An Embedded Network Simulator to Support Network Protocols' Development

Luigi Rizzo*.

Dip. di Ingegneria dell'Informazione, Università di Pisa
via Diotisalvi 2 – 56126 Pisa (Italy) – email: l.rizzo@iet.unipi.it

Abstract. The development of network protocols, especially if designed for use in very large scale networks, generally requires extensive simulation and tests in operational environments to assess their performance and correctness. Both approaches have limitations: simulation because of possible lack of accuracy in modeling the system (and, especially, traffic generators), tests in operating networks because of the difficulty of setting up and controlling the experimental testbed.
In this paper we propose to embed network simulators in operational systems, so as to get the advantages of both simulators and real testbeds. Such simulators can be built with minimal modifications to existing protocol stacks. They work by intercepting communications of the protocol layer under test and simulating the effects of finite queues, bandwidth limitations, communication delays, noisy channels. As a result, experiments can be run on a standalone system, while simulating arbitrarily complex networks. Thanks to the ability of using real traffic generators and protocol implementations, doing experiments becomes as simple as running the desired set of applications on a workstation.
An implementation of such a simulator, targeted to TCP and compatible with BSD-derived systems, is available from the author.
Keywords: Protocol evaluation, TCP/IP, simulation

1 Introduction

Network protocols, especially if designed to be used on very large scale systems such as the Internet, require careful analysis, both in design and implementation, to ensure that they can work properly even in unusual operating conditions. Almost unavoidably, theoretical analysis must be accompanied or followed by simulations and tests in operational systems to evaluate the actual performance.

Experimental testbeds, when available, are extremely useful because they allow testing real implementations of the protocol, and with real applications used as traffic generators. This way, no effort is required in modeling any part of the system under test, increasing the confidence in the tests' results. On the other hand, often testbeds are hard to setup with the desired features, because of cost or unavailability of suitable hardware/software. Also, experiments done

* Paper presented at the 9^{th} International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, St. Malo, France, June 1997, LNCS-1245 pp.97-107, Springer Verlag

in real testbeds might suffer from the lack of adequate control over operating conditions (queue sizes, delays, external traffic sources).

Simulations have complementary properties. They make it possible to overcome the lack of a testbed with the desired features, at the expense of a greater effort in modeling the whole system under test. As an example of the difficulties in building an accurate simulator, consider a simple FTP transfer over TCP. The flow of data is regulated by a number of factors, such as the speed of disks at the sender and the receiver side, the scheduling of processes at the two nodes, the size of send and receive windows, the acknowledgement generation policy. The latter, in turn, depend on the behaviour of different protocol layers, or even on the result of previous communication.

In many cases, building an accurate model of a system is an extremely challenging task, and the unavoidable simplifications that are introduced might possibly result in inaccurate or unreliable results. Also, simulated environments might not be available to debug and test the final implementation of a protocol, when it is easy to introduce subtle implementation bugs. Nevertheless, the difficulties of setting up a real testbed with the desired features has stimulated the development of a number of network simulators, such as REAL [1], Netsim [2, 3] and `ns` [4]. The *x*-kernel framework [5] has also been used for the implementation and testing of network protocols.

Experiments on network protocols are usually aimed to determine protocols' behaviour in complex networks made of many nodes, routers and links, with different queueing policies, queue sizes, bandwidths, propagation delays. The problems discussed above in building and/or modeling complex environments, and, especially, in interpreting experimental results obtained in such settings, often suggests the use of simplified networks such as the ones shown in Figure 1. There, a bottleneck router followed by a link with given bandwidth and delay models the overall features of the network, and additional nodes simply generate background traffic on the network.

Quite often, in experiments on real testbeds, the bottleneck router is also modified to act as a "flakeway", introducing artificial delays, random packet losses and reordering. In some cases, the effects of bandwidth limitations can be simulated [6].

