**Note: Quantum tunneling is not supported.**
**(– quoting net/sched/sch_blackhole.c)**

# Traffic Shaping

**... mit Linux**

Florian Westphal

15.1.2010

1024D/F260502D <fw@strlen.de>
1C81 1AD5 EA8F 3047 7555
E8EE 5E2F DA6C F260 502D

## Zusammenfassung

Traffic Shaping erlaubt es, ausgehenden Traffic so zu beeinflussen, dass die Sendegeschwindigkeit bestimmten Vorgaben angeglichen werden kann. Auch kann die Leitungsauslastung verbessert bzw. bestimmte Arten von Netztraffic z.B. bevorzugt versand werden.

Unter Linux wird dies über die `NET_SCHED` (Traffic control/"QoS and/or fair queueing") Packet-Scheduler umgesetzt.

# Agenda

1. Begriffsdefinitionen

2. Shaping vs. Scheduling vs. Policing

3. Exkurs: Warteschlagentheorie

4. qdiscs

   - Einsortierung im Netzwerkstack
   - qdiscs in TC, beispiele
   - Classless/Classful queues
   - Komplexere Setups

5. Sonstiges

# Begriffe

- `skb` (`struct sk_buff`) Linux-Kernel Datenstruktur: repräsentiert ein Datenpaket
- `skb->priority` Member-Variable d. besagten Datenstruktur.
- TOS/DSCP: Feld im IPv4-Header, kann Flow/Pakettyp-Typ codieren
- Flow: eine "(TCP)-Verbindung"

# Traffic Shaping

Optimierung / Sicherung bestimmter Eigenschaften, z.B:

- Bandbreitenbeschränkung
- Senken der Latenz
- "Glättung" der Auslastung (vermeiden von Lastspitzen)
- erfolgt durch Verzögerung (delay) von bestimmten Paketen
- für (sinnvolles) Shaping muss der Flaschenhals bekannt sein ('next Hop hat 1-MBit-Uplink')
- nur für Egress (=ausgehenden) Traffic

# Scheduling

- Verbesserung d. Flow-Fairness
  - Verhindern, d. einzelne Flows/Endpunkte den Link "monopolisieren"
- Priorisierung v. bestimmtem Traffic
- bezieht sich auf zu sendende Pakete $\rightarrow$ Egress

# Traffic Policing

- verwerfen von Paketen bei Überschreiten eines bestimmten Grenzwerts
- kann daher auch bei Ingress (=eingehender) Traffic verwendet werden (keine ingress qdisc)
- shaping: Verzögern v. Traffic (zur einhaltung bestimmter Senderate)
- policing: Verwerfen von Paketen um unter bestimmter Bandbreitengrenze bleiben

# Warteschlangentheorie (Bedientheorie)

- zentrale Elemente:

  1. Bedienstelle (z.B. Ampel, Kasse, . . . )
  2. Warteraum

- Grössen: Eingangsstrom, Ausgangsstrom

- Eingangsstrom: Mittlere Anzahl Anünfte pro Zeiteinheit

- Warteschlange, wenn Bedienstelle besetzt (Eingangsstrom $>$ Ausgangsstrom)

# Warteschlange

- . . . legt fest, welche Pakete bevorzugt, verzögert bzw. verworfen werden
- beeinflusst daher auch ggf. die Sendereihenfolge (von Paketen)
- endliche Größe
- verschiedene Wartedisziplinen (= Queueing Disciplines): FIFO, Round-Robin, Selection-In-Random-Order, . . .
- umstellen: `tc` (aus dem `iproute2`-Paket).

# qdiscs

- qdisc: Algorithmus, welcher Warteschlange verwaltet
- Netwerkinterfaces haben (normalerweise) mindestens eine qdisc
  - `lo` (und Tunnelinterfaces) sind queue-less
- qdisc: letzte Station eines Paketes vor dem Versand
- Standard-qdisc: `pfifo-fast` (*p*acket FIFO)
- Unterscheidung zwischen classful und classless qdiscs

# qdiscs im Netzwerkstack

```
L3-Protokoll ->  net/dev/core.c:dev_queue_xmit(skb)
                        rc = qdisc_enqueue_root(skb, q);
                        qdisc_run(q);
                        return rc;
```

- Die qdisc "hängt" – grob vereinfacht – an '`struct net_device`'.
- Das Paket wird erst in die Warteschlange eingefügt (`qdisc_enqueue`)
- `qdisc_run` verschickt dann ein Paket in der Queue, oder auch nicht (abh. v. Algorithmus)

