TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

DIPLOMOVÁ PRÁCE

Traffic Control

Liberec 2006 Petr Klejna

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Diplomová práce

Vypracoval: Petr Klejna

Studijní program: M 2612 – Elektrotechnika a informatika

Obor: Automatické řízení a inženýrská informatika

Název tématu: Traffic Control

Vedoucí diplomové práce: Mgr. David Kmoch

Konzultant: RNDr. Pavel Satrapa, Ph.D.

Rozsah práce:

Počet stran:	64
Počet stran přílohy:	19
Počet tabulek:	17
Počet obrázků:	14
Počet grafů:	4

Datum: 6. ledna 2006

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum 6. ledna 2006 Podpis

Poděkování

Na tomto místě bych chtěl poděkovat Mgr. Davidu Kmochovi za odborné vedení, rady a připomínky k diplomové práci.

Mé poděkování rovněž patří rodině za podporu po celou dobu studia.

Anotace

Diplomová práce se zabývá řízením datových toků (Traffic Control) na počítačové síti v prostředí operačního systému Linux.

Cílem je poskytnout informace k praktickému využití traffic control, neboť dostupná dokumentace je dosti neúplná, a tím je ztížena prvotní orientace v problematice. Práce popisuje filosofii implementace, sémantiku a syntaxi příkazů, a je doplněna několika názornými příklady.

Práce je rozdělena do několika kapitol, z nichž každá obsahuje informace vztahující se k jistému tématu. Je postupně popsáno nastavení podpory traffic control v jádře operačního systému a případná instalace chybějícího softwaru, mechanismy filtrování paketů, koncové a hierarchické disciplíny, včetně jejich vzájemného porovnání.

Annotation

This Diploma Thesis deals with controlling data flows (Traffic Control) in a computer network using the Linux operating system.

The aim is to provide information for the practical use of traffic control, because contemporary documentation is not complete and therefore outlining the problems is quite complicated. The thesis describes the philosophy of implementation, semantics and syntax of commands, and it has been completed by several clear examples.

The thesis is divided into several chapters, each of them is devoted to one theme. It describes the setting-up of traffic control support in the kernel of an operating system and the felicitous installation of missing software, the mechanisms of packet filtering, classless and classfull queuing disciplines respectively, including their comparison.

Obsah

Úvod

V současné době stále rostou požadavky na rychlost a propustnost počítačových sítí. Fyzická řešení bývají mnohdy velice nákladná a ne vždy zcela realizovatelná. Jiným řešením je zefektivnění již existující technologie.

Co je traffic control?

Traffic control se všeobecně vztahuje k hodnotám počítačové sítě jako je latence a propustnost, věcem které přímo ovlivňují uživatelovu spokojenost. Běžné řízení paketů nemusí být ve všech případech ideální. Navíc každá síť může mít odlišné nároky a je svým způsobem specifická. Pomocí nástrojů traffic control máme možnost vytvářet svá řídicí pravidla šitá na míru našim individuálním nárokům.

Třídění a usměrňování datových toků se provádí na výstupu směrovače (dále *router*). K pochopení výhod traffic control použiji následující příklad. Mějme router se základním nastavením snažící se o co nejlepší řízení toků dat (režim *best effort*). Linka je zcela nebo téměř využita a tvoří se fronta. Představme si frontu paketů jako řadu aut čekajících na zaplacení mýtného na most. Do prostoru výběru mýta se najednou vejde omezený počet aut. Ostatní auta a nově přijíždějící se řadí do fronty za nimi. Najednou přijede ambulance s pacientem v kritickém stavu a zařadí se na konec fronty. Na základě daných kritérií je ambulance přemístěna na začátek fronty a je jí umožněn rychlý průjezd. Ambulance v našem případě představuje pakety s požadavkem na rychlou odezvu (nízkou latenci). Příkladem by mohl být audio nebo video přenos (je potřeba nastavit správné omezení, aby nedošlo k zahlcení linky) nebo interaktivní (SSH, telnet) spojení. Je také vhodné upřednostnit malé pakety před velkými, které můžeme přirovnat k nákladním automobilům. Ty totiž mohou výrazně zpomalit provoz, jelikož hlavně v případě pomalejší linky ostatní pakety musí čekat, než projde velký paket. Dále můžeme nastavovat jednotlivým stanicím na lokální síti nebo službám minimální (garantovanou) a maximální šířku pásma. Toho využívají především *ISP* (*Internet Service Provider* – poskytovatelé internetu), kteří svým klientům omezí šířku pásma, a jednu linku pak sdílí více uživatelů.

Co se týče příchozího datového toku (datový tok neboli anglicky traffic, dále jen tok), jedná se o jiný případ. Těžko můžeme ovlivňovat příchozí tok, dokud nemáme kontrolu nad rozhodovacími pravidly traffic control mimo naší síťovou infrastrukturu.

Traffic control je nazýván také jako *traffic shaping*, nebo *QoS* (*Quality of Service*).

Komu je diplomová práce určena?

Z jejího názvu je patrné, že je určena pro lidi pracující s počítačovou sítí (síťaře). Pokud řešíte problémy s přetížením sítě a pokud pracujete v Linuxu, je tato práce určena právě vám. Práce je zaměřena na tři skupiny lidi:

- Na zvídavé lidi, kteří by si chtěli vyzkoušet něco nového a zvýšit si tak komfort služeb na své síti,
- na síťové administrátory, kteří chtějí mít souhrn základních informací po ruce,
- a v neposlední řadě na síťaře pracující s jinými operačními systémy, aby mohli srovnat, jak se s problémy řízení toku dat vypořádává GNU/Linux a jejich systém.

Použité konvence

V textu jsou použity následující typografické konvence:

- Zdrojový kód, příkazy, proměnné, programové výstupy a text, který se zobrazí na obrazovce, je vytištěn `neproporcionálním písmem`.
- Vše, co zadává uživatel, je tištěno **`tučným neproporcionálním písmem`**.
- Zástupné názvy v popisech syntaxe jsou vytištěny *`neproporcionální kurzívou`*. Tento odkaz nahraďte skutečným názvem souboru, argumentu nebo jiného elementu, který odkaz reprezentuje.
- Termíny, které se v textu objevují poprvé, jsou vytištěny *kurzívou*. Tento typ je také někdy použit pro zdůraznění důležitých částí.

