

Dummysnet Revisited

Marta Carbone, Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione
Università di Pisa*

Technical Report, 31 may 2009
<http://info.iet.unipi.it/~luigi/dummysnet/>

ABSTRACT

Dummysnet is a widely used link emulator, developed long ago to run experiments on networks with user-defined features. Since its original design, Dummysnet has been extended in various ways, and has become very popular within the research community due to its features and to the ability to emulate even moderately complex network setups within unmodified operating systems (FreeBSD and Mac OS X).

We have recently enhanced the emulation capabilities to improve the modeling of MAC layers; we have ported the tool to Linux; and, as part of the Onelab2 project, we are using Dummysnet to add emulation to PlanetLab nodes. Such works should greatly increase the use of the tool, and make it necessary to give a more comprehensive description of its features and limitations.

In this paper we will present current Dummysnet features, including recent work of ours mentioned above; make an extensive discussion of related work; and document how the emulator can be extended to implement more features.

1. INTRODUCTION

Live testing has always been an important part of the research and validation of network protocols and applications. While analysis and simulators can provide important insight on the behaviour of a system, the interaction of the system under test with the possibly unknown features of the external environment are difficult to evaluate without real experiments. Live experiments, in turn, pose the problem of controlling the environment to achieve reliable and reproducible results. So, researchers often use a combination of simulators [10, 11], emulators [13, 22] and real testbeds [9, 24] to perform their experiments in a more controlled way, and exploit the advantages of the various techniques.

The focus of this paper is on network emulators, and in particular on a network emulator called Dummysnet [22], developed over a decade ago by one of the authors, and become very popular since then [3].

Much of a product's popularity comes from three factors: availability, learning curve, feature set. These properties do not come for free, but are usually the result of design choices and implementation effort made by the authors.

In terms of availability, Dummysnet has been a standard component of FreeBSD for over ten years, and of Mac OS X since 2006. Hence, researchers could find the tool readily

*The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n.224263 - Onelab2.

available on their systems. Additionally, since the beginning we distributed bootable disk images to create dummysnet-enabled bridges using existing PC hardware and without disrupting existing software installations. This way, emulation could be made available within a network regardless of the operating systems in use. These distribution channels contributed significantly to the diffusion of the tool.

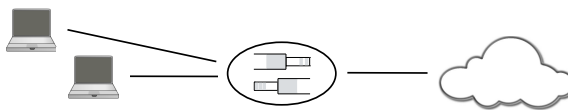


Figure 1: A Dummysnet-enabled bridge can be inserted between some nodes and the rest of the network without any change to the configuration of the network or the software running on the nodes.

In terms of learning curve, the user interface was carefully designed so that one can set up the emulator with as few as two commands, such as the following:

```
# define bandwidth and delay of the emulated link
ipfw pipe 1 config bw 3Mbit/s delay 32ms
# pass all traffic through the emulator
ipfw add pipe 1 ip from any to any
```

Additional features can be learned and used incrementally starting from this simple case, as we will show.

As for the feature set, Dummysnet has been significantly extended since its original design, and it currently includes various queue management schemes (FIFO, RED, WF²Q+), has the ability to create per-flow emulated links, can emulate complex topologies, and can reproduce multipath effects. We have recently added some support for better MAC emulation. Some third party extensions have also been developed, to introduce programmable or trace-driven packet dropping or alterations.

Dummysnet usage has exceeded by far our initial expectations. Besides the use by individual researchers to perform their experiments, Dummysnet is also a core component in the popular Emulab [24] testbed, and has become extremely popular as a traffic shaper. We expect that our recent work (more support for MAC emulation, described in Section 3.6; the Linux port; and its upcoming use in PlanetLab [9] nodes as part of the Onelab2 project [7]) will boost even more the usage of the tool.

This paper makes two major contributions: in Section 3, we give a comprehensive and up to date presentation of features and limitations of Dummysnet, so that researchers can

make the best use of the tool; in Section 4, we present a detailed discussion of the internal structure of the system, to show how and where new features can be added in the most effective way.

Before proceeding with the above, we start with an overview of related work, to put Dummynet in context with similar existing tools, and to present the various techniques that can be used in network emulators.

2. RELATED WORK

Up the mid '90s, and to some degree even nowadays, researchers often evaluated their systems with the help of custom network emulators, sometimes embedded in their applications. Especially for research on congestion control or error recovery schemes it is important, and sometimes sufficient, to induce controlled packet drops on the traffic. For this reason some emulation systems just cause packet drops, following some computed pattern or using a trace from an external source (a simulator or a real system). However, losses are normally dependent on actual traffic patterns, and for systems or protocols (such as TCP) that react to losses by altering the traffic they produce, a pure trace-driven loss generation may be insufficient. This has prompted the development of emulators that also model the behaviour of a link with finite queues, limited bandwidth and perhaps some propagation delay.