# qdisc im Netzwerkstack (cont).

- qdisc kann Pakete zwischenspeichern, z.B. falls Netzwerkinterface "busy"
- qdisc verwirft Paket ggf. (zumindest dann, wenn Queue zu voll)
- `qdisc_run` schickt die Pakete zur Hardware

```
net/sched/sch_generic.c:qdisc_run(qdisc q) {
    skb = dequeue_skb(q);
    dev = qdisc_dev(q);
    ret = dev_hard_start_xmit(skb, dev, ... );
    switch (ret) {
       case NETDEV_TX_OK: return qdisc_qlen(q);
       case NETDEV_TX_BUSY: return dev_requeue_skb(skb, q);
    }
}
```

# qdisc im Netzwerkstack (cont).

`qdisc_run()` läuft solange, bis entweder:

1. `dequeue_skb()` liefert kein Paket mehr (qdisc leer, throttling, . . . – Algorithmusabh.)

2. rescheduling nötig / Zeitquantum überschritten

3. Hardware "busy"

Bei 2 & 3: tx-softirq schedule

# Terminologie

- root qdisc - diejenige qdisc, die an ein Netzwerkadapter angeschlossen ist
- classless qdisc: qdisc ohne (konfigurierbare) interne Unterteilung

# classless qdisc Beispiel: pfifo-fast

- Jedes interface hat `pfifo_fast` qdisc, kann nur überlagert werden
- 3 Bänder:

  ```
  0 interactive
  1 best-effort / interactive bulk
  2 bulk
  ```

- Aufteilung erfolgt nach `skb->priority` (IPv4 setzt das aus TOS/DSCP)
- `pfifo_fast_ops->dequeue()` liefert ältesten `skb` aus niedrigsten Band
- *kein* delay – "nur" Scheduling

# classless qdisc Beispiel 2: (G)RED

- RED: Pakete probalistisch verwerfen, wenn qgröße $>$ Wert
- gut für Router mit (sehr) viel Traffic (Glättung der Leitungsauslastung)
- Unterstützt auch ECN-Markierung
- GRED: Pakete werden anhand TOS/DSMARK in Queues vorsortiert
- RED-Parameter können dann per-queue getrennt eingestellt werden
- GRED-alternative: `sched_drr` mit RED-Childs

# classless qdisc Beispiel 3: SFQ

(Stochastic Fairness Queueing)

- reines Scheduling: faire Behandlung aller Flows
  - verteilt Pakete auf FIFO (1024) Queues
  - enqueue: hash ermittelt Queue
  - dequeue: Round-Robin Abarbeitung der Queues
  - qdisc voll: paket aus der längsten queue wird verworfen
- nutzlos wenn Link-Flaschenhals woanders liegt (da scheduling only)

# Erste Schritte...

```
$ /sbin/tc -s qdisc show dev eth0
qdisc pfifo_fast 0: root bands 3 priomap  1 [ .. ]
 Sent 4933482 bytes 53025 pkt (dropped 0, overlimits 0 requeues 0)
 rate 0bit 0pps backlog 0b 0p requeues 0
```

(pfifo_fast kann nicht entfernt werden). qdisc Ändern: `tc qdisc add dev eth0 root bfifo limit 300` Parameter reparieren:

```
$ tc qdisc change dev eth0 root bfifo limit 1500000
```

```
$ tc -s qdisc dev eth0
qdisc bfifo 8001: dev eth0 root limit 1514000b
 Sent 80900 bytes 1044 pkt (dropped 0, overlimits 0 requeues 0)
```

8001: ist das (hier automatisch vergebene) qdisc-handle.
Löschen: `tc qdisc del dev eth0 root`

# Traffic Control Elemente

1. qdisc

2. Klassifizierung

   - bestimme Paketeigenschaften (Größe, Protokoll, TCP-Flags, . . . )
   - Flow-Verhalten, zB. "$>$ 1MB übertragen"
   - Applikationsprotokoll (Layer 7)

# Classful qdiscs

- Problem: Bestimmte Pakete/Flows unterschiedlich behandeln
- Lösung: Classful qdiscs
- Grob: mehrere qdiscs, welche (baumartig) miteinander Verbunden sind
- Nach "aussen" (d.h. aus Netzwerk-API Sicht) gibt es nur eine qdisc

# Classful Qdiscs (cont.)

- `enqueue()`: paket muss ggf. an eine der "inneren" qdiscs weitergereicht werden (Klassifizierung).

