Dummynet: a simple approach to the evaluation of network protocols

Luigi Rizzo[1]
Dipartimento di Ingegneria dell'Informazione, Università di Pisa
via Diotisalvi 2 – 56126 Pisa (Italy)
email: l.rizzo@iet.unipi.it

Abstract

Network protocols are usually tested in operational networks or in simulated environments. With the former approach it is not easy to set and control the various operational parameters such as bandwidth, delays, queue sizes. Simulators are easier to control, but they are often only an approximate model of the desired setting, especially for what regards the various traffic generators (both producers and consumers) and their interaction with the protocol itself.

In this paper we show how a simple, yet flexible and accurate network simulator – `dummynet` – can be built with minimal modifications to an existing protocol stack, allowing experiments to be run on a standalone system. `dummynet` works by intercepting communications of the protocol layer under test and simulating the effects of finite queues, bandwidth limitations and communication delays. It runs in a fully operational system, hence allowing the use of real traffic generators and protocol implementations, while solving the problem of simulating unusual environments. With our tool, doing experiments with network protocols is as simple as running the desired set of applications on a workstation.

A FreeBSD implementation of `dummynet`, targeted to TCP, is available from the author. This implementation is highly portable and compatible with other BSD-derived systems, and takes less than 300 lines of kernel code.
Keywords: Protocol evaluation, TCP/IP, simulation

1 Introduction

Studies on the effectiveness and performance of network protocols are usually done in operational networks, and/or through simulations. The former approach is limited by the availability of an operational network with the desired features. Also, the actual operating conditions (queue sizes, delays, external traffic sources) are often unknown and/or not easily controllable. On the other hand, testing in a real network relieves the researcher from the need of modeling and building the various traffic sources to be used in the experiments, as real world applications can be used for this purpose.

Simulations have the advantage of being much more controllable and reproducible. However, a simulated environment is always just an approximation of the reality; this is especially true for the various traffic generators [4] which usually interact with the protocol itself. As an example, consider a simple FTP transfer over TCP. The flow of data is regulated by a number of factors, such as the size of windows, the speed of disks at the sender and the receiver side, the CPU load and scheduling of

[1]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.

processes at the two nodes, the acknowledgement generation policy, etc. Such factors are often hard to model accurately in simulators, possibly resulting in inaccurate or unreliable results. Nevertheless, the difficulties of setting up a real network with the desired features has stimulated the development of a number of network simulators, such as REAL [11], Netsim [7, 8] and `ns` [12]. The x-kernel framework [9] has also been used for the implementation and testing of network protocols.

In most cases, the aim of experiments on network protocols is to determine their behaviour in a complex network made of many nodes, routers and links, with different queueing policies, queue sizes, bandwidths, propagation delays. The difficulty of modeling complex environments often suggests the use of simplified networks (Figure 1) where the connection under observation goes through a pair of routers (which model the presence of queues of finite size) and a bottleneck link with given bandwidth and delay, modeling the overall bandwidth and delay. Quite often, in experiments on real networks, one of the routers is 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 [15]. Still, a couple of workstations and a bottleneck link or a local network are generally necessary to set up an experimental testbed.