2.1 Single-link emulators

The two most popular and flexible representatives of this class are Dummynet [22], which has been available for over a decade as part of FreeBSD, and NISTNet [13], which is available as an add-on module for Linux. In addition to the simple link features described before (programmable rate, delay and queue size), both Dummynet and NISTNet can create multiple emulated links. They also include packet classifiers to select traffic subject to emulation, and support some additional emulation features such as loss generation, queue management policies, and more. Both tools are readily available, with little or no installation effort, on popular OS platforms.

The importance of having a packet classifier system becomes evident looking at the limitations of systems that *do not* have this feature. As an example, EMPOWER [29] does not use or implement a packet classifier, and has to rely on multiple physical interfaces to do the traffic selection, with a clear impairment on the ease of use.

Commercial products such as LANforge-ICE [6] also implement similar features, providing multiple emulated links with configurable rate, latency, and packet loss. In addition to these basic functions, LANforge-ICE lets the user configure network attributes such as jitter, and provides a wide set of packet corruption techniques.

Commercial products also include hardware-based emulators, which can provide much higher timing accuracy, support higher data rates and more detailed emulation of MAC features. As an example, the AnueSystem ethernet emulator [2] provides detailed emulation of multiple ethernet-like networks, also including a playback option to record and replay traffic on a network segment.

Some feature-rich simulators such as Ns-2 [10] and Ns-3 [11] provide an emulation feature to drive the simulation with live traffic or, vice-versa, generate live traffic from a run of the simulator.

Finally, some emulators do not use an analytical model of the system they reproduce, but resort instead to traffic traces, or even to monitoring live systems, to drive the emulation. Two systems that belong to this class are ENDE [27] and SatelliteLab [14]: both monitor the behaviour of an actual communication link subject to traffic patterns similar to the ones going through the emulator, and use the measurement results to reconfigure the emulator in real time.

Link emulation is a close relative to traffic shaping, a feature available in several systems and routers to enforce service limitations (or guarantees, depending on the point of view). Because of the similarities, some emulation systems are built around traffic shapers¹, adding the missing features. As an example, netem [16] implements link delays, while relying on the Linux package “tc” for traffic shaping and packet classification.

Some researchers have done work to extend Dummynet to match their needs more closely, using one of two techniques:

- using external programs to dynamically reconfigure the emulator, e.g., dynamically changing a link’s bandwidth. This approach does not require kernel modifications, even though the external programs will have little or no information on the traffic flowing through the emulator;
- implementing kernel changes to extend the emulation capabilities, e.g., adding selective or trace-driven packet drops. This approach is more intrusive in terms of modifications to the system, but it can make full use of the information on traffic and emulator state.

There are several instances of these approaches documented in the literature, mostly focused on specific research projects. In the first category we find the SEDLANE [23] testbed, which runs Ns-2 simulations to extract the time-dependent features (delays and loss rates) of an ad-hoc network, and then uses the results of the simulations to change over time the configuration of a dummynet-based testbed where the real experiments are run. As an example of the second category we can cite the KauNet [15] tool, which extends Dummynet to push error/drop patterns into the kernel, and applies them to the packet flowing through the emulator. The extension is complemented by a pattern generation tool to provide a compact representation of the information used by the kernel.

2.2 Emulating multihop networks

An often needed feature is the emulation of topologies involving multiple links, and routers or other intermediate nodes interconnecting them. Dummynet implements this feature through the ability to reinject traffic into the emulator multiple times. With this approach, a single physical machine can emulate lightly loaded networks with simple topologies, at least when the action performed by intermediate nodes can be described in terms of rules of the packet classifier (this includes basic forwarding and queueing).

When the intermediate nodes must perform more complex actions, some form of virtualization can be useful to model an entire network within a single system. An interesting approach is represented by Imunes [28], which supports multiple, virtual network stacks within one instance

¹The converse is also true: Dummynet is widely used as a traffic shaper, even more than as an emulator which was the original purpose of the tool.

of the FreeBSD operating system. Each virtual stack can implement a node in the emulated topology, and connect to other nodes through its own instance of Dummynet. The obvious extension of this concept is to run multiple emulator instances within virtual machines (Xen, VMWare, Virtual-Box, Qemu) and connect them as required.

For more complex configurations, e.g. the modeling of large or heavily loaded networks, one may want to distribute the emulation on multiple physical systems. This can be done nicely in the context of network testbeds. EmuLab [1] is one of the most popular topology-aware network testbed, able to run experiments on physical devices in a real network infrastructure. The system uses a topology description to program a set of configurable switches, connected to FreeBSD nodes that use Dummynet to emulate the various links involved. Planetlab [9] nodes are also being extended by the authors to use Dummynet for the same purpose.