V každé kapitole najdete poznámky, upozornění, které obsahují důležité informace.

Anglické termíny jsou ponechány většinou bez překladu, jelikož se domnívám, že snaha o překlad by mohla vést k nesrovnalostem a byla by spíše matoucí. Seznam použitých termínů a zkratek je na straně 71.

Všechny zdrojové kódy a soubory najdete na disku CD-ROM, který je součástí této diplomové práce.

1 Než začneme

Pokud chcete využít možností traffic control, budete potřebovat jádro operačního systému (*kernel*) s vhodnou podporou. Ta je součástí jádra od verze 2.2, pro verzi 2.0 byla dostupná jako patch jádra. Front-end část traffic control je obsažena v programovém balíku *iproute* (nyní ve verzi 2.6.14), který nebývá součástí standardní instalace.

Následuje výběr disciplín a filtrů, které mohou být začleněny do jádra nebo přeloženy jako moduly.

Nastavení podpory QoS

Volby pro traffic control v jádře verze 2.6 jsou pod **`Device Driver > Networking support > Networking options > QoS and/or fair queuing`**. Je potřeba povolit minimálně následující vybrané položky ve výpisu 1.1.

Výpis 1.1: Nastavení podpory QoS v jádře

```
[*] QoS and/or fair queueing
<M>   HTB packet schedule
<M>   The simplest PRIO pseudoscheduler
<M>   RED queue
<M>   SFQ queue
<M>   TBF queue
<M>   Ingress Qdisc
[*]   QoS support
[*]     Rate estimator
[*]   Packet classifier API
<M>     Firewall based classifier
<M>     U32 classifier
[*]     Traffic policing (needed for in/egress)
```

Pro jádro 2.6.9 a novější je k dispozici další volba určující **scheduler clock source**. Volba se nachází opět v sekci **QoS and/or fair queuing**.

```
Packet scheduler clock source (Timer interrupt)  --->
 ( ) Timer interrupt
 ( ) gettimeofday
 (X) CPU cycle counter
```

Volba **Packet scheduler clock source** nahrazuje dosavadní editaci souboru `include/net/pkt_sched.h` popsanou níže, kde najdete i popis jednotlivých položek.

Nastavení podpory netfilter

Volby pro netfilter v jádře verze 2.6 jsou pod **Device Driver > Networking support > Networking options > Network packet filtering (replaces ipchains) -> IP: Netfilter Configuration**.

Výpis 1.2: Nastavení podpory netfilter v jádře

```
<M> Connection tracking (required for masq/NAT + layer7)
<M>   FTP protocol support
<M>   IRC protocol support
<M> IP tables support (required for filtering/masq/NAT)
<M>   limit match support
<M>   IP range match support
<M>   Layer 7 match support (EXPERIMENTAL)
[ ]     Layer 7 debugging output(2048)  Buffer size for application
layer data
<M>   Packet type match support
<M>   netfilter MARK match support
<M>   Multiple port match support
<M>   TOS match support
<M>   LENGTH match support
<M>   Helper match support
```

```
<M>   Connection state match support
<M>   Connection tracking match support
<M>   Packet filtering
<M>     REJECT target support
<M>   Full NAT
<M> MASQUERADE target support
<M> Packet mangling
<M>   TOS target support
<M>   MARK target support
<M>   CLASSIFY target support
<M> LOG target support
<M> ULOG target support
```

Poznámka:

Kompilací jádra se nebudeme zabývat, pokud si někdo nebude vědět s něčím rady, nechť si projde *Kernel HOWTO* na internetu. Jádro se vším potřebným může být i v instalačním balíčku.

Nároky jednotlivých nástrojů `tc` na konfguraci jádra shrnuje tabulka 1.1.

Tabulka 1.1: Požadované nastavení jádra pro jednotlivé nástroje `tc`

Parametr	Konfigurace jádra
`tc`	**QoS support, QoS and/or fair queueing**
`qdisc`	**QoS support, QoS and/or fair queueing**
`qdisc sfq`	**QoS support, QoS and/or fair queueing, SFQ queue**
`qdisc red`	**QoS support, QoS and/or fair queueing, RED queue**
`qdisc tbf`	**QoS support, QoS and/or fair queueing, TBF queue, Rate estimator**

qdisc cbq	**QoS support, QoS and/or fair queueing, CBQ packet scheduler, Rate estimator**
qdisc prio	**QoS support, QoS and/or fair queueing, The simplest PRIO pseudo scheduler**
qdisc csz	**QoS support, QoS and/or fair queueing, CSZ packet scheduler**
class	**QoS support, QoS and/or fair queueing, Packet classifier API**
filter	**QoS support, QoS and/or fair queueing, Packet classifier API**
estimator	**QoS support, QoS and/or fair queueing, Rate estimator**

Změny ve zdrojových souborech jádra

Ve zdrojových souborech jádra je přednastaveno několik údajů, které bychom měli změnit.

Parametr PSCHED_CLOCK_SOURCE

Při standardní konfiguraci jádra nebyl do verze 2.6.9 přístupný parametr, který určuje zdroj času pro *scheduler*. Parametr se jmenuje PSCHED_CLOCK_SOURCE a má zásadní vliv na funkčnost scheduleru. Je definován v souboru include/net/pkt_sched.h.

Tabulka 1.2: Volby pro sheduler

Parametr	Význam
PSCHED_GETTIMEOFDAY	Čas je získáván voláním funkce do_gettimeoday(). Toto varianta je velmi pomalá a autoři ji nedoporučují používat.
PSCHED_JIFFIES	Čas je odvozen od základního časovače jádra - parametr HZ. Protože pro platformu i386 je standardně HZ = 100, získáme časovou základnu s dosti špatnou granularitou 10

	ms, což je hodnota nepoužitelná pro traffic control při rychlostech nad 1.5 Mb/s. Tato hodnota je přednastavena.
PSCHED_CPU	Čas je odvozen od čtení registru TSC, který je čítačem taktu procesoru. Registr TSC existuje na platformě Intel od doby Pentia, ale někdy není použitelný jako časovač, např. při zapnutém APM (advanced power management). Volba PSCHED_CPU je jednoznačně preferována, ale bohužel není přednastavena.

