M.I.T. Laboratory for Computer Science

Request for Comments No. 216
February 16, 1982

On UNIX Network Software Performance or, Who Knows Where The Time Goes...

by Larry W. Allen

Abstract

The Computer Systems and Communications Group has been working on a new implementation of the DoD standard Internet and Transmission Control protocols for the UNIX operating system. This document describes the results of some recent performance testing on this implementation. It includes descriptions of some pitfalls we found in the course of the testing.

1. Introduction

In recent months the Computer Systems and Communications Group has been working on a new implementation of the DoD standard Internet and Transmission Control protocols [3, 4] for UNIX[1] specifically for the PDP-11[2] . One of the goals of the new implementation is to obtain better performance than existing PDP-11 UNIX implementations. To be more specific, a (somewhat arbitrary) goal of 10 milliseconds to turn around an incoming echo packet was specified.

This paper contains a brief description of the network software, a discussion of the experimental procedures used in monitoring its performance, and the conclusions we reached. We experienced a number of difficulties in evaluating the system performance; these are described in detail. Finally, some suggestions are given for further work.

---

[1] UNIX is a trademark of Bell Laboratories.

[2] PDP-11 is a trademark of Digital Equipment Corporation.

2. Overview of the Network Software

This section contains a brief overview of the CSC implementation of Internet and TCP. The network software is divided among a kernel-level packet driver and user-level code which calls on the kernel to send and receive packets.

The UNIX kernel has been characterized as more an I/O multiplexer than a complete operating system [8]. The CSC network software also follows this philosophy. The kernel driver basically functions as a packet switch, multiplexing outgoing packets from user processes to the network and demultiplexing incoming packets to the appropriate recipients. In addition, it performs reassembly of fragmented internet packets. Finally, it contains the driver for the local network hardware. Incoming packets are deposited into kernel buffers from which they must be copied into the user's buffers when the user performs a read system call; outgoing packets are transmitted directly from the user buffers without an intervening copy.

All other work is done in user-level code. This includes filling and validating InterNet headers and all higher-level protocol work, such as packaging and delivering TCP segments. This approach is in contrast to many other TCP/IP implementations in which the TCP is implemented in the kernel [1, 2]; it was chosen because of the limited kernel address space on the PDP-11 and because of a desire to make the kernel code as independent as possible of the particular version of the UNIX kernel being used.

At present, there are user-level implementations of the Trivial File Transfer Protocol (TFTP) [6] and the Internet Control Message (ICMP) [5] and Gateway-to-Gateway (GGP) [7] Protocols. The TFTP implementation is a daemon which performs both server mode and user mode transfers. The ICMP/GGP implementation is also a daemon, handling incoming ICMP and GGP packets of all types, and also performing user-requested echoes and timestamps to other hosts. Most of the performance testing which is described below involved measurements of the ICMP/GGP daemon.

3. UNIX Performance Measurement Facilities

The UNIX operating system provides a number of facilities for measuring and evaluating the performance of applications software [9]. The system provides a 16.6 millisecond (1/60 second) interrupting clock, which may be read by user programs. A record is kept of the total CPU time used by each process; this record is further divided into time spent executing in kernel mode and

time spent executing in user mode. The times may be displayed while a process is in execution; also, it is possible to have the total times for a process displayed when the process exits.

A second and very important facility for performance monitoring supplied by UNIX is that of execution profiling. An *execution profile* is a display of the approximate length of time spent executing in each subroutine in a program. In UNIX, the profile is generated by sampling the process' program counter on each clock tick and producing an execution time histogram; later this histogram is correlated with the monitored program's symbol table to produce the profile. Also, it is possible to cause the C compiler to produce code which counts the number of times each subroutine is called; if this is done, the execution profile will also display the average time per call for each subroutine.

A limitation of execution profiling is that the execution profile only includes the time a process spends running in user mode; time spent in system calls is not included. It is possible to build a version of the UNIX operating system which includes kernel profiling. This is similar to process execution profiling as described above, but provides a histogram of the time spent in the operating system kernel (i.e.. performing system calls, servicing interrupts, and scheduling processes). We have not as yet found it necessary to profile our UNIX kernel, although this was considered for a while as will be described below.

4. Performance of the ICMP/GGP Daemon

The very early testing of our implementation revealed no immediately obvious performance problems, so we did not at first worry about testing the performance of the packet driver. However, when the ICMP/GGP implementation began to become operational, its performance was apparently quite poor. For example, to send a timestamp packet from our machine to itself took between 250 and 300 milliseconds, for a round-trip time of 500 to 600 milliseconds. Responses to echo requests from other hosts also took a minimum of around 250 milliseconds to perform. As a result, we thought it necessary to take a much closer look at the execution times to attempt to determine where the time was being spent.

We began the testing by creating a version of the ICMP/GGP daemon which would run with execution profiling enabled, including the counting of calls to each subroutine. The program was run for several hours with profiling on, repeatedly performing a series of user requests. Because

servicing of user requests exercises substantially different code paths than handling of incoming packets, the test was repeated with no user requests present but with a foreign host repeatedly requesting echoes. These tests were done under a variety of system loads, ranging from an unloaded system up to a normal daily user load (on our PDP-11/45, about six users running mostly the EMACS editor and the C compiler).

The results of the profiling tests raised more questions than they answered. The execution profile indicated that the daemon was spending only about 25 milliseconds of CPU time per packet executing in user mode. We thus had to account for about 225 milliseconds per packet of time spent elsewhere, either running in the kernel or lost to other processes due to context switching.