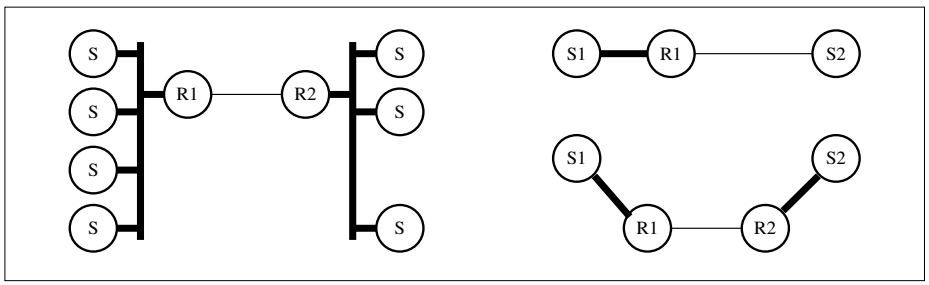

Fig. 1. Typical settings used in the study of network protocols. The thin line represents the bottleneck link.

In this paper we extend the concept of a flakeway in order to build a testbed which gives the advantages of both simulation and real-world testing. Our proposal consists in embedding a flexible network simulator into an operational protocol stack. Under normal operating conditions, the simulator is disabled and introduces a negligible overhead in system's operations. When running experiments, the simulator can be configured to simulate arbitrary networks with the desired features. Hence, experiments can be run on a standalone system, without the need of a real testbed. Yet, real traffic generators and protocol implementations can be used to run experiments. This allows the tests to cover the final implementation as well, something that is not generally possible by using simulators. Also, the researcher has full control over the testbed, which makes experimental results easier to understand.

In the next section we show the principle of operation of our simulator, showing the way to simulate its basic components (routers and links). Section 3 shows how arbitrary networks can be built with a proper composition of the basic components. Finally, we present some examples showing the ease of use and the little intrusivity of the simulator, and illustrate some possible applications.

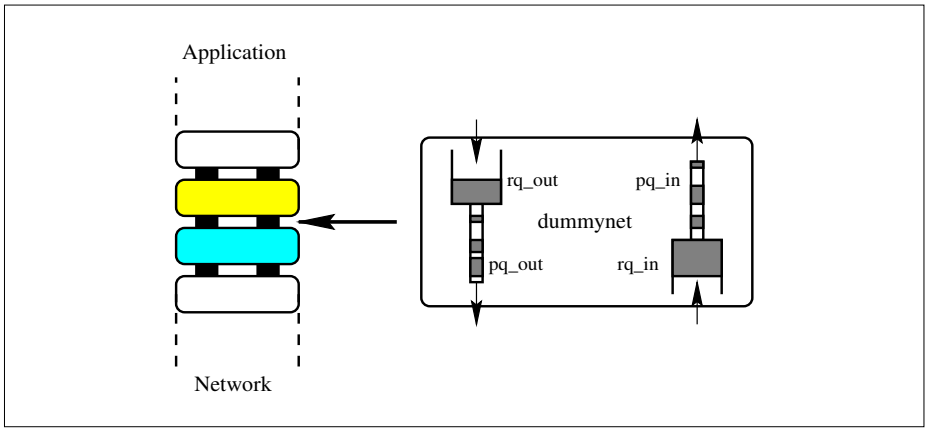

Fig. 2. The principle of operation of our simulator

2 Principle of Operation

In a typical protocol stack, each layer communicates with the adjacent ones (Figure 2), where the upper layer is generally on the path to one of the communicating peers, and the lower layer leads to the other peer via "the network". The latter is made of two types of components, namely *routers* with bounded queue size and a given queueing policy, and communication links (*pipes*) with given bandwidth, delay and loss rate. A network is an arbitrary graph of routers

and pipes, and the presence of multiple paths between two nodes can lead to out-of-order delivery of packets.