Jestli váš procesor podporuje registr TSC můžete zjistit příkazem:

```
# cat /proc/cpuinfo
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep myrt \
pge mca cmov pat pse36 mmx fxsr sse syscall mmxext 3dnowext 3dnow
...
```

Parametr HTB_HYSTERESIS

Definuje vztah mezi přesností řízení a rychlostí výpočtu. Pokud je maximální šířka pásma menší než 1MB/s, není přednastavená volba HTB_HYSTERESIS příliš vhodná. Otevřete soubor net/sched/sch_htb.c a změňte hodnotu HTB_HYSTERESIS na 0.

```
#define HTB_HYSTERESIS 0
```

Parametr SFQ_DEPTH

Pokud naší sítí prochází malé množství dat, je přednastavená délka fronty 128 příliš velká. Datové toky s požadavkem na rychlou odezvu mohou být zbytečně brzděny, když SFQ začne plnit jejich frontu. Parametr SFQ_DEPTH je definován v souboru net/sched/sch_sfq.c. Frontu můžete zkrátit například na 10.

```
#define SFQ_DEPTH               10
```

Iproute2

Iproute2 je soubor programů určených k řízení TCP/IP a traffic control v Linuxu. Momentálně je pod správou Stephena Hemmingera. Původní autor Alexey Kuznetsov je známý hlavně díky implementaci traffic control do linuxového jádra.

Ve většině síťových manuálů je stále uváděn `ifconfig` jako konfigurační nástroj, ale to je špatně, protože `ifconfig` nedostačuje modernímu síťovému prostředí. Od jeho používání by se mělo upustit, ale většina linuxových distribucí ho stále obsahuje. Z toho důvodu je stále mnoho síťových konfiguračních systémů založených na používaní `ifconfig`, a proto mají omezené možnosti. Cílem */etc/net* projektu je podpora moderních síťových technologií, nepoužívá `ifconfig` a umožňuje systémovým administrátorům využít všech vlastností iproute2, zahrnujících traffic control.

Iproute2 se obvykle nachází v balíčku s názvem iproute nebo iproute2 a skládá se z několika nástrojů, z nichž nejdůležitější jsou `ip` a `tc`. `ip` konfiguruje IPv4 a IPv6 a `tc` traffic control. Oba nástroje mají podrobné návody a jsou doprovázeny manuálovými stránkami. Na internetu je k iproute2 spousta informací.

Download

Současná a dřívější verze jsou ke stažení na adrese:
`http://developer.osdl.org/dev/iproute2/download/`

K dispozici je také CVS s nejnovějším kódem:

```
$ cvs -d :pserver:cvsanon@cvs.osdl.org/repos login
Password: cvsanon
$ cvs -d :pserver:cvsanon@cvs.osdl.org/repos co iproute2
```

2 Nástroj tc

Nástroj `tc` zajišťuje založení, zrušení, modifikaci nebo výpis disciplín, tříd a filtrů.

K rozlišení jednotlivých disciplín a filtrů slouží parametr `handle`, identifikátor ve tvaru *X*:*Y*. Základní disciplína každého rozhraní má atribut `root` s identifikátorem ve tvaru *X*: nebo *X*:0 (oba zápisy znamenají totéž). Hierarchie mezi disciplínami, třídami a filtry je zajištěna parametrem `parent`, jehož hodnotou je identifikátor nadřazeného objektu. Pořadí vykonávání filtrů je určeno parametrem `prio` a v případě splnění podmínky je paket předán disciplíně nebo třídě podle hodnoty parametru `flowid` (resp. synonymum `classid`).

Nástroj tc používá několik zkratek, jež shrnuje tabulka 2.1. Jedná se o zdroj poměrně častých chyb, mnohdy se zaměňují kilobajty za kilobity. Poznamenejme, že 1Mbitu odpovídá 128KBajtům (u kvantifikátorů kilo a mega se nerozlišují velká a malá písmena).

Tabulka 2.1: Zkratky užívané nástrojem tc

Přípona	Význam	Přepočet na základní jednotku
Rychlost		
`bps`	bytes per second	
`kbps`	kilobytes per second	*1024 = bps
`mbps`	megabytes per second	*1024*1024 = bps
`kbit`	kilobits per second	*1024/8 = bps
`mbit`	megabits per second	*1024*1024/8 = bps
Velikost		
`k, kb`	kilobytes	*1024 = b (bytes)

`m, mb`	megabytes	*1024*1024 = b
`kbit`	kilobits	*1024/8 = b
`mbit`	megabits	*1024*1024/8 = b

Čas

`s, sec, secs`	sekunda
`ms, msec, msecs`	milisekunda
`us, usec, usecs`	mikrosekunda

Syntaktické detaily příkazu `tc` (např. volby pro jednotlivé typy disciplín) a zejména jejich přesná sémantika je dosti složitá a nepřehledná záležitost, která není nikde dostatečně dokumentována. Někdy je jedinou metodou získání informací prohlídka souborů se zdrojovým kódem, kde se občas vyskytuje i nějaký vysvětlující komentář. Výpisy ze zdrojových souborů jsou v příloze.

Tabulka 2.2: Parametry užívané nástrojem `tc`

Parametr	Význam
`handle`	Identifikátor třídy nebo filtru. Nemá stejný význam u všech filtrů.
`parent`	handle (identifikátor) třídy, ke které se pravidlo vztahuje. Třída musí již existovat.
`classid`	Identifikátor třídy
`class`	Třída, viz. kapitola věnující se disciplínám.
`prio`	Priorita pravidla. Nižší čísla jsou testována dříve.
`protocol`	Určuje protocol, na jehož pakety se pravidlo vztahuje.

Výpis ze souboru `iproute2/tc/tc.c`

```
Usage: tc [ OPTIONS ] OBJECT { COMMAND | help }
where  OBJECT := { qdisc | class | filter }
       OPTIONS := { -s[tatistics] | -d[etails] | -r[aw] }
```

3 Filtrování paketů IP

Dnes se uživatelé internetu proti hrozbě síťových útoků zcela běžně chrání prostřednictvím techniky nazývané *filtrování* IP. Tento mechanismus znamená, že jádro systému kontroluje každý odeslaný nebo přijatý paket, a samo se rozhoduje, jestli jej propustí dále, zahodí jej, anebo jej určitým způsobem pozmění. Filtrování IP se často označuje za firewall, protože pečlivým filtrováním všech paketů, které do počítače vstupují nebo z něj vystupují, fakticky vytváříme ochranné rozhraní umístěné mezi vlastním systémem a internetem. Filtrování paketů IP nás neochraní před útoky virů a trojských koňů, případně před poškozením aplikací, ale na druhé straně nás dokáže chránit před celou řadou útoků vedených ze sítě, jako jsou určité typy útoků odepření služeb a falšování IP (to jsou pakety, které za svůj zdroj označují systém, z něhož ve skutečnosti vůbec nepocházejí). Filtrování IP představuje také další úroveň řízení přístupu, která zabraňuje uživatelům ve vstupu do systému.

