New developments in link emulation and packet scheduling in FreeBSD, linux and Windows

Luigi Rizzo, Università di Pisa

March 29, 2010

Summary

Some recent, network related projects at UNIPI:

- **The dummynet emulator: new features, performance, Linux and Windows ports**
  (mostly supported by the ONELAB2 project - European Commission)
- **Fast packet scheduling algorithms: QFQ**
  (joint work with Fabio Checconi and Paolo Valente, partly supported by the NETOS project - Univ. di Pisa)

Dummynet

Why emulation

Emulation is a standard tool in protocol and application testing. It gives you:

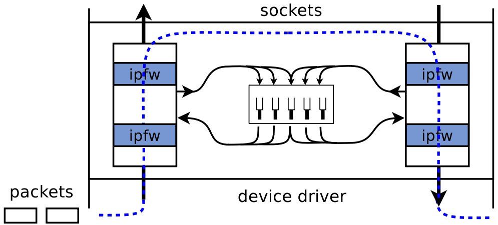
- ease of configuration/setup;
- reproducibility;
- more realistic results than simulation.

Several existing options:

- dummynet, NISTnet, tc+netem, netpath...

Dummynet

Dummynet is a network emulator developed in 1997 on FreeBSD, and substantially revised in recent years.
Now available on FreeBSD, OSX, Linux/Openwrt, Windows.

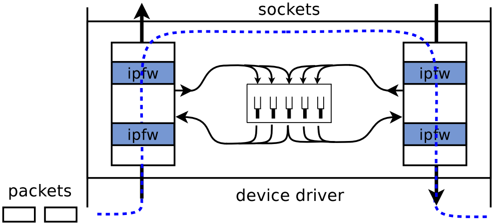


- intercepts packets in various points of the protocol stack;
- passes packets through a classifier and then to pipes, which model communication links;
- on exit, packets are reinjected in the protocol stack or in the classifier.

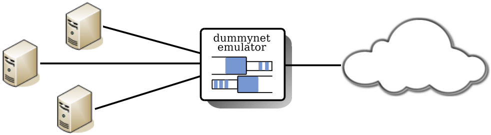
User interface

/sbin/ipfw is the main user interface for the system.
Use is very simple:

```
ipfw add 100 pipe 1 out dst-ip 1.2.3.4
ipfw add 100 pipe 2 in src-ip 1.2.3.4
ipfw pipe 1 config bw 256Kbit/s delay 12ms
ipfw pipe 2 config bw 4Mbit/s delay 2ms
```



Main applications

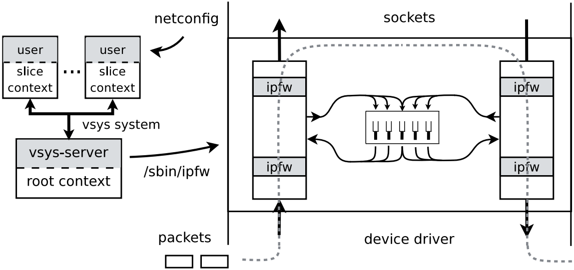


- link emulator (protocol/app testing):
  - in-node emulator (workstation, Planetlab);
  - transparent bridge;
- local traffic shaping:
  - share or reserve bandwidth for certain apps;
  - outgoing or incoming traffic shaping;
- traffic shaper in testbeds (Emulab/Planetlab) or ISPs:
  - must scale to thousands of pipes;
  - needs extra features for quick classification/demux.

Emulation in Planetlab

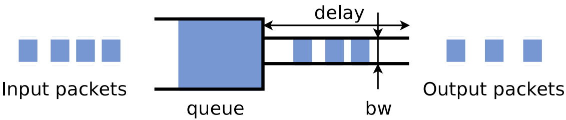
As part of the ONELAB2 project, we added dummynet as an in-node emulator in Planetlab:

- users can define independent emulated links;
- a frontend hides the complexity of configuration:
- **client** and **server** modes create typical configurations.