The simplest topologies used in experiments usually include just two routers and one pipe. Both elements can be easily modeled (see Figure 2) by two pairs of queues, `rq` and `pq`, inserted at some point in the protocol stack (typically, below the transport layer). Let k be the maximum size of `rq`, B and t_p the bandwidth and propagation delay of the pipe, respectively. Traffic exchanged between the two layers is then subject to the following processing:

1. packets are first inserted in `rq`; insertions are bounded by the maximum queue size, k, and are performed according to the queueing policy of choice (usually FIFO with tail-drop, but other policies are also possible, such as RED [7]).
2. packets are moved from `rq` to `pq` at a maximum rate of B bytes per second. `pq` uses a FIFO policy;
3. packets remain in `pq` for t_p seconds, after which they are dequeued and sent to the next protocol layer. Random losses can be introduced at this stage, by dropping packets according to the loss rate instead of delivering them to the next stage.

Steps 2 and 3 can be performed by running a periodic task whose period T is a suitable submultiple of t_p. In this case, at each run at most BT bytes are moved from `rq` to `pq`, while packets remain in `pq` for t_p/T cycles.

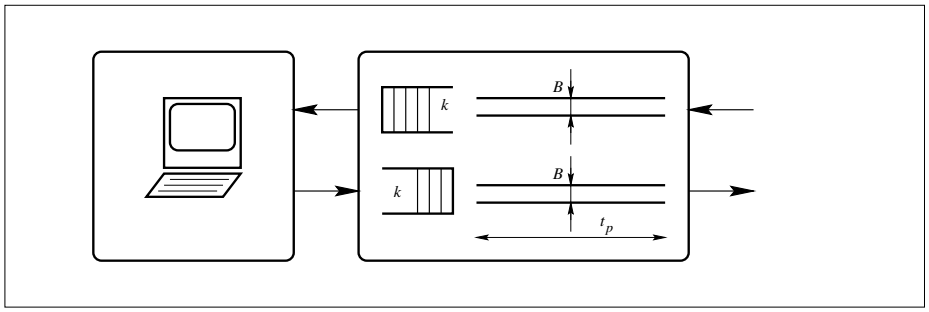

Fig. 3. Structure of a node using the basic simulator

The basic component of our simulator comprises two pairs `rq`/`pq` and is inserted between protocol layers as shown in Figure 2. Since most systems implement a loopback at the bottom of the stack, local communication (through the loopback) is also subject to the queueing and delay introduced by the simulated network, and the resulting system looks like the one in Figure 3. The presence of the loopback is what lets experiments to be run on standalone systems, without the need for a real network. By using such a simple setting, we can simulate most of the settings used in the literature on TCP congestion control [5, 8, 9, 10, 11, 12].

3 Simulating Arbitrary Networks

More complex structures, involving multiple queues and links, can also be simulated, even when the simulator is running on a single node. To this purpose, a model of the system must be defined in terms of queues and links. On each router, and possibly on each simulated node, queues of bounded size are associated to the output links (or, depending on the buffer-management policy used in the router, a single queue may be used for all output links). Unidirectional pipes with given bandwidth and delay are used to simulate communication links: two pipes are used for full-duplex links (e.g. a point-to-point connection), while a single one suffices for half-duplex, shared links such as an Ethernet LAN (in this case, we implicitly assume that all nodes transmitting to the shared medium queue their data into an additional, shared queue). Routing tables must also be defined, so that traffic can be forwarded through the appropriate paths depending on the source and destination addresses. The system where the experiment is run can be assigned multiple addresses in order to simulate a complete network on a standalone system, or it can use a single address and simply simulate the effect of different paths to different destinations.