3. DUMMYNET

In this Section we will describe the main components of our system: the actual emulation component, *dummynet*, and its associated packet classifier, *ipfw*. An additional component, `/sbin/ipfw`, or *ipfw* for brevity, is the frontend program used to configure the classifier and the emulator. We will use the term “Dummynet” (uppercase initial) to refer to the system as a whole.

The list of features and options available in Dummynet is very long and growing over time. For a complete and up-to-date description, the reader is referred to the *ipfw* manual page [4].

As a design principle, when building and extending Dummynet, we decided to focus on emulating the basic components of a communication network, and providing flexible and hopefully simple tools to connect these components to each other. This contrasts with another popular approach, which is to emulate the *aggregate effects* (such as loss patterns, delays, reordering, etc.) induced by a certain network configuration. The reason for our choice is that often such effects are heavily dependent on actual traffic patterns, and trying to model them independently would introduce too large approximations.

3.1 Pipes

The basic object that Dummynet makes available is called *pipe*. It combines some of the main features of a communication device: a queue with finite size, and a communication link with fixed bandwidth and programmable propagation delay.

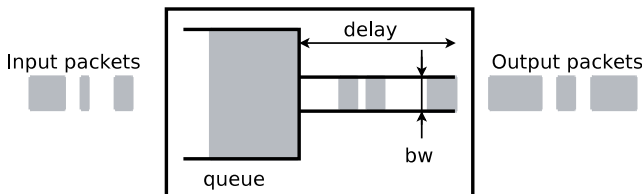


Figure 2: The structure of a dummynet “pipe”. Configurable parameters include bandwidth, delay and queue size.

As many as 2^{32} independent pipes can be created in the same system, each identified by a different integer. Pipes’

parameters can be set and reconfigured on the fly with commands such as the following:

```
ipfw pipe 3 config bw 640Kbit/s delay 30ms queue 20
```

A packet going through a pipe is queued or dropped depending on the occupation of the queue. The queue is drained at a rate corresponding to the link’s bandwidth, B . Once outside the queue, a packet is staged in a delay line for a time t_D (the propagation delay of the link), and then reinjected into the network stack. The total delay that a packet will experience is $T = t_D + (l + l_Q)/B$, where l is the packet length, and l_Q is the total length of the data in the queue before the packet was inserted.

In some situations it is useful to split traffic into multiple flows, and pass each flow through a separate pipe. This is achieved with a feature called “dynamic pipes”: additional mask parameters are specified in the configuration of a pipe, which indicate which bits in the 5-tuple of a packet (protocol, addresses and ports) should be used to group packets into flows. For each of the patterns resulting after masking, a new pipe will be created, and matching traffic will be directed to it. As an example, the rule

```
ipfw pipe 4 config mask src-ip 0x000000ff bw 1Mbit/s
```

will group packets with the same value of the least significant 8 bits in the source address, and direct each flow to a new instance of pipe 4. The bandwidth of each instance is 1Mbit/s. This feature is mostly of interest when Dummynet is used as a traffic shaper, and limitations need to be enforced independently on the various flows.

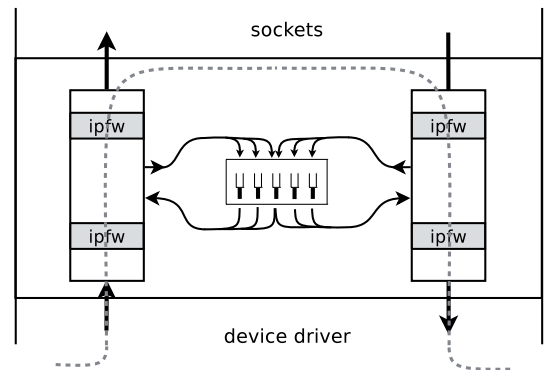


Figure 3: Hooking pipes to firewall rules. The dashed line represents the traffic flow when a node acts as a bridge or a router.

3.2 Traffic selection: the packet classifier

Traffic is passed to pipes using the packet classifier, *ipfw*, which in turn uses a list of numbered rules (*ruleset*) to match packets and decide their fate. The command to insert a rule in the ruleset is

```
ipfw add rule-number action options
```

Whenever the classifier is invoked on a packet, rules are evaluated in *rule-number* order. For each rule, zero or more *options* define the match criteria (e.g., match addresses, ports, protocols, even metadata such as direction, incoming inter-

face, related sockets). Options are evaluated on the current packet, and if they all match the *action* specifies what to do with it (accept, drop, pass to a pipe, etc.).

Within the system, traffic is intercepted and passed to the packet classifier at various points in the network stack, as shown in Figure 3. This typically happens during layer 2 and/or layer 3 processing, both in the inbound and outbound path. By means of appropriate actions in the rules, the classifier can pass matching traffic to different pipes. Coming out of the pipe, packets will be reinjected into the stack (or possibly into the classifier, see Section 3.3) after the point of intercept. The same ruleset is used at all points, so rules should contain options such as “in” and “out” to make a precise selection of where a certain rule should act.