- Entscheidung erfolgt durch *filter*
  - Filter sind qdisc zugeordnet
  - Filter-Rückgabewert entscheidet über die Klassenzugehögigkeit
  - jede Klasse kann weitere Filter/Unterklassen haben

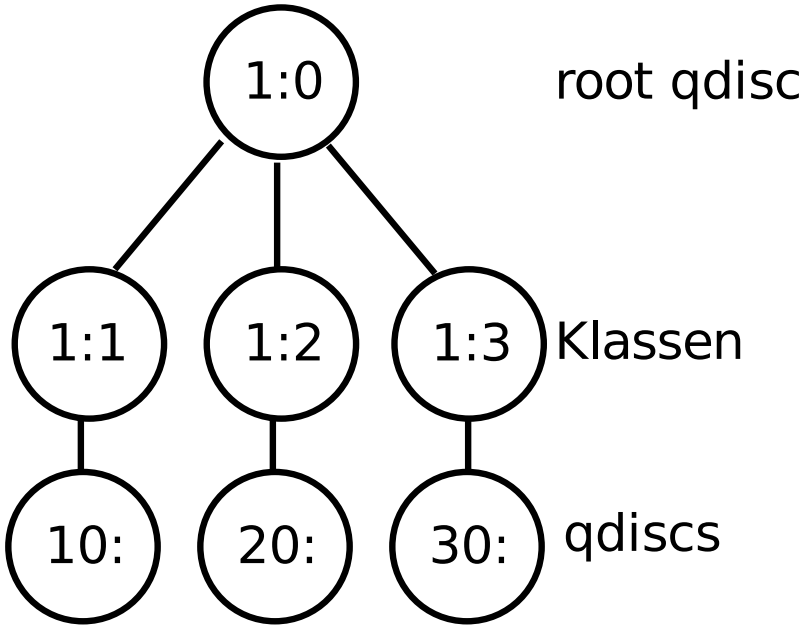
# Root, handle, sibling, parent...

- `pfifo_fast`: Standard root qdisc, pro interface
- handle: ID – identifiziert qdisc

  major/minor nummern:

- root qdisc – für gewöhnlich '$1:$' ($=1:0$).
- die Minor-Nummer einer qdisc ist *immer* 0.
- Klassen haben die Major-Nummer ihres parents ($=$ übergeordnete qdisc).
- en/de-queue erfolgt an der root qdisc

# Beispiel




Einfache Hierarchie: Aufteilung in 3 Klassen.
Einfachste (Classful) Qdisc hierfür: `prio`

# Umsetzung

Zuerst wird die root qdisc (hier `prio`) angelegt:

```
$ tc qdisc add dev eth0 root handle 1: prio
```

- `prio` erzeugt *implizit* 3 Bänder (ähnlich `pfifo_fast`) – aber als Klassen
- Im Gegensatz zu `pfifo_fast` kann jetzt jedes Band unterschiedlich behandelt werden:

```
$ tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 25kbit ...
$ tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate ...
```

# classless qdisc Beispiel 4: Token Bucket Filter

- reines Shaping, kein Scheduling
- Token: "Währung"
  1. Rate eingehender Pakete $=$ Rate eingehender Tokens $\rightarrow$ Pakete passieren Queue ohne Verzögerung
  2. Rate eingehender Pakete $\leq$ Rate eingehender Tokens $\rightarrow$ Siehe 1; tokens Akummulieren sich (bis zur maximalen Bucket-Grösse).
  3. Rate eingehender Pakete $\geq$ Rate eingehender Tokens $\rightarrow$ throttling, ggf. Packetdrops
- Im Gegensatz zu bisher vorgestellten qdiscs *nicht* "Work-Conserving": `->dequeue()` auf nicht-leere tbf-qdisc liefert nicht immer ein Paket

classful-Version: HTB (Hierarchial Token Bucket)

# Komplexere Setups

Man nehme eine classful qdisc:

- CBQ ("class-based-queueing")
- HTB ("CBQ-Nachfolger")
- HFSC
- DSMARK
- DRR (flexiblerer Ersatz für classless SFQ)

# Klassen hinzufügen

```
tc qdisc add dev eth0 root handle 1: drr
tc class add dev eth0 parent 1: classid 1:1 drr
```

Die Standard-qdisc ist in diesem Fall wieder `pfifo_fast`.
Ändern: `tc qdisc add dev eth0 parent 1:1 <qdisc-alg> ..`

Man kann classful qdiscs auch zu anderen classful qdiscs hinzufügen:

```
tc qdisc add dev eth0 parent 1:2 handle 10: drr
```

Bisher werden noch alle Pakete via eth0 weggeworfen. Warum?

