Toolbox to analyze computer-network interaction

Markus Peuhkuri

Helsinki University of Technology, Laboratory of Telecommunications Technology
P.O. Box 3000 02015 HUT FINLAND

ABSTRACT

Understanding the nature of data communications requires tools to collect data from the computer, the network and their interaction. A tool is needed to get better understanding of the processes generating traffic to the network.

The main components are an instrumented Linux kernel, a synthetic benchmark program, a system call tracer and a set of analysis programs for the post processing. The data collected from the network can be synchronized with the data collected from computer with an adequate accuracy without expensive hardware.

Changes on the operating system (e.g. scheduling algorithm) or on the network can be easily evaluated by the synthetic benchmark where it is possible to modify CPU/IO-intensity ratio and the number of processes each type thus emulating different real-world applications. The data and the code size can be modified to evaluate the memory system performance over different working set sizes.

The early measurements on the Ethernet indicate that this toolbox is useful. Measurements have revealed how network traffic is affected as number of processes changes. The toolbox development continues on ATM environment.

Keywords: Teletraffic characterization, Operating systems, Process scheduling, Traffic measurements

1. INTRODUCTION

Usually traffic in a network is described by Markovian models. While this kind of model is easy to handle mathematically and holds well for traditional telecommunications, recent studies have shown that data traffic does not generally fit well in this model.[1,2] Computer-generated data traffic is very bursty by nature, especially bulk transfers where applications send data as fast as possible. Many CPU-intensive and network I/O-intensive tasks cause the traffic to be even more bursty. When a computer outputs many flows by different processes, a single flow may be very bursty even if the overall traffic is smooth.[3]

Bursty traffic is difficult to handle by the network elements if the same network must simultaneously support real-time traffic with strict delay requirements and small packet loss. The on-off ratio or burst size of traffic can be one measure when determining the cost or the QoS of data transfer so ways to smooth traffic may be economical.[4] To be able to understand how data traffic can be made network friendly, the function of operating system must be understood, specially the process scheduling and the buffering.

Contemporary operating systems are quite complex. Version 2.0.29 of Linux operating system has some 740,000 lines of C and assembler code, including comment lines. One fourth of this is processor dependent. Half of the code is drivers for various hardware (including network devices). The actual operating system and the networking both contribute over 50,000 lines, a bit less than the file systems. Modeling operating system's time-based behavior is difficult task as it is affected both by even more complex applications programs with system calls and hardware with interrupts and DMA transfers. Contemporary applications also handle complex data, which nowadays includes large multimedia components.

This gave motivation to create a tool which makes it possible to gather information from network, operating system and applications simultaneously. Afterwards the data collected from different sources must be combined and analyzed to get the full picture.

In this paper we examine first structure of the toolbox and it's design. Secondly we study the preliminary measurements and the results from them. Finally we judge the toolbox and find areas to develop it further.

Other author information: Email: *Markus.Peuhkuri@hut.fi*; Telephone: +358-9-451 2467; Fax: +358-9-460 224; Project home page: URL:*http://keskus.hut.fi/tutkimus/mitta/*.

Table 1. Stored process information

Variable	Previous	Next
Process identifier (PID)	X	X
CPU usage counter	X	X
Process status	X	
Bending signals		X
Program counter (user space)	X	X

2. STRUCTURE OF THE TOOLBOX

The toolbox has several components which are useful also at they own. As the process scheduling is an important area in our research, we wanted to find information when a process was changed and why. To get this information out, we had to modify the operating system kernel and add our own instrumentation in it.

In addition of scheduling, we also want to log actions of applications. The most interesting are system calls dealing with I/O: `write()`, `read()` and specialized variations for network traffic. This can be done with the system call tracer which has at least two possible implementations. The benchmark program gives us good control over various parameters: it should be configurable to mimic real applications.

Because we are interested on network traffic, we need also tools to dump traffic from network. There are many suitable programs and equipment available. After all the data has been collected, it must be combined and analyzed.

2.1. Instrumented kernel

While evaluating suitable platform to experiment, some factors were essential:

1. Operating system source code available and redistributable. As we wanted the results and the developed environment to be public, no proprietary operating system could be chosen.
2. Must run on an inexpensive hardware.
3. Not restricted to a specific vendor hardware, like network interface cards. As companies are bought by other companies, some hardware may suddenly become unavailable.
4. A good ATM support is preferred.