Při zavádění techniky filtrování IP musíme nejdříve vědět, co s kterým paketem budeme dělat. Firewall určuje, které pakety budeme povolovat a které budeme zamítat, kdežto pro traffic control využijeme jiných pravidel filtrování. Rozhodování o filtrování konkrétního paketu vychází obvykle z údajů v jeho hlavičce, mezi než patří zdrojová a cílová IP adresa, typ protokolu (TCP, UDP atd.) a zdrojové a cílové číslo portu (toto číslo označuje příslušnou službu, do které paket směřuje). Různé síťové služby používají přitom ke své činnosti různé protokoly a různá čísla portů, ty nejznámější shrnuje tabulka 3.1. V Linuxu jsou porty definované v souboru `/etc/services`.

Tabulka 3.1: Čísla portů TCP and UDP

Port / protokol	Popis
Porty 0 až 1023	
20/tcp	FTP (File Transfer Protocol) – data port
21/tcp	FTP – řídicí (příkazový) port

22/tcp	SSH (Secure Shell) – používán pro zabezpečené přihlašování, přenos souborů (scp, sftp) a port forwarding
23/tcp	Telnet protocol – nekryptovaná textová komunikace
25/tcp	SMTP (Simple Mail Transfer Protocol) – používán pro posílání e-mailů
53/tcp & udp	DNS (Domain Name Server)
67/udp	BOOTP (BootStrap Protocol) server; také používán DHCP (Dynamic Host Configuration Protocol)
68/udp	BOOTP client; také používán DHCP
80/tcp	HTTP (HyperText Transfer Protocol) – používán pro přenos webových stránek
110/tcp	POP3 (Post Office Protocol version 3) – používán pro příjem emailů
123/udp	NTP (Network Time Protocol) – používán pro synchronizaci času
143/tcp	IMAP4 (Internet Message Access Protocol 4) – používán pro příjem emailů

Porty 1024 až 49151

1214/tcp	Kazaa (port může být uživatelem změněn)
4662/tcp	eMule (často používaný port)
4672/udp	eMule (často používaný port)
6881/tcp	BitTorrent (často používaný port)
6969/tcp	BitTorrnet tracker port
8080/tcp	HTTP Alternate (http-alt) – používán v případě, že na tom samém počítači běží druhý web server (na portu 80), pro web proxy a caching server, nebo pro web server běžící pod jiným uživatelem než root.

Poznámka:

IANA je zodpovědná za přidělování TCP a UDP čísel portů ke specifickému užití. Známé porty jsou v rozmezí 0-1023. Registrovaná čísla portů jsou v rozmezí 1024-49151. Porty s čísly v rozmezí 49152-65535 jsou soukromé či dynamické, jež nepoužívá žádna definovaná aplikace. Dle definice žádný port nemůže být registrován z rozsahu dynamických portů.

Seznam TCP a UDP portů je na adrese

`http://en.wikipedia.org/wiki/TCP_and_UDP_port_numbers`

Implementace v jádře

Filtrování IP je implementováno v samotném jádře Linuxu, které obsahuje potřebný programový kód pro kontrolu jednotlivých přijatých a odeslaných paketů a pro aplikaci filtrovacích pravidel, jež určují další osud paketu. Pro konfiguraci pravidel slouží uživatelský konfigurační nástroj, který přebírá argumenty z příkazového řádku a převádí je do specifikace filtrů – ty se pak ukládají a jádro je používá jako pravidla.

Součástí Linuxu jsou celkem tři generace filtrování IP v jádře, z nichž každá má svůj vlastní konfigurační mechanismus. První generace se označovala jako *ipfw* (IP firewall) a nabízela jisté základní funkce filtrování. Pro složitější konfigurace byla ale málo flexibilní a neefektivní, takže se dnes už prakticky nepoužívá. Druhou generaci filtrování IP představují takzvané řetězy pravidel *IP chains*. Vznikly zásadním zdokonalením ipfw a dodnes se běžně používají. Nejnovější generací filtrování jsou pak tabulky *netfilter/iptables*. Název netfilter zde označuje komponentu jádra a tabulky iptables jsou uživatelským konfiguračním nástrojem (oba pojmy se poměrně často volně zaměňují). Mechanismus netfilter nabízí nejen možnost velice flexibilní konfigurace, ale také je dobře rozšiřitelný. V následujících částech textu si popíšeme činnost netfilter a v příkladech uvedeme několik jednoduchých konfigurací.

Základy mechanismu netfilter

Filtrovací mechanismus netfilter je implementován v linuxovém jádře verze 2.4.0 a novějších. Nejdůležitější nástroj pro manipulaci s filtrovacími tabulkami a jejich zobrazování se nazývá iptables a je součástí všech současných distribucí Linuxu. Příkaz iptables umožňuje konfiguraci bohaté a komplexní množiny pravidel a má tudíž velké množství různých voleb příkazového řádku. My probereme alespoň ty nejběžnější. Úplný výklad příkazu najdete v manuálové stránce iptables.

Důležitým principem činnosti netfilter je takzvaný *řetěz* (chain), který tvoří seznam pravidel aplikovaných na pakety v okamžiku vstupu, výstupu nebo průchodu systémem. Řetězce se sdružují do tří tabulek. První je tabulka mangle a je zodpovědná za změnu bitů týkajících se QoS v TCP hlavičce paketu. Druhou je tabulka filter a je zodpovědná za filtrování paketů. Poslední je nat tabulka, která má na starosti překlad síťových adres. Tabulka 3.2 ukazuje řetězy implicitně definované jádrem a jejich příslušnost k jednotlivým tabulkám. Administrátor může specifikovat i nové řetězy pravidel a propojit je s předdefinovanými řetězy.