# Auflösung

drr kann die eingehenden Pakete keiner Klasse zuordnen $\rightarrow$ drop
Pakete werden mit *filtern* klassifiziert. Manche classful qdiscs (z.B. `hfsc`)

kennen eine "default" Klasse:

```
tc qdisc add dev eth0 root handle 1: $name default 1:100
```

# filter syntax

1. Filter
   - `protocol` (für welches Netzwerkprotokoll der Filter greifen soll)
   - `priority` (= `preference` – legt Evaluierungsreihenfolge fest)
   - `parent` (Klassen-ID, zu der filter gehoeren soll)

   ```
   tc filter add dev eth0 protocol ip pref 10 parent 1:
   ```

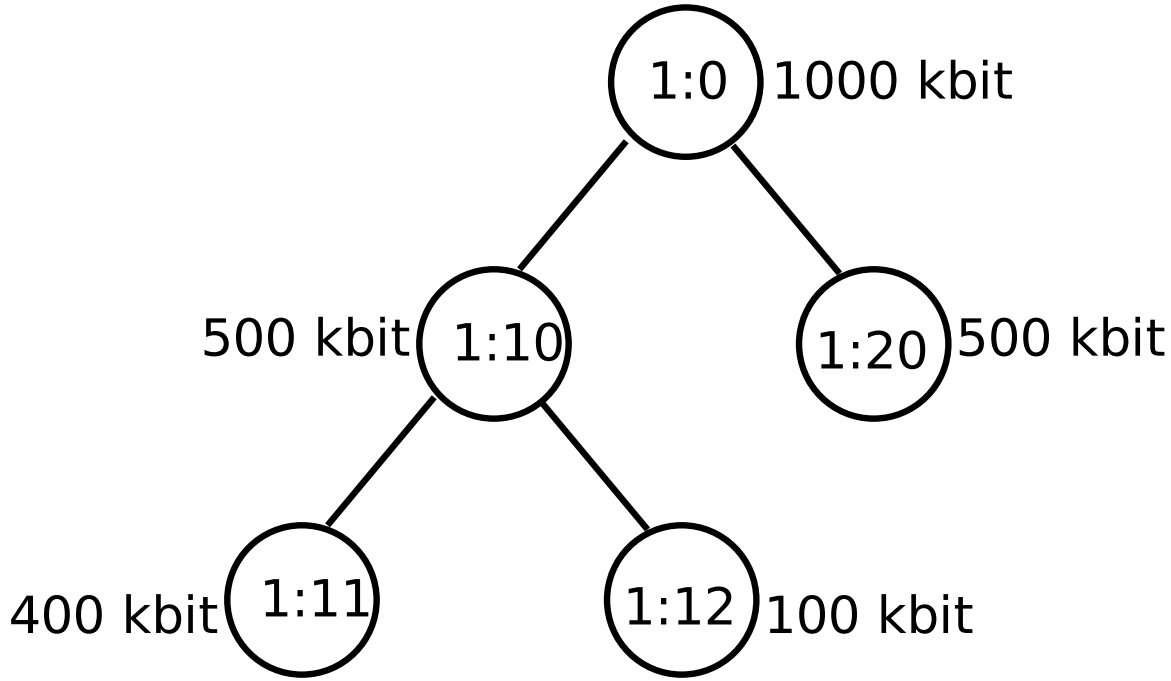
2. Selektor
   - u32 ("`match u32 00100000 00ff0000 at 0`" – pattern, mask, offset – mächtig aber komplex)
   - fw ("`handle 0xa2 fw classid 1:1`")
   - route, tcindex, . . .

3. Aktion – z.B. port mirroring/redirection (`tc .. mirred`)

# Shaping mit HFSC

- erlaubt autfteilung der Bandbreite und Vergabe von Latenzzeiten
- Modelliert mit sog. 'Service Kurven': $f(t)$ ist Senderate zum Zeitpunkt $t$
- Zwei Kriterien (Kurven):
  1. Link-Sharing Kurve: faire Verteilung (rel. zu benachbarten Klassen)
  2. Realtime-Kurve: (nur Leaf-klassen – einhaltung von Garantien)
- RT-Kurve hat höhere Gewichtung – Klasse kann Paket aufgrund RT-Kriterien auch senden, wenn dadurch Linksharing verletzt wird

# Beispiel Leitungskapazitäten aufteilen




Quelle: [HFSCL]

# HFSC

- jeder Klasse der (HFSC-)Hierarchie ist "Virtual-Time" zugeordnet
- Wenn Paket gesendet werden soll: (Leaf-)Klasse mit . . .
  - geringstem RT-Zeitpunkt
- . . . oder (falls abgelaufen bzw. nicht konfiguriert):
  - geringster VT
- auswählen. Danach wird in dieser (und allen Parentklassen incl. root) die VT erhöht.

