Dummynet Revisited

Marta Carbone, Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione
Universita' di Pisa*
30 november 2009 – rizzo@iet.unipi.it

ABSTRACT

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

The goal of this paper is to present in detail the current features of Dummynet, compare it with other emulation solutions, and discuss what operating conditions should be considered and what kind of accuracy to expect when using an emulation system.

1. INTRODUCTION

Live testing has always been an important part of the research and validation of network protocols and applications. Analysis and simulators can provide important insight on the behaviour of a system, but the interaction of the system under test with the possibly unknown features of the external environment is difficult to evaluate without live experiments. These, in turn, pose the problem of controlling the test environment to achieve reliable and reproducible results. Researchers often use a combination of simulators [5,6], emulators [9,18] and real testbeds [10,21] 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 system called Dummynet [18], developed over a decade ago by one of the authors, and become very popular since then [2]. Three factors have contributed mainly to its diffusion: availability, learning curve, and feature set.

In terms of availability, Dummynet has been a standard component of FreeBSD for over ten years, and of Mac OS X since 2006. Hence, researchers could find it readily available on their systems. Additionally, since the beginning we distributed bootable disk images to create dummynet-enabled bridges using existing PC hardware without disrupting their existing software installations. This way, emulation could be made available within a network as shown in Fig.1, regardless of the operating systems in use.

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.

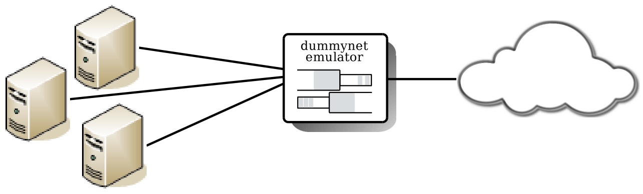

Figure 1: A Dummynet-enabled bridge can be inserted in an existing the network without any change in the configuration.

As for the feature set, Dummynet has been significantly extended since its original design, and it currently includes various queue management schemes (FIFO, RED, WF^{2}Q+), 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 [12,19], to introduce programmable or trace-driven packet dropping or alterations.

Dummynet usage has exceeded by far our initial expectations. Besides the use by individual researchers, Dummynet is also a core component in the popular Emulab [21] 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; a Linux port; and the inclusion in PlanetLab [8] nodes as part of the Onelab2 project [4]) will boost its use even more.

This paper makes two major contributions: in Section 3, we give a comprehensive and up to date presentation of features and limitations of Dummynet, so that researchers can make the best use of the tool; in Section 4.1, we make a detailed analysis of the accuracy and performance of Dummynet.

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. For research on congestion control or error recovery

*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.

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 links with finite queues, limited bandwidth and propagation delay.

2.1 Link emulators

The two most popular and flexible representatives of this class are Dummynet [18], which has been available for over 10 years as part of FreeBSD, and NISTNet [9], which is available as an add-on module for Linux. In addition to the link features described before (programmable rate, delay and queue size), both Dummynet and NISTNet can create multiple emulated links, 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.

An interesting proposal is Netpath [7], which relies on the Click modular router software to build emulated links. Click [15] is a system that lets the programmer build datapaths by glueing together “elements”. These can represent input or output interfaces, classifiers, queues, delays lines. By constructing a graph of Click elements, Netpath builds a full emulator with functionalities similar to Dummynet. The Netpath approach is interesting because the emulator can be extended with relative ease by writing new Click elements with the desired functionalities. Also, the ability to completely bypass the operating system and its device drivers makes this a very interesting solution for a high performance, standalone emulator box.

Commercial products such as LANforge-ICE [3] 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 functions.

Hardware-based emulators can provide much higher timing accuracy, support higher data rates and more detailed emulation of MAC features. As an example, the AnueSystem Ethernet emulator [1] 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 [5] and Ns-3 [6] include an emulation mode 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 [22] and SatelliteLab [11]: both monitor the behaviour of an actual communication link subject to interesting traffic patterns, and use the measurement results to configure the emulator in real time to generate a similar behaviour.