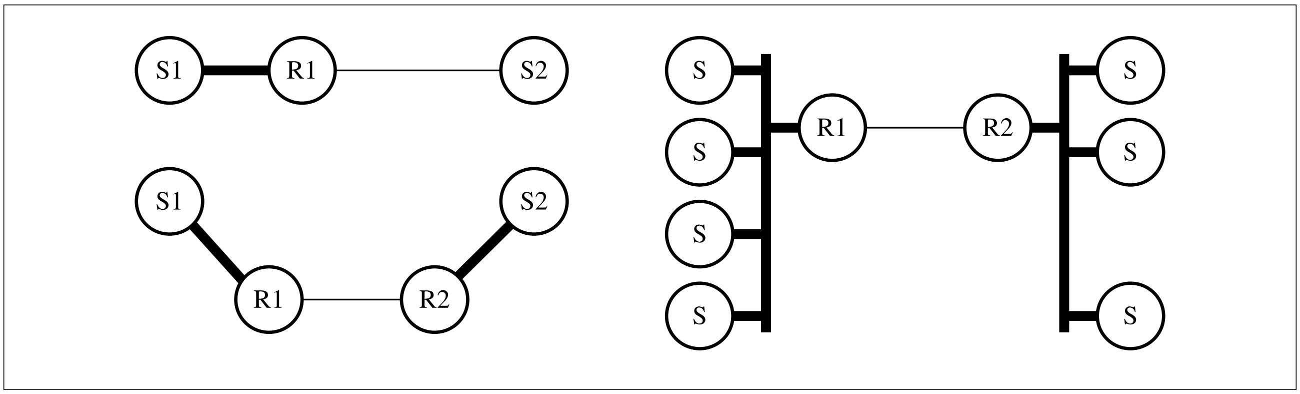


Figure 1: Typical settings used in the study of network protocols. The thin line represents the bottleneck link.

In this paper we extend the concept of a flakeway and propose a simple yet very effective approach to insert a model of this simplified network in an operational protocol stack, and run experiments on a standalone system. Our approach, called `dummynet`, is applicable to any protocol. It works by simply intercepting communication between the protocol layer under analysis and the underlying one (Figure 2), and simulating the presence of a real network with finite-size queues, bandwidth limitations, communication delays, and possibly lossy links. The `dummynet` approach gives most of the advantages of both simulation and real-world testing: great control over operating parameters, simplicity, ability to use real traffic generators. With our tool one can run experiments such as the one mentioned in [2, 3, 5, 9, 10, 14] on a single workstation, by using unmodified real-world applications (e.g. FTP, Telnet, Web browsers) as traffic generators. As a consequence, running an experiment is as easy (and quick) as running the desired set of applications on a workstation. Since `dummynet` introduces almost no overhead in the communication, experiments can be done up to the maximum operating speed supported by the system in use.

In the next Section we discuss the principle of operation of `dummynet` and its limitations. A brief description of an actual implementation is then provided, followed by an example of use and a discussion of possible extensions.

The work presented in this paper has some similarity with the one presented in [15]. The main difference lies in the fact that our approach allows experiments to be run on a standalone system, without the need of a network, and can be used to simulate networks with arbitrary topologies, as will be shown in Section 2.5.