Tabulka 3.2: Předdefinované řetězy

Tabulka	Určení	Řetěz	Použití
`filter`	Filtrování paketů	`INPUT`	Tento řetěz platí pro všechny pakety určené pro lokální systém.
		`OUTPUT`	Tento řetěz platí pro pakety, které lokální systém vysílá ven.
		`FORWARD`	Tento řetěz se uplatňuje v situaci, kdy paket procházejí systémem z jednoho jeho síťového rozhraní na druhé. Používá se u systémů, které pro pakety fungují jako směrovač nebo brána (router nebo gateway), a platí pro pakety, které z lokálního systému ani nepocházejí, ani v něm nemají cíl.

nat	Překlad adres	PREROUTING	Překlad adresy před routováním. Tímto řetězem procházejí jak pakety určené pro lokální systém, tak pakety směrované jinam.
		POSTROUTING	Překlad adresy po routování. Tímto řetězem procházejí jak pakety odcházející z lokálního systému, tak pakety směrované jinam.
mangle	Modifikace hlavičky paketu	INPUT OUTPUT FORWARD PREROUTING POSTROUTING	

Každé pravidlo v řetězu definuje jistou množinu kritérií, která blíže popisuje odpovídající pakety a operaci (akci, cíl), která se má s příslušnými pakety provést. Seznam akcí prováděných nad pakety je v tabulce 3.3.

Tabulka 3.3: Operace s pakety

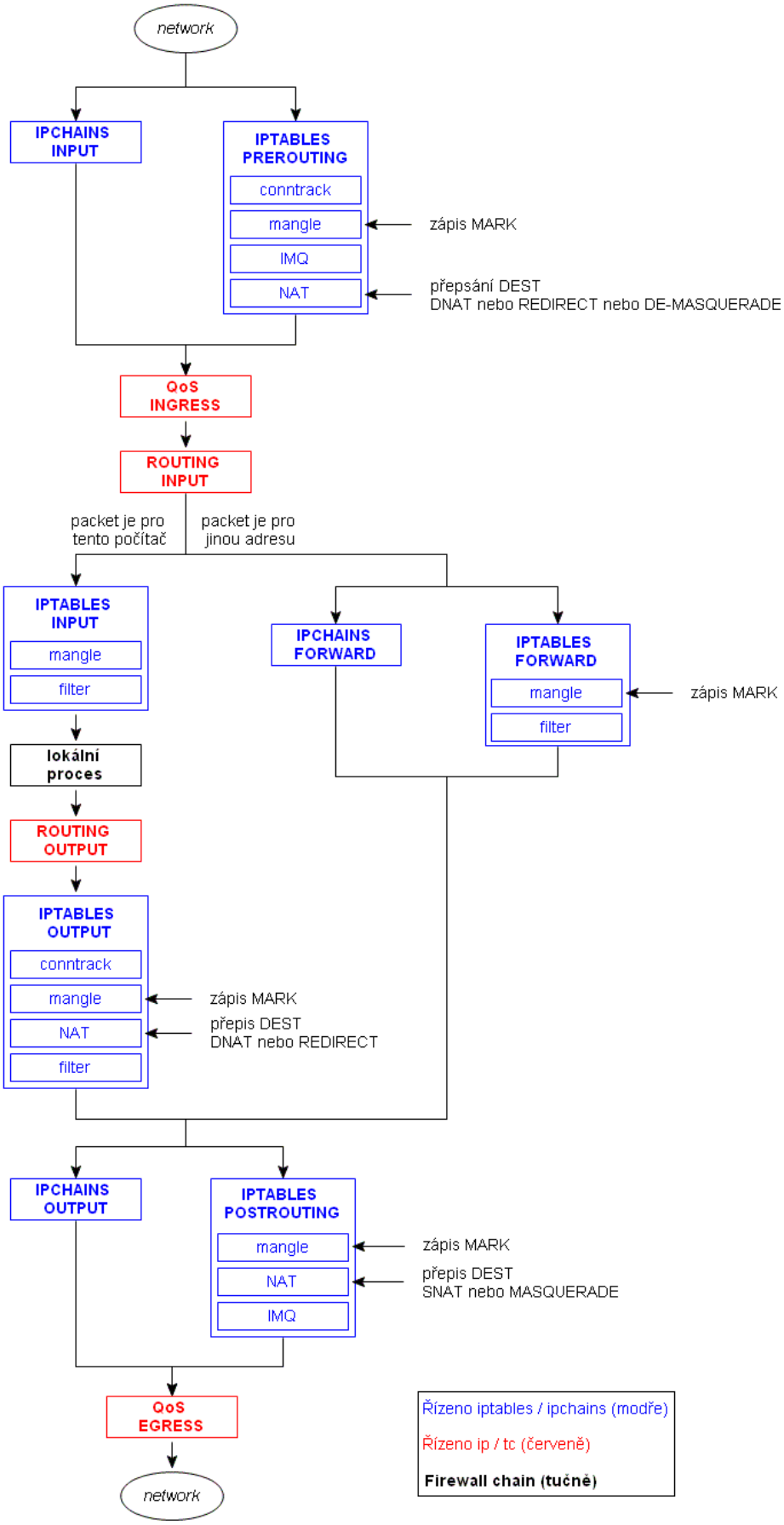
Cíl (akce)	Význam	Popis
Tabulka mangle		
TOS	*Type Of Service*	Změna TOS paketu.
TTL	*Time To Live*	Změna TTL paketu.
MARK	*označení* paketu	
CLASSID	*předání* paketu	
Tabulka NAT		
DNAT	*Destination Network Address Translation*	Změna cílové adresy paketu.

SNAT	*Source Network Address Translation*	Změna zdrojové adresy paketu. Lokální síť v nadřazené síti (internetu) vystupuje pod jedinou IP adresou routeru.
MASQUERADE		Podobné jako SNAT, ale sám si zjišťuje současnou IP adresu počítače (funguje i pro dynamické IP).
REDIRECT	*přeposlání* paketu	

Tabulka filter

ACCEPT	*příjetí* paketu	Paket je schválen a může být přijat systémem nebo z něj naopak být odeslán. Iptables zastavují další zpracování paketu.
DROP	*zahození* paketu	Požadavek odeslání nebo příjmu paketu je zamítnut. Iptables zastavují další zpracování paketu.
REJECT	*odmítnutí* paketu	Stejné jako DROP, ovšem odesilateli je zaslána zpráva (defaultně icmp-port-unreachable).
RETURN	*návrat* paketu	Packet se vrátí do volajícího řetězce.
LOG	*záznam* paketu	Informace o paketu jsou poslány démonu syslog Zpracování pokračuje dalšími pravidly v tabulce.
QUEUE	*předání* paketu	Poslední operace je užitečná při vytváření uživatelsky definovaných řetězů, se kterými je možné hierarchicky definovat složitá pravidla filtrování paketů.