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 shapers[1], adding the missing features. As an example, netem [13] implements link delays, and relies on the Linux package “tc” for traffic shaping and packet classification.

Some researchers have done work to extend Dummynet 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 [19] 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 [12] 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 connected by routers or other intermediate nodes. Dummynet implements this feature by reinjecting traffic into the emulator multiple times. With this approach, a single physical machine can easily emulate lightly loaded networks involving multiple network hops and routers.

When the functions of the routers cannot be modeled by classifier rules, some form of virtualization can be useful to model an entire network within a single system. Imunes [23] implements multiple, virtual network stacks within one instance 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, VirtualBox, 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. Modelnet [20] supports the definition of complex topologies using a cluster of routers running a modified version of Dummynet. The Modelnet preprocessor, using a topology description, maps

[1]The 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.

links to Dummynet pipes (Sec.3.1), places pipes on individual routers to distribute the load, and computes the sequence of nodes/pipes that a packet must traverse. If pipes are on different routers, Modelnet only needs to transfer the control information (not the payload) from one router to another, which reduces the run-time overehead of the system.

The EmuLab [21] testbed acts in a similar way. 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 [10] nodes have been recently extended by the authors to use Dummynet for the same purpose [8].

3. DUMMYNET

In this section, we will describe the main components of our system: the actual emulation engine, *dummynet*, and its associated packet classifier, *ipfw*. Configuration of both components is done running the `/sbin/ipfw` program (`ipfw` for brevity). We will use the term "Dummynet" (uppercase initial) to refer to the system as a whole. Both the classifier and the emulation engine have a large (and growing) set of features, so we refer the reader to the online page [17] for a complete and up-to-date description.

Our design principle, when building and extending Dummynet, was to reproduce only the basic components of a communication network, and provide flexible and simple tools to connect these components to each other. For the reproduction of high level phenomena such as congestion-related loss or routing across multiple paths, we rely on a proper composition of Dummynet components and traffic sources to achieve the desired results.

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 made available by the emulation engine is called *pipe* (Fig. 2). It combines a queue with finite size, and a communication link with fixed bandwidth and programmable propagation delay. Each pipe is identified by a different integer, and their number is only limited by available memory.

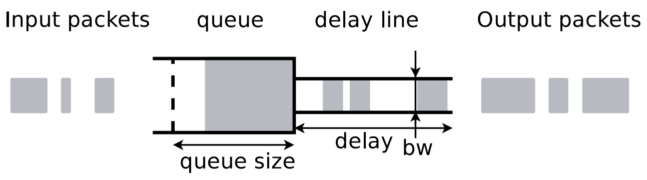

Figure 2: The structure of a dummynet "pipe". Configurable parameters include bandwidth, delay and queue size.

Pipes' parameters can be set and reconfigured dynamically with one-line commands such as the following:

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

Packets are intercepted in the network stack as discussed in Sec.3.2, and possibly sent to pipes. A packet going through a pipe is queued if there is capacity, and 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. As a result of this process, the pipe will delay each packet i by a time $T_i = (l_i + Q_i)/B + t_D$, where l_i is the length of the packet, Q_i is queue occupation when the packet was queued, and B and t_D are the bandwidth and propagation delay of the link.

In some situations it is useful to aggregate traffic into flows, and pass each flow through a separate pipe. This is achieved with a feature called "dynamic pipes": "mask" parameters can be specified in the configuration of a pipe, which indicate the bits in the 5-tuple of a packet (protocol, addresses and ports) that should be used to group packets into flows. For each pattern 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.

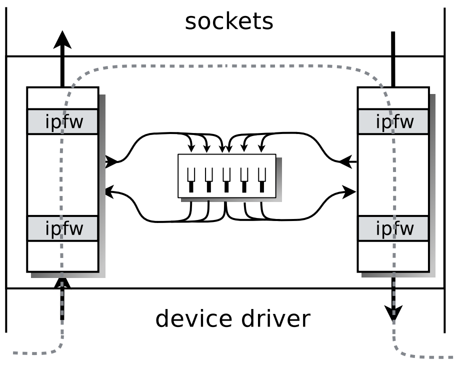

Figure 3: Classifier rules send packets to specific pipes. On exit, the traffic is reinjected into the network stack.