The following is a sample ruleset and pipe configuration:

```
ipfw add 100 pipe 10 out dst-ip xy.it
ipfw add 200 pipe 11 in src-ip xy.it
ipfw add 300 pipe 24 out dst-ip 192.168.4.0/24
ipfw add 400 pipe 24 in dst-ip 192.168.4.0/24

ipfw pipe 10 config bw 2000Kbit/s delay 3ms
ipfw pipe 11 config bw 256Kbit/s delay 12ms
ipfw pipe 24 config bw 10Mbit/s
```

Here, rules 100 and 200 pass traffic for host xy.it (the name is resolved when the rule is inserted) through two different pipes, depending on the direction. This is a typical configuration for bidirectional links, possibly with different settings for each pipe in case the link is asymmetrical. This part of the configuration could emulate the case where local host is connected through an ADSL link to the remote server xy.it .

Rules 300 and 400 direct to pipe 24 all inbound and outbound traffic for subnet 192.168.4.0/24, irrespective of the direction (there is no restriction on what goes to which pipe; the system will take care of reinjecting packets in the right path on exit from the pipe). Using a single pipe for bidirectional traffic is sometimes useful when emulating shared media such as an ethernet segment.

It is very common to use DummyNet on a machine acting as a router or bridge, inserted on an existing network segment as shown in Figure 1. In this case one must remember that the traffic traversing the emulator goes through the classifier multiple times (see the dashed line in Figure 3); once again, the rulesets should be designed carefully to pass traffic through the pipes only once.

3.3 Multipath and multihop networks

The existence of multiple (wired) paths between source and destination can lead to phenomena such as packet reordering and losses. We can model multiple paths using a classifier option that matches with a given probability, resulting in traffic being directed to one of multiple links. As an example, the rules

```
ipfw add 1000 prob 0.2 pipe 10 src-port 80 in
ipfw add 1010 prob 0.7 pipe 20 src-port 80 in
ipfw add 1020 pipe 30 src-port 80 in
```

randomly send 20% of incoming HTTP traffic to pipe 10, another 56% (0.7 of the remaining 80%) to pipe 20, and the remaining part to pipe 30. If pipes have different bandwidth or delays, or they are subject to other interfering traffic, one can cause a wide range of effects from selective packet loss to jitter and reordering.

Topologies where packets must traverse multiple links (and queues) can be emulated by sending packet that emerge from a pipe back into the classifier, and in turn into other pipes. The `sysctl`² variable `net.inet.ip.fw.one_pass` controls the reinjection of the packets in the classifier after they emerge from a pipe. The reinjection is at the rule following the one that originally matched the packet, to prevent the creation of loops. Reinjected packets are subject to the usual classifier processing: they are compared against subsequent rules, and possibly sent through pipe(s) again.

3.4 Packet dropping

Packet drops in a wired network are usually due to queue overflows, queue management schemes (e.g. RED), or routing problems. Radio links add noise and interference as other potential causes of drops. Competing traffic also has a big role on the actual drop patterns experienced by a flow.

Once again, DummyNet focuses on the emulation of basic mechanisms that cause drops (queueing, routing), and on supplying tools to combine them effectively (classifiers, reinjection). We expect researchers to generate congestion-related drops by driving pipes with suitable traffic patterns, from the application under test and possibly from other competing sources. For non-congestion related losses we need different mechanisms. Drops due to routing misconfigurations can be emulated with the probabilistic match option described in the previous Section, which can also prove handy to run experiments with fixed loss rates³, e.g.:

```
ipfw add 400 prob 0.05 deny src-ip 10.0.0.0/8
```

More options can be specified to cause different drop probabilities depending on, say, packet lengths or other attributes. It is also very easy, as shown in Section 4.2, to add new classifier options and implement different packet dropping patterns. This approach has been used in the past by other researchers [15, 21].

3.5 Queue management and packet scheduling algorithms

DummyNet includes various queue management policies and one packet scheduling algorithm, all with configurable parameters. FIFO queues are the default, with size configurable either in bytes or number of slots. We also implement RED queues, whose parameters are also configurable. Other queue management schemes such as ABE [17] have been implemented in DummyNet in the past.

DummyNet implements the WF^2Q+ packet scheduling algorithm, and the infrastructure used for this can be easily reused to build different schedulers, or even to emulate multiaccess networks, as described in Section 3.6.2.

A general requirement of packet scheduling algorithms is to group packets into flows according to some criteria, queue flows separately, and let the scheduler select which queue to serve next.

²`sysctl` is the mechanism used on FreeBSD and Mac OS X to manage system-wide variables and statistics. This is similar to `procfs` entries on Linux, and registry entries on Windows.