`netconfig config client 22,80 IN bw 6Mbit/s OUT bw 256Kbit/s`

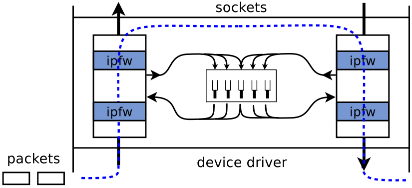
Dummynet internals: Pipes



- Only model basic features of a link:
  - queue with configurable size and management policy (FIFO, RED);
  - programmable link bandwidth;
  - deterministic propagation delay;
- avoid non deterministic behaviour:
  - do not deal with error/loss/delay models;
  - use real traffic to cause perturbations;
- ... except for some useful features:
  - random packet drop and random rule match;
  - you don't have to use them if you don't like the model.

Classifier

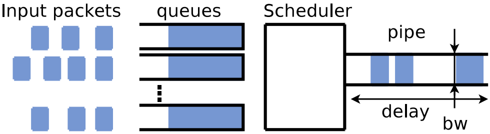
A classifier is used to send traffic to different pipes.



- we use FreeBSD's ipfw, which is easy to use and has a large number of packet matching options;
- ipfw has been extend with custom features:
  - multiple passes, to emulate complex networks;
  - probabilistic match, to emulate multipath and reordering;
  - table lookup, to speed up classification and dispatch.

Beyond pipes: queues, schedulers, links

More complex configurations require to split a pipe in its components – queue, scheduler, link – so we can:



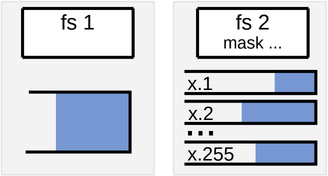
- attach multiple queues to one scheduler;
- configure scheduler features (algorithm, weights, etc.);
- model more complex links (e.g. radio).

Per-flow queues / Flowsets

A flowset is an abstraction used to model per-flow queues. It has several attributes:

- a *flow-mask*, used to create per-flow queues;
- a *scheduler* to which queues are attached to;
- weight/priority and other scheduling parameters;

```
ipfw queue 1 config sched 5 weight 10
ipfw queue 2 config sched 5 mask dst-ip 0xff weight 1
ipfw add 100 queue 1 src-ip luigi-pc
ipfw add 100 queue 2 src-ip my-subnet/24
```



Links

Links can model more than bandwidth and delay:

- uniform random loss;

```
ipfw pipe 1 config plr 0.06 // 6% loss on this link
```

- reordering (through probabilistic matching):

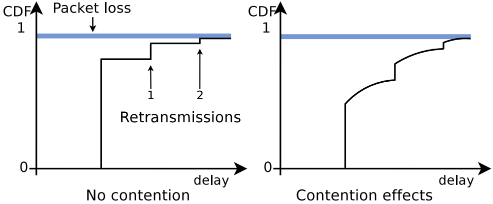
```
// 30% of packets go to pipe 1, 70% go to pipe 2
ipfw add 100 prob 0.3 pipe 1 dst-ip 1.2.3.4
ipfw add 100 pipe 2 dst-ip 1.2.3.4
ipfw pipe 1 config delay 100ms
ipfw pipe 2 config delay 20ms
```

- MAC overheads (preambles, contentions, link-level rxmit):
  - use *profiles* or model the MAC as a scheduler.

Link Profiles

Profiles model the extra air-time for a packet transmission

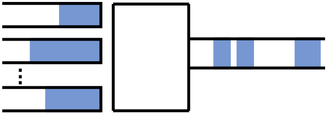
- an empirical function gives the distribution of extra air-time;



- not tied to a specific technology. Can be used for wireless or wired links of various kinds;
- can model low level features (preambles, inter-frame gaps...) or more complex ones (contentions, retransmissions, collisions);
- of course it is not as precise as full emulation.

Schedulers

Schedulers arbitrate access of multiple flows to the same link



- newly designed API supports configurable schedulers: FIFO, DRR, PRIO, WF2Q+, QFQ, KPS;
- a MAC layer is a scheduler, too. An 802.11b scheduler will be available shortly;
- schedulers have masks, too:

```
ipfw queue 1 config sched 5 weight 10 // used for ssh
ipfw queue 2 config sched 5 weight 1  // all other traffic
ipfw add 100 queue 1 out proto tcp src-port 22,53
ipfw add 100 queue 2 out
// each /24 subnet has its own instance
ipfw sched 5 config type QFQ mask src-ip 0xffffff00
```

Schedulers (cont)