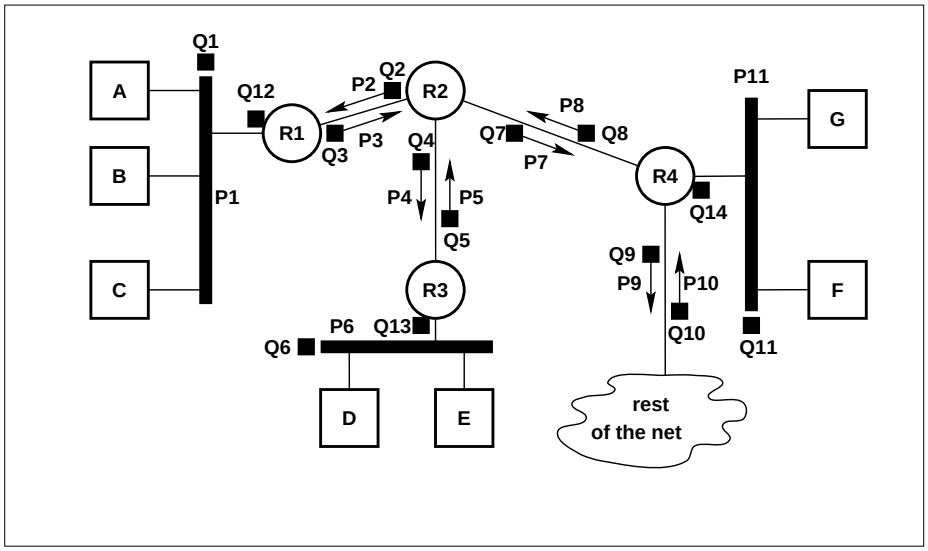

Fig. 4. A sample network and the corresponding structure of the modules.

Figure 4 presents a sample topology, with three Ethernet networks (between A, B, C and R1; D, E and R3; F, G and R4), and four point-to-point links. In order to run a standalone simulation, the system is assigned seven addresses, corresponding to the simulated end-nodes A..G. A total of 11 unidirectional pipes is used to represent the various links and Ethernets. Queues are assigned to the outputs of each router (and to the link coming from the rest of the network,

P10). Q1, Q6 and Q11 are the queues associated with the three Ethernets in the system, whereas we have not provided queues for the end nodes. Finally, a routing table is associated to the output end of each pipe to determine the flow of packets, depending on source and destination addresses, and the direction of flow in the protocol stack (e.g. from TCP to IP or from IP to TCP).

Traffic is routed in the following way. Packets generated and received by "internal" nodes (A..G) are subject to queueing and delay on one direction only (e.g. going from TCP to IP). Once they have reached the destination queue, they are forwarded to the lower protocol layer. From here, through the loopback interface, they move upwards and straight to the final destination. Packets originating from or directed to an external destination (the cloud labeled "rest of the net") are subject to queueing and delays in the inbound and outbound path, respectively.

4 Limitations

Our simulation technique produces, of course, only an approximate model of a real network with given features. Most of the approximations introduced by our technique derive from the granularity, T, and the precision of the operating system's timer, and in many cases they have little influence on the experiments.

T sets the resolution of all timing-related measurements. On modern systems, a granularity of 1 ms or even smaller is easy to achieve, and is suitable for the vast majority of networks except, perhaps, those with very high bandwidths and short pipes.

On a non real-time system there is no guarantee that deadlines are honored; thus, depending on the overall system's load, the periodic task might be run late, or even miss one or more ticks. In our experiments, however, these events have been extremely rare even on a relatively slow system running FreeBSD, which is not a real-time OS. Besides, the same errors affect all protocol timers, which are driven by the same clock interrupt.

Finally, it should be noted that network-related events occur synchronously with the system's timer. This might hide or amplify some real-world phenomena which occur because of race conditions. Such a problem can only be of some concern in very special situations.

The accuracy of the simulation also depends on the correct computation of packet sizes (needed in the computation of the simulated bandwidth). Our technique makes it possible to account for link-layer "overheads", including the effect of link-layer compression, a feature which makes the link appear as a variable-bandwidth channel, and might have significant effects on performance.

5 A Sample Implementation

We have developed a basic simulator [13] using the technique described in this paper, working at the interface between TCP and IP; it intercepts calls to `ip_output()` made by TCP modules, and those to `tcp_input()` made by the

protocol demultiplexer in IP. All the basic functionalities required to build arbitrary networks are included in less than 300 lines of kernel code.

When disabled (i.e. no buffering, delay or bandwidth limitation is required), the overhead introduced by our tool corresponds to one function call per packet. This is a negligible overhead, and allows one to have the simulator compiled into the kernel at all times. When enabled, the overhead is directly proportional to the work required to route packets through the simulated network: all queue manipulations are in fact constant time operations, and no copies of data are done at all. As it is shown from Figure 6, even when the simulator is enabled the available communication bandwidth still remains much larger than the typical Ethernet bandwidth, so that no sensible performance degradation is perceived.