³Pipes also have a `plr` (packet loss rate) attribute, which can be set to a value between 0 and 1 to causes random packet drops with that probability.

To this purpose, Dummynet implements an object called *queue*, which is used to create one or more physical queues that store packets belonging to individual flows. Queues also store the relevant parameters for the scheduling algorithm in use, such as weights or priorities. The mechanism to group packet into flows and create multiple queues is similar to the one used for dynamic pipes: a mask is applied to the 5-tuple, and a new instance of a queue is created for each of new pattern resulting after masking.

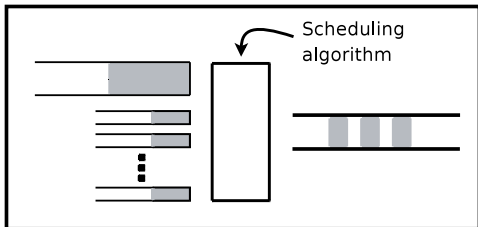


Figure 4: The binding between queues, scheduling algorithm and the corresponding pipe.

Queues are bound to a *pipe* object with a configuration parameter. The pipe, in turn, defines the data rate and the delay of the link, and how to select the next queue to serve according to the scheduling algorithm in use. In the case of WF²Q+, the algorithm will serve all backlogged queues according to their respective weights.

As an example, the following configuration defines a 4Mbit/s pipe, serving one queue with weight 20, and multiple (dynamic) queues with weight 10 each. The resulting arrangement is shown in Figure 4.

```
ipfw pipe 5 config bw 4Mbit/s
ipfw queue 10 config weight 20 pipe 5
ipfw queue 8 config weight 10 pipe 5
    mask src-ip 0xffffffff
ipfw add 1000 queue 10 out proto udp
ipfw add 1010 queue 8 out proto tcp src-ip
10.8.5.0/24
```

The parameter `mask src-ip 0xffffffff` tells the queue object to create one instance for each source IP, and rule 1010 limits the number of different source IP addresses sent to the queue object.

3.6 Emulating MAC layer effects

Precise emulation of MAC layer effects, such as framing (gaps, preambles, checksums), channel scheduling or link level retransmissions was not supported in the original version of Dummynet. This was partly because of a different focus of the emulator, and partly because the task can become extremely complex. Especially for shared media, the behaviour of a communication channel is heavily dependent on the interaction between the MAC layer and all network nodes (*stations*) sharing the channel. Without modeling these additional actors, the results could be heavily inaccurate, or at least they will be valid only in certain conditions.

There are two basic approaches that can be followed to address these problems:

- reproduce the behaviour of the channel as seen from one station, assuming a limited interference from other stations. This is an approximate solution but a relatively simple one to implement;

- implement a more complete model that considers all the actors in the system, namely the MAC layer and other stations. This approach can give much more precise results, and it is also very much in line with the philosophy of emulating mechanisms, not effects.

3.6.1 Single station model

The first approach has been recently added to the system, and is suitable for networks where the number of stations is small, or they are well decoupled from each other, e.g. due to MAC layer scheduling.

In our model we assume that, in addition to the transmission time l/B , a packet transmission will take some extra time given by the sum of busy intervals, contentions, backoffs, preambles, framing, possibly link level acknowledgments and retransmissions. Our extension takes as input the Cumulative Distribution Function of this extra “airtime”, as shown in Figure 5, and uses it in the computation of how long the channel will be busy for a packet transmission. An accurate determination of this curve is key to achieve realistic emulation, and this clearly depends on the type of MAC protocol and the load on the channel.

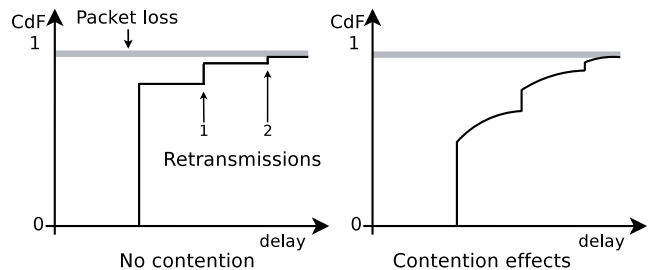


Figure 5: Specification of the additional airtime used for packet transmissions.

As a first approximation, one could start by modeling a non-contended channel with deterministic delays, in which case the Cdf will resemble the curve in Figure 5, left: with a probability corresponding to the successful transmission at the first attempt, transmissions will have a fixed overhead. Link level retransmissions, if present, will cause extra delays (due to timeouts on link level acks), giving the curve a staircase shape. Finally, packets may be lost due to excessive retransmissions, and we can specify the probability of this event. The curve for the non-contended case can be computed, at least as a first approximation, looking at the specifications of the MAC protocol involved.