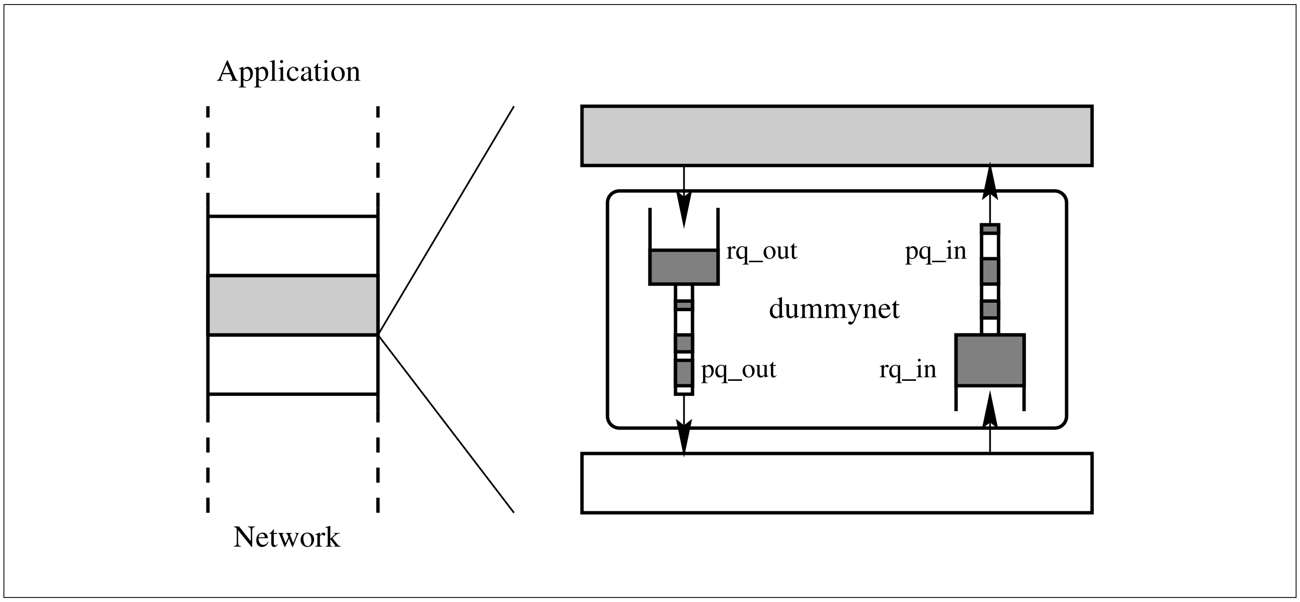


Figure 2: The principle of operation of `dummynet`

2 Principle of operation

In a typical protocol stack, each layer communicates with the adjacent ones (Figure 2), where the upper layer is generally connected to one of the communicating peers, and the lower layer connects to "the network". In order to simulate the presence of a network between two communicating peers, we need to insert the following elements in the flow of data:

- *routers* with bounded queue size and a given queueing policy;
- communication links (*pipes*) with given bandwidth and delay.

Additionally, random packet reordering and losses can be introduced to simulate the unreliability of real networks. Such features are important when the real network has redundant paths, and/or wireless or other noisy links (losses due to congestion are better simulated by bounded size queues).

As discussed before, the simplest setting usually includes one or two routers and one pipe. Both elements can be easily modeled by a couple of queues, `rq` and `pq`, inserted between the protocol layer under observation and the lower layer (Figure 2); a pair `rq/pq` is needed in each direction of the communication. In the following, we call k the maximum size of `rq`, B and t_p the bandwidth and propagation delay of the pipe, respectively. The following processing is necessary on traffic exchanged between the two layers:

1. when a layer communicates with the next one (above or below), packets are 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 possible such as RED [6]). Random reordering of packets can be introduced at this stage.

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.

Steps 2 and 3 can be performed by running a periodic task with period T, where 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.

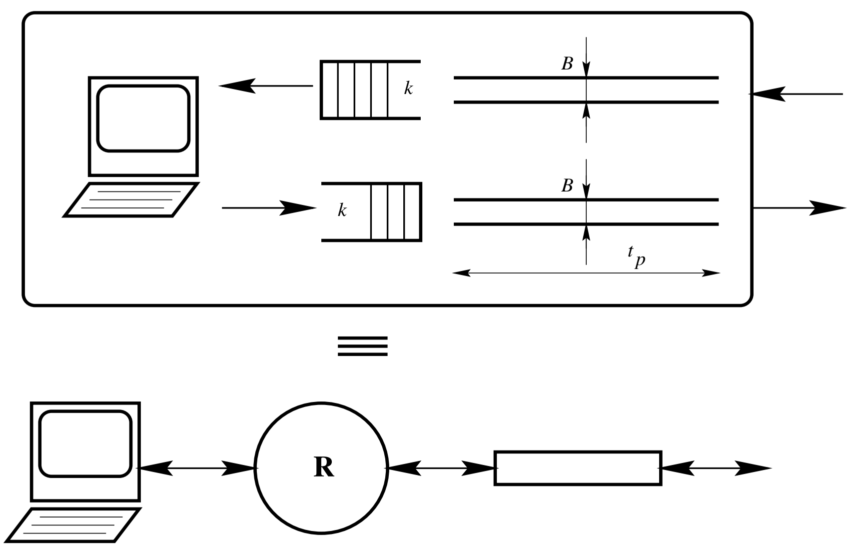


Figure 3: Structure of a node using `dummynet`

The resulting system looks like the one in Figure 3: local communication is also subject to the queueing and delay introduced by the simulated network. Such a simple setting is able to simulate almost all settings used in the literature where a bottleneck link can be identified, e.g. those used in [2, 3, 5, 9, 10, 14] to cite only some.

