason is that for issuing comparable tests the overall network traffic must be the same, and given that IPv6 has a header of 40 bytes in front of the 20 bytes of IPv4, the packet sizes have been tunned accordingly.

The way for generating up to 100Mbps will be using all three traffic generators shown in the previous section (*xarello, malvasia, carretell*). Given that *carretell* and *malvasia* have only a 10Mbps link, both of them will generate this amount of traffic, and *xarello* will do 80Mbps.

4.2 Test details

As stated before, given that there are two machines with only an ethernet link (10Mbps), and those machines have low computational power (about Pentium 133Mhz) the tests have been distributed the following way:

- **xarello:** [*Linux*], good computation power, so, will generate a total amount of 80 Mbps of traffic the following way: 7.150pkt/sec with a size of 1.400b per packet, that at physical level is an amount of about 82Mbps.

  For small packet size the traffic is specified as 100.100pkt/sec and 100b per packet (80b for IPv6).

- **carretell:** [*FreeBSD*], low computational power, this one will generate over 8Mbps, that is 850pkt/sec with 1.400 bytes per packet, which at physical level equals to 9Mbps.

  The small packet side is 11.900pkt/sec with a size of 100b (80b for IPv6).

- **malvasia:** [*Linux*], same traffic parameters as `carretell`.

The time duration of each test is 60 seconds. This duration is good for analysing all the kernel behaviour because is:

- Enough time for having many context switches that can introduce delay variation.
- High packet generation for cache failures and memory traffic with the network card.
- Enough packets for extracting statistics and having good conclusions of the general behaviour.
- Time enough for overflowing all the kernel buffers due to all the generated traffic.
- Not too much packet workload for later processing in a reasonable amount of time.

4.2.1 Test problems

The main problem that has been found is the fact that for small packet size the hardware used isn't powerful enough for generating a 100.100's packet rate per second. At *xarello* the maximum amount of packets per second have been around 39.500. This highlights the main problem found, due the huge overhead leaded by the small packet generation. The problem is obvious, given that for the same effective bytes transmitted the required computational power is much bigger, most of the time the kernel is only generating packets and calculating CRCs, at the core network, besides, the problem arises due to the huge amount of headers to process by the routers.

Another, less important, problem found has been the high CPU usage on the destination machine, that's because the destination is a single machine that receives all three flows simultaneously.

4.3 Results

The results are divided in two main parts, the first one describes the reported delay from the *application level* point of view. The second part, on the other hand, focuses the explanation within the *kernel level* point of view.

All the results that can be seen at this section show only the flow from *xarello* to *pampol*, the point is that the other generated flows are used only for collapsing the physical link and the router. But studying such flows won't add any value to the analysis.

4.3.1 NetMeter results

This part of the study takes into account only the application point of view. There are parts where more precise analysis is needed, such analysis is done on the kernel section below.

This subsection is divided into two parts, first the small versus big packet differences are discussed; then the explanation focusses on the efficiency comparison between IPv4 and IPv6.

Packet size comparison

As stated before, the packet size has great impact on network behavior, the main reason is the huge impact on the kernel for processing all the IP headers. This section is devoted to explain all the pros and cons of application's level packet handling. The results obtained in this section reflect this impact on the system under test.

Figure 4 shows the one way delay distribution of the tests referring small packets and big packets. The graphical representation is done in distribution form because the number of packets of each test is very diferent, given that the small packet rate is about 100.100pkt/sec and the big packet part is about 7.150pkt/sec.

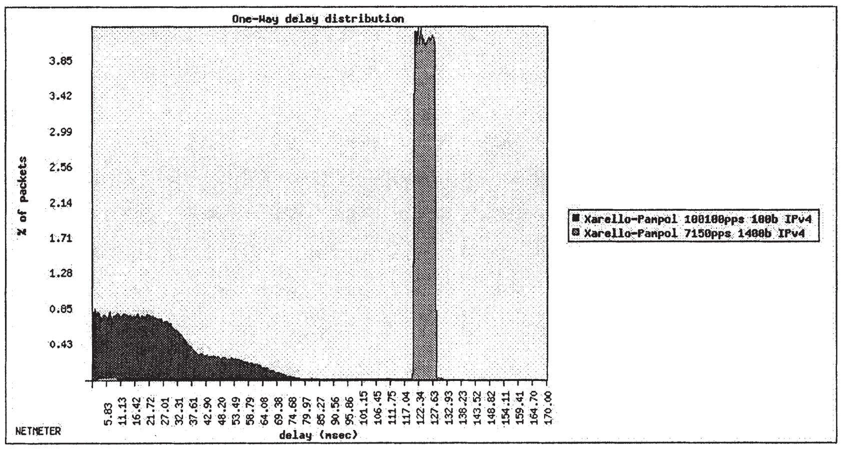

Figure 4: Big vs small packet size delay distribution

This distribution holds in the *y-axis* the percentage of packets with a given delay and in the *x-axis* the actual delay. This graphical representation hints the most natural behavior of both packet sizes. The main tendency is that small packets have lower latencies, given that the kernel has less information to move from user space to the network, or inside the router from one network interface to another. On its side the big packets have bigger delay. For a clearer representation, Table 2 shows the statistical results of the tests corresponding the above figure. All the time related values are expressed in milliseconds.