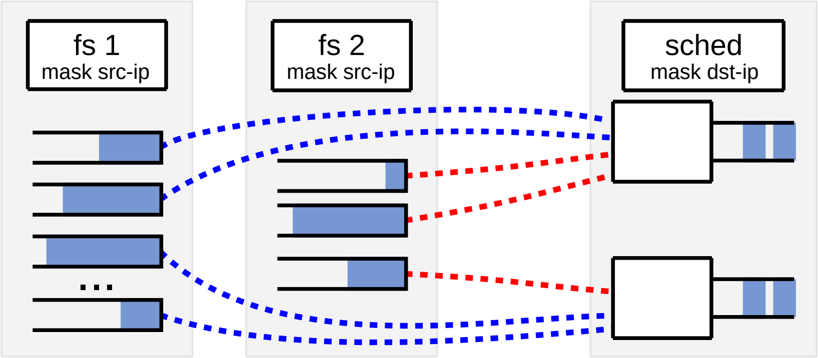
The scheduler API makes dummynet a tool for testing schedulers, too:

- adding a new scheduler is straightforward;
- you can concentrate on your algorithm, don't have to worry about classification, getting traffic, locking, etc..

```
> wc dn_sched*.c
    120     553    3766 dn_sched_fifo.c
    229     939    6367 dn_sched_prio.c
    653    2225   16724 dn_sched_kps.c
    864    3466   23302 dn_sched_qfq.c
    307    1110    7297 dn_sched_rr.c
    373    1854   12080 dn_sched_wf2q.c
```

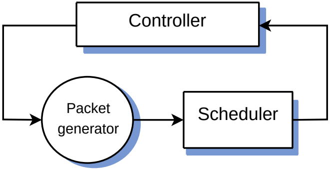
Overall structure

Relation between flowsets, masks, queues and schedulers.



Testing framework

We have support to run schedulers in user space.



- generate traffic for a programmable number of flows, packet size and weight distribution;
- carefully control the operating point of the scheduler;

```
./test -alg rr -qmin 4n -qmax 30n -flowsets 1::512,8::64
dn_rr     n 5004288 10000000 time 0.683968  136.676
./test -alg qfq -qmin 4n -qmax 30n -flowsets 1::512,8::64
dn_qfq    n 5004288 10000000 time 0.974142  194.661
./test -alg kps -qmin 4n -qmax 30n -flowsets 1::512,8::64
dn_kps    n 5004288 10000000 time 2.855963  570.703
```

Accuracy

At least three main factors influence the accuracy of emulation:

- timer accuracy (20 μs .. 1 ms or less);
- competing traffic (120 μs .. 1.2 ms per competing link);
- Operating System interference (virtually unbounded; normally in the 30 .. 200 μs range).

Accuracy can be improved addressing these three factors.
100 μs is a reasonable target on modern hardware.

Performance

Per-packet processing is the main factor limiting performance.

- detailed analysis in April 2010 CCR paper;
- split classifier + scheduling + emulation cost;
- classifier cost is C + O(R) (number of rules). Normally 400 .. 1000 ns with up to 20 rules.
- scheduling from $O(1)$ to $O(\log N)$;
- emulation: $O(\log N)$, 700 .. 1500 ns with 1 .. 1000 flows.

Overall, 2-3 μs/pkt on entry level PC hardware.

Porting

Dummynet has been recently ported to Linux and Windows.

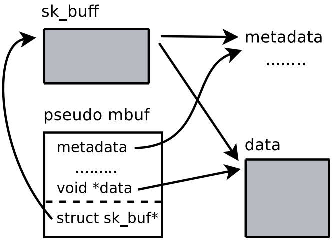
- We use the same codebase for all platforms;
- very little conditional code (except in headers);
- glue libraries to map FreeBSD kernel APIs to underlying OS APIs.

Main differences between platforms:

- internal packet representation;
- locking;
- packet filtering hooks;
- timers (API and resolution);
- module loading/unloading;
- userland/kernel communication.

Packet representation

In-kernel packet representation is similar in principle, different in details between BSD (mbufs), Linux (skbufs) and Windows (NDIS_PACKET).



- create mbuf lookalikes on entry, fill with metadata from native representation;
- internally, only use mbufs;
- destroy the wrapper on exit.