Na obrázku 3.1 je diagram průchodu paketu jádrem. Diagram zobrazuje dvě možnosti filtrování paketu pomocí nástrojů iptables a ipchains. Třetí, nejstarší, filtrovací mechanismus zobrazen není. Lze ovšem používat buď iptables, nebo ipchains, a nebo ipfwadm, ale není možné je vzájemně kombinovat.



Obrázek 3.1: Průchod paketu jádrem

Paket postupně prochází všemi pravidly v řetězu, dokud není definitivně rozhodnuto jaká akce se provede, nebo dokud se nedostane na konec řetězu. Jestliže projde množinou pravidel až na konec, určí jeho osud implicitní akce řetězu.

Linuxový mechanismus netfilter podporuje v pravidlech filtrování celou řadu dalších zajímavých věcí. Jednou z největších výhod netfilter je jeho rozšiřitelnost. Můžeme navrhnout rozšíření, která zdokonalují jeho činnost.

Příkaz iptables

Příkaz iptables slouží k provádění změn v řetězech a množinách pravidel netfilter, můžeme pomocí něj vytvářet nové řetězy, odstraňovat je, vypisovat pravidla v řetězu, vyprázdnit řetěz (neboli odstranit z něj veškerá pravidla) a definovat implicitní neboli výchozí akci. Dále můžeme příkazem iptables vkládat nová pravidla do řetězu, přidávat je na konec, odstraňovat je a nahrazovat.

Příkaz iptables má velké množství různých argumentů a voleb příkazového řádku. V tabulce 3.4 je přehled argumentů příkazu iptables pro práci s řetězy pravidel a tabulka 3.5 shrnuje argumenty stejného příkazu pro práci s jednotlivými pravidly.

Tabulka 3.4: Operace s řetězy

Argument	Popis
`-L [řetěz]`	Vypíše pravidla v řetězu, případně ve všech řetězech, není-li řetěz definován.
`-F [řetěz]`	Odstraní pravidla z řetězu, případně ze všech řetězů, není-li řetěz definován.
`-P řetěz akce`	Nastaví implicitní akci u zadaného řetězu na *akce*. Platné akce jsou `ACCEPT`, `DROP`, `QUEUE`, `RETURN`.
`-Z [řetěz]`	Vynuluje bajtové čítače v řetězu nebo ve všech řetězech.
`-N řetěz`	Vytvoří nový řetěz se zadaným názvem. Řetěz se zadaným názvem nesmí existovat. Tímto příkazem se vytváří uživatelem

	definované řetězy.
`-X [řetěz]`	Odstraní řetěz definovaný uživatelem, případně všechny uživatelem definované řetězy, není-li název řetězu uveden. Aby příkaz uspěl, na řetěz nesmí existovat žádné odkazy z jiných tříd.

Tabulka 3.5: Operace s pravidly

Argument	Popis
`-A řetěz specifikace_pravidla`	Přidá nové pravidlo na konec řetězu.
`-D řetěz číslo_pravidla`	Odstraní pravidlo určené svým číslem z řetězu.
`-R řetěz číslo_pravidla specifikace_pravidla`	V řetězu nahradí pravidlo se zadaným číslem za pravidlo popsané ve *specifikaci_pravidla*.
`-I řetěz číslo_pravidla specifikace_pravidla`	Na pozici *číslo_pravidla* v řetězu vloží pravidlo podle *specifikace_pravidla*. Není-li číslo pravidla nastaveno, předpokládá se 1.

Každé filtrovací pravidlo obsahuje parametry, které popisují množinu vyhovujících paketů. V tabulce 3.6 je seznam všech parametrů obsažených v každém pravidle (pokud některý z parametru není zadán, použije se jeho implicitní hodnota), a dále pak seznam rozšiřujících parametrů. Jak už bylo zmíněno, nástroj netfilter je rozšiřitelný pomocí modulů sdílených knihoven. Existuje několik standardních rozšíření, která implementují funkce programu ipchains. Společně s rozšiřujícími parametry jsou uvedeny parametry `-m` a `-p`, jež nastavují kontext rozšíření.

Tabulka 3.6: Parametry pravidel

Parametr	Význam
`-p [!] protokol`	Protokol paketu. Platné hodnoty jsou `tcp`, `udp`, `icmp` nebo `all` (všechny).
`-s [!] zdroj[/maska]`	Zdrojová adresa paketu, určená jako hostitelský název

	systému nebo IP adresa. */maska* popisuje nepovinnou masku sítě, a to jako literál nebo jako počet bitů. Literálový zápis představuje například výraz `/255.255.255.0`, zatímco počet bitů v masce popisujeme výrazem `/24`.
`-d [!]` *cíl*`[`*/maska*`]`	Cílová adresa paketu. Platí pro ni stejná syntaxe jako pro zdrojovou adresu.
`-i [!]` *rozhraní*	Síťové rozhraní, na němž byl příchozí paket přijat. Pokud název rozhraní končí symbolem +, vyhovují všechna rozhraní začínající zadaným řetězcem. Například `-i eth+` definuje všechna ethernetová rozhraní, `-i ! eth+` definuje všechna rozhraní kromě ethernetových.
`-o [!]` *rozhraní*	Síťové rozhraní, na němž bude odchozí paket odeslán. Použití je stejné jako u parametru `-i`.
`-j` *cíl*	Zkratka j znamená jump neboli skok. Definuje akci (cíl) která se má provést, jestliže paket vyhovuje kritériím pravidla.
`-f`	Pravidlo se týká pouze druhého a následujících fragmentů fragmentovaného paketu.
`-t` *table*	Definuje tabulku. Platné hodnoty jsou `mangle`, `nat`, `filter`. Pokud není tento parametr uveden, pracuje se s tabulkou `filter`.