2.1 Applications

`dummynet` is particularly useful when studying the interactions between protocols and application programs in unusual or hard-to-reproduce settings. We can identify the following broad classes of applications:

- **debugging**. The use of real implementations of all the elements which take part in a communication can in fact evidence bugs or even unexpected features of the implementations which would almost unavoidably escape in a simulation (unless explicitly modeled, of course). As an example, consider a TCP implementation which caches route metrics from previous connections. Such a system has memory of the previous status of the network, but a simulation might not model properly this behaviour, hence failing in reproducing the response to sudden changes in the features of the network.

- **study of new protocols**. The ability to simulate unusual or hard to reproduce settings eases the study of new protocols. As an example, such a tool can be used to study the behaviour of existing or new congestion control mechanisms in presence of bottleneck links (e.g. the slow modem connections which have become so widespread nowadays, or asymmetric links).

- **performance evaluation**. As a third broad class of applications, this tool can be used to answer questions such as "How would my application run over a network with bandwidth B and delay τ ?". This is extremely useful for users or developers of, say, conferencing applications or some other kind of collaborative software, which want to evaluate the performance of a their product before (or when) setting up a possibly expensive communication link. Differently from the previous applications, in this case simulation is not an option because the performance metrics are often extremely subjective and the test environment must be as close as possible to the real setting.

Although not its primary purpose, a trivial extension to our `dummynet` (in order to select traffic basing on addresses and ports) allows its use as a simple selective bandwidth limiter for servers which do not explicitly provide such a feature either in the application or in the operating system itself.

2.2 Limitations

As all simulations of reality, `dummynet` can only approximate the behaviour of a real system with given features. Most of the approximations introduced by our tool derive from the granularity and the precision of the operating system's timer, and in many cases they have little influence on the experiments.

The granularity of the timer, T, limits the resolution in all timing-related measurements. In practice, this only constitutes a problem when simulating fast networks and short pipes, resulting in an overall packet delay comparable with T.

A second problem is that the periodic task might be run late, or even miss one or more timer ticks, depending on the overall system load. 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 events in `dummynet` (in particular, extractions from queues and pipes) occur synchronously with the system's timer. In principle, this might hide or amplify some real-world phenomenona which occur because of race conditions. However, it must be kept in mind that in a real system there are so many asynchronous events (corresponding to interrupts generated by external events, e.g. disk or network I/O, console activities, etc.) that the presence of `dummynet` does not change the essentially non-deterministic behaviour of a system when analysed with a fine-grain (e.g. microseconds) resolution.

When determining packet sizes (needed in the computation of the simulated bandwidth), `dummynet` normally uses the size of the payload at the lower layer (IP in our case of `dummynet`-TCP). It is straightforward to include link-layer overheads in the computations, if these are known. In particular, it is possible to simulate the effects of link-layer compression, which makes the link appear as a variable-bandwidth channel.

2.3 Implementation details

We have developed a basic version of `dummynet` working at the interface between TCP and IP. Our implementation [13] takes less than 300 lines of kernel code in FreeBSD; it intercepts calls to `ip_output()` made by TCP modules, and those to `tcp_input()` made by the protocol demultiplexer in IP.

Under normal conditions, there is no system overhead for using `dummynet`. When a TCP communication occurs, packets are queued according to the description given in Section 2, and a periodic task is run every T seconds to perform the required queue management. Note that the periodic task only needs to run when there is data in any of the queues of the system. The only system requirement for `dummynet` is the availability of a timer with a granularity – T – sufficiently small for the purposes of the simulation. Many BSD-derived systems have a granularity of 10 ms. Some, as in the case of FreeBSD, allow the use of a different granularity by simply using a kernel compile option, e.g. `option "HZ=1000"` sets T to 1 ms, which can be useful for experiments where a higher resolution (or bandwidth) is required. With current workstations, the overhead deriving from a faster clock rate is reasonably low.