Locking and other OS APIs

- mostly dealt with through macros, preprocessor magic and wrapper functions;
- a 1:1 mapping between equivalent functions was almost always possible;
- hardest part was *locating* the right API to use (e.g., ExSetTimerResolution() on Windows);
- changing kernel APIs are very challenging too (Linux netfilter API is a moving target even within 2.6.X);

Availability and Credits

See http://info.iet.unipi.it/∼luigi/dummynet/

Supported operating systems:

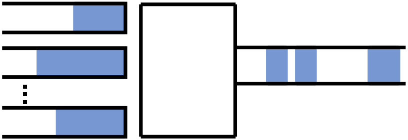
- FreeBSD (since 1998), OSX (2006)
- Linux/OpenWRT (2009)
- Windows XP, Windows 7 (2010)

Credits:

- Marta Carbone (Linux port)
- Fabio Checconi (QFQ, KPS)
- Riccardo Panicucci (scheduler API)
- Francesco Magno (Windows port)

O(1) packet scheduling at high data rates

O(1) packet scheduling at high data rates

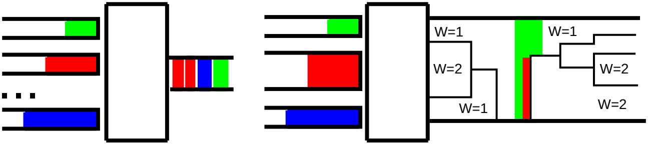


Why do we care about packet scheduling ?

- arbitrate access to common resources;
- provide service guarantees and resource isolation;
- overprovisioning is not always possible/desirable, today's CPUs are too fast;
- links are very fast too, so schedulers must keep up with high data rates and number of flows.

Problem setting and definitions

Many definitions for Service Guarantees. We consider the deviations of our actual scheduler (Packet System) from the service offered by an Ideal Fluid System.



- each flow has a weight Φ_i, and *should* receive a fraction $\Phi_i / \sum_j \Phi_j$ of the total link capacity at any time;
- the Fluid System serves all flows simultaneously;
- the Packet System serves one packet at a time, is non preemptable, online, and possibly work-conserving;

Service Guarantees

Because of its nature, a Packet System cannot guarantee perfect sharing at all times. The magnitude of deviations is an indicator of the quality of the scheduler.

- various quality metrics including

$$\text{B-WFI} = \max_{k,\Delta t} [\Phi_k W(\Delta t) - W_k(\Delta t)]$$

- in the best possible Packet System (e.g. WF^2Q), B-WFI = 1 MSS (*Optimal B-WFI*);
- tradeoff between guarantees and complexity:
  Xu-Lipton 2002: optimal B-WFI requires $\Omega(\log N)$ time;
  Valente 2004: an $O(\log N)$ version of WF^2Q;
- breaking the $O(\log N)$ barrier implies relaxed guarantees.

State of the art of fast schedulers

- Priority-based schedulers are fast but give no guarantees except to the flow with highest priority;
- Round Robin schedulers have $O(1)$ time but poor guarantees ($O(N)$ B-WFI);
- some *timestamp-based* schedulers such as WF^2Q give optimal service guarantees in $O(\log N)$ time;
- approximated variants of timestamp-based schedulers (KPS - Karsten 2006; GFQ - Stephens,Bennet,Zhang 1999) have near-optimal guarantees and $O(1)$ time complexity (but several times slower than RR).

Our result

QFQ is a practical O(1) approximated timestamp-based scheduler with

- near-optimal guarantees (B-WFI $\sim$5 MSS);
- truly constant complexity, independent of number of flows and configuration parameters;
- uses very simple CPU instructions;
- 110 ns/pkt on common workstations, compared to 55 ns for DRR and 400 ns for KPS.

Fair Queueing in software (or inexpensive hardware) is feasible at GBit/s rates.