At this point the UNIX facilities for determining the total CPU times used by a process became important. We discovered that the daemon was spending about 25 milliseconds per packet in user mode (which confirmed the data obtained by the execution profiling), but it was in addition spending between 225 and 300 milliseconds per packet executing in kernel mode! In other words, 90% of the time spent in processing an echo packet was being spent running in the operating system kernel. We feared that this time was being spent in the actual packet driver, processing incoming and outgoing packets.

If this were correct, it would have a very bad effect on the performance of higher-level protocols like TCP. Indeed, with the usual maximum size of TCP packets at 512 bytes, this would mean a maximum achievable transfer rate of around 16,000 bits/second. We knew, however, that there was something wrong with our measuring techniques; for the Trivial File Transfer daemon was at the same time consistently achieving 40,000 bit per second transfer rates using 512 byte packets. We would have to dig deeper into the question of where the kernel time was going.

5. Performance of the Kernel Packet Driver

At this point in the testing, we knew that it was taking around 250 milliseconds to turn around an incoming echo packet, and that of this time, around 25 milliseconds was being spent in user level code with the balance spent executing in the kernel; virtually no time was lost due to process context switching. We had no idea, however, where the kernel time was being spent. For a while, we seriously considered building a version of the kernel which included kernel profiling, despite the fact that this was likely to present us with a mound of data from which it would be very difficult to

extract any relevant information. Fortunately, we were able to obtain more useful performance data without resorting to kernel profiling.

Instead, we wrote a test program which was intended to be run on a completely unloaded system. The program attempted to find the actual elapsed time (not CPU time) required by the network send and receive system calls; this would place an upper bound on the CPU time required and perhaps tell us that the packet switching code was not the culprit. The program was very simple; it just read the 1/60 second system clock, sent a packet (to itself), read the clock, received the packet, and read the clock again; then it looped. The elapsed times were accumulated along with the total number of packets sent; then after sending several tens of thousands of packets the average times were calculated.

The results, while very encouraging, were also somewhat perplexing. The average time to send a packet was about 9 milliseconds; the average time to receive a packet, about 4 milliseconds. These times were quite repeatable; moreover, they did not vary much when the test program was run on a loaded system. In fact, even with four copies of the test program running simultaneously sending packets as fast as possible, the average times were only about twice the above figures. We could conclude, therefore, that the time spent in the packet driver was a negligible proportion of the total time required to process an incoming packet. We were still lost, however, as to where the other 200 milliseconds per packet of kernel time was going.

At this point, fortunately, a brainstorm struck. The ICMP/GGP daemon has fairly extensive logging facilities built in to simplify debugging. These include the ability to log all transactions (both local requests and foreign requests); to log all errors; and to dump all incoming and outgoing packets. Moreover, it is possible to individually control each type of logging. During all of the ICMP/GGP performance testing reported above, logging had been on at the transaction level. This means that for each incoming request received and for each local transaction originated a record was being written to the log file. Also, the log file was unbuffered to insure that all logging information would be preserved in the event of a program crash. It occurred to us that all of this logging might be quite costly in performance.

So we reran the ICMP/GGP performance tests with all logging turned off. We were astonished to find that the round-trip time for a timestamp from our machine to itself was now about 32

milliseconds. Also, the daemon could handle a packet every 10 milliseconds, as verified by running a program on another machine which generated echo requests at that rate. This is as good as or better than any other GGP implementation to which we presently have access. These times have been confirmed by further testing.

6. Conclusions and Further Work

In conclusion, the new network software performs as well as or better than the original specifications. Unfortunately, determining this fact took much more effort than we would have liked. We found a number of pitfalls in our performance testing, most of which involved timing things which shouldn't have been happening anyway. We hope that this recounting of our experiences will help others avoid the same problems.

The time and effort spent in profiling and performance testing was by no means wasted. We discovered a number of areas in which the network software's performance could be improved, and also learned which areas to watch in the future. We believe that we can easily double the performance of the ICMP/GGP daemon.

In the future, we will be working on implementations of TCP and several protocols which use it, like Telnet. We will closely monitor the performance of these protocols to see if the conclusions of this paper are substantiated. We would also like to improve the accuracy of our timing measurements by taking advantage of a higher-resolution clock which has become available to us. We will report on this work in future papers.

References

1. Gurwitz, R. F. VAX-UNIX Networking Support Project Implementation Description. IEN 168, Computer Systems Division, Bolt Beranek and Newman Inc., January, 1981.

2. Lyons, D. The DECSYSTEM-20 TCP/IP User Interface. IEN 176, Digital Equipment Corporation, March, 1981.

3. Postel, J. Internet Protocol. RFC 791, Defense Advanced Research Projects Agency, Information Processing Techniques Office, September, 1981.

4. Postel, J. DoD Standard Transmission Control Protocol. RFC 793, Defense Advanced Research Projects Agency, Information Processing Techniques Office, September, 1981.

5. Postel, J. Internet Control Message Protocol. RFC 792, Defense Advanced Research Projects Agency, Information Processing Techniques Office, September, 1981.

6. Sollins, K. The TFTP Protocol. RFC 782, Defense Advanced Research Projects Agency, Information Processing Techniques Office, June, 1981.

7. Strazisar, V. How To Build A Gateway. IEN 109, Defense Advanced Research Projects Agency, Information Processing Techniques Office, August, 1979.

8. Thompson, K. UNIX Implementation. *Bell System Technical Journal 57*, 6 (July-August 1978), 1931-1946.

9. Thompson, K. and Ritchie, D. M. *UNIX Programmer's Manual.* DSSR edition, 1978.