There are only two practical limitations in running a simulation: the CPU power required by all producers/consumers to generate the required traffic, and the memory required to buffer all packets in transit (both in queues and pipes).

Since the simulator only intercepts calls between selected protocol layers, other traffic is left unmodified. As an example, our implementation does not interfere with UDP traffic, allowing a system to mount disks using NFS over UDP, yet leaving a clean simulation environment with only TCP traffic.

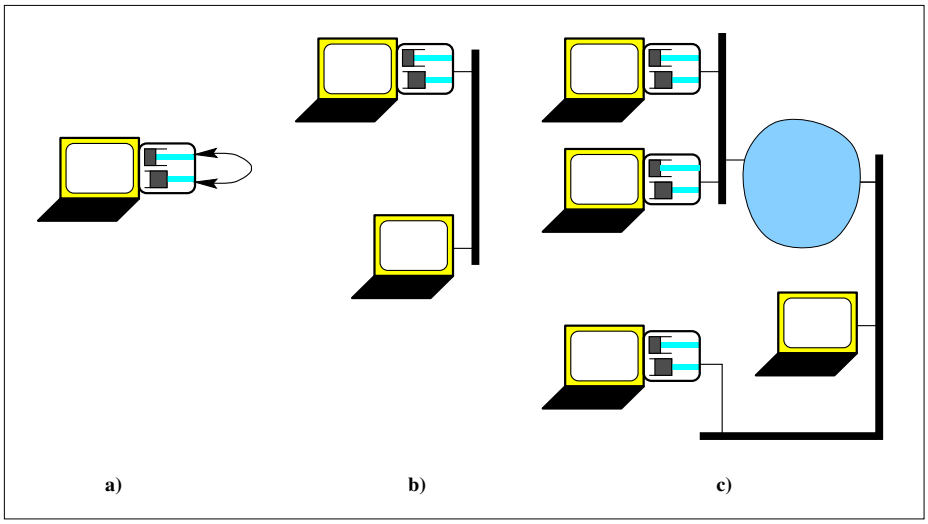

Fig. 5. Various configurations used for experiments

5.1 Examples of use

Depending on the complexity of the simulated network, different techniques can be used to configure the simulator. For the simple case shown in this Section, comprising one pipe and one router, the operating parameters (k, B, t_p) are set

using a single kernel variable (`net.inet.tcp.dummynet`). The `sysctl` command allows an easy setting of the parameters. The value of `net.inet.tcp.dummynet` is given as the decimal number `BBBBkkddd`, where `BBBB` is the bandwidth in KB/s, `kk` is the queue size, `ddd` is the value of t_p in units of T seconds (1 ms in our case). For more complex setups, such as those shown in Section 3, a more flexible setup utility is required, to define the structure of the simulated network and set up the routing tables.

The simplest way of doing an experiment consists in running a communication between two processes on the same system. Since the loopback occurs at the end of the pipe (Figure 5a), buffering and delays occur twice, and buffers are shared by traffic in the two directions.

An example of use of this setting is shown in Figure 6, where some FTP transfers are done using `ncftp`; for each configuration (except for the last three), we show the average throughput value of 10 tests, to compensate for the variations deriving from concurrent network and CPU activities. Lines beginning with `---` are normally part of the system's logfile. The system used for the experiments is a Pentium100 with 32MB RAM, running FreeBSD 2.1. During the experiments, both client and server were running on the same system (the author's workstation), together with the usual workload consisting of an X Server, a number of X applications, a Web server and various other applications.

TCP communications use a 16KB window in this example, so in some cases the throughput is limited by the window size rather than the available bandwidth. The MSS for the interface is set to a low value, which limits performance further but allows a larger number of packets to fit in the window in use.