QFQ overview

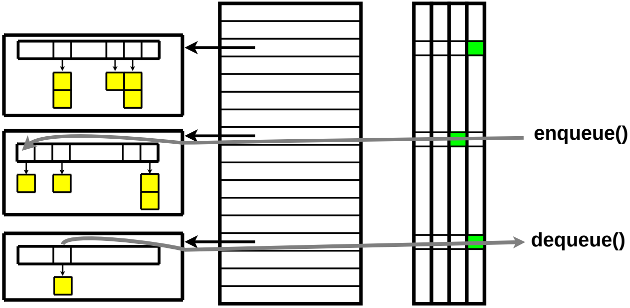
QFQ operates as other timestamp-based schedulers:

- track the behaviour of a Fluid System;
- for each packet, compute *Virtual* Start and Finish times;
- schedule in Finish time order among packets that are i) available and ii) already started in the Fluid Server.

The sorting steps imply a $O(logN)$ complexity.

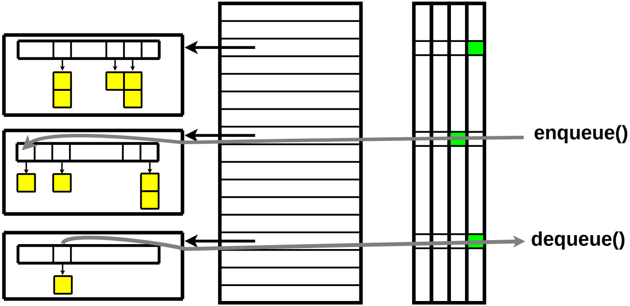
- use approximated sorting to reduce complexity;
- use careful approximations to preserve guarantees;
- use extra data structures to reduce constants.

QFQ data structures



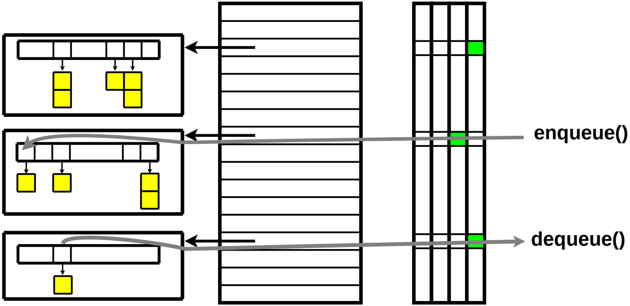
- Approximated sorting based on rounded timestamps and splitting flows into a constant number of groups;
- flow i belongs to group $\lceil \log_2 L_i/\Phi_i \rceil$;
- rounding intervals grow exponentially.

QFQ data structures – sorting



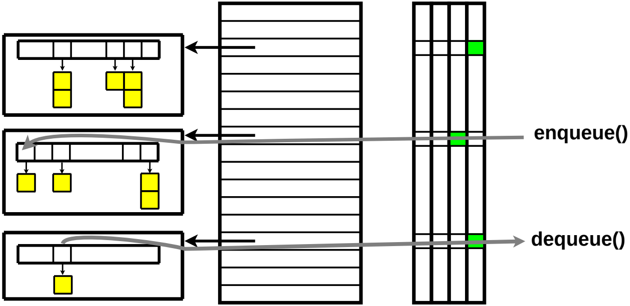
- Use approximate timestamps for sorting, but exact values when computing timestamps;
- within each group, there is only a finite number of slots, so we can use bucket sort;
- for selection purposes, use same $(F - S)$ for all flows in a group, so the order on F and S is the same.

QFQ data structures – selection



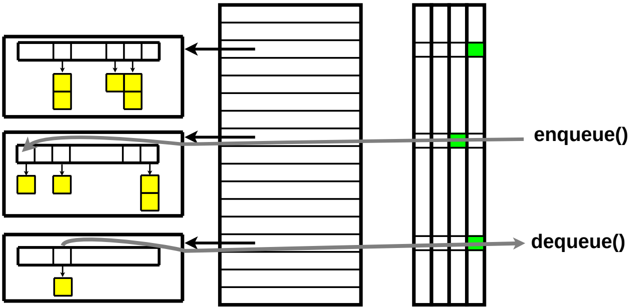
- Manage four Set of Groups. In each set, index reflects Virtual Time ordering;
- the eligible flow with minimum F can be found with one FFS instruction instead of scanning the groups;
- moving groups between sets does not require loops, either.

QFQ – enqueue



- bucket-insert in the group;
- update group state and sets.

QFQ – dequeue



- locate first bit in set ER;
- serve first flow in the first slot of the corresponding group;
- possibly put the flow in a new slot;
- update group state and sets.