After all these requirements were evaluated, the Linux was chosen. It is a POSIX-conformant free operating system, running on industry-standard PCs.[5,6] Experimental OS's, such as X-kernel,[7] were dropped because only large user and developer community will keep system with current standards. Ability to run popular real-world applications is important: otherwise one is just simulating, not collecting real data for models. The other popular free operating system family, BSD-derived[8] systems, failed to provide as good support for ATM as Linux.

Few routines was added to the Linux kernel to store information from previously run and next-to-run processes as listed in Table 1. Other information stored includes time stamp, "goodness" of process chosen and lengths of various task queues. The network statistics stored includes number of network buffers in use and different buffer allocations.

Collecting data traces from computer systems always brings problems with storing of the measurement data. Usually the storage is either limited by the size or by the bandwidth. As we wanted to have minimal disturbance to system, the measurement data was stored in central memory as it does not cause disturbing delays or interrupts.

The results can be printed out by reading a specific file from a `/proc` file system. The `/proc` file system provides easy and convent way to deliver data from kernel.[5] The routine prints the results as ASCII text; one line for each record for easy post processing.

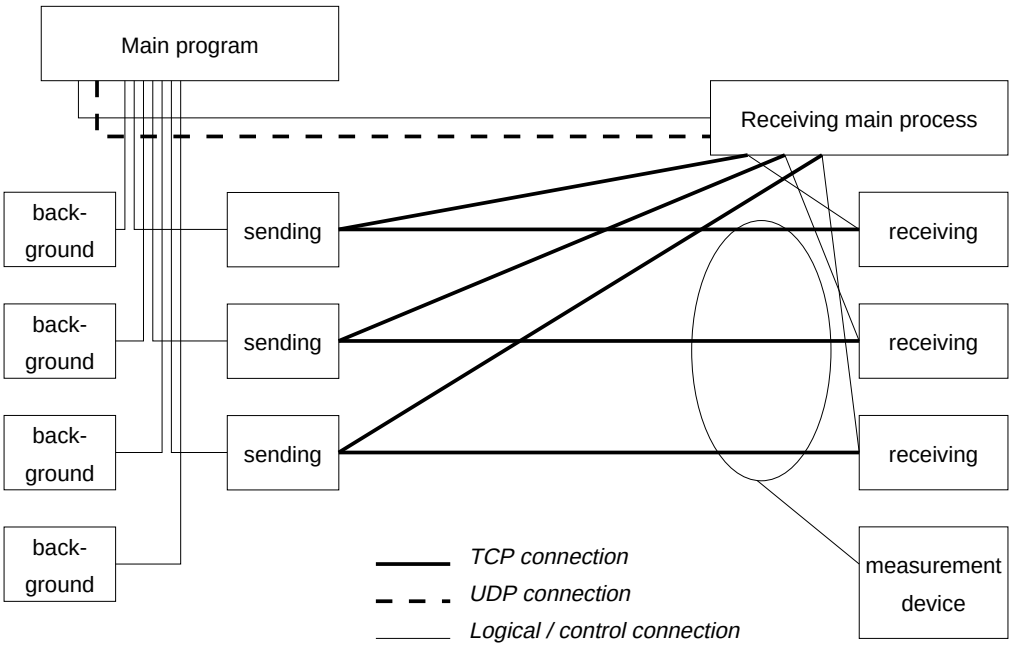


Figure 1. Functional organization of benchmark program.

2.2. Synthetic benchmark

The program for measurements can be divided into five part:

1. Main process.
2. Background processes.
3. Sending processes.
4. Receiving processes.
5. Auxiliary routines.

The program flow is visualized in Figure 1. Execution starts with the main process. It takes care of argument parsing and starts the remote receiving process which reserves listening UDP and TCP ports and sends their addresses to the main process. The main and the remote processes exchange UDP datagrams with time stamps to synchronizes clocks.

The remote process waits for incoming TCP connections. The main process creates the all background and sending processes alternating and then waits them to terminate. As the all child processes have terminated, the main process sends stop character to remote process. The program takes care that all the subprocesses exit properly to find out if there are some problems invalidating measurements. As remote process exits, the main process exits also.