Since no copies of data are done, and all operations require constant time, the overhead introduced by `dummynet` is almost negligible (a few function calls per packet) even when communications are in progress. It should be kept in mind that the system using `dummynet` should have sufficient buffer space to store both packets in `rq` (at most $k \cdot MSS$ bytes) and those in transit in the pipe (Bt_p bytes) in each direction.

`dummynet` only intercepts calls between adjacent protocol layers, leaving other traffic unmodified. As an example, our implementation of `dummynet` only acts between TCP and IP, meaning that a system mounting disks using NFS[1] over UDP can continue to work without performance problems, leaving a clean simulation environment with only TCP traffic. Some filtering rules could also be used, to make `dummynet` affect only selected traffic (e.g. TCP traffic to/from a given port, or all traffic through a given interface, etc.)

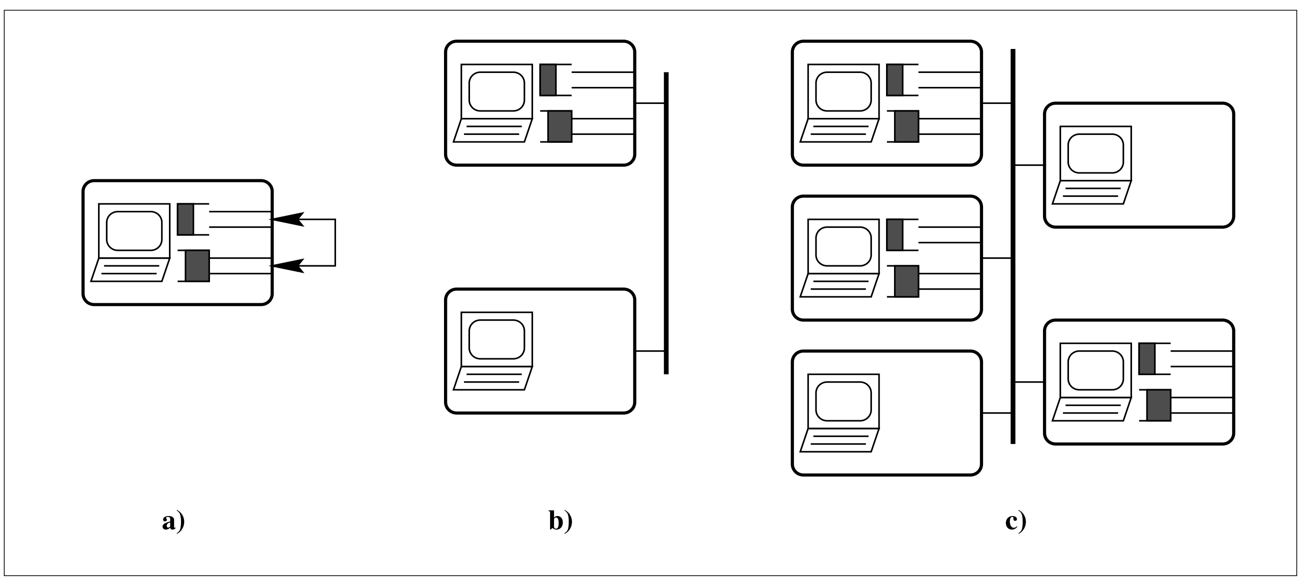


Figure 4: Various configurations for experiments with `dummynet`

[1] `dummynet` development was done on a completely diskless system, where crashes due to unstable kernels would not cause annoying disk corruptions

```
prova# ifconfig lo0 127.0.0.1 mtu 576   # this is to have many pkts per window
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.
```

Figure 5: A sample session showing the use of `dummynet`

2.4 Examples of use

Most features of the simulated network (k, B, t_p) are controlled by a kernel variable which can be modified with the `sysctl` command[2]. Ideally, the operating parameters of the protocol under test (e.g. MSS, max window size, TCP options in use, etc.), should also be configurable at runtime without the need of rebooting the system; in the case of BSD-derived system they often are via kernel variables.