# HFSC-Konfiguration

1. hfsc-anlegen: `tc qdisc add dev eth0 root handle 1: hfsc`

2. Klassen einfügen:

```
tc add class dev eth0 parent 1: classid 1:1 hfsc \
     [ [ rt SC ] [ ls SC ] | [ sc SC ] ] [ ul SC ]
```

   wobei `SC` (Service-Curve) `:= [ umax bytes dmax ms ] rate BPS` (=Eigenschaften)

   - ls: link-sharing, us: upper limit
   - Leaf-Klassen kann auch eine rt-Kurve zugewiesen werden
   - sc: erzeugt zwei identische rt- und ls-Kurven

# HFSC-Konfiguration (2)

```
tc qdisc add dev eth0 root handle 1: hfsc
tc class add dev eth0 parent 1: classid 1:1 \
   hfsc sc rate 1000kbit ul rate 1000kbit
tc class add dev eth0 parent 1:1 classid 1:10 \
   hfsc sc rate 500kbit ul rate 1000kbit
[ 1:20 ]
```

Zusätlich kann die max. Verzögerung (`dmax`) für bestimmte Grenze (`umax`) konfiguriert werden:

```
tc class add dev eth0 parent 1:10 classid 1:11 \
   hfsc sc umax 1500b dmax 53ms rate 400kbit \
                              ul rate 1000kbit
[1:12 rate 100kbit]
```

# Weitere Konfiguration ...

- es könnten nun weitere qdiscs angeschlossen werden, z.B. SFQ oder RED
- Verständnisfrage: Warum ist TBF als HFSC-Leaf keine gute Idee?

# Sonstiges...

- es könnten nun weitere qdiscs angeschlossen werden, z.B. SFQ oder RED
- Verständnisfrage: Warum ist TBF als HFSC-Leaf keine gute Idee?

# Sonstiges: mirred

mirred: mirror/redirect (kopieren/Umleiten auf anderes Interface)

```
tc qdisc add dev eth0 ingress
tc filter add dev eth0 parent ffff: protocol \
   all u32 match u32 0 0 action mirred egress mirror dev eth1
```

"mirred ingress" nicht implementiert $\rightarrow$ aber Umweg über
`tc qdisc add ... ingress` möglich

# Sonstiges: ifb (Intermediate Functional Block)

- pseudo-device, erlaubt shaping auch auf eingehenden Traffic

- `modprobe ifb; ip link set dev ifb0 up`

```
tc qdisc add dev eth0 ingress
tc filter add dev eth0 parent ffff: \
   protocol ip u32 match u32 0 0 flowid 1:1 action \
    mirred egress redirect dev ifb0
tc qdisc add dev ifb0
```

Einfachere Alternative zu inbound-shaping: `tc filter ... policy`

```
tc qdisc add ... ingress
tc filter add dev eth0 parent ffff: protocol ip u32 match ip \
    src 0.0.0.0/0 police rate 8192kbit burst 10k drop
```

# Sonstiges: netem

qdisc zum Testen von Netzwerkprotokollen

- packet delay, loss, duplizierung, reordering, . . .
- jitter, Datenkorruption, . . .
- Vorsicht vor 'loss'-Anwendung auf lokalem System: Netzwerstack 'sieht', wenn das Paket verworfen wurde und reagiert darauf
- daher besser auf Middlebox einsetzen

```
tc qdisc add dev eth0 root netem delay 100ms loss 0.1%
```

# Und zum Schluss... – EINVAL-Ursachen

```
RTNETLINK answers: Invalid argument
We have an error talking to the kernel
```

Beliebte Fehler:

- in `parent`-Parameter angegebenes qdisc-handle existiert nicht
- `tc class`, aber angegebene qdisc ist classless
- erforderlicher scheduler-Parameter Fe lt (parameter `help`, man page, lartc.org)

# Lektüre

LARTC "Linux Advanced Routing & Traffic Control",`http://www.lartc.org`

iproute2 Verzeichnis "doc/" in den iproute2-Sourcen, `tc/*.c`

RED S. Floyd, V. Jacobson. "Random Early Detection gateways for Congestion Avoidance.",
`http://www.aciri.org/floyd/papers/red/red.html`

HFSC Ion Stoica, Hui Zhang, T.S. Eugene Ng.
"A Hierarchial Fair Service Curve Algorithm for Link Sharing, Real-Time and Priority Services"
`http://www.cs.rice.edu/~eugeneng/papers/SIGCOMM97.pdf`

HFSCL Klaus Rechert, Patrick McHardy, "HFSC Scheduling mit Linux"
`http://klaus.geekserver.net/hfsc/hfsc.html`