A new packet scheduling architecture for FreeBSD

Luigi Rizzo, Università di Pisa

May 13, 2010

Introduction

Packet scheduling is necessary when demand exceeds available resources. So far, FreeBSD had only limited packet scheduling support:

- ALTQ, three schedulers: PRI, CBQ, HFSC (only on output interfaces, device-specific);
- Dummynet, in/out but only one scheduler (WF2Q+);

We need something better, to support:

- more modern (and efficient) schedulers;
- more complex usage scenarios;
- research on packet schedulers;
- customer demand.

Overview of the talk

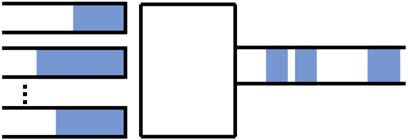
Topics covered in this talk:

- some packet scheduling theory, discussing architectures, service properties, and performance;
- scheduling support in Dummynet – user view (how to make use of the new features);
- scheduling support in Dummynet – kernel side (how to extend/build new schedulers).

Work supported by the ONELAB2 project - www.onelab.eu
QFQ is joint work with Fabio Checconi and Paolo Valente, partly supported by the NETOS project - Univ. di Pisa.

Packet scheduling background

Packet scheduling background

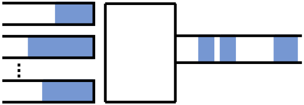


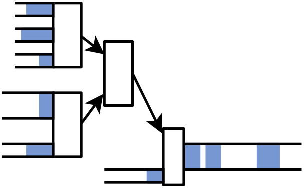
Why do we care about packet scheduling ?

- 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

- Basic building block: flat scheduler;
- characterised in terms of "service guarantees", memory and time complexity;
- basic schedulers can be composed in a hierarchical fashion;
- again, we can try to characterise the aggregate scheduler.





[Non-]Schedulers

Some solutions are not real schedulers:

- Policers:
  - each flow is given a maximum bandwidth (e.g. using leaky bucket, $O(1)$ time);
  - make sure total bandwidth $\leq$ link capacity $\rightarrow$ there is never congestion;
  - excess bandwidth is not used.
- Queue management policies (RED, RIO, ...):
  - randomly mark/drop packets as queue size grows;
  - responsive flows (e.g. TCP) will react reducing their rate. The feedback stabilizes the system;
  - ineffective on non-responsive flows.

Real Schedulers

- Priority-based schedulers:
  - simple to implement, often $O(1)$ time complexity;
  - one gets guarantees, other may starve.
- Round Robin (and variants):
  - also $O(1)$ time complexity;
  - no starvation, but $O(N)$ delays due to RR policy.
- Fair Queueing (and variants):
  - small, well defined service/delay guarantees;
  - $O(\log N) \ldots O(1)$ time complexity.
- Hierarchical schedulers: compositions of the above.
  - much harder to analyse;
  - often, $O(N)$ time complexity.

Priority and Round Robin

Each flow is assigned a Priority. Flows are served in strict priority order (Round Robin within the same priority).

- priority management is $O(1) \dots O(\log N) \dots O(N)$ (`ffs()`, binary heap, linear scan);
- Round Robin management is always $O(1)$;

In Round Robin variants (Deficit RR, Weighted RR), a flow's "weight" indicates the share of bandwidth it should receive:

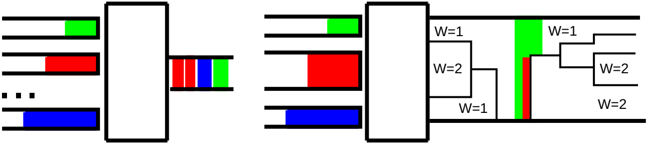
- in each slot, give service proportional to the weight;
- inherent $O(N)$ delay and burstiness:

```
...  A B CCCCCC D E F ...  Z A B CCCCCC ...
```

- reducing the delay requires more complex (and time-consuming) data structures to serve high-weight flows more often.