In case of contention, the additional delay becomes variable, because we must now account for channel-busy times and collisions that depend on the presence of other stations. The regions corresponding to the various retransmissions will likely overlap, and the curve (to be determined experimentally or based on empirical considerations) should become similar to the one in Figure 5, right.

In the current code, the parameters associated to this empirical delay distribution must be stored in a file that is passed as a `profile` argument to a pipe’s configuration:

```
ipfw pipe 3 config profile /data/test/foo
```

The file includes a profile name, the data rate, a loss

threshold, and the number and position of corners in the polyline describing the curve:

```

name test_profile
bandwidth 2Mbit/s
loss-level 0.99
samples 5
prob delay # delay in us
0.2 100
0.4 130
0.5 130
0.8 1500
0.98 2000

```

3.6.2 Emulating multiple stations

A more precise emulation of a network with multiple stations can be achieved⁴ by reusing a lot of the infrastructure already present to implement scheduling algorithms. In fact, a MAC protocol is not so different from a scheduling algorithm, and can be modeled in a way similar to Figure 4: in this case each queue represents a separate station, and the scheduling algorithm implements the MAC protocol.

We can reuse the *queue* object and its `mask` mechanism to group traffic generated by each station into a separate queue. Instead of scheduling queues with WF²Q+, we will use the specific MAC algorithm to compute which queue should be served next, how long the transmission will take, and any other additional information that affects the various queues. The actual MAC protocol to emulate, and its parameters, will be configured as attributes of a *pipe*, whereas queues will be linked to pipes using the mechanisms already seen in Section 3.5.

3.7 Dummynet accuracy

Emulation accuracy has two aspects, one related to how detailed is the model of the system, and one referring to how closely the emulation system can reproduce the timing computed by the model.

The link model implemented by Dummynet pipes is rather simplistic as it does not account for overheads or features introduced by the MAC layer, such as framing, compression and retransmission. From an application's point of view, these approximations are often acceptable, and they are of the same order of magnitude or smaller than the timing errors induced by the underlying operating system, as discussed below.

A more significant source of approximations is the fact that, as in many emulation systems, timings are rounded to multiples of the system timer's quantum. The recommended value for this quantum is *1ms* or less. In practice, on modern hardware, the resolution can be significantly increased, up to 100-200 μ s. Interference from other kernel activities may induce extra delays of up to a few tens of microseconds or more, depending on systems type and load. For measurements on individual packets, this kind of delays is adequate for rates up to a few Mbit/s. Errors are not accumulated though, so aggregate effects (such as overall throughput, or effects of congestion related losses) can be measured with reasonable confidence even at higher rates.

A precise measurement of the actual accuracy is beyond the scope of this paper, and also not significant unless we

⁴Note that this Section describes work in progress not yet present in the system.

specify a lot of context, such as the platform (FreeBSD, Mac OS X, Linux), and the competing system load on the machine running the emulator.

3.8 Scalability

The applicability of a network emulator is strictly related to its performance and scalability. Depending on the configuration of the emulator and on the characteristics of the underlying hardware and operating system, there are Dummynet instances in use with thousands of active pipes and queues, and a total throughput across all pipes well over 100,000 packets per second⁵. But more than absolute performance, it is important to know about the scaling properties of the system.

Packet classification is done entirely in the *ipfw* firewall, whose internal architecture is in line with many modern systems such as iptables [5] and BPF [20]: individual options are translated into microinstructions that can be processed very efficiently. The cost of the classifier is linear in the number of options that must be evaluated on each packet.

Queue management and scheduling costs depends on the algorithm used, and range from $O(1)$ for FIFO and RED, to $O(\log N)$ for WF²Q+, where N is the number of backlogged queues attached to one pipe. Managing multiple pipes also introduces an $O(\log P)$ overhead, where P is the total number of active pipes. Idle pipes and queues (those with no backlogged traffic) normally cause only a memory overhead because they are managed using hash tables. Speaking of memory usage, one must remember that packets consume memory while they are stored into queues and pipes, and the amount of storage is, in the worst case, proportional to the bandwidth \cdot delay product of each pipe.

4. EXTENDING DUMMYNET

Dummynet is made of three pieces of code, closely reflecting different functionalities. One kernel module implements a programmable packet classifier, *ipfw*, in charge of selecting the traffic subject to emulation. Another kernel module, *dummynet*, is in charge of all actions that modify the timing of packets as they traverse the system. A userspace program, `/sbin/ipfw`, implements the frontend of the system and is used to read or modify the configuration of both kernel components.

This distinction of roles is very useful when it comes to decide where to add new features to the system.

Note that all Dummynet components are written in a way that supports smooth operation upon configuration changes. Modifications to the parameters of a pipe take effect starting from the next packet that reaches the head of the queue. Even changing the delay is processed in a way to avoid re-ordering. It follows that users can liberally alter the configuration of the system to reproduce certain phenomena that may occur in a real network.