3.1.1 Complexity

Traversing a pipe has a constant per-packet cost, with slightly optimized code paths in case of pipes with zero delay or no bandwidth limitations. The presence of multiple pipes (either statically defined, or dynamic ones) gives an $O(\log N)$ factor in the number of active pipes, which arises because internally Dummynet uses a priority queue to determine the next pipe to serve. Actual performance numbers are measured and discussed in Sec.4.2.

3.2 Traffic selection: the packet classifier

As mentioned earlier, traffic is passed to pipes using the packet classifier, *ipfw*, which uses a list of numbered rules (called *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 interface, related sockets). Options are evaluated on the current packet, and if they all match the *action* specifies what to do with the packet itself (accept, drop, pass to a pipe, etc.).

Traffic is intercepted and passed to the packet classifier at various points in the network stack (Figure 3) – typically 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 following is an example ruleset and pipe configuration which emulates two distinct links:

```
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, one for each direction. This is a typical configuration for bidirectional links, possibly with different settings for each pipe in case the link is asymmetrical, e.g. an ADSL.

Rules 300 and 400 direct to pipe 24 all inbound and outbound traffic for subnet 192.168.4.0/24, irrespective of the direction. Using a single pipe for bidirectional traffic is sometimes useful when emulating shared media such as an Ethernet segment.

3.2.1 Complexity

Classifier rules are based on microinstructions, so their cost is proportional to the number and complexity of the options they include: basic address or port matching is very simple, whereas looking up sockets or routes associated to packets is more expensive also due to the locking requirements. Typically, the classification costs are linear in the number of rules, but the classifier includes mechanisms, such as lookup tables and stateful entries, that permit $O(1)$ dispatching of the traffic to the appropriate pipe. A measurement of actual processing time for the simplest rules is reported in Sec. 4.2.

3.3 Multipath and multihop networks

The existence of multiple (wired) paths between source and destination can lead to packet reordering and/or losses. We can model multiple paths using a classifier option that matches packets with a given probability; this allows traffic to be randomly 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
```

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 variable `net.inet.ip.fw.one_pass` controls the reinjection of the packets in the classifier after they emerge from a pipe. 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.

As said before, Dummynet focuses on the emulation of basic mechanisms that cause drops (queueing, routing), and on supplying tools to combine them effectively (classifiers, reinjection). Congestion-related drops can be induced by driving pipes with suitable traffic patterns, from the application under test and possibly from other competing sources. For non congestion-related drops, we can use the probabilistic match option described in Sec. 3.3 to emulate links with uniform random loss patterns, e.g.:

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

More classifier options can be used to create different drop probabilities depending on addresses, packet lengths or other attributes. It is also easy to create new classifier options and implement different packet dropping patterns, an approach that has been used in the past by other researchers [12, 16].

3.5 Queue management and packet scheduling

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 and GRED queues, whose parameters are also configurable. Other queue management schemes such as ABE [14] have been implemented in Dummynet in the past.

Research on packet scheduling is supported by two mechanisms: an object called *queue*, used to create one or more physical queues that store packets belonging to individual flows, and a mechanism to load and configure at runtime specific link scheduling algorithms. The connection between queues, pipes and schedulers is shown in Figure 4.

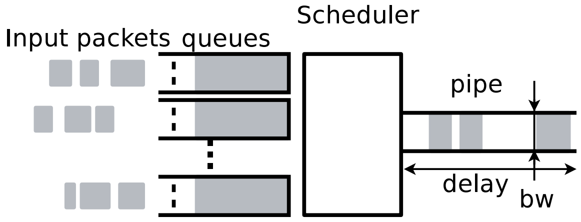

Figure 4: The binding between queues, scheduling algorithm and the corresponding pipe.