Rozšiřující parametry

`-p TCP --sport [!]` *port*`[:`*port*`]`	Zdrojový port TCP paketu. Zadává se jako číslo portu, nebo jako název služby podle údajů v souboru `/etc/services`. Lze specifikovat i rozsahy portů zadáním spodní a horní meze oddělených dvojtečkou. Například hodnota 20:25 znamená porty od 20 (včetně) do 25 (včetně).

`-p TCP --dport [!] port[:port]`	Cílový port TCP paketu. Platí pro něj stejná syntaxe jako pro zdrojový port.
`-p TCP --syn`	Pravidlu vyhovují pakety pouze s nastaveným příznakem SYN a nenastavenými příznaky ACK. Využíva se pro identifikaci žádosti o navázání TCP spojení.
`-p UDP --sport [!] port[:port]`	Zdrojový port UDP paketu. Platí pro něj stejná syntaxe jako pro zdrojový port TCP.
`-p UDP --dport [!] port[:port]`	Cílový port UDP paketu. Platí pro něj stejná syntaxe jako pro zdrojový port TCP.
`-m multiport --sport <port, port>`	Seznam zdrojových TCP/UDP portů oddělených čárkou.
`-m multiport --dport <port, port>`	Seznam cílových TCP/UDP portů oddělených čárkou.
`-m multiport --ports <port, port>`	Seznam TCP/UDP portů oddělených čárkou. Předpokládá se, že zdrojové i cílové porty jsou shodné.
`-m --state state`	Testování stavu. Parametr `state` může nabývat následujících hodnot: `ESTABLISHED` paket je součástí již existujícího spojení. `NEW` úvodní paket nového spojení. `RELATED` úvodní paket dalšího nového spojení k již existujícím spojením. Například u protokolu FTP. `INVALID` paket nelze identifikovat, což může být způsobeno nedostatečným výkonem počítače, nebo chybou ICMP (paket neodpovídá žádnému existujícímu toku)
`-m --mac-source [!] adresa`	Ethernetová adresa odesilatele paketu.

Upozornění!

Vykřičník `!` **před parametrem obrací (neguje) jeho význam. Zápis parametru** `-dport 80` **tak například znamená „pravidlu vyhovuje cílový port 80“, zatímco parametr** `-dport ! 80` **znamená „pravidlu vyhovuje jakýkoli cílový port s výjimkou portu 80“.**

Další důležité volby v tabulce 3.7 slouží pro vytváření množin pravidel.

Tabulka 3.7: Ostatní volby příkazu iptables

Volba	Popis
`-v`	Zapne podrobný výpis (nejužitečnější je při výpisu pravidel s volbou `-L`)
`-n`	Vypisuje IP adresy v číselné podobě, neprovádí se tedy vyhledávání podle služby DNS.
`-m` *`modul`*	Zavádí rozšiřující modul iptables s názvem *`modul`*.

Manipulace s bity typu služby

Bity typu služby (*TOS – Type Of Service*) jsou čtyři bitové příznaky v hlavičce paketu. Každý z bitů má svůj vlastní význam a v jednom paketu může být nastaven pouze jeden z nich, kombinace příznaků nejsou povoleny. Pokud je některý z těchto bitů nastaven, mohou směšovače zpracovávat pakety odlišně od zpracovávání paketů s jiným příznakem nebo od paketů bez těchto příznaků. Bity se označují jako bity typu služby, protože umožňují, aby odesílající aplikace síti sdělila, kterou službu vyžaduje.

Existují následující skupiny služeb:

Minimum delay

Používá se, pokud nejdůležitějším kritériem je čas přenosu paketu od zdroje k cíli, tedy pokud se požaduje co nejmenší zpoždění.

Maximum throughput

Používá se, pokud je důležitý objem dat přenášený v jednotlivých časových intervalech. Existuje celá řada síťových aplikací, pro než není kritické zpoždění přenosu, je však pro ně důležitá dostatečná propustnost sítě. Například online přenosy audia a videa.

Maximum reliability

Používá se, pokud je důležité, aby data v mezích možností dorazila k cíli bez nutností jejich opakovaného odesílání. Jako příklad je možné uvést protokoly PPP a SLIP.

Minimum cost

Používá se, je-li nutné minimalizovat přenosové náklady.

Nastavení bitů typu služby

Pomocí ipfwadm a ipchains

Pomocí ipfwadm a ipchains se změna TOS bitů provádí parametrem `-t`. Změny se provádí dvěma bitovými maskami. První maska se logicky ANDuje s příznaky v paketu, druhá se s nimi logicky XORuje. Bitové masky se definují osmibitovou šestnáctkovou soustavou. Jak ipfwadm, tak ipchains používají stejnou syntaxi:

`-t` *`andmaska xormaska`*

Doporučené použití a odpovídající masky jsou uvedeny v tabulce 3.8.

Tabulka 3.8: Doporučené nastavení TOS bitů

TOS	Maska AND	Maska XOR	Doporučené použití
Normal Service	0x00	0x00	
Minimize Cost	0x01	0x02	NNTP, SMTP
Maximize Reliability	0x01	0x04	SNMP, DNS
Maximize Throughput	0x01	0x08	FTP-data, WWW
Minimize Delay	0x01	0x10	FTP, SSH, telnet

Pomocí iptables

Pomocí iptables můžeme definovat pravidla, kterým budou vyhovovat pakety s určitými příznaky, pomocí parametru `-m tos` a nastavovat TOS bity paketům, vyhovujícím danému pravidlu, pomocí parametru `-j akce`. TOS bity je možné nastavovat pouze v řetězech OUTPUT a FORWARD. Testování a nastavování probíhá nezávisle. Můžeme vytvářet libovolné skupiny pravidel. Můžeme například vytvořit pravidlo, které nastaví TOS u paketů pocházejících z určitého počítače nebo portu.

Namísto složité dvoumaskové konfigurace programů ipfwadm a ipchains používá program iptables jednodušší řešení, které přímo říká, které TOS bity musí vyhovovat nebo které TOS bity mají být nastaveny. Navíc se nemusí specifikovat jejich šestnáctkovými hodnotami a mohou se použít názvy uvedené v tabulce 3.9.