Proportional share

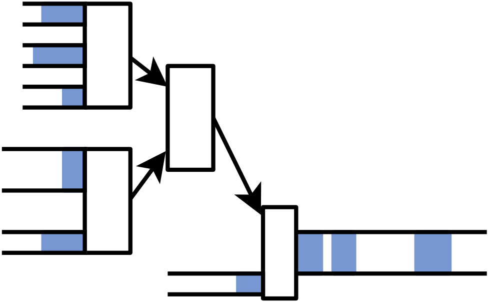
Proportional share schedulers try to emulate, on a Packet-by-Packet basis, the behaviour of a “Fluid System”:



- label each flow with a weight Φ_i;
- assign bandwidth to backlogged flows proportionally to their weight: $\boxed{R_i = R\Phi_i / \sum_{j \in B} \Phi_j}$
- ideally, this should be true over any time interval;
- in practice, some difference is unavoidable;
- the emulation cost ranges from $O(1)$ to $O(N)$ depending on the algorithm.

Hierarchical schedulers

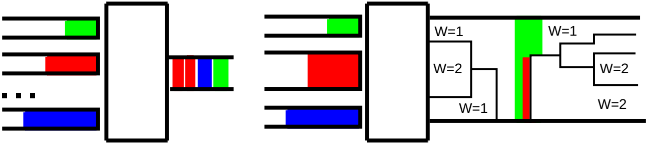
- Useful to implement different aggregation of resources



- at least $O(\text{depth})$ complexity for the infrastructure;
- very likely to go to $O(N)$ if nodes do not scale (e.g. need to explore all children on a dequeue).

Service Guarantees

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 (2)

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

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

  B-WFI is the maximum lag in terms of service, there is a similar definition in terms of time (T-WFI).
- In the best possible Packet System (e.g. WF^{2}Q), 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^{2}Q;
- breaking the $O(\log N)$ barrier implies relaxed guarantees.

Do we really care about WFI ?

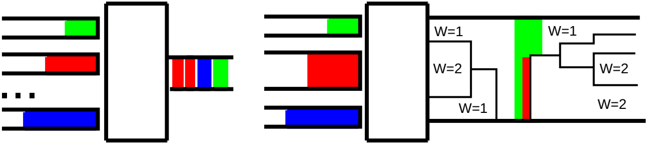
Is the WFI an invention of bored academics ?

- No. A large WFI means that a flow needs to have a large queue to store traffic while it is not served.
- Example: a round-robin scheduler has $O(N)$ WFI. With 50K queues, a flow using half of the link's capacity needs a queue of 25K packets;
- traffic goes out in huge bursts;
- the burstiness propagates downstream.

Timestamp based schedulers

Timestamp based schedulers emulate a fluid scheduler as follows:

- compute, at each time, how much service the flow would receive in the Fluid system (Virtual Time);
- mark packet with their Start and Finish time in the fluid system;
- schedule packets according to their Finish times;
- to reduce burstiness, do not consider packets that have not started yet in the fluid system (Eligibility)



Guarantees of Timestamp based schedulers

Computing the timestamps, and sorting on them, is what creates the complexity bound $\Omega(\log N)$.

- Schedulers using exact timestamps (WF^{2}Q) have B-WFI$=$1 MSS (optimal B-WFI);
- cannot do better due to non-preemption, and work-conserving policy;
- the use of approximate timestamps reduces complexity, but causes slightly larger lags:
  B-WFI $\leq c\cdot$MSS for some constant c (near-optimal B-WFI).

State of the art of packet 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^{2}Q 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).
- QFQ (Checconi, Valente, Rizzo 2010) has $O(1)$ time and is almost as fast as RR.

QFQ features

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;
- it's real, not just a paper design;
- 110 ns/pkt on common workstations, compared to 55 ns for DRR and 400 ns for KPS.
- more details on "GoogleTechTalks qfq" and `http://info.iet.unipi.it/~luigi/qfq/`

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

Choosing the right scheduler

We have many algorithms with different features. How do we choose ?