The most impressive result of the table is the fact that small packets have a much bigger chance of being discarded. That's because there are lots of them collapsing the source machine and the router. Another related matter that can be seen on the results is that the effective bandwidth used is much bigger on the big packet test, given the fewer packet losses and higher payload weight of each packet.

Measure	Big size	Small size
Avg. Delay (msec)	121,411	29,685
Max. Delay (msec)	1.133,459	1.026,001
Min. Delay (msec)	2,192	0,707
Delay variation (msec)	1.131,267	1.025,294
Packet Loss (pkts)	80.325	5.255.815
Used bandwidth (Kbps)	64.992	3.983

Table 2: Big vs small packet statistical results

Protocol comparison

In this section is discussed the user level's protocol comparison. This study has the goal of comparing the implementation efficiency of both protocols. This is done because the number of Internet's hosts using IPv6 is growing. Moreover this version is still under development and in the future will substitute IPv4, so it makes sense to check its efficiency.

Given that the badwdith utilisation at physical level will be the same for both protocols as told in section 3.1.2, the obtained results can be compared.

Figure 5 shows the per packet delay representation. As can be seen the representation of both tests is very similar, this states that the current IPv6 implementation is efficient. This is really a handicap due to the fact that IPv6 is a protocol with 128 bits of address space in front of the 32 bits of IPv4.

Table 3 shows the mathematical results of the same tests. As can be seen the results are somewhat better for IPv4 in terms of effective bandwidth, that's caused by the bigger header of IPv6. The results show smaller delays for

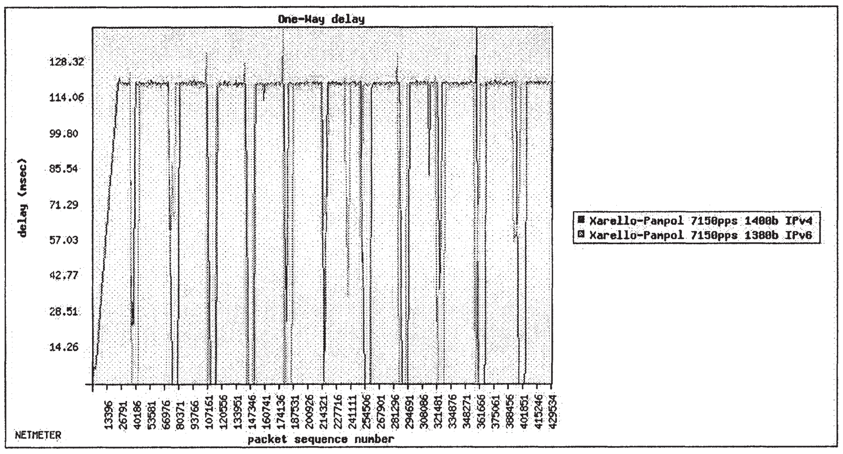

Figure 5: IPv4 vs IPv6 per packet delay

IPv6, which is good given the conditions for IPv6, those results remark the efficient implemetation of Internet's Next Generation Protocol stack under Linux.

Measure	IPv4	IPv6
Avg. Delay (msec)	121,411	121,127
Max. Delay (msec)	1.133,459	1.129,252
Min. Delay (msec)	2,192	1,850
Delay variation (msec)	1.131,267	1.127,402
Packet Loss (pkts)	80.325	78.285
Used bandwidth (Kbps)	64.992	64.362

Table 3: IPv4 vs IPv6 statistical results

If we take a closer look at the Figure 5 there are burst of traffic losses, at what seems a near constant interval. This peculiar results are obtained with all the tests and all the protocols (not included for space constrains).

For analysing this packet losses is necessary to focus the explanation on the kernel implementation.

4.3.2 Kernel results

The kernel study is done for understanding the low network efficiency in some situations. The problems found range from bursty packet losses to low bandwidth utilisation, due to time spent by the kernel processing packet headers.

For examining the kernel code, the easiest way is using coverage tools. The procedure used for looking at the parts of the kernel code that are being used is to capture the executed code during the tests and then comparing the cases of big versus small packet (GCOV).