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 4a), 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 5, where some FTP transfers are done using `ncftp` (we have used this applications because its timings are slightly more accurate than those of `ftp`). The system used for the experiments is a Pentium100 with 32MB RAM, running FreeBSD 2.1. Both client and server run on the same system, together with an X Server and a number of X applications.

[2] This is just the low level interface. A trivial C program or shell script can be written to parse a more user-friendly format such as "`dummynet bw 2.6 Mbps queue 30 delay 45 ms`"

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. 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 the experiment). Lines beginning with `---` are normally part of the system's logfile.

For each configuration (except the last three), we show the average throughput value of 10 tests, to compensate for the variations deriving from concurrent network and CPU activities.

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 `dummynet` running on one of them (Figure 4b). 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 `dummynet` configured with different parameters (Figure 4c).

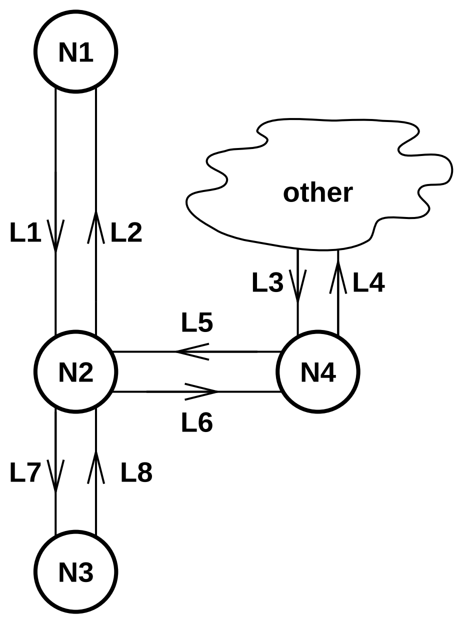
One would expect that experiments with `dummynet` – especially those performed with a single workstation – are completely deterministic and reproducible, since the behaviour of the network is simulated. These expectations are wrong, but not because of the presence of our tool: even without it, a real system is intrinsically not deterministic because of the presence and interaction of many asynchronous components (e.g. network traffic, disks and other peripherals). This non-determinism is less and less evident when the analysis becomes more coarse, but there might still exist some microscopic race conditions which cause a different ordering of events in different experiments with possibly macroscopic effects (e.g. occasional queue overflows). This is an intrinsic limitation of asynchronous systems which cannot be overcome by our tool.

2.5 Extensions

Even a very basic version of `dummynet`, comprising one queue and one pipe, suffices to run on a single workstation many of the simulations found in the literature. The addition of a module providing programmable (random or deterministic) packet losses and reordering is straightforward. The use of such a module is standard practice in simulations of noisy environments (e.g. wireless networks [1]).

Our tool is not limited to the simulation of networks with a single communication link. More complex topologies can be modeled by defining multiple data structures, each comprising one router, one pipe and a routing table (a bidirectional link is made of two of these objects).

Figure 6 shows a sample network and the routing table associated to each link. When a packet is extracted from a pipe, its next destination is computed according to the corresponding routing table. The ability to assign multiple addresses to one interface (e.g. the loopback interface), present in many modern systems, permits the simulation of complex topologies involving multiple hosts still using a standalone system. This extension is relatively straightforward: in the current implementation, adding a router/pipe pair requires just a few lines of code in addition to the tests required to route traffic into the appropriate path. Obviously, configuring such a setting requires a more powerful mechanism (such as a dedicated socket type with a special `setsockopt()` call) than a simple kernel variable.



L1	
DST	next
N1	L2
N2	TCP
N3	L7
N4	L6
other	L6

L2	
DST	next
N1	TCP
N2	L1
N3	L1
N4	L1
other	L1

L3	
DST	next
N1	L5
N2	L5
N3	L5
N4	TCP
other	L4

L4	
DST	next
N1	L3
N2	L3
N3	L3
N4	L3
other	IP

L5	
DST	next
N1	L2
N2	TCP
N3	L7
N4	L6
other	L6

L6	
DST	next
N1	L5
N2	L5
N3	L5
N4	TCP
other	L4

L7	
DST	next
N1	L8
N2	L8
N3	TCP
N4	L8
other	L8

L8	
DST	next
N1	L2
N2	TCP
N3	L7
N4	L6
other	L6

Figure 6: A method for building a complex network with `dummynet`. Each unidirectional link is associated with a suitable routing table used to determine the next hop for each packet. “TCP” and “IP” mean that the packet is forwarded to that layer in the protocol stack.