In the first test, bandwidth and queue limits are set to a large value in order to determine the maximum throughput. The second experiment limits the bandwidth to 200KB/s, but the actual throughput is lower because the channel is shared by data and ACKs, and the TCP header (including RFC1323 and RFC1644 extensions) consumes a portion of the bandwidth. In the third experiment a short propagation delay is introduced, which has negligible effect on the throughput. Increasing the delay to 50 ms (making the RTT 200 ms) causes the connection to be limited by the window size (roughly one window per RTT or 80 KB/s, with various overheads and the cost of slow start reducing the throughput even further). The next two experiments are run with very limited queue sizes: here, frequent overflows occur which reduce the throughput significantly. In the last run, Selective Acknowledgments are enabled.

In single-system experiments, both communication peers usually run the same implementation of a protocol (unless the system allows the protocol parameters to be set individually for each process). Interoperability tests can be done by using two nodes on the same LAN, with the simulator running on one of them (Figure 5b). This resembles the typical setting for protocol evaluation in real networks, consisting in two nodes on different LANs connected by one or two routers and a bottleneck link. Finally, more complex simulation settings can be built by using several systems, some of which use the simulator configured with different parameters (Figure 5c).

One would expect that the use of our simulation technique – especially when working on a single workstation – leads to completely deterministic and repro-

```
prova# ifconfig lo0 127.0.0.1 mtu 576 # small packets --> large windows
prova# ncftp -u localhost
...
ncftp> !sysctl -w net.inet.tcp.dummynet=999900000
--- 0 ms, 9999 KB/s, 0 buffers
ncftp> get 1M a
a: 1048576 bytes received in 0.66 seconds, 1552.17 K/s.

ncftp> !sysctl -w net.inet.tcp.dummynet=20000000
--- 0 ms, 200 KB/s, 0 buffers
ncftp> get 1M a
a: 1048576 bytes received in 6.17 seconds, 166.10 K/s.

ncftp> !sysctl -w net.inet.tcp.dummynet=20000001
--- 1 ms, 200 KB/s, 0 buffers
ncftp> get 1M a
a: 1048576 bytes received in 6.21 seconds, 165.01 K/s.

ncftp> !sysctl -w net.inet.tcp.dummynet=20000050
--- 50 ms, 200 KB/s, 0 buffers
ncftp> get 1M a
a: 1048576 bytes received in 15.53 seconds, 65.96 K/s.

ncftp> !sysctl -w net.inet.tcp.dummynet=20007050
--- 50 ms, 200 KB/s, 7 buffers
ncftp> get 1M a
--- tcp_ip_out drop, have 7 packets (3 times)
a: 1048576 bytes received in 28.01 seconds, 36.56 K/s.

ncftp> !sysctl -w net.inet.tcp.dummynet=20007001
--- 1 ms, 200 KB/s, 7 buffers
ncftp> get 1M a
--- tcp_ip_out drop, have 7 packets (40 times)
a: 1048576 bytes received in 10.88 seconds, 94.09 K/s.

ncftp> !sysctl -w net.inet.tcp.sack=0x10 # enable SACK
ncftp> get 1M a
--- tcp_ip_out drop, have 7 packets (40 times)
a: 1048576 bytes received in 10.14 seconds, 101.01 K/s.
```

Fig. 6. A sample session showing the use of the simulator

ducible results, since the behaviour of the network is simulated. These expectations are wrong, because the traffic sources and, especially, their interactions with the simulator, are not fully deterministic.

5.2 Applications

The simulation technique used in this paper has been used extensively in the development of Selective Acknowledgement options for TCP [14, 15], and is being actively used in experiments on new congestion control strategies. These two applications reflect the typical cases where such a simulated environment is most useful. In the former (implementation of a protocol extension), building a real testbed would be hard because it would require the availability of other implementations. In the latter (analysis of the behaviour of a modified or new protocol), tests needs to be done first in a controlled environment, in order to get a better understanding of the protocol's behaviour; only at a later time the effects of (unknown) external traffic can be accounted for. In both cases, it is also very important to make sure that the final implementation has no undesired interaction with other mechanism already present in the protocol stack. Such experiments will be more and more necessary in the development of new protocols such as IPv6, or multicast extensions, because of the unavailability of a suitable infrastructure.