Obecná syntaxe pravidla, kterému vyhovují pakety s určitým nastavením TOS, je:

`-m tos --tos` *`název`* `[`*`další parametry`*`] -j` *`akce`*

Obecná syntaxe pravidla pro nastavení TOS bitů je:

`[`*`další parametry`*`] -j TOS --set` *`název`*

Tabulka 3.9: Doporučené nastavení TOS bitů

Název	Maska
Normal-Service	0x00
Minimize-Cost	0x02
Maximize-Reliability	0x04
Maximize-Throughput	0x08
Minimize-Delay	0x10

Filtry provádějí klasifikaci paketů založenou na jejich specifických vlastnostech. Filtry, příslušející téže disciplíně nebo třídě, jsou podle své priority seřazeny do seznamu a sekvenčně aplikovány, dokud není podmínka některého filtru splněna.

Používání cíle CLASSIFY

Od verze 2.6 linuxového jádra je cíl `CLASSIFY` standardní součástí distribucí. Nástroj Netfilter byl rozšířen o akci `CLASSIFY` ve verzi 1.2.9.

Používaní cíle `CLASSIFY` je poměrně jednoduché, pokud známe parametry iptables.

```
iptables -t mangle -A POSTROUTING -o eth0 -p tcp --sport 80 -j \
CLASSIFY --set-class 1:10
```

Tento příkaz přidává pravidlo do řetězu `POSTROUTING` tabulky `mangle`. Pravidlo se vztahuje na TCP pakety se zdrojovým portem 80, které jsou odesílany ze síťové karty eth0. Cílem pravidla je `CLASSIFY`, který roztřídí pakety do předem definovaných tříd disciplín popsaných v kapitole 5. V našem případě se jedná o třídu s hlavním číslem 1 a vedlejším číslem 10.

Cíl `CLASSIFY` lze použít pouze u řetězu `POSTROUTING` tabulky `mangle`. Pokud budeme chtít filtrovat toky jinde, můžeme použít cíl `MARK`.

Používání cíle MARK

Pokud nemůžeme použít cíl `CLASSIFY`, můžeme použít cíl `MARK` ve spojení s tříděním toků nástroje tc.

```
iptables -t mangle -A POSTROUTING -o eth0 -p tcp --sport 80 -j \
MARK --set-mark 1
```

Tento příkaz nastaví „neviditelnou“ značku každému paketu, který pravidlu vyhovuje. Značka existuje pouze v jádře systému (paket není nikterak modifikován). Nástroj `tc` na základě těchto značek třídí toky.

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 handle1 \
fw classid 1:10
```

Parametrem `parent` je definována takzvaná *root* třída disciplíny, která musí existovat před vytvořením filtru. Parametr `handle` odkazuje na značku, kterou jsme přiřadili toku již v předešlém příkazu. Parametrem `classid` určíme třídu, do které tok patří.

Klasifikátor u32

Filtr *u32* (často uváděn jako *Selector u32*) porovnává určité bity v hlavičce IP, TCP, UDP a ICMP paketů. Nejčastější klasifikace paketů se zakládá opět na porovnávání zdrojové a cílové adresy a zdrojového a cílového čísla portu.

```
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip \
sport 80 0xffff classid 1:10
```

Jak je vidět, oproti předešlým příkladům filtrů je v syntaxi tohoto příkazu několik změn. Opět pracujeme s `filter` na síťovém rozhraní eth0 s protokolem IP a identifikátorem rodiče 1:0. Druhá, odlišná, část příkazu specifikuje selektor u32 ve formátu klíčového slova `match` následovaného klíčovým slovem `ip` se čtyřmi možnými parametry: `src` (*source* – zdrojová adresa), `dst` (*destination* – cílová adresa), `sport` a `dport` (*source* a *destination port*). V našem případě pravidlu vyhovují všechny IP pakety vyslané z portu 80. Parametrem `classid` na konci příkazu určíme třídu, do které tok patří. Kritérií s klíčovým slovem `match`, podle kterých se vybírá vyhovující paket, může být několik za sebou.

```
tc filter add dev eth0 parent 1:0 protocol ip u32 match ip \
src 1.2.3.4 match ip dst 4.3.2.0/24 match ip tos 0x10 0xff \
classid 1:10
```

Klasifikátor route

Klasifikátor *route* využívá znalosti, která řádka routovací tabulky byla použita pro daný paket. Tím se může eliminovat opakované rozhodování na základě cílové adresy.

```
ip route add 10.0.1.0/24 via 10.0.2.2 dev eth1 realm 10
```

Tímto příkazem označíme cílovou síť 10.0.1.0/24 pomocí parametru `realm` a jeho hodnotou 10. Toto číslo nám potom reprezentuje síť nebo stanici při přidávání filtru `route`.

```
tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
route to 10 classid 1:10
```

Klasifikátor `route` lze použít na zdrojové routy.

```
ip route add 10.0.2.0/24 dev eth0 realm 2
```

```
tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
route from 2 classid 1:2
```

Pakety z podsítě 10.0.2.0/24 budou předány třídě 1:2.

Klasifikátor L7

L7 je nový rozšiřující modul pro Netfilter, který umožňuje rozpoznat toky aplikační (sedmé) vrstvy. L7 prozkoumává obsah paketů a hledá shodu se specifikovaným vzorem. Součástí L7 je databáze srovnávacích vzorů pro různé protokoly aplikační vrstvy. L7 je velice užitečný nástroj při rozpoznávání *peer to peer* (*p2p*) aplikací, z nichž spousta používá náhodné porty, které ovšem nešifrují svá data ve snaze vyvarovat se odhalení.

Klasifikátor L7 zatím není oficiální součásti Netfilter. Lze stahnout jeho zdrojový kód a provést patch jádra a iptables. Podrobný návod (HOWTO) popisující jednotlivé kroky kompilace je na internetové adrese `http://l7-filter.sourceforge.net/L7-HOWTO-Netfilter`. Pokud budete mít nainstalovaný patch jádra a iptables s podporou L7, můžete ho začít používat .

```
iptables -t mangle -A POSTROUTING -m layer7 --l7proto edonkey \
-j CLASSIFY --set-class 2:1
```

Tento příkaz umožňuje detekci datových toků pocházejících z populárního p2p programu eMule.

Shrnutí

Filtry jsou svázány s třídami a disciplínami. Umožňují klasifikování paketů a jejich rozdělení do