A "queue" is created or configured with the command

```
ipfw queue N config pipe X weight Y mask ...
```

which additionally defines the pipe (and scheduler) that the queue is attached to, sets the relevant parameters (such as weights or priorities) for the scheduling algorithm, and defines how packets should be grouped into flows, The mechanism to group packets into flows (and create one queue per flow) is similar to the one used for dynamic pipes: a mask is applied to the 5-tuple of the packet, and a new instance of a queue is created for each pattern resulting after masking.

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.

```
ipfw pipe 5 config bw 4Mbit/s // default: WF2Q+
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.

The configuration of the scheduler type and of the output link is done using the `ipfw pipe X config ...` command. Support for loadable scheduling algorithms has been added recently, and this will allow complementing the existing WF^{2}Q algorithm with other schedulers.

3.5.1 *Complexity*

The cost of scheduling multiple queues into the same pipe is completely determined by the scheduling algorithm in use. In this respect, the literature reports a wide range of options from $O(1)$ to $O(\log N)$ to $O(N)$. The WF^{2}Q+ algorithm implemented as default has logarithmic complexity with constants similar to those related to the management of multiple pipes.

3.6 Emulating MAC layer effects

Precise emulation of MAC layer effects, such as framing (gaps, preambles, checksums), channel scheduling or link level retransmissions is not present in Dummynet. This is 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 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.

Given that links which deviate significantly from our basic link model are more and more common, we recently introduced some limited support to model these media. To keep the complexity within acceptable limits, we compromised on an approximate solution that reproduces MAC effects in a probabilistic way and assumes only limited interference from other stations.

In our model, a packet transmission will keep the channel busy for the transmission time l/B plus some "extra airtime", given by the sum of busy intervals, contentions, backoffs, preambles, framing, possibly link level acknowledgments and retransmissions. The emulator takes as input the Cumulative Distribution Function of this extra "airtime", 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.

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

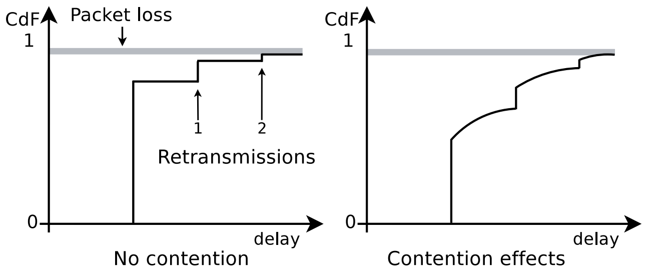

Figure 5: Specification of the additional airtime used for packet transmissions.

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
```

4. ACCURACY AND PERFORMANCE

One of the main reasons for using an emulator is to provide a reasonably reproducible environment for experiments. It is then important to understand the limits of operation, and the extent of the approximations and errors introduced by the emulator itself. In the following we performs this analysis for Dummynet, but note that most of the issues discussed here apply, with similar or the same effects, also to the other emulation systems presented in the literature or available on the market.

The measurements presented here were performed on an entry-level desktop machine (AMD BE-2400 CPU, 2.3GHz, 800MHz FSB) running FreeBSD 8. Some of the results (e.g. those of Sec.4.1.1 and Sec.4.1.3) are relatively independent of the OS type and CPU speed. Others are instead heavily influenced by these factors.

4.1 Emulator accuracy

Two aspects influence the accuracy of an emulator: how detailed is the model of the system, and how closely the hardware and software can reproduce the timing computed by the model.

For the first aspect, the basic link model implemented by Dummynet pipes (and by other common emulators such as Nistnet and Netpath) is limited to a fixed-rate channel. None of these systems make any attempt to emulate MAC level features such as framing, slotting, collision handling, link level retransmissions. The reason for the omission is that a detailed modeling of these features would be prohibitively expensive, unless one limits the model to simple or approximate features (for Dummynet, see Sec. 3.6).