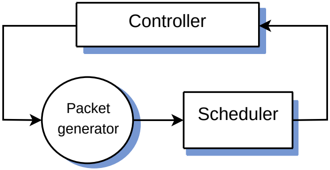
- the decision depends a lot on the operating conditions. For large N, asymptotic complexity is important. For small N, or certain weight distributions, guarantees or actual run times are more important;
- theory can tell us about worst-case service guarantees and asymptotic complexity;
- we need measurements to determine the run-time constants.

Testing scheduler performance

In-kernel measurements are very hard:

- very difficult to set up a suitable test environment;
- packet generation, reception, device drivers and other costs dominate the measurements;

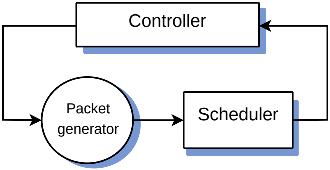
We make our measurements by running the kernel code in userspace:



- can easily generate traffic at 40Mpps and more (compare to 200..500Kpps on the wire for the same hardware);
- can generate traffic for a programmable number of flows, packet size and weight distribution;
- can control the operating point of the scheduler during tests.

Test harness

The kernel sources also include some test code to build and run schedulers in user space. This is very useful both for correctness and performance testing.



Controller

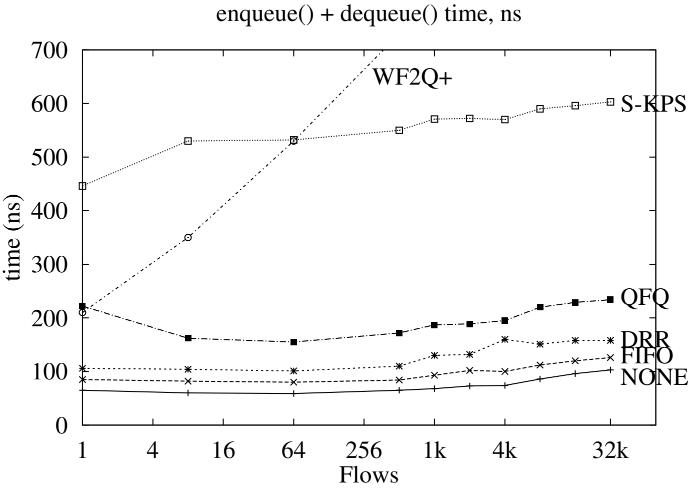
Packet generator

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
```

Scheduler Performance comparison

Sample results on a 2.3GHz Athlon with 667MHz memory.



Scheduler Performance comparison (2)

Same data in tabular format – average time (ns) for an enqueue()/dequeue() pair and packet generation. StD 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

More on performance

CPU type and speed, memory (and cache) size and speed has a big impact on performance. As an example, QFQ with 32k flows:

- 220ns on 2.3GHz Athlon w/ 666 MHz RAM;
- 110ns on fast Nehalem w/ 1.3GHz RAM;
- 8000ns on Asus WL500GP (240 MHz MIPSEL);

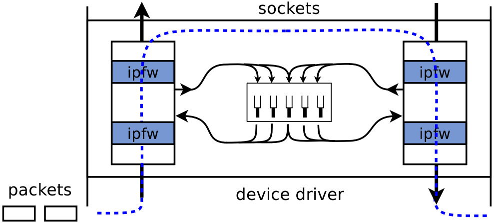
Remember that scheduling is only one block in the packet processing chain. On the same 2.3GHZ Athlon:

- 500..1000ns within the device driver;
- another 500..2000ns within ipfw and IP layer.

Packet scheduling in Dummynet – User view

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.

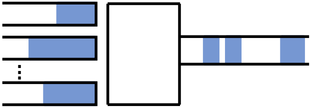


- intercepts packets in various points of the protocol stack;
- passes packets through a classifier (ipfw) and then to pipes or queues, which model communication links;
- on exit, packets are reinjected in the protocol stack or in the classifier.

Packet scheduling using dummynet

- Use dummynet to create bottleneck link(s):
  - the bottleneck can be a close approximation of another bottleneck downstream;
  - it can be used to enforce service limitations;
  - we can also enforce limitations on incoming traffic.
- use the classifier to select packets subject to scheduling, and group them into flows.

User interface



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

- define a link and its scheduler