For avoiding extrange behaviors kernel tests have been done separatedly from the application level ones. So, only one flow has been studied, that reduces the covered code and eases the work of detecting the critical parts. Having that in mind, the Table 4 shows the amount of recevied packets on several points inside the scenario, at source, at IP level on the router and at destination. As can be observed there are many packets that get lost on the way to the destination. The higher packet loss can be seen on the router with small packet tests, there the loss is about 80%, which is too much for being acceptable.

The results of the Table 4 reflect the GCov results (on the router) and NetMeter results (source and destination). For the interpretation of the GCov results the analysed functions will run once for packet received/transmitted, so, the text always refer those counters as packets.

Looking at the above data is clear that there is some problem inside the router box. So, let's look deep inside the kernel code.

Test	Source report (Application Level)	Coverage report	Destination report
IPv4 small packet (pkts)	6.006.000	1.203.782	1.087.749
IPv4 big packet (pkts)	429.001	429.001	396.819

Table 4: Kernel tests data

Driver level analysis

Looking at the code coverage data obtained from the small packet test, at the router machine:

- Interruptions received at Network Interface Card (NIC): 5.845.227[1]
- Packets processed at NIC level: 6.006.117
  - When inside the interruption the driver loops while there is data on the reception buffer, so, it can process several packets per interruption.
  - There are more packets than the test because the machine has two network cards with the same driver and the other interface can receive packets during the test, besides, the FastEthernet switch, sends control frames periodically (at level 2 as shown later).

The above list shows that the low level drivers of the kernel are capable of handling all the packet workload, at least at the first interruption level stage.

Protocol independent level

Still inside the interrupt handling level the `netif_rx` function has a counter of 6.006.134. There are more packets because the other interfaces are also receiving packets (potentially). Besides, this code is driver independent, so global to all the system.

If we take a closer look at the `netif_rx` function is possible to see that the fallback option, that is packet dropping, has a counter of 4.802.247 packets, so if there were 134 extra packets on the counter, we take $6.006.134 - 4.802.247 = 1.203.887 packets$, but the Coverage at IP level reports 1.203.782 packets.

So, the first look shows that the main problem of the packet dropping is due to the slow handling of the interruption, which is critical.

Protocol dependent level

Continuing the ascendent way through the IP stack, the next step is the `ip_rcv` which holds a counter of 1.203.875 packets, still far from the 1.203.782 stated on Table 4. This value (1.203.875) differs from the given at the above section, the reason is because the switch that interconnects the source machine and the router sends Level 2 frames for detecting network loops, so those frames don't reach the IP level.

The rest of the missing packets at the output get lost during the call to packet forwarding, this is because some packets have the router as destination. Those packets belong to different interfaces.

So, all the tests done, focus the main problem to the network interface interrupt handling routine.

5 Conclusions and Further work

For giving good conclusions further work in the kernel internals is necessary, but in the current position is easy to tell that the interruption frequency is a critical issue in the networking code. A possible solution to that problem would be to move out some work done in the interruption code, an example of that is the socket buffer that is allocated inside the interruption for each sent/received packet.

As opened lines of further research there are:

1. Kernel level:
   (a) Buffers size for preventing packet losses.

[1]All this data is directly extracted from GCov tests.

   (b) Check all kernel code memory accesses alignment.
   (c) More extensive tests for overloading much more the machine.
   (d) Try different NIC drivers with more optimised code, and smarter hardware to avoid interruption bursts.
   (e) Besides packet forwarding, the kernel level packet generation/reception would be a good analysis also.
   (f) GTrace analysis [8]
   (g) OProfile analysis, knowing the used time on each part could help to focus further study.

2. User level:
   (a) TCP tests for more realistic Internet's generic traffic.
   (b) Extra flows to stress more the routing cache.

References

[1] Daniel P. Bovet & Marco Cesati. *Understanding the Linux Kernel (Second Edition).* O'Reilly, 2003.

[2] Craig Hunt. *TCP/IP Network Administration.* O'Reilly, April 2002.

[3] Harald Welte. Netfilter. Technical report, http://www.netfilter.org, 2003.

[4] David Mills. Network time protocol. Technical report, http://www.ntp.org/, 2003.

[5] René Serral. Netmeter a network metering tool. Technical report, http://www.ccaba.upc.es/netmeter, 2004.

[6] R. Hinden [Nokia] S. Deering [Cisco]. Internet protocol, version 6 (ipv6) specification. Technical report, IETF - RFC - 2460, December 1998.

[7] GCC Crew. Code coverage project. Technical report, http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html, 2001.

[8] Galderic Punti Marisa Gil Xavier Martorell Nacho Navarro. gtrace: function call and memory access traces of dynamically linked programs in ia-32 and ia-64 linux. Technical report, UPC-DAC-2002-51, November 2002.