The current implementation of `dummynet` lacks any automated tool to setup an arbitrary network topology (basically, producing the routing tables shown in Figure 6) starting from a graphical or textual description. We are dubious about the real usefulness of such a tool for the following reasons:

- most works in the literature, especially regarding protocol implementation, use the simplified setting that we have implemented. Thanks to the dynamic configurability of k, B and t_p, most simulations can be run with just a few command lines; complex runs involving different settings may be run by using existing scripting languages;

- really complex networks might make the experimental results very hard to interpret, unless the structure of the network is carefully selected and understood by the researcher. Producing a low-level description of the network such as the one shown in Figure 6 helps in gaining such a knowledge.

Traces of the traffic in the network can be produced by using standard tools such as `tcpdump`, or by instrumenting `dummynet` to log all significant events on queues and pipes. The type of information which can be collected (and the overhead for collecting it) is very similar to that available from a simulator.

`dummynet` ought to be implemented as a Loadable Kernel Module (LKM), so as to ease modifications of the simulated network and different tracing options. Since facilities for LKMs are provided by several operating systems (e.g. SunOS, FreeBSD, Linux) and `dummynet` has very little interaction with the rest of the system, such an implementation poses no practical difficulty.

3 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.

The convenience of use of a system such as `dummynet` really encourages to make experiment with different system configurations, and to try variations to existing protocols. `dummynet` is especially useful when developing completely new protocols, as a suitable testbed might simply not exist. The use of `dummynet` 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 author thanks Lorenzo Vicisano, Jeff Mogul and an anonymous reviewer for their comments on an earlier version of this paper.

References

[1] H.Balakrishnan, S.Seshan, E.Amir, R.Katz, "Improving TCP/IP Performance over Wireless Networks", *Proc. of the 1st ACM Int'l Conf. on Mobile Computing and Networking (MOBICOM)*, Nov.95

[2] L.S.Brakmo, L.Peterson, "Performance Problems in BSD4.4. TCP", 1994 (`ftp://cs.arizona.edu/xkernel/Papers/tcp_problems.ps`)

[3] 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 (`ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps`)

[4] P.Danzig, S.Jamin, "A library of TCP Internetwork Traffic Characteristics", Technical Report CS-SYS-91-495, Computer Science Dept., USC, 1991

[5] 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`

[6] 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`

[7] A.Heybey, "The network simulator", Technical Report, MIT, Sept.1990

[8] J.Hoe, "Startup dynamics of TCP's Congestion Control and Avoidance Schemes", Master's Thesis, MIT, June 1995

[9] 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

[10] V.Jacobson, "Congestion Avoidance and Control", *Proceedings of SIGCOMM'88* (Stanford, CA, Aug.88), ACM

[11] S.Keshav, "REAL: A Network Simulator", Technical Report 88/472, Dept. of Computer Science, UC Berkeley, 1988. (`http://netlib.att.com/~keshav/papers/real.ps.Z`)
Simulator sources available as `ftp://ftp.research.att.com/dist/qos/REAL.tar`

[12] S.McCanne, S.Floyd, ns-LBNL Network Simulator (`http://www-nrg.ee.lbl.gov/ns/`)

[13] L.Rizzo, Sources for `dummynet` (`http://www.iet.unipi.it/~luigi/dummynet.diffs`)

[14] Z. Wang, J. Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm", ACM Computer Communications Review, Apr '92.

[15] E.Limin Yan, "The Design and Implementation of an Emulated WAN", Tech. report, CS Dept., USC, 1995 (`http://catarina.usc.edy/lyan/delayemulator.tar.gz`)