



Technischer Bericht

Project Report: Performance Analysis of the TCP/IP Stack of Linux Kernel 2.6.9

Jan Demter, Christian Dickmann, Henning Peters, Niklas Steinleitner,
Xiaoming Fu

Technische Berichte
des Instituts für Informatik
an der Georg-August-Universität Göttingen

April 2005

Georg-August-Universität Göttingen
Institut für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14

Fax +49 (5 51) 39-1 44 15

Email office@informatik.uni-goettingen.de

WWW www.ifi.informatik.uni-goettingen.de

Abstract

This document reports the project “performance study of the TCP/IP stack for the Linux kernel” which we performed during the practical course *Computer Networks* in winter semester 2004/05, including its design, implementation and performance results. We analyzed the packet processing time traversing each layer of the Linux kernel 2.6.9 TCP/IP stack (socket, TCP/UDP, IP and Ethernet) and the influence of multi-threading and different packet sizes. The design is based on the idea of inserting probing points via hooks in the kernel code and export timing data to a user-space application. A packet generator and analysis tools were also developed. The results demonstrate a number of key concepts in TCP/IP networking, such as layering, user-system interface, connection versus datagram modes, processing routines and their overhead in different layers. Some preliminary results reveal the system has its bottlenecks in different situations, and our tools released under GPL-license have been designed in such a way that allows easy extensibility for other networking diagnostics purposes.

Project Report
Performance Analysis of the TCP/IP Stack of
Linux Kernel 2.6.9

Jan Demter Christian Dickmann Henning Peters
 Niklas Steinleitner

Instructor: Dr. Xiaoming Fu

Institute for Informatics
University of Göttingen, Germany

April 19, 2005

Contents

| | | |
|----------|---------------------------------------|-----------|
| 1 | Introduction | 2 |
| 2 | Problem description | 3 |
| 3 | Technical approach | 4 |
| 3.1 | Packet tracking | 4 |
| 3.2 | Packet identification | 4 |
| 3.3 | Packet generation | 5 |
| 3.4 | Timestamps | 5 |
| 3.5 | Development environment | 6 |
| 4 | Implementation details | 7 |
| 4.1 | Packet Generator | 7 |
| 4.2 | Probing points | 8 |
| 4.3 | Timestamping code | 9 |
| 4.4 | Errors in measurement | 10 |
| 4.5 | Automated measurement | 10 |
| 4.6 | Evaluation considerations | 11 |
| 5 | Results | 13 |
| 5.1 | Increasing the packet size | 13 |
| 5.2 | Increasing the thread count | 15 |
| 5.3 | TCP vs. UDP | 15 |
| 6 | Summary and future work | 17 |

Chapter 1

Introduction

The objective of this project is to investigate the path TCP/IP packets take on a single node through each layer of the TCP/IP stack. Analyzed aspects are the queueing behavior in socket layer, UDP versus TCP processing, TCP fragmentation and IP and Ethernet layer processing. The performance metric we selected is the packet processing time, since it is the most noticeable behavior for a packet from an end-to-end point of view.

Papadopoulos and Parulkar [Pap93] presented a performance study concerning UNIX IPC including the underlying TCP/IP protocol on SunOS 4.0.3 running on Sun Workstations with 10Mbps Ethernet. With regards to TCP/IP protocol evaluation, they studied performance of queueing in layers, buffer requirements, protocol control mechanisms and the interaction with the operating system. In order to do so, several probes were inserted in the SunOS TCP/IP source code at different locations. However, they mainly focused on using IPC to trigger TCP/IP processing, with little consideration on UDP and the influence of other applications running directly above TCP. Moreover, the packet sizes used in their tests were rather small. Welte [Wel00] also described his method on how to determine the packet cross-layer processing times inside a router and a receiving host running Linux Kernel 2.4 in x86 architecture. In contrast, in this project we not only look at per-packet processing in these TCP/IP layers, but also study the difference between the uses of connection-less transport protocol UDP and connection-oriented TCP, as well as other scenarios such as larger packet size (which requires TCP fragmentation and reassembly), multiple TCP connections and multiple threads over a single TCP connection on a link. The operating system we used is the recent Linux kernel 2.6.9; all machines used are Via Eden 553MHz PCs with 256MB RAM and RTL-8139 100Mbps Ethernet NICs.

Chapter 2

Problem description

Basically there are at least two alternative approaches to study the per-packet, per-layer processing times. The first approach is to use APIs for different layer interfaces to filter and collect the performance data in the program level. Another approach is to follow the methods proposed in [Pap93], inserting certain probing points, placing timestamps and recompiling the kernel.

Netfilter is one method belonging to the first group. It is a packet filtering engine of kernel version 2.4 and 2.6. The netfilter code is executed via hooks that can be enabled during compilation at specific code locations that were made for packet filtering, but are not suitable for our project which targets at per-packet, per-layer and per-routine (if necessary) performance analysis. Furthermore, enabling netfilter creates unnecessary packet processing overhead which has an impact on the measurement results. These disadvantages can be avoided by following the second approach. Our investigation further shows that the probing point hooks can be selected with a focus on performance measurement only without introducing additional processing overhead, and our code can be flexibly extended for other analysis.