The second aspect – reproducing the timing computed by the model – is mainly affected by the three factors: competing traffic on the physical links, interference of Operating System activities, and timers resolution. These factors will be discussed in detail in the rest of this Section, and affect any emulation system, not just Dummynet. It is useful to summarize their impact in the following table:

Cause	Error introduced
Competing traffic	120..1200 μs per emulated link
OS interference	30 μs or more, OS-dependent
Timer resolution	25..1000 μs, constant, OS-dependent

We should emphasize that competing traffic, which introduces a large and often neglected error component, *affects all emulators* that support multiple emulated links. The effect of the other two components can be reduced by running the operating system in carefully controlled load situation, or bypassing the OS altogether.

4.1.1 Competing traffic on the output interface

Emulators normally compute a "due time" for each packet that goes through them, and arrange transmissions on the physical output links so that packets are released at their due time. When an emulator supports multiple emulated links, and packets on different emulated links have the same or very close due time, the serialization of the transmission on the physical output interface, introduces an unavoidable error which in the worst case can be $T = (N-1)L/B$ seconds. Here N is the number of emulated links that may conflict, L is the maximum packet size, and B is the bandwidth on the output interface. To put the numbers in context, L/B can be as large as 1.2ms with packets of 1500 bytes on a 100Mbit/s Ethernet, and 120μs on a Gigabit interface. This error can be reduced or removed by working on the factors of the formula above.

4.1.2 Operating System interference

Non Real Time Operating Systems do not give guarantees on when kernel tasks (such as those related to emulation) will be processed, as they might be preempted by other OS activities with higher priority. To determine at least a lower bound for the delays that can be experienced in the network stack due to OS interference, we measured the variations in the ping response time of a system under some representative load conditions

- IDLE. Completely idle system, no extra process is running except the basic system services;
- USER. Several processes run the loop
  ```
  extern volatile a; for(;;) a++;
  ```
  consuming the full CPU available, and accessing the memory bus, in user space;
- KERNEL. A number of process are accessing devices and memory filesystems, continuously issuing system calls that cause heavy kernel load;

and with three different configurations of the network stack: no firewall, 1 ipfw rule, 100 ipfw rules. This test is significant because it represents very closely the work done in the kernel by the various configurations of the emulation code.

The results (both average and standard deviations) are presented below (all times are in μs):

	IDLE		USER		KERNEL	
	avg	sd	avg	sd	avg	sd
no ipfw	27.2	1.94	27.6	1.72	49.7	2.50
IPFW-1	28.1	2.82	28.2	1.36	55.1	2.62
IPFW-100	36.2	1.82	36.3	1.78	71.0	2.60

More than the absolute times, we are interested in the difference between the baseline case (IDLE) and the columns corresponding to other load conditions. We see that a USER load, no matter how large, has negligible impact on the system, whereas a KERNEL load introduces a lot of additional delay: individual packets were delayed by 20-35 μs in our experiments, and we can easily generate much higher delays with appropriate kernel loads.

This type of error can be reduced by carefully controlling the operating conditions of the Operating System, or bypassing it altogether. This is however not always possible when the emulator must run on the same machine used for experiments.

4.1.3 Timer resolution

Dummynet (and many other emulators), internally round times to multiples of the quantum of the system timer, which runs HZ times per second (in our case $HZ = 1000$). This introduces a timing error of $1/HZ = 1$ ms, which can be greatly reduced by running the system timer at a higher rate (we have often used HZ=10000 to 40000 corresponding to 25..100 μs error) or relying on the high resolution timers that many operating system make available.

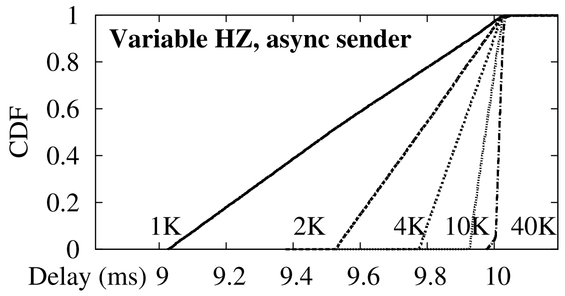

Figure 6: Distribution of delays for a 10ms pipe and different HZ values, asynchronous senders.