4.1 Dynamic reconfiguration

The simplest Dummynet extensions are related to features that operate on relatively long timescales (100's of milliseconds or more), and do not need precise information on the status of the emulator. They can be conveniently implemented by periodically invoking the configuration com-

⁵The emulation overhead, as in most network devices, is mostly per packet and not per byte.

mands to alter the configuration as desired. Examples in this category include the simulation of mobility, routing flaps, and related long term bandwidth variations. As mentioned, this is an approach that has been already used by some researchers [23].

A non trivial example that can use this approach is a simplistic simulation of rate adaptation schemes [8, 18, 25] for multirate wireless channels. In this example, a master process is in charge of modeling the radio channel, taking as input the SNR values over time. Using that, and periodically monitoring the pipe's rate, the process can set the expected loss rate accordingly (see Figure 6, left).

The actual rate adaptation algorithm can be implemented as a second process (Figure 6, right). Starting from an initial state S , the process periodically reads the packet counters associated to the rules and pipes, determines how many packets have been transmitted or dropped in the last interval, updates its state S , and possibly updates the pipe's rate accordingly.

Channel Model	Rate adaptation scheme
<pre> for (;;) { usleep(10000); SNR = <next SNR value>; B = <read pipe rate>; L = loss(SNR, rate); <set drop prob. to L> } </pre>	<pre> S = <initial state> B = <starting rate> for (;;) { pipe 3 config bw B usleep(1000); X = <pass/drop counters>; B = new_rate(B, X, &S); } </pre>

Figure 6: The processes to model a rate Adaptation Scheme. Left: the process modeling the channel. Right: the process implementing the rate adaptation scheme.

It is important to note that while the reconfiguration is trivial, the difficulty in this approach is usually the mapping of the input data into parameters that can be injected into the emulator (in the above example, mapping SNR values at the receiver into packet loss rates).

4.2 Adding classifier options

Dynamic reconfiguration is not practical when it must be done too frequently (possibly on individual packet arrivals), and/or needs information associated to the individual packets. In these cases, we can further differentiate between what should be implemented in the classifier and what belongs to the *dummysnet* part.

In general, anything that does not depend on or modify the internal status of pipes and queues should be implemented as part of the classifier. Such features include e.g. selective packet dropping policies, or even error injection.

It is particularly convenient to add features to the classifier, because it has a very extensible structure. Each rule is represented by a sequence of options, which in turn have a “Type-Lenght-Value” (TLV) description. Each option is processed by a per-type handler, which is the only piece of code that needs to know about the structure of the Value field.

New classifier options can be added by defining a new TLV structure to store the relevant information, and a handler that performs the required operation. The handler is invoked during the evaluation of a rule that contains the option, and should return a boolean indicating whether the

packet did or did not match the specified condition. Normally, options (and their associated handlers) are completely stateless, but nothing prevents the handler from having side effects such as storing state or altering the packets. In fact, one such option (`keep-state`) already exists in *ipfw*, and is used to create stateful firewalls.

As an example, one could easily create a modified “conditional match” option where the match probability is a function of the packet length; or implement probabilistic packet corruption using the bit error rate as the input parameter. These options would be completely stateless and essentially similar to the “probabilistic match” shown in Section 3.3.

At another extreme in terms of implementation complexity, one could create an option to generate deterministic packet drops on a per-flow basis. This feature is sometimes used to test or validate the behaviour of data recovery schemes such as SACK [19, 21]. In this case, the option would receive as input a specific drop pattern, and a mask indicating how to aggregate flows, similar to the one used for dynamic pipes or queues. The code can use the already existing mechanisms to create records for new flows, store in those records the necessary state information (which could be as simple as a count of received packets) and then apply the drop pattern to the current flow based on its state.

Care must be taken with extensions that induce packet modifications. Within the kernel, packets are generally represented by a structure (`mbufs` [26] on FreeBSD and Mac OS X; `skbufs` [12] on Linux) that contains packets' metadata and a pointer to a shared buffer with the actual packet data. Altering bits in the shared buffer might affect other recipients of the same data. In this case, developers should be careful about data sharing and make private copies of the packet data to be modified.

4.3 Extending the emulation part

Any extension that needs access to queue or pipe state, needs to be implemented in the *dummysnet* part of the system. The extension described in Section 3.6.2 falls perfectly in this category. New queueing policies or scheduling algorithms also need to be implemented there.

In terms of implementation, these changes normally require to define the following:

- a set of parameters that adequately describe (in a form usable by the emulator) the features of the queues, of the link and of the algorithm used to select the queue to serve (the “scheduling algorithm” in Figure 4);
- a function to be run on packet arrivals, to select or create the proper queue to use and implement drop policies;
- a function to be run when the channel becomes available, to compute the next queue to serve, the transmission time, the fate of the packet at the head of the queue, and any possible alteration of the channel's state.