```
ipfw sched 4 config type qfq bw 4Mbit/s
```

- define the weight of each queue

```
ipfw queue 1 config weight 10 sched 4
ipfw queue 2 config weight 3  sched 4
```

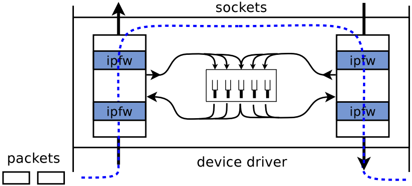
- send traffic to the queues using the classifier

```
ipfw add 100 queue 1 out src-ip 1.2.3.4
ipfw add 100 queue 2 out src-ip 1.2.3.5
```

More details later.

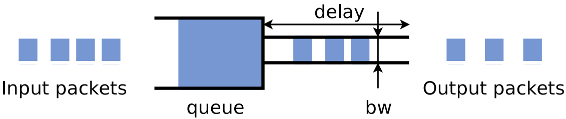
Classifier

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.

Pipes



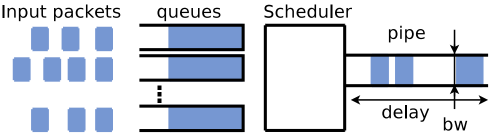
A pipe models basic features of a link:

- queue with configurable size and management policy (FIFO, RED);
- programmable link bandwidth;
- deterministic propagation delay;

In this context (scheduling) we are only interested in bandwidth.

Queues and Schedulers

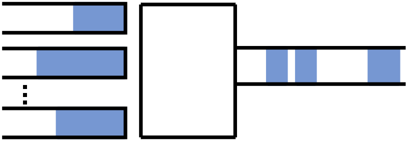
We can split a dummynet pipe into components – queue, scheduler, link – so we can:



- attach multiple queues to one scheduler;
- configure scheduler features (algorithm, weights, etc.);
- dynamically generate queues and scheduler instances.

Queues

A queue contains all packets for the same flow;



- identified by a numeric ID, carries a `weight` and other per-flow scheduling parameters;
- multiple queues are attached to one scheduler
- ipfw rules send traffic to the queues.

```
ipfw queue 1 config sched 5 weight 10
ipfw queue 2 config sched 5 weight 1
ipfw add 100 queue 1 src-ip 1.2.3.4
ipfw add 100 queue 2 src-ip 1.2.3.6
```

Dynamic creation of queues

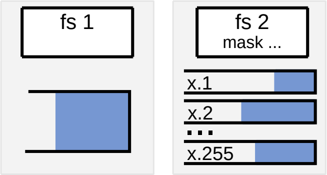
To configure multiple, per-flow queues with similar features we can use a *flow-mask*:

```
ipfw queue 1 config weight 4 sched 5
ipfw queue 2 config weight 1 sched 5 mask dst-ip 0xff
```

(this “template” is called a flowset in the code).

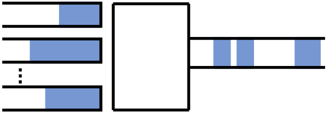
- the mask is applied to the 5-tuple of each packet;
- a new queue is created for each different value after masking;

Weight and other parameters are the same for all queues created through masking. In particular, all such queues talk to the same scheduler.



Schedulers

Schedulers arbitrate queues accessing the same link



- users can define the scheduler type and link speed
  `ipfw sched 5 config type QFQ bw 4Mbit/s`
- currently available choices are FIFO, DRR, PRIO, WF2Q+, QFQ, KPS;

Schedulers can also be used as generic flow-processing hooks (e.g. for deep packet/flow inspection, ...).

Dynamic creation of schedulers

Schedulers have a scheduler mask, used for dynamic creation of scheduler instances:

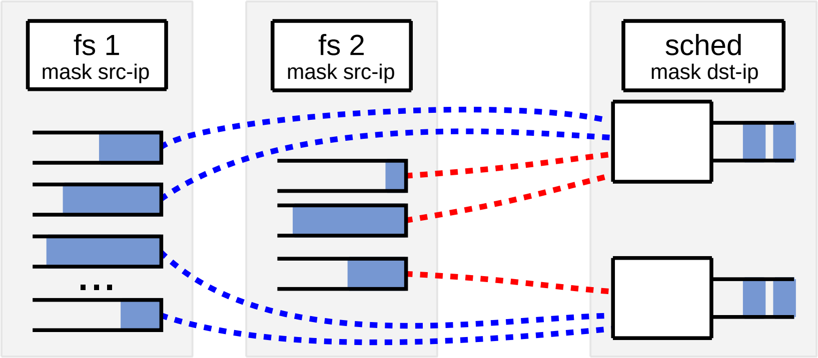
```
ipfw sched 5 config type QFQ bw 4Mbit/s mask src-ip 0xff
```

- the mask is applied to the 5-tuple of packets;
- a new scheduler instance is created for each value after masking;
- the various instances do not share anything (unlike dynamic queues, which are attached to the same scheduler);

Useful e.g. for ISPs with multiple independent customers.

Overall structure

Relation between flowsets, masks, queues and schedulers.



Advanced configurations

Configurations should exploit masks and ipfw tables to reduce the cost of the classifier.
Often, one table per direction can be used for most of the dispatching:

```
ipfw add queue tablearg out src-ip table(1)
ipfw table 1 add 1.2.3.0/24 20 // this goes to queue 20
ipfw table 1 add 1.2.3.8 21 // privileged IP to queue 21
ipfw table 1 add 1.2.4.0/20 25 // this goes to queue 25
...
// flowset 20 creates one queue per IP
ipfw queue 20 config sched 3 weight 5 mask src-ip 0xff
ipfw queue 21 config sched 3 weight 20
// flowset 25 creates one queue per /24 subnet
ipfw queue 25 config sched 4 mask src-ip 0xf00
```

Summary (user view)

- use ipfw rules to pass packets to queues;
- ipfw tables very useful to produce compact and efficient rulesets;
- use masks on queues and schedulers to dynamically create instances of flows and schedulers with similar attributes;
- pick one of many schedulers according to your requirements.

Packet scheduling in Dummynet – Kernel view

Dummynet – Packet flow within the kernel

Packets going through dummynet normally follow this path:

- input interface or local source;
- pfil hooks ipfw_check_hook() ;
- ipfw processing ipfw_chk() ;
- initial dummynet dispatch dummynet_io() .
  Enqueue into the scheduler occurs here;
- delayed reinject dummynet_task(), dummynet_send() .
  Dequeue from the scheduler occurs here.

Dummynet – Internal data structures

Internally, most dummynet structures (including scheduler-related ones) are managed through hash tables:

- a global hash table contains flowsets. Initial packet dispatch always searches this table;
- a global hash table contains schedulers. This is used during configurations to attach queues to schedulers;
- per-flowset hash table contains queues. Looked up after applying masks;
- per-scheduler hash table contains scheduler instances. Looked up when a new queue is created.

A priority queue (heap) is used to store pending events.

Interfacing with schedulers

The packet scheduling infrastructure takes care of all common operations:

- module management;
- applying queue and scheduler masks;
- creation of queues and scheduler instances;
- locking and memory allocations;
- commonly used data structures (hash tables, heaps, hashing);

Schedulers only need to provide enqueue() and dequeue() handlers, plus a few callbacks called by constructors and destructors of the various data structures.

Packet scheduler descriptor and module glue