Figure 6 shows the measured distribution of delays for a pipe with a nominal 10 ms delay, using different values of the timer tick. As expected, using higher HZ values reduces the error, up to the point where the error component due to the OS interference, discussed in the previous Section, becomes dominant.

It is important to realize that the effect of timer resolution depends on the behaviour of the traffic sources. Quite often, sources tend to synchronize with the emulator (as an example, a TCP receiver responds immediately to packets released by the emulator), and this can produce peculiar results in the measurements. One such example is presented in Figure 7, where we show the delay introduced by the same 10 ms pipe with HZ=1000 and synchronous senders (in this case, a `ping` with an interval of 10.2 or 10.5ms, or a `ping -f` which responds as soon as the reply comes in). As you can see, the 1 ms error does not show at all in the `ping -f` case, and the periodic pings give a peculiar staircase pattern.

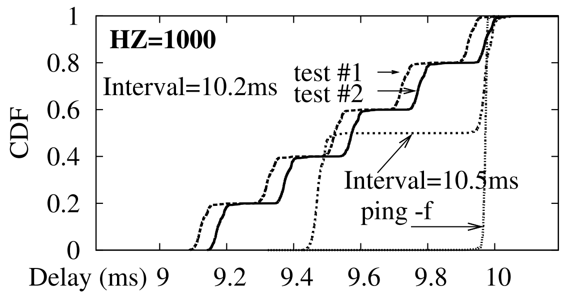

Figure 7: Distribution of delays for a 10ms pipe, HZ=1000, synchronous senders.

4.2 Performance

Emulators are often characterized with their maximum throughput, in packets per seconds (PPS) [2]. However, this value is the result the various contributions shown in Fig.8, top. Only classification and emulation belong to the emulator itself, whereas the others are inherited by the underlying operating system. In the case of dummynet, which is a) designed to run on top of an existing OS, and b) runs on different platforms, we have no chance to optimize any of these phases.

To better identify the overhead introduced by the emulator, we measured the per-packet processing time (the inverse of the PPS rate) of a system with a local traffic generator which is discarding traffic at various point of the processing chain (Fig.8 bottom): before the classifier (A), after one or more classifier rules (B), or after going through the emulation engine (C) with different configurations. From the difference between the various time we can derive the average cost of each processing stage. The use of a local generator removes two very expensive and variable parts of the processing chain, thus permitting more accurate measurements. The generator transmits minimum-sized UDP packets to one or more different ports.

Experiments lasted 1..5 sec each, were repeated at least 100 times, and always on a freshly booted system. Cases B and C are run with different configurations as explained later. Also, for reference, we have also made some measure-

[2]the per-packet overhead is normally the dominant factor in limiting performance.

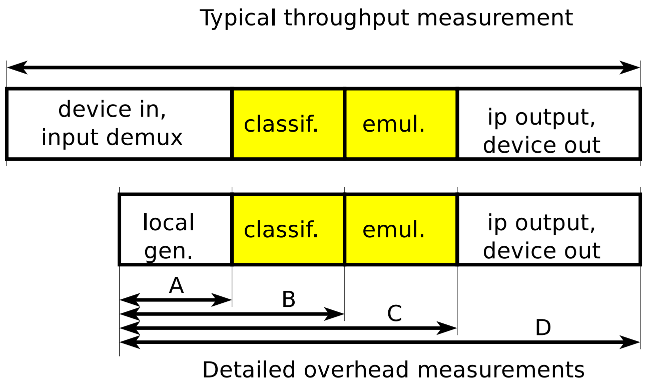

Figure 8: Measurements to determine the emulator's overhead. Top: typical components in PPS measurements. Below: our detailed measurements, to extract the cost of classification and emulation.

ments for case D (packets going all the way through a Gbit interface), to compare the emulation overhead with the cost of device output, which is completely dependent on the OS in use. The results are summarized in the following table:

Case	avg/sd (ns)	1 flow
A	643 / 16.9	Drop before classifier.
B	1044 / 30.0	Drop in first rule.
B_{100}	4668 / 59.3	Drop after 100 rules.
C_0	1740 / 44.8	pipe with 0 delay, unlimited bw.
C_d	2360 / 33.2	pipe with 20ms delay
C_r	1877 / 44.7	pipe with bw limit
D	2212 / 125	No pipe, output on Gbit interface
Case	avg/sd (ns)	1000 flows
A'	1704 / 64.6	Drop before classifier
B'	2095 / 73.4	Drop in first rule
C'_r	3549 / 93.2	1000 pipes with bw limit

First, cases A and B let us determine the classification cost, which has a fixed component related to entering the classifier ($B - A = 400$ ns), and a variable component that depends on the number of rules in the classifier. From the difference between case B with 1 rule, and case B_{100} with 100 rules, we derive that the simplest rules requires $(B_{100} - B)/99 = 36.6$ ns.

If a packet goes to a pipe, we have an additional component between roughly 700 and 1300 ns ($C_0 - B$ and $C_d - B$, respectively). While the complexity of pipe processing is $O(1)$, there are different code paths depending on the parameters and queue occupation. A pipe with unlimited bandwidth and no-delay (C_0) and a pipe with pure delay (C_d) represent the two extreme cases for the complexity of the emulation code.

When multiple pipes are involved, the processing time of the emulation engine is $t_1 + k \log N$ where N is the number of pipes with backlogged traffic, and t_1 corresponds to the 1-pipe case just discussed. To estimte the weight of the logarithmic component, we ran another series of experiments generating traffic to 1000 different ports[3], passed to 1000 bandwidth-limited pipes. This resulted in $C'_r - B' = t_1 + k \log 1000 \approx 1500$ ns, which is less than twice the value of $C_r - B = t_1 \approx 840$ ns.

[3]The local source, in this experiment, is more time-consuming ($A' = 1704$ ns vs $A = 643$ ns) because it has to send packets to non-connected sockets.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an overview of emulation solutions used in networking research, and discussed in detail Dummynet emulator. In addition to its current features, we have analysed its scalability, accuracy and performance and presented measurement results showing its behaviour in various operating conditions.

Even though it is a mature and already widely used piece of software, we have recently done some enhancements on Dummynet, which include a Linux and OpenWRT port, the inclusion in PlanetLab as in-node emulator, support for more accurate modeling of MAC layers. Upcoming work includes a Windows port, and support for dynamically loadable packet scheduling algorithms.

This collection of features should help researchers in doing live testing of their protocols and applications, without being constrained by the use of a specific OS platform, or by the limited features of the tool they use.

6. REFERENCES

[1] Anuesystems. http://anuesystems.com/Products_NetworkEmulator_Ethernet.shtml.

[2] Dummynet references according to citeseer. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.2969.

[3] Lanforge-ice. http://www.candelatech.com/lanforge_v3/datasheet.html#ice.

[4] The onelab2 project. http://www.onelab.eu/.

[5] The ns-2 Network Simulator. http://nsnam.isi.edu/nsnam/index.php.

[6] The NS-3 Network Simulator. http://www.nsnam.org/.

[7] S. Agarwal, J. Sommers, and P. Barford. Scalable network path emulation. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 219–228, Washington, DC, USA, 2005. IEEE Computer Society.

[8] M. Carbone and L. Rizzo. Adding Emulation to PlanetLab Nodes. *CoNEXT09, Rome 1 Dec 2009*, 2009.

[9] M. Carson and D. Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[10] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[11] 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.

[12] 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. ACM, 2007.

[13] S. Hemminger. Network emulation with NetEm. In *Linux Conf Au*, 2005.

[14] 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.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek. The Click modular router. ACM Transactions on Computer Systems 18(3), August 2000, pages 263-297.

[16] 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.

[17] L. Rizzo. Dummynet home page. http://info.iet.unipi.it/ luigi/dummynet/.

[18] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[19] 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)*, pages 3224–3231, June 2007.

[20] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[21] 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.

[22] I. Yeom and A. N. Reddy. Ende: An end-to-end network delay emulator tool for multimedia protocol development. *Multimedia Tools Appl.*, 14(3):269–296, 2001.

[23] 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.