As usual we give a few examples of what can be done in this part of the system. The effects of link level compression can be emulated by computing the length of the compressed packet, and using that value in the evaluation of the packet transmission time. As a slightly more complex example, the same approach described in Section 3.6.1 could be used to model a network with slotted channel access. In this case

the overheads would be computed and not tabulated, and would probably rely on the packet length in addition to other channel parameters. Finally, we can add a new scheduling algorithm, or emulate a multiaccess network as described in Section 3.6.2. In this case, most of the additional complexity (both in terms of coding and in runtime costs) will be in the function implementing the scheduling part.

5. CONCLUSIONS AND FUTURE WORK

We have presented the current features of the Dummynet emulator, comparing it with other similar systems, and showing how its features can be used to model a number of useful network settings.

Dummynet has been originally developed on FreeBSD, and as such it is also available in Mac OS X. Recently we have completed a port to various Linux versions, including OpenWRT. A Windows port is also in the works.

The use of within Emulab, and its upcoming inclusion in PlanetLab, will make it very important that the features of the emulator stay in touch with current network technologies. To this purpose, we have provided detailed indications on how the emulator can be extended, and what is the most proper place to do the work depending on the feature to be added.

6. REFERENCES

- [1]
- [2] Anuesystems. http://anuesystems.com/Products_NetworkEmulator_Ethernet.shtml.
- [3] Dummynet references according to citeseer. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.2969>.
- [4] ipfw manual page. <http://www.freebsd.org/cgi/man.cgi?query=ipfw>.
- [5] iptables. <http://www.netfilter.org/projects/iptables/index.html>.
- [6] Lanforge-ice. http://www.candelatech.com/lanforge_v3/datasheet.html#ice.
- [7] The onelab2 project. <http://www.onelab.eu/>.
- [8] The onoe rate adaptation algorithm. <http://sourceforge.net/projects/madwifi>.
- [9] Planetlab. an open platform for developing, deploying, and accessing planetary-scale services. <http://www.planet-lab.org/>.
- [10] The ns-2 Network Simulator. <http://nslam.isi.edu/nslam/index.php>.
- [11] The NS-3 Network Simulator. <http://www.nslam.org/>.
- [12] C. Benvenuti. *Understanding Linux network internals*. O'Reilly, 2005.
- [13] M. Carson and D. Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.
- [14] M. Dischinger, A. Haeberlen, I. Beschastnikh, K. P. Gummadi, and S. Saroiu. Satellitelab: adding heterogeneity to planetary-scale network testbeds. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 315–326. ACM, 2008.
- [15] J. Garcia, E. Conchon, T. Pérennou, and A. Brunstrom. Kaunet: improving reproducibility for wireless and mobile research. In *MobiEval '07: 1st int. Workshop on System Evaluation for Mobile Platforms*, pages 21–26, 2007.
- [16] S. Hemminger. Network emulation with NetEm. In *Linux Conf Au*, 2005.
- [17] P. Hurley, J. Le Boudec, P. Thiran, and M. Kara. ABE: Providing a low-delay service within best effort. *IEEE Network*, 15(3):60–69, 2001.
- [18] A. Kamerman and L. Monteban. Wavelan-ii: A high-performance wireless lan for the unlicensed band. *Bell Laboratories Technical Journal*, 1997.
- [19] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Rfc2018: Tcp selective acknowledgment options. <http://www.ietf.org/rfc/rfc2018>.
- [20] S. McCanne and V. Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter Conference*, 1993.
- [21] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 257–267, New York, NY, USA, 2001. ACM.
- [22] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.
- [23] S. Seddik-Ghaleb, A. Ghamri-Doudane Y. Senouci. Emulating end-to-end losses and delays for ad hoc networks. In *IEEE International Conference on Communications 2007 (ICC'07)*, June 2007.
- [24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.
- [25] S. H. Y. Wong, H. Yang, S. Lu, and V. Bharghavan. Robust rate adaptation for 802.11 wireless networks. In *MobiCom '06: Proceedings of the 12th annual international conference on Mobile computing and networking*, pages 146–157, New York, NY, USA, 2006. ACM.
- [26] G. Wright and W. Stevens. TCP/IP illustrated, Vol.2: The implementation. *Addison-Wesley*, 1995.
- [27] I. Yeom and A. Narasimha Reddy. Ende: An end-to-end network delay emulator tool for multimedia protocol development. In *Multimedia Tools and Applications*, volume 14, pages 269–296. Kluwer Academic Publishers, 2001.
- [28] M. Zec and M. Mikuc. Operating system support for integrated network emulation in imunes. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Boston, MA, 2004.
- [29] P. Zheng and L. Ni. Empower: a scalable framework for network emulation. In *International Conference on Parallel Processing*, pages 185–192, 2002.