Service guarantees

Service guarantees for QFQ:

$$\text{B-WFI}^k = 3\phi^k\sigma_i + 2\phi^k L$$

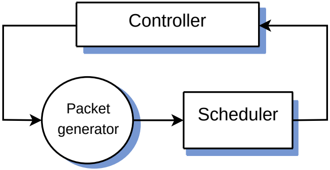
(remember that $L^k/\Phi_k < \sigma_i \leq 2L^k/\Phi_k$)

$$\text{T-WFI}^k = \left(3\left\lceil\frac{L^k}{\phi^k}\right\rceil + 2L\right)\frac{1}{R}$$

(R is the link's rate).

Experimental results

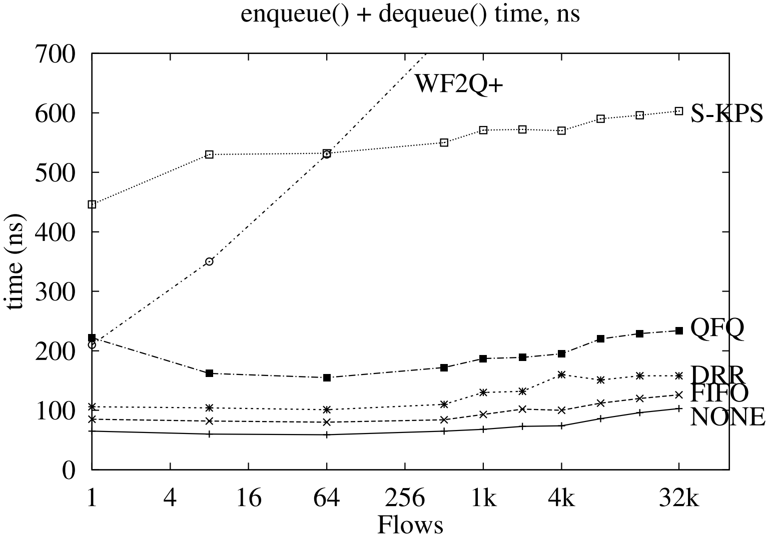
Measurements taken by running the kernel code in userspace:



- generate traffic for a programmable number of flows, packet size and weight distribution;
- carefully control the operating point of the scheduler;

```
./test -alg rr -qmin 4n -qmax 30n -flowsets 1::512,8::64
dn_rr     n 5004288 10000000 time 0.683968  136.676
./test -alg qfq -qmin 4n -qmax 30n -flowsets 1::512,8::64
dn_qfq    n 5004288 10000000 time 0.974142  194.661
./test -alg kps -qmin 4n -qmax 30n -flowsets 1::512,8::64
dn_kps    n 5004288 10000000 time 2.855963  570.703
```

Performance comparison – scalability



Mixed workloads

Measurement results in ns for an enqueue()/dequeue() pair and packet generation. Standard deviations are within 3% of the average.

Flows	NONE	FIFO	DRR	QFQ	KPS	WF2Q+
1	62	83	105	**221**	450	210
8	60	80	102	**163**	543	344
64	59	80	100	**158**	540	526
512	64	85	110	**175**	560	740
4k	74	102	157	**197**	590	1110
32k	62	117	147	**222**	601	1690
1:32k,2:4k,4:2k,8:1k,128:16,1k:1 flows						
mix	92	119	160	**255**	612	1715

Conclusions

- QFQ is a Timestamp-based scheduler with near optimal service guarantees and true $O(1)$ run time;
- 110 ns/pkt, only 2 times slower than RR and 4 times faster than comparable algorithms;
- already available as part of dummynet, together with several other schedulers;
- technical report and code at `http://info.iet.unipi.it/~luigi/qfq/`
- Joint work with Fabio Checconi and Paolo Valente;
- soon available as a Click module;

Future work

Future work:

- detailed performance analysis on low-end hardware (OpenWRT platforms);
- identify performance bottlenecks, memory access patterns;
- investigate feasibility of hardware implementations (including NETFPGA).

Links and further info

- For dummynet
  `http://info.iet.unipi.it/~luigi/dummynet/`
- For QFQ
  `http://info.iet.unipi.it/~luigi/qfq/`

For everything else, there's `www.google.com`