We would like to remark that, since the network simulator we have shown introduces very little overhead, it can also be used during normal operations, e.g. as a tool to provide rate-limitation for selected traffic.

6 Conclusions

We have shown how experiments on network protocols can be done easily on a standalone system using real world applications as traffic generators. Our approach gives the advantages of both real-world testing and simulation: simplicity of use, high control over operating parameters, high accuracy, no need for complex hardware settings, no overhead for running simulations. Especially, experiments can be run using a single workstation and do not require the presence of a real network or expensive devices such as routers and delay emulators.

The convenience of use and the little intrusivity of the technique described in this paper really encourages in having the network simulator available as a standard part of the system, so that experiments with different system configurations can be done as soon as there is a need, without requiring long times to setup a suitable testbed. The simulator is especially useful when developing completely new protocols, as a suitable testbed might simply not exist. The use of our technique can speed up dramatically the analysis and development of protocols, making the simulation environment readily available in a production environment and easily interfaced with other working systems.

Acknowledgements

The work described in this paper has been supported in part by the Commission of European Communities, Esprit Project LTR 20422 – "Moby Dick, The Mobile Digital Companion (MOBYDICK)", and in part by the Ministero dell'Università e della Ricerca Scientifica e Teconologica of Italy.

References

1. S.Keshav: "REAL: A Network Simulator", Technical Report 88/472, Dept. of Computer Science, UC Berkeley, 1988.
   Available as (`http://netlib.att.com/~keshav/papers/real.ps.Z`)
   Simulator sources available as `ftp://ftp.research.att.com/dist/qos/REAL.tar`
2. A.Heybey: "The network simulator", Technical Report, MIT, Sept.1990
3. J.Hoe: "Startup dynamics of TCP's Congestion Control and Avoidance Schemes", Master's Thesis, MIT, June 1995
4. S.McCanne, S.Floyd: ns-LBNL Network Simulator.
   Available from (`http://www-nrg.ee.lbl.gov/ns/`)
5. N.C.Hutchinson, L.L.Peterson: "The *x*-kernel: An architecture for implementing network protocols", IEEE Trans. on Software Engineering, 17(1):64-76, Jan.1991.
6. E.Limin Yan: "The Design and Implementation of an Emulated WAN", Tech. report, CS Dept., USC, 1995.
   Available from `http://catarina.usc.edy/lyan/delayemulator.tar.gz`
7. S.Floyd, V.Jacobson: "Random Early Detection Gateways for Congestion Avoidance", IEEE/ACM Trans. on Networking, 1(4):397-413, Aug.1993.
   Available from `http://www-nrg.ee.lbl.gov/nrg-papers.html`
8. V.Jacobson, "Congestion Avoidance and Control", *Proceedings of SIGCOMM'88* (Stanford, CA, Aug.88), ACM.
9. Z. Wang, J. Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm", ACM Computer Communications Review, Apr '92.
10. L.S.Brakmo, L.Peterson: "Performance Problems in BSD4.4. TCP", 1994.
    Available as `ftp://cs.arizona.edu/xkernel/Papers/tcp_problems.ps`
11. L.S.Brakmo, S.W.O'Malley, L.Peterson: "TCP Vegas: New Techniques for Congestion Detection and Avoidance", *Proceedings of SIGCOMM'94 Conference*, pp.24-35, Aug.94. Available as `ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps`
12. K. Fall, S.Floyd: "Comparison of Tahoe, Reno and SACK TCP", Tech. Report, 1995. Available from `http://www-nrg.ee.lbl.gov/nrg-papers.html`
13. L.Rizzo, Simulator's sources.
    Available as `http://www.iet.unipi.it/~luigi/dummynet.diffs`
14. M. Mathis, J. Mahdavi, S. Floyd, A. Romanow: "RFC2018: TCP Selective Acknowledgement Option", Oct.1996.
15. L.Rizzo: Sources for a SACK implementation for FreeBSD.
    Available as `http://www.iet.unipi.it/~luigi/sack.diffs`

This article was processed using the LaTeX macro package with LLNCS style