2.2.1. Background process

The background process just takes CPU time by calculating the sieve of Eratosthenes over dataset of 2^N bytes (configurable $1 \leq N \leq 25$: 2 bytes – 32 MB) in repetition. The process logs wall clock time. Both the total number of primes found and the largest one found are compared to the precalculated value to catch possible compiler optimization errors after each round. Different sizes of dataset can be used to evaluate data cache performance. If a code size variations are required, this can be done by making copies of routine.

2.2.2. Sending process

The sending process makes connection to remote, reads data from disk, scans it for line feeds and replaces them by carriage returns and sends data to network. The size for reads and writes is configurable. Time and bytes read or written is recorded before and after each system call making three events for each round. This process thus somewhat emulates typical ASCII mode ftp transfer or WWW server with on-the-fly server-side document processing.

When many processes are sending, each process starts reading from a different position evenly spaced over the file. If a large enough file is used then no process gets data read previously by other process from the disk cache.

2.2.3. Receiving process

The remote main process creates a new process for each incoming connection. Each receiving child process reads data from network, records time stamp and logs it with the number of received bytes. When connection is closed by sending party, data is stored to disk. The data is stored with corrected time stamps.

The sending and receiving computers do not have to be similar as there are routines which automatically convert required information between different byte orders.

2.2.4. Results output

Logging of events is controlled by a initial time what is waited before the logging starts and by the duration of logging. The initial time is used to let processes to get in a steady state. At the end there is configurable quiet time before auxiliary routines store the results to disk.

2.3. System call tracer

There are two methods to trace system calls: modifying the operating system and an external program. The first alternative provides better performance as no additional switch between user space and kernel space (system call) is needed: only one call ($< 2\mu$s) to a routine to store required information at the system call interface. The logging uses the same logging buffer as scheduling trace.

The latter alternative is the only possibility when no source code is available for the operating system. External program traps all operating system calls from the program to instrument. Any operating system which supports `ptrace()` system call or has similar mechanism is usable, there are some ready made tools to trace system calls at least for Linux and SunOS. These generally do not include exact time stamps, so some modifications are needed. The performance penalty comes from a process switch and a needed system call to get a current time. System call overhead is about 10 μs in Linux on 180 MHz Pentium Pro.

2.4. Protocol analyzer

To investigate network traffic it must be captured. Basically any TCP/IP capable recording protocol analyzer will do. If the analyzer can controlled by a remote system, it will help to automate the measurements.

One must pay some attention on a time stamp accuracy of used measurement tool. Many systems give the time stamps for frames with no better than 1 – 10 ms accuracy. For example, the tcpdump[9] suffers from this on many platforms, like on Linux. Some systems might show that they give accurate results but careful inspections shows impossible frame intervals. One way to check for proper time stamps is to measure traffic from a busy network. The frame intervals should be reasonable considering frame lengths.

2.5. Data analysis

At first the data collected from the different sources are combined. As every environment uses different identifiers — process identifiers in two independent computer system and TCP port numbers in network — one must identify which stream belongs to which pair of processes. Also the wall clock time is different in every system. With synthetic benchmark the sending and receiving times are already synchronized. The same synchronization datagrams can be used to synchronize packet capture as benchmark program prints out their send times.

The first analysis program is written with perl,[10] mainly because of easy string manipulation and matching. It corrects all times and outputs then frame time stamps, available window sizes, time stamps of network writes and disk reads. From scheduler trace program prints out time stamps and information about process changes and communications buffer usage.

The data is then ready to be included in analysis programs, like matlab. Basic routines are available to calculate some characteristic numbers and to visualize data.

Table 2. Time of one-character write – read loop.

	Time (μs)
Original 2.0.29 kernel	28.5
Instrumented kernel	31.9
Difference per context switch	1.7

3. PRELIMINARY MEASUREMENTS

By these measurements we wanted to find out how the network traffic is affected if number of sending and background processes is changed. The benchmark program was run with all combinations of 1, 2, 5, 10, 15 and 20 sending and 0, 1, 2, 5, 10 and 20 background processes.

3.1. Environment

Computers used for sending and receiving data were both equipped with 180 MHz Pentium Pro processors, the operating system was an instrumented Linux 2.0.29. Network cards used were SMC/WD 8013 ISA cards. These two computers were connected to a single 10Base2 segment with about 2 meters of cable between them.

Data was captured from network with 170 MHz Sun Ultra 1, the operating system was SunOS 5.5 (Solaris 2.5) and tcpdump[9] version was 3.3. This system was connected via a 10BaseT hub with a transceiver to the 10Base2 segment. There were no other active equipment in the measurement network.

The initial time value used in these measurements was 25 seconds. Processes collected time stamps for 20 seconds. After end of logging each process waited for two seconds to let system calm down before storing log to disk.

Both the receiving and the sending computers were rebooted after each measurement. All measurements were controlled by script and after start, no user interaction was needed before all measurements were completed.

3.2. Sanity check of the results

As the measurement system accuracy was not evaluated and verified by other measurement tools, the results were inspected to find any errors. The network traces were studied to find impossible frame intervals. The minimum interframe time for maximum size Ethernet frames is less then 1.6 ms on 10 Mbit/s. The shortest time measured was over 1.7 ms, which suggest that the time stamps recorded for frames were accurate enough for this measurement.

A possible bottleneck disturbing measurements could be receiving computer's capacity to handle data fast enough. Available TCP window size was all the time in all measurements over 20,000 bytes, so sending computer could send every time it wanted to.

3.2.1. System performance analysis

The effect of the instrumentation on the performance was evaluated by creating two processes, connected with two pipes: one for reading and one for writing.[11] The processes wrote one character and then read one character. The wall clock time for 100,000 loops was measured, values presented in Table 2 are averages from ten runs.

In the first measurements the `schedule()` was called 16.3 – 41.3 times per second on average over measurement interval. If we assume the `schedule()` is called 50 times per second, then the performance lost is less than $1/10,000$. Nearly 6,000 rescheduling per second is required to make this performance loss to one percent.

3.3. Observations

3.3.1. Single process

The most trivial case is when there is just one sending process and no background processes. The traffic generated is very stedy, the most of frame intervals ($> 97\%$) are below 1,9 ms. The minimum frame interval is 1,811 ms and the median value is 1,882. However, every now and then, about eight times a second, the frame interval is about 25 ms. This longer time is caused by operating system which does some administrative tasks.

The system operation is synchronous in this case: as application does write system call, control goes right through TCP and IP layers to network device driver and hardware.

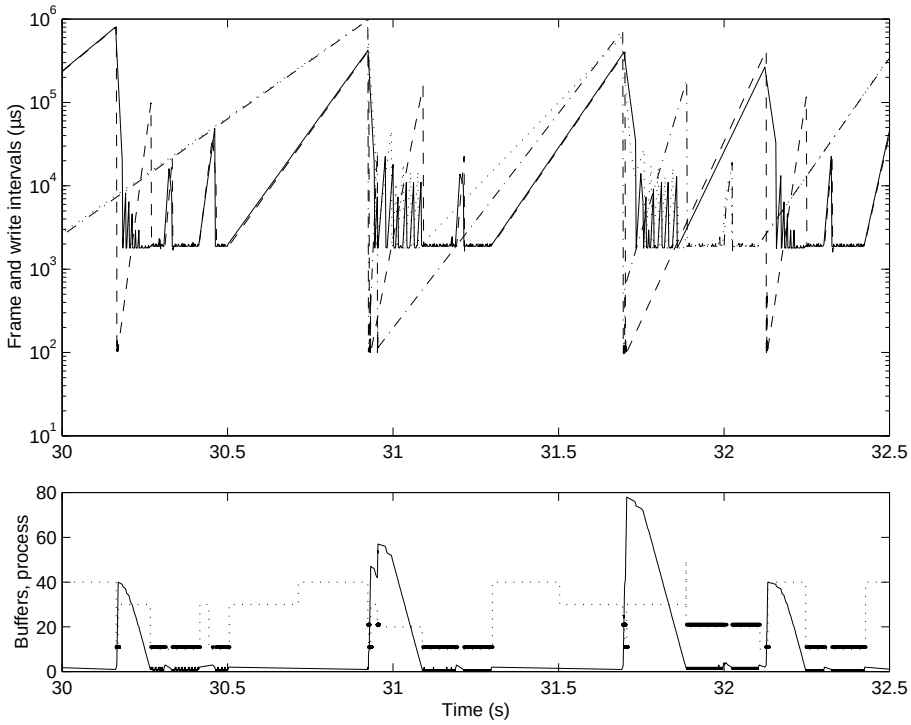


Figure 2. Process trace: two sending and two background processes. See Chapter 3.3.2 for legend.

3.3.2. Several processes

The effects of scheduling and TCP timers can be noted already with two processes (both with one sending and one background process and with two sending processes). The effects can be clearly seen by studying Figure 2. It shows a trace from two sending and two background processes. The frame intervals (solid and dot lines) are mostly steady around 1.9 ms, except when stream has been off for about 0.5 seconds. After the off period the network writes (dash and dash-dot lines) happen very fast with only 0.1 ms intervals, usually 40 times a row filling the network buffers. In normal steady state the write interval is about the same as the frame interval. The number of fast writes can be best seen from a solid line that shows allocated buffers (in a lower part). In the lower part there is also a step line (dash) that shows currently running process. Sending processes are at 10 and 20, background processes are at 30 and 40. A kernel support process is run around 31.85 and is shown as a peak to 50. Writes to network are shown by dots just above process line.

The frame interval distribution of a one stream is very similar to one sending and one background process as with two sending process. This can be seen by comparing a write interval distribution with two sending processes (Figure 3(a)) to one with one sending and one background process (Figure 3(b)). Also the minimum, median and maximum values are very much the same.

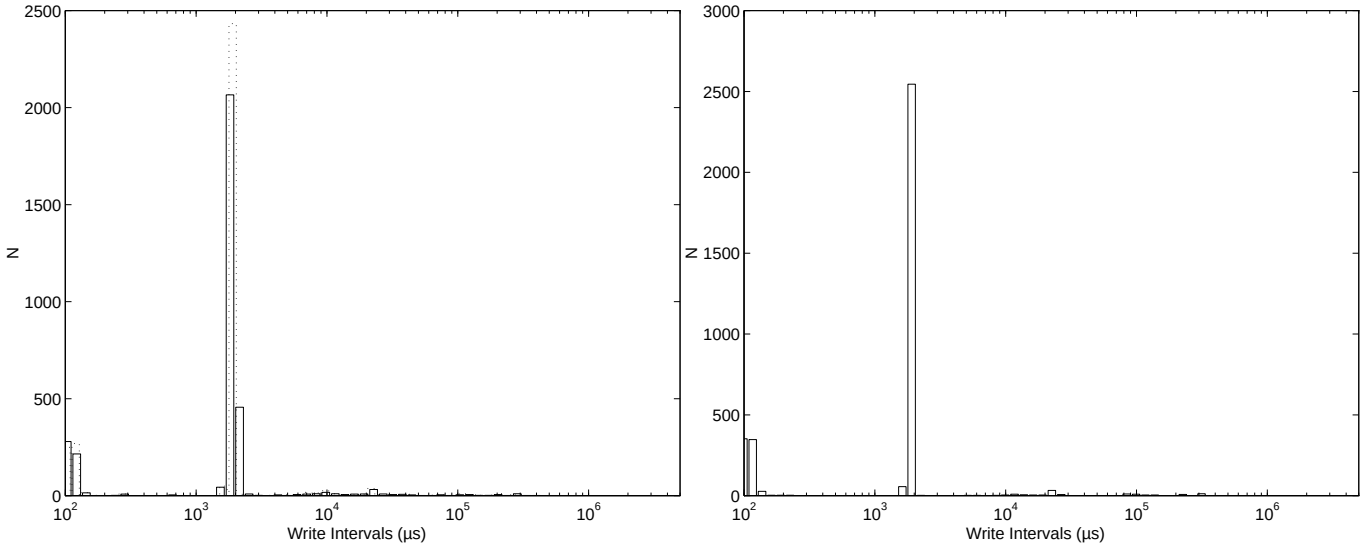
This suggest that with current Linux kernel there is no difference if a process is just using CPU or doing network I/O when one considers its impact on other network I/O processes. Only the total number of processes is what matters.

3.4. Characteristic properties of the generated network traffic

3.4.1. Zero-loss traffic contract

The measured traffic was then converted to ATM traffic. Each frame was converted to corresponding number of cells at speed 10 Mbit/s. The standard dual leaky bucket[12,13] usage parameter control (UPC) was then applied by changing either maximum burst size (MBS) or sustainable cell rate (SCR) until satisfying, minimal traffic contract was found.* Each stream was evaluated individually and an average over all streams was calculated.

*The peak cell rate (PCR) and cell delay variation tolerance (CDVT) were fixed.



(a) Two sending processes, no background processes.

(b) One sending process, One background process.

Figure 3. Write interval distribution.

Observation made based on frame intervals was further strengthened: the required traffic contract was similar with same number of processes. However, with large number of processes this does not hold. For example, with 20 sending processes and one background processes the required burst size is smaller than with inverted configuration. A possible explanation is that with many sending processes the streams interleave and thus reduce the burst size of an individual stream.

3.4.2. Index of dispersion for counts

One measure for burstness of traffic is index of dispersion (for counts) (IDC).[14,15] As defined on Equation 1, it is ratio between the variance and the expected value of number of arrivals (N_t) over interval t.

$$IDC(t) = \frac{Var(N_t)}{E(N_t)}. \tag{1}$$

The IDC is either constant or converges to a fixed value quite rapidly with conventional traffic models such as Poisson or Poisson-like processes and the Markov modulated Poisson processes. For self-similar traffic the IDC is monotonically increasing.[1,2]

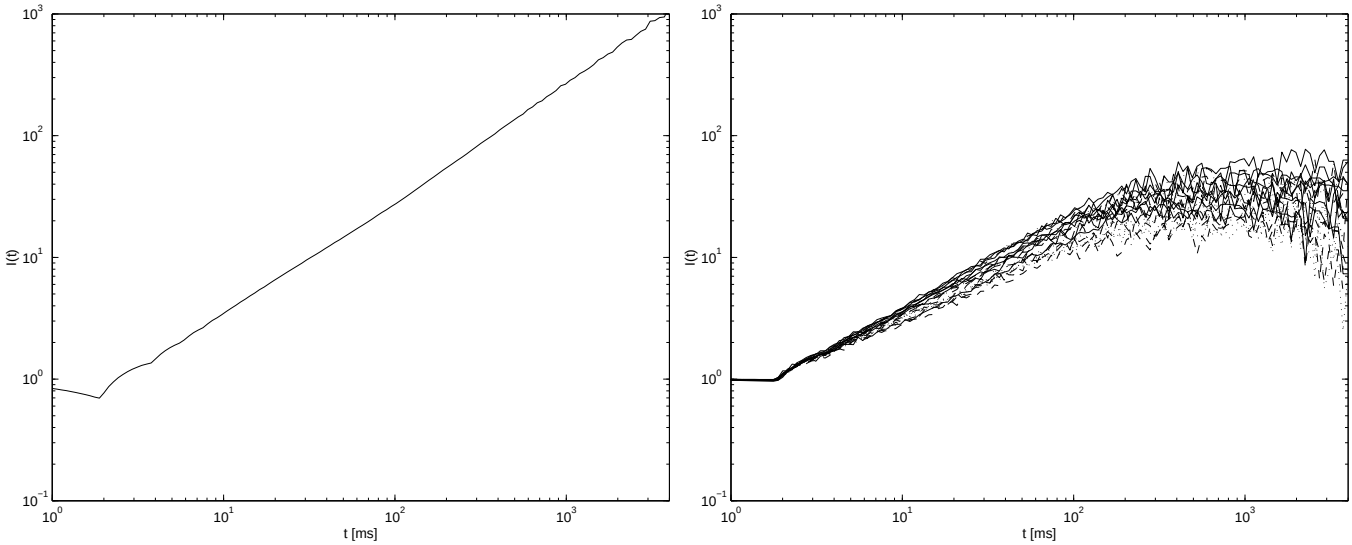
As can be seen from Figure 4(a), in case of one sending process the traffic seems to be self-similar over used time scale. As the number of processes increases (Figure 4(b)), the generated traffic is more like poissonian process. Further studies are needed to find out if this is only because of relatively short measurement period or true nature of traffic.

3.5. Transfer performance

An interesting observation was that transfer performance is strongly correlated[†] with the ratio of sending processes to total number of processes as can be seen from Figure 5. This gives an indication that the Linux network IO is not optimal in this case: even if the network was the bottleneck resource, it was underutilized. However, one cannot make general conclusions based on this single experiment.

From process traces (Figure 2 and similar) one excepts that the TCP Slow Start[16] after 0.5 second off period would cause degeneration on transfer performance. However, the total transfer performance with 20 sending processes was as good as 97.3 % of performance with one process. With first case, the slow start occurred practically every time a process started to send.

[†] Linear correlation 0.99.



(a) One sending process.

(b) 20 sending processes, 20 background processes.

Figure 4. Curves of index of dispersion for counts.

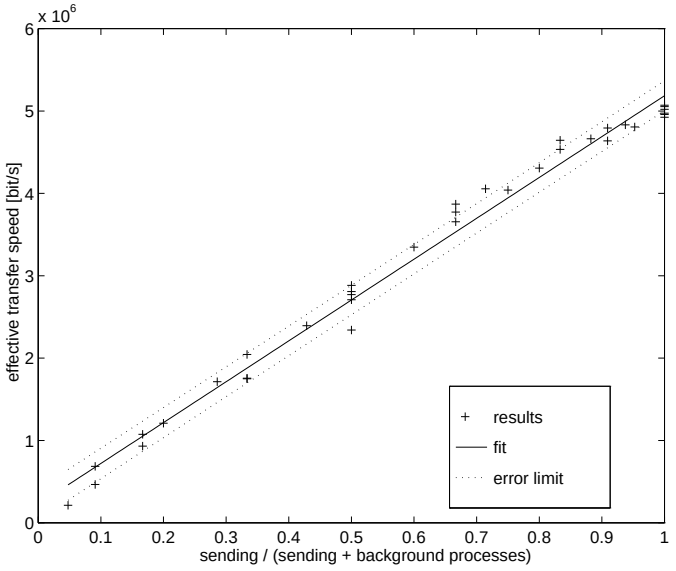


Figure 5. Total goodput compared to ratio of process types.

Table 3. Fairness Index over whole measurement time (20 s)

	Number of sending processes					
	1	2	5	10	15	20
0	1.0000	0.9997	0.9848	0.9648	0.9593	0.9413
1	1.0000	1.0000	0.9951	0.9406	0.9494	0.9076
2	1.0000	0.9997	0.9876	0.9728	0.9130	0.9482
5	1.0000	0.9996	0.9842	0.9751	0.9080	0.8862
10	1.0000	0.9971	0.9913	0.9524	0.8893	0.8887
20	1.0000	0.9919	0.9787	0.9060	0.8864	0.8700

3.5.1. Fairness

The fairness between streams was calculated with Equation 2 which applies well because all streams were equal.[17] From Table 3 we can see that the all streams get about the same service. The more there are processes the lower the fairness index is. This can be explained with short measurement interval: with 20 sending and 20 background processes the longest interframe time for one stream was 8.9 seconds, nearly half of the measurement interval.

$$\text{Fairness Index} = \left(\sum x_i\right)^2 / \left(n \times \sum x_i^2\right) \tag{2}$$

4. CONCLUSIONS AND FUTURE DEVELOPMENT

An toolbox was developed to study interaction between operating system and network. A preliminary measurement was made and results analyzed.

The developed toolbox has helped us to understand interaction between network and operating system. Longer test runs must be used to find more about the true nature of traffic.

The first measurements were made with the synthetic benchmark before the system call tracer was fully functional. The developed system call tracer must be verified with similar measurements. Currently, the code size for the synthetic benchmark can be changed only manually by recompiling, so automation for it may be required if code size dependent measurements are frequent.

The analysis program must be adapted for other protocol analyzers to measure traffic on ATM networks. Thanks to perl's string functions, it shouldn't be a difficult task. One can use benchmark synchronization method to synchronize systems in real-world applications measurements also.

As new Linux kernel versions come out, the instrumentation must be adapted again. This is not very difficult task if no major changes to kernel organization are made. Applying the changes made for the 2.0.25 kernel to the 2.0.29 kernel was very straightforward.

Modeling the real-world applications is important to improve the synthetic benchmark. This requires detailed analysis of applications programs, a good tool for this is ATOM[18] running on the Alpha AXP under OSF/1. The used network was low latency, zero loss[‡] network: it would be interesting to study traffic over longer networks with packet loss and it's effect on scheduling.

While our research project continues, the development of the toolbox will continue also as it is by now proven its usefulness. The patches for the operating system, source code for the benchmark and analysis programs will be made available at project home page.

ACKNOWLEDGEMENTS

This work was supported by Finnish Academy of Research under contract 34097 (MITTA).

[‡] No TCP retransmissions occured, so network was zero loss on IP level, collisions did happen on Ethernet level.

REFERENCES

1. W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic," *IEEE/ACM Transactions on Networking* **2**, 1 1994.
2. A. Vidcs, "Self-similar traffic modeling techniques in atm networks," master's thesis, Technical University of Budapest, May 1996.
3. E. C. Lin, *The Effects of Scheduling Contention on Workstation Traffic*. Licentiate thesis, Kungliga Tekniska Hgskolan, May 1996.
4. K. Kilkki, "Simple integrated media access (sima)," Internet-Draft 00, Nokia Research Center, March 1997. Expire in 20th September 1997.
5. M. Beck, H. Bhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, Addison-Wesley Publishing Company, Inc., 1996.
6. W. Almesberger, "High-speed atm networking on low-end computer systems," DI-EPFL Technical Report 95/147, Laboratoire de Rseaux de Communication (LRC), EPFL, CH-1015 Lausanne, Switzerland, August 1995.
7. L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach*, Morgan Kaufman Publishers, Inc., 1996.
8. M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing Company, Inc., 1996.
9. V. Jacobson, C. Leres, and S. McCanne, *tcpdump – dump traffic on a network*, Jun 1994. A Program Manual Page.
10. L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl*, O'Reily & Associates, Inc, 2nd ed., September 1996.
11. L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 279–294, January 22-26 1996.
12. ITU-T, "B-isdn atm layer cell transfer performance," ITU-T Recommendation I.356, International Telecommunication Union, 1993.
13. ITU-T, "Traffic control and congestion control in b-isdn," ITU-T Recommendation I.371, International Telecommunication Union, 1994.
14. H. Heffes and D. M. Lucantoni, "A markov modulated characterization of paketized voice and data traffic and related statistical multiplexer performance," *IEEE Journal of Selected Areas in Communications* **SAC-4**, pp. pp. 856 – 868, Sept 1986.
15. J. Roberts, U. Mocci, and J. Virtamo, eds., *Broadband network teletraffic: performance evaluation and design of bradband multiservice networks; final report of action COST 242*, Springer, 1996.
16. V. Jacobson, "Congestion avoidance and control," in *Proceedings of the ACM SIGCOMM Conference*, pp. 314–329, August 1988.
17. R. Goyal, R. Jain, S. Kalyanaraman, S. Fahmy, and S.-C. Kim, "Performance of tcp over ubr+," Tech. Rep. 96-1269, ATM Forum, October 1996.
18. A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," Research Report 94.2, Digital Western Research Laboratory, March 1994.

Contents

1 INTRODUCTION **1**

2 STRUCTURE OF THE TOOLBOX **2**

2.1 Instrumented kernel . . . 2

2.2 Synthetic benchmark . . . 3

2.2.1 Background process . . . 3

2.2.2 Sending process . . . 4

2.2.3 Receiving process . . . 4

2.2.4 Results output . . . 4

2.3 System call tracer . . . 4

2.4 Protocol analyzer . . . 4

2.5 Data analysis . . . 4

3 PRELIMINARY MEASUREMENTS **5**

3.1 Environment . . . 5

3.2 Sanity check of the results . . . 5

3.2.1 System performance analysis . . . 5

3.3 Observations . . . 5

3.3.1 Single process . . . 5

3.3.2 Several processes . . . 6

3.4 Characteristic properties of the generated network traffic . . . 6

3.4.1 Zero-loss traffic contract . . . 6

3.4.2 Index of dispersion for counts . . . 7

3.5 Transfer performance . . . 7

3.5.1 Fairness . . . 9

4 CONCLUSIONS AND FUTURE DEVELOPMENT **9**