Before describing the detailed design, we identify the following project tasks which are critical to evaluate a TCP/IP packet processing across networking layers and required functional routines:

- Probing points at each layer-to-layer transition (Christian Dickmann, Henning Peters and Bernd Schlör).
- Accurate and efficient measurement of processing with a minimal additional overhead (Jan Demter and Henning Peters), which implies the performance data that may be exported to the user-space, avoiding unnecessary data processing in the kernel space.
- Reliable packet identification at all probing points (Christian Dickmann and Henning Peters).
- Easy and automatic evaluation and illustration of the collected data (Christian Dickmann and Sebastian Vogelsang).
- Ability to generate packets in different modes (single process, single thread vs. multiple threads vs. multiple processes; data rate and size, etc.) (Niklas Steinleitner).

In addition, Ubbo Veentjer worked on a development environment using User-Mode Linux [UML]. Bernd Schlör was the project and development leader. Michael Sobol contributed to the investigation of previous approaches, general discussions and project presentation.

Chapter 3

Technical approach

3.1 Packet tracking

Our main problem was how to track packets traversing the Linux kernel. As far as we know there are no suitable hooks already implemented in kernel code that can be used to find out which packet passed which probe at what time. Netfilter is a packet filtering subsystem implementing several such hooks, but in our experiments it showed that required processing overhead is non-marginal. Moreover, netfilter code is limited to the interface between the transport layer and the IP layer, while hooking at other layer boundaries, such as socket layer and ethernet layer, are not supported.

To measure further aspects of packet traversal we decided to implement our own timestamp probing code that takes timestamps at all layers (and required routines) of the network stack directly into the kernel. In theory this seems to be trivial, we simply need to insert such a probe before and after each layer/routine border. But it turned out that finding the right position is like searching for a needle in a haystack. The core part of Linux kernel 2.6.9 network stack consists of thousands of lines of code, and (to the best of our knowledge) existing books just describe the basic functionality from a high abstraction level.

Finally, we inserted thirteen probing points at relevant positions. The exact locations can be found in section 4.2 and a patch for the kernel 2.6.9 is available freely in [Kperf].

3.2 Packet identification

A second problem was how to identify a packet in the network stack. Various network protocols (e.g. TCP) implement a sequence number which one could use, but that number is not accessible from all positions where we would like to measure. We discussed various ways and they boil down to two basic approaches: “payload independent” measurement and “payload dependent” measurement.

1. For “payload independent” we would not use the payload of a packet to make any assumptions of its identity. Therefore we would use other characteristics such as writing hidden information into unused segments of IP datagrams. Because we are only interested in the way from socket to TCP/IP network kernel and vice versa, but not in measuring network latency, this approach could be used, but it has a drawback that it would be hard to see how fragmentation occurs.

2. In the “payload-dependent” case, we need to define a simple payload format that could be used to identify the packet uniquely. In order to study the fragmentation issue, we follow this approach and the payload format is defined to consist of 16 bytes. The first 8 bytes are used to mark the beginning of a so-called “frame” in our context (here, 0xFFFFFFFFFFFFFFFF), the other 8 bytes are left for our identification number (4 bytes sending thread number and 4 bytes packet number). Thus, in case of large packets we are able to check in which particular frames fragmentation and reassembly occur.

Because we are more interested in packet-processing performance (e.g., fragmentation, IP processing) than in being dependent on the used payload, we chose the second approach. It is anticipated the first approach could potentially be useful in analyzing real traffic generated from high-layer protocols like HTTP or RTP.

3.3 Packet generation

Next, we needed to generate packets, so that we were able to analyze how the Linux network stack handles packets of different sizes, how fragmentation occurs and how it behaves under different speeds and end program models. To accomplish these tasks, a multithreaded client and a polling server were written in C. The client is able to generate a adjustable number of packets. Each packet is filled with the 16-byte frames as described in section 3.2. All frames belonging to a same packet share a unique combination of thread and packet numbers. Once a client generates such packets, they are sent via a TCP socket or a UDP socket to the server. The server accepts TCP and UDP traffic on a given IP address and port number. It recognizes the reception of packets and then silently discards them in the application layer. We measured the end-to-end traversal time for different scenarios with the concentration on data packets containing a payload.

3.4 Timestamps

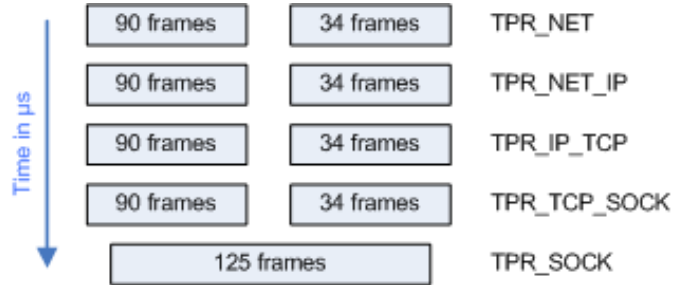
Finally, we had to find a way how to store our obtained timestamps and make the results accessible for further analysis in the user-space. The data analysis was implemented via simple scripts which take the raw data as input and output the (calculated) results in a graph or full-text presentation. Writing those scripts can be easily done in high level scripting languages, such as Perl or PHP. We used PHP to implement an automated test environment, which is described in detail in section 3.2. For storage and access, we found a simple solution. First, we allocate a virtual memory segment at initialization. Each time the timestamping code is called, we write into this segment and update the tail pointer. This can be done very fast in constant linear time. Of course, the write procedure should only have a minimal influence on the timing results. See the topic 4.4. for information about this issue.