```
static struct dn_alg rr_desc = {
        _SI( .type = ) DN_SCHED_RR,
        _SI( .name = ) "RR",
        _SI( .flags = ) DN_MULTIQUEUE,
        _SI( .schk_datalen = ) 0,
        _SI( .si_datalen = ) sizeof(struct rr_si),
        _SI( .q_datalen = ) sizeof(struct rr_queue) - sizeof(struct dn_queue),
        _SI( .enqueue = ) rr_enqueue,
        _SI( .dequeue = ) rr_dequeue,
        _SI( .config = ) rr_config,
        _SI( .destroy = ) NULL,
        _SI( .new_sched = ) rr_new_sched,
        _SI( .free_sched = ) rr_free_sched,
...
}
DECLARE_DNSCHED_MODULE(dn_rr, &rr_desc);
```

Enqueue() and dequeue()

enqueue(si, q, m) enqueues mbuf m on queue q;

- normally, just enqueue packet into q, and possibly do some housekeeping on internal data structures;
- q is only a hint, the scheduler can put the packet somewhere else (e.g. priority or FIFO);
- return 0 on success, 1 on drop;

After enqueue, m = dequeue(si) is called repeatedly to return the next packet to transmit:

- m == NULL means no more packets queued;
- packets can be tagged as "to be dropped" (for schedulers that may lose packets, e.g. those emulating wireless links);
- special values can request to postpone a transmission (non work conserving schedulers).

Example scheduler code: enqueue

```
#ifdef _KERNEL
... a ton of kernel headers
#else
#include <dn_test.h>
#endif
...
static int
rr_enqueue(struct dn_sch_inst *_si, struct dn_queue *q, struct mbuf *m) {
        struct rr_si *si = (struct rr_si *)(_si + 1);
        struct rr_queue *rrq = (struct rr_queue *)q;
        if (m != q->mq.head) {
                if (dn_enqueue(q, m, 0)) /* packet was dropped */
                        return 1;
                if (m != q->mq.head) /* already backlogged */
                        return 0;
        }
        /* If reach this point, queue q was idle */
        if (rrq->status == 1) /* Queue is already in the queue list */
                return 0;
        /* Insert the queue in the queue list */
        rr_append(rrq, si);
        return 0;
}
```

Source file organization

Most files are in sys/netinet/ipfw/

- ip_dn_private.h basic data structures (queues, flowsets, scheduler instances);
- dn_sched.h scheduler API and related macros;
- dn_sched_FOO.c implementation of algorithm FOO;
- test/ code for testing schedulers in userland.

Headers and basic schedulers (RR, PRIO, ...) are heavily documented so they can be used as a reference to develop new modules.

Available Schedulers

The current set of schedulers covers a wide range of options: FIFO, DRR, PRIO, WF2Q+, KPS, QFQ.

- more are coming (e.g. an 802.11b/g scheduler);
- 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
```

Conclusions ...

theory	There are many ways to do packet scheduling;
theory	one size does not fit all;
user	the packet scheduling architecture in dummynet permits very flexible configurations;
user	works on Linux/OpenWRT and Windows, too (more on this tomorrow);
kernel	adding a new scheduler is relatively straightforward;
kernel	very useful tool for researchers on traffic scheduling;
kernel	and there is some testing harness so you can debug and evaluate your algorithms in user space.

… credits, and future work

Thanks to the OneLab project (www.onelab.eu), the NETOS project (info.iet.unipi.it/∼luigi/netos/) and Riccardo Panicucci who did a lot of the coding.

Future work will cover:

- more performance measurements;
- optimize generic code paths;
- support for hierarchical schedulers;
- more schedulers (e.g. 802.11b/g almost complete);

More info at
`http://info.iet.unipi.it/∼luigi/dummynet/`