Furthermore, we create a proc file entry. A proc file is a virtual file residing in the proc directory. Reading this file is like stepping through our kernel memory segment in a formatted way. From this output one can determine which parts of a packet traversed a probing point at a given time. For implementation we used the Linux kernel sequential file interface (seq_file) for handling large proc file entries. Here is an example of how this output looks like if the client is invoked to send 30 TCP packets each consisting out of 2000 bytes:

```

278 id 6 seq 31 thread 1 ts 1105888396.088332 x90
279 id 7 seq 31 thread 1 ts 1105888396.088341 x90
280 id 8 seq 31 thread 1 ts 1105888396.088345 x90
281 id 6 seq 31 thread 1 ts 1105888396.088511 x34
282 id 7 seq 31 thread 1 ts 1105888396.088520 x34
283 id 8 seq 31 thread 1 ts 1105888396.088524 x34
284 id 9 seq 31 thread 1 ts 1105888396.088649 x90
285 id 9 seq 31 thread 1 ts 1105888396.088663 x34
286 id 10 seq 31 thread 1 ts 1105888396.088691 x125

```



As one can tell, there are currently 2 fragments of sequence 31 passing all receiving probing points (id 6, 7, 8, 9) but the socket layer itself (id 10). In TCP layer, the fragments are first collected and then combined into a bytestream that is read sequentially by the user-space application. These timing results can be processed for further interpretation.

3.5 Development environment

Because we did a lot of work in kernel space, we knew very early that doing those tests on a normal Linux box would be too time consuming. A debugging session could still be done via serial interface, but if we would screw up something, and we knew that we would do so sooner or later, each time a reboot would be necessary. User-mode Linux [UML] seemed the way to go for kernel development. UML is a patch for Linux kernel that enables it to be run as a Linux virtual machine. The increase of productivity was proved several times. We set up a networked server where several persons were able to work on their own custom testing kernels in parallel and commit their changes to a subversion (SVN) repository for version control. Of course, this setup was only intended for development purposes and later replaced by a non-uml testbed consisting out of three distinct machines.

Chapter 4

Implementation details

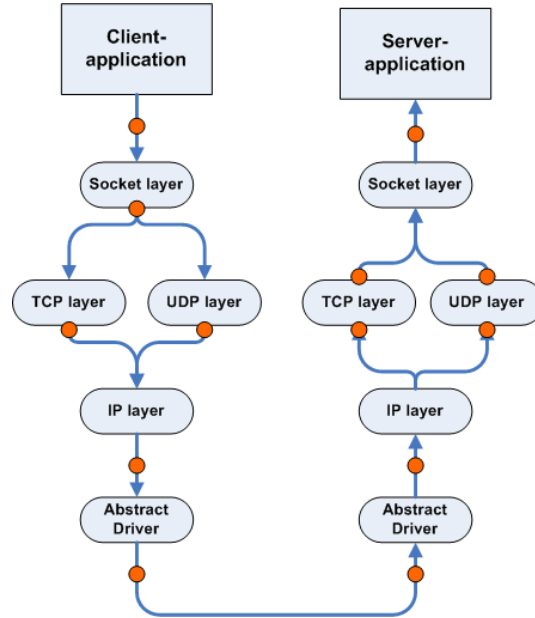
4.1 Packet Generator

After a review of several packet generators, we found that none of them met all of our requirements. Therefore, we developed our own packet generator (including receiver). It consists of two parts, a server-application part and a client-application part. The packet generator is implemented in ANSI C.

- Server
The server accepts all incoming TCP connection requests and UDP data requests on a specify port. Then it opens a socket for each session and listen for packets on this socket. If packets come in over this socket, the application receives all packets and drops them.
- Client
We implemented a multithreaded client which can initiate both TCP and UDP sessions to a given \langle IP address, port number \rangle pair if the server application is running at this port. The client provides the use of different numbers of threads and sockets, so we can cause the following cases:
 - each thread uses only one socket
 - one thread uses many sockets
 - one socket is shared by many threads

The packet generator basically does the following jobs: sends a user-specified number of packets, fills all these packets with our own 16 bytes frames payload (see 3.2) and records threads and sockets numbers in each of these frames. For more realistic emulation of real traffic, the client can be further developed to wait a randomized idle time within a range between two consequent packets.

4.2 Probing points



In detail, our probing points are located in Linux kernel network stack at following positions:

- net/socket.c:1554 TPS_SOCKET
- net/ipv4/af_inet.c:661 TPS_SOCKET_TRANS
- net/ipv4/tcp_output.c:374 TPS_TCP_IP
- net/ipv4/udp.c:646 TPS_UDP_IP
- net/ipv4/ip_output.c:240 TPS_IP_NET
- net/ipv4/ip_output.c:224 TPS_NET
- net/core/dev.c:1843, 1849 TPR_NET
- net/ipv4/ip_input.c:293 TPR_NET_IP
- net/ipv4/tcp_ipv4.c:1744 TPR_IP_TCP
- net/ipv4/udp.c:1137 TPR_IP_UDP
- net/ipv4/tcp_input.c:4362, 4398, 4449 TPR_TCP_SOCKET
- net/ipv4/udp.c:1161 TPR_UDP_SOCKET
- net/socket.c:1601 TPR_SOCKET

TPS_SOCKET_TRANS is the same probing point for both, UDP and TCP protocol. At probing point TPR_TCP_SOCKET there are three timestamps implemented, at probing point TPS_NET, we use a function pointer to get the timestamp when the device driver is called.

4.3 Timestamping code

Figure 4.1 shows a sequence diagram of the timestamping code. After the first call, memory (space for 500000 measurement values) and user-space interface (seq_file) is initialized implicitly and calibration is started. The calibration is finished after a certain amount of measured timestamps and is used for elimination of errors in measurement.

The timestamp functions are defined as follows:

- tp_timer_data: direct access to payload
- tp_timer_seq: access to sk_buf structure
- tp_timer: timestamp code (do_gettimeofday)

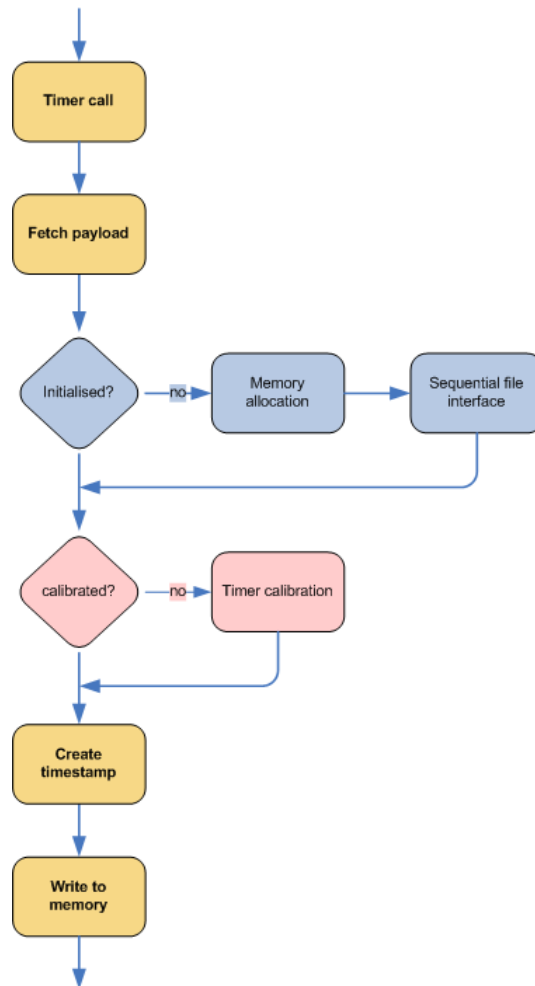


Figure 4.1: Timestamping code sequence diagram

4.4 Errors in measurement

Every measurement has an influence on the results because of its nature that the measurement code itself is taking up CPU cycles. Usually we invoke one, in some cases two (inline) functions at each measurement point. This depends on the direct availability of payload access. In some cases we have to prepare our accessible data (i.e. accessing payload at receiving socket layer when the network stack has not identified any protocol header, yet). This preparation is done in linear time and it only consists of pointer arithmetic (`tp_timer_seq`). In tests we found out that our measurement resolution (1 microsecond) is not good enough for this task. On the other hand, reading the data and counting the received fragments is much more expensive. That is the reason why we only focused on this single function (`tp_timer_data`).

To make it more accurate, we implemented a so-called “timer calibration”. The time that is needed for our measurement code depends on the hardware and software configuration of our testbed. We were seeking an approach that would automatically calibrate our measurement depending on real-world testing data. In our current version, the first 100 function invocations are tracked and the time needed to run the measurement code is taken for a 5% trimmed-mean calculation. Finally, an informative message is sent to the kernel logger so that one knows when to start with real measurement. When calibration is finished, the calibration code is not called anymore. Only the calibration mean value is subtracted from each result.

Here is a shortened example of the kernel logger calibration message:

```
cal.list: 3 3 3 3 3 3 ... 8 8 8 8 8 9 9 12 13 13 22 12738 calibration finished.  
runtime (5% trimmed mean): 4  $\mu$ s
```

`cal.list` is an array of measurement values. Each number represents the runtime of one measurement call in microseconds.

The reason for the long runtime of the outlier (last `cal.list` value) is the implicit initialization (`tp_timer_init`) we are doing on first invocation. By using a trimmed mean, this value is ignored amongst others.

Compared to the small size of the implementation we should keep in mind that the gained accuracy is probably not significant. In our development environment, the measurement code itself takes about 4-6 microseconds per invocation. Unfortunately, these small values have demonstrated their impact, sometimes comparable to the usual noise of our results. One thing we learned from this is that we do not have to pay too much attention to our implementation-related measurement errors.

4.5 Automated measurement

We needed to run the packet generator for many times with different parameters to get the results we desired. So we implemented a script for automated measurement and evaluation of the results. For this task PHP was our first choice. The script uses two computers, one running the server, the other running the packet-generator. SSH is used to run these tools on the specified machines and SCP is used to download the measured timestamps. Figure 4.2 shows the operation of the automated measurement done in our testbed.

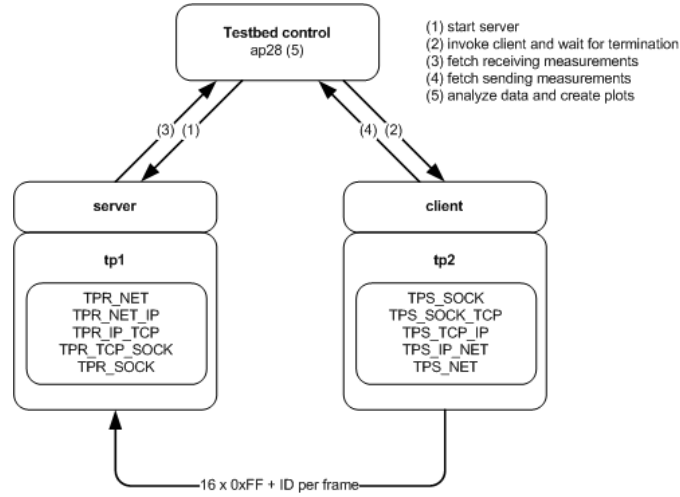


Figure 4.2: Testbed setup

The script is capable of running the same scenario with a range of used packet sizes and threads. For each combination of packet size and thread count the measured timestamps are downloaded and processed into a PHP array by the script. This array is then used to generate plots and average times. The collected average times are used to create plots for packet size or thread ranges. We use gnuplot to generate the plots and as an additional feature even a HTML page is created.

4.6 Evaluation considerations

In order to generate more useful plots further interpretations and well chosen metrics are required. In our case we had to deal with fragmentation. Consider two TCP packets which are merged in the socket layer when received at the server side (shown in Fig. 4.3). As one can see, packet 1 and 2 have timestamps at probes 6 to 9 but they share one timestamp at probe 10, where both packets are merged and delivered to the user-space application.

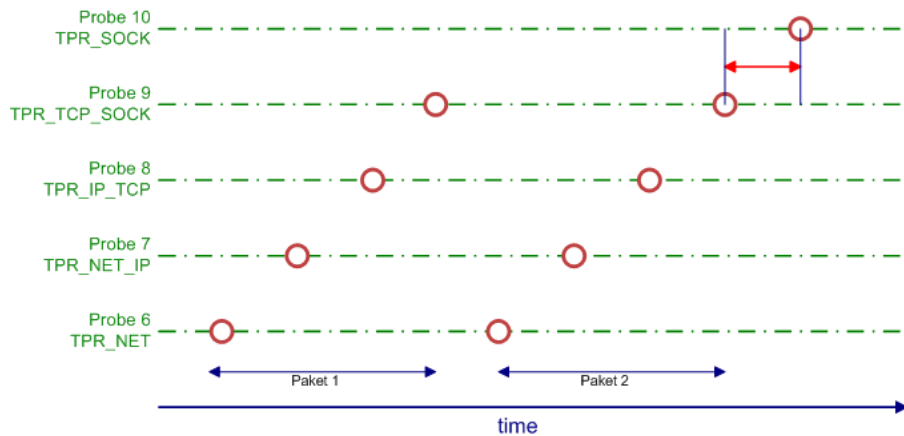


Figure 4.3: First approach - Just measure packet 2

The first approach (Figure 4.3) just measured socket layer time for packet 2. This approach has the obvious problem of lacking a socket layer value for packet 1. A quick solution to this problem is approach 2, which is shown in figure 4.4.

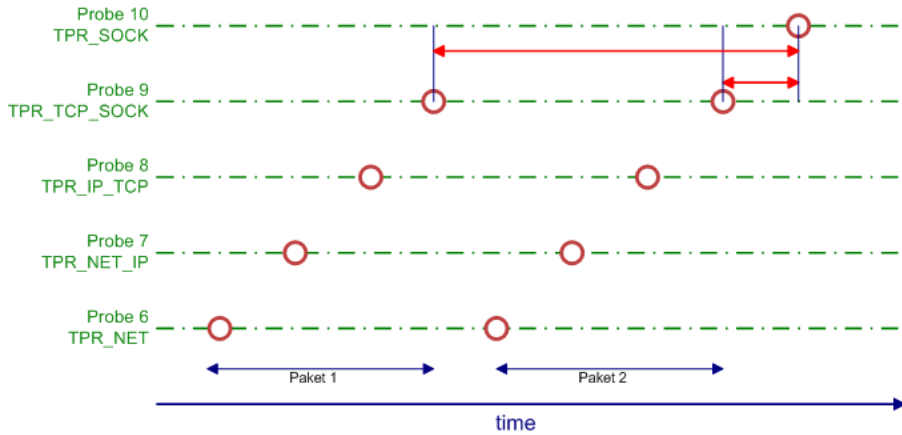


Figure 4.4: Second approach - Measure both packets

Both timestamps at probe 9 are compared to the one timestamp at probe 10 and the difference is the socket layer processing time. This metric measures the socket layer delay, but it has a drawback: Delays of other layers are obviously added to the socket layer delay, which makes the detection of bottleneck difficult. As an solution to this problem we came up with the third approach shown in figure 4.5.

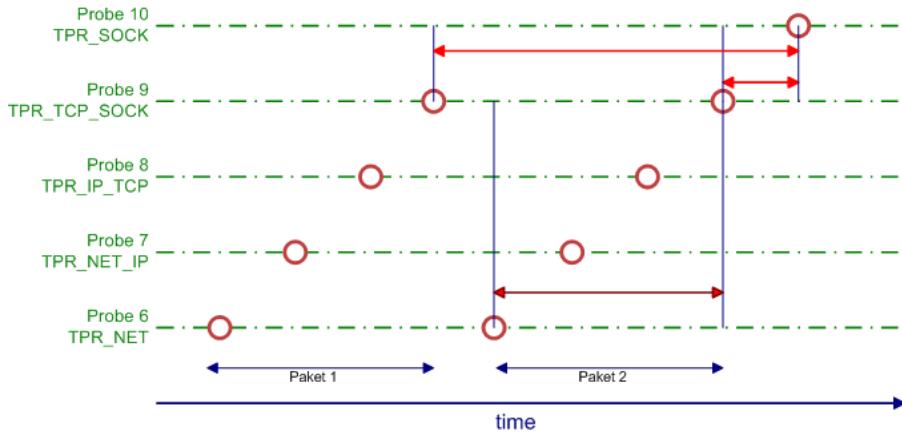


Figure 4.5: Third approach - Subtract non-socket layer time

The idea is to subtract the non-socket layer time between probe 9 and 10 for packet 1. This way we are able to approximate the time the CPU was busy processing the socket layer. All graphs shown in this document use the third approach, although the second one is not less interesting.

Chapter 5

Results

5.1 Increasing the packet size

In this section we want to discuss the plots our measurements produced. We used our automated measurement tools to run the tests and generate the plots. The first measurement - shown in figure 5.1 - shows the impact of increased packet size to the average layer time on the receiver side. The packet size was increased in steps of 5000 bytes. To reduce the impact of outliers (small bursts in measured values) we used a 5 percent trimmed mean for the average layer processing time.

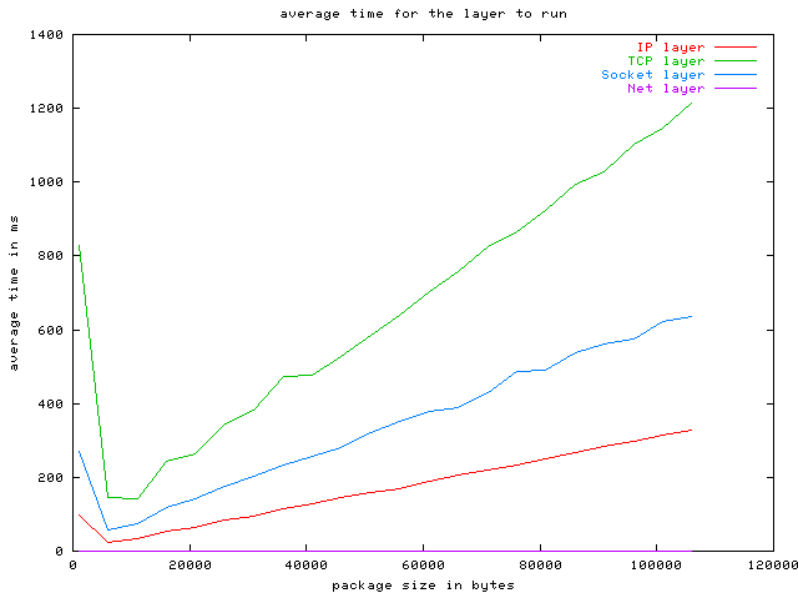


Figure 5.1: 20 threads, average layer processing time on receiver side

This shows the layer processing time increases in a linear fashion. This indicates a good scalability of the network stack in terms of user packet sizes, on the testbed machines. The peak result at 1000 bytes can be ignored, as it turned out to be a small glitch in measurement we later fixed.

We also measured the performance for larger packet sizes than the link MTU. Figure 5.2 shows such a measurement done with packet size of 26000 bytes. The burst at the first packet can be due to internal parameter initialization and has been ignored in the final evaluation.

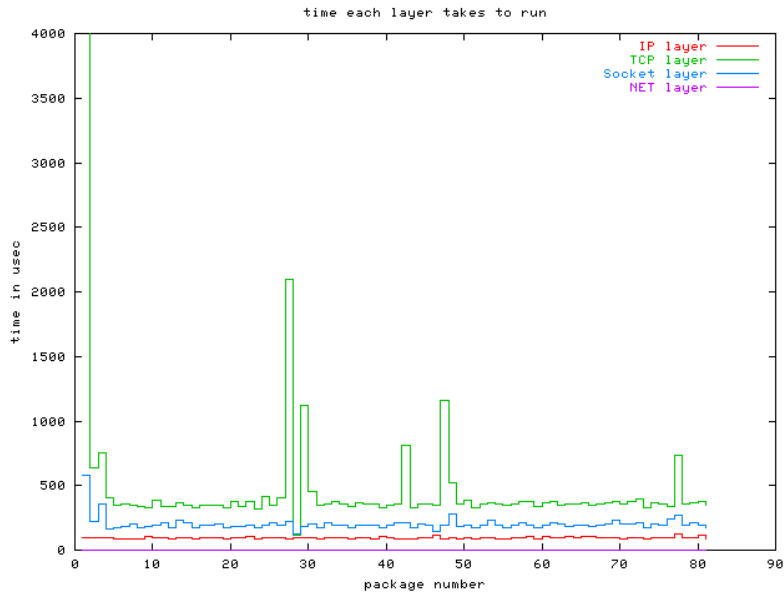


Figure 5.2: 20 threads, 26000 bytes per packet, receiver side

This plot clearly shows that TCP takes a lot more time than IP to compute, which is logical due to the complexity of TCP. The main purpose of IP is routing of packets. We measured endpoints, so no routing had to be applied and therefore the IP layer was not used. The complexity of TCP is discussed in the next section, where TCP is compared to UDP.

We now switch to the sender side and look at the same scenario used above. So we got a thread count of 20 again and increase the packet size in steps of 5000 bytes.

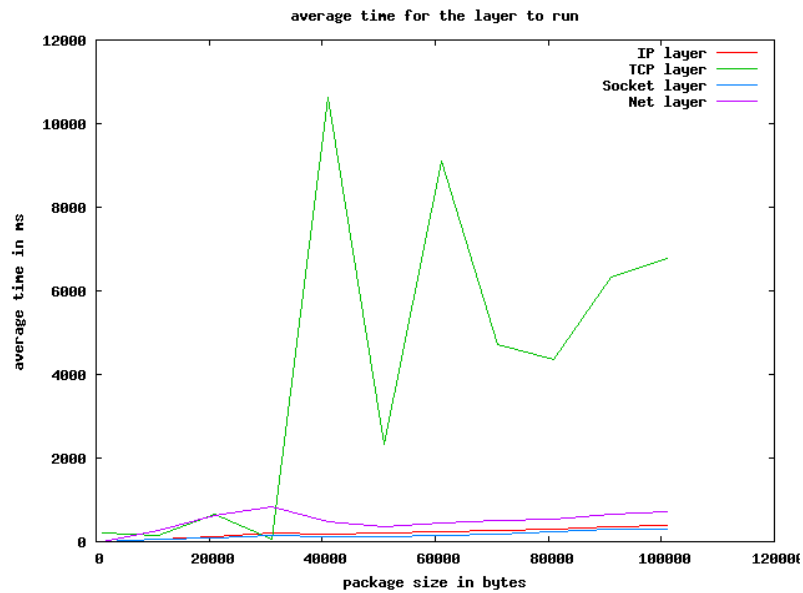


Figure 5.3: 20 threads, average layer processing time on sender side

Looking at figure 5.3 one can see how unpredictable the runtime of TCP on sender side can be.

5.2 Increasing the thread count

In the following measurement we increased the thread count in steps of 50 from 20 to 520. Therefore a constant packet size of 2000 bytes was used. Figure 5.4 again shows the average layer time on the receiver side with a 5 percent trimmed mean.

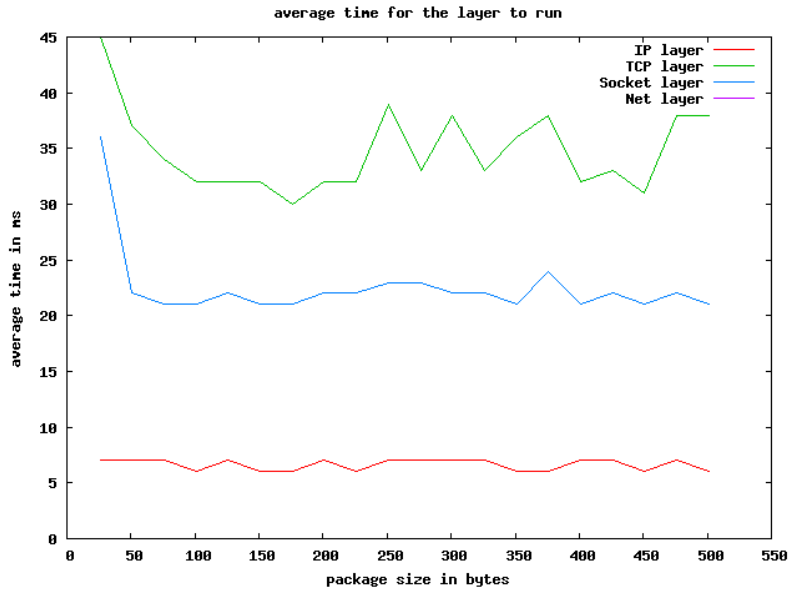


Figure 5.4: 2000 bytes/packet, average layer processing time on receiver side

As we do not measure delays in this plot, but approximate real processing time, the thread count has no big impact on the layer processing time.

5.3 TCP vs. UDP

In this section we want to compare TCP to UDP in terms of processing time. The measurements were done with 1400 bytes per packet, so without any fragmentation, and only one thread was used. First of all we focus on the receiver side.

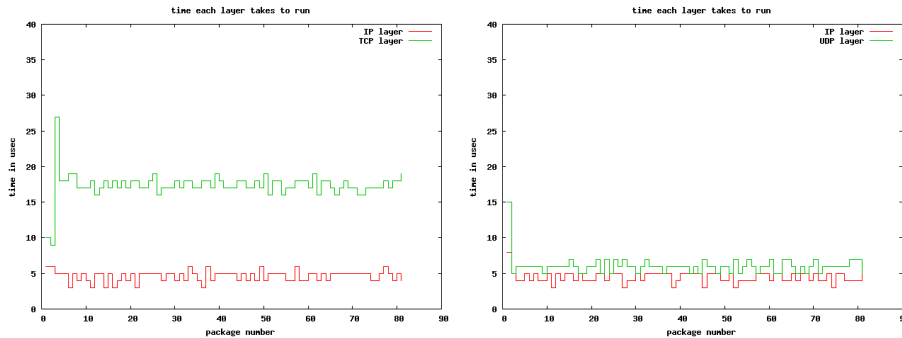


Figure 5.5: Comparison of TCP (left) and UDP (right) processing, receiver side

The result of this test is obvious. UDP is much faster than TCP on the receiver side, approximately 6 μ s vs. 17 μ s on average, respectively.

Looking into UDP source code makes clear why this is the case. UDP processing is not much more than checksum calculation and a lookup routine in the socket-to-port mapping table. TCP has a lot more work to do, such as flow and congestion control and reordering of packets. Our test network was so simple that it is likely that some of these TCP features may not be necessary at all, but the overhead of maintaining states and verifications of variables involved in these TCP features seems to have a great impact on the overall TCP processing time.

In addition to the receiver side, we also compared TCP and UDP on the sender side. The scenario is the same as what we used above. Figure 5.6 shows the different plots. As one can see, UDP is faster on the sender side too, even if the difference is marginal in comparison to the receiver side. The overhead of the TCP features mentioned above is not very significant, which is due to the fact that the TCP features on the sender side only operate noticeably when needed, which was not the case in our test scenario.

Chapter 6

Summary and future work

To summarize, we conclude that the TCP/IP implementation of the Linux Kernel 2.6.9 performed very well, in consideration of the fact that an increase of load had a linear impact on processing time. It should be noted, that this result might be related to the machines we used. The machines were not able to use their 100 MBit/s connections, instead we managed to use only up to 40 MBit/s. This result is likely due to the relatively poor performance of the used Realtek RTL-8139 network cards.

As a result our analysis could be done again on more powerful machines capable of using their connection speed. In order to get more realistic results the connection could be artificially congested. This way TCP's control mechanisms actually take place and its scaling under a more realistic scenario could be tested.

Additionally other implementation and versions of TCP can be tested. Older or future versions of the Linux Kernel along with other operating systems like FreeBSD and NetBSD could be an interesting subject to analyze. Further analysis on UDP and the impact of socket buffer size are topics we initially intended to cover (however was left open due to a time limitation).

Finally, due to the increasing demand to IPv6 and IP security, performance comparison between IPv6 and IPv4, and between IP and IPsec will be very useful. We believe the results and tools developed in this project will be helpful for these purposes with certain extensions.

Acknowledgement

We would like to thank Prof. Dr. Dieter Hogrefe for providing the nice working environment for this project. We sincerely thank the teaching assistants, Sebastian Willert and Ingo Juchem, for sharing their experience and providing effective assistance throughout the project, and Hannes Tschofenig (Siemens) for his comments to the project. The instructor would like to thank Prof. Dr. Henning Schulzrinne (Columbia University) for the discussions which came up with initial project ideas.

Bibliography

- [Pap93] C. Papadopoulos and G. Parulkar,
Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(2): 199-216, Apr 1993.
- [Wel00] H. Welte,
The journey of a packet through the Linux 2.4 network stack, Oct 2000.
URL: <http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html>
- [Kperf] A tool for Linux kernel performance analysis, Apr 2005.
URL: <http://user.informatik.uni-goettingen.de/~kperf>
- [UML] User-mode Linux Kernel (UML).
URL: <http://user-mode-linux.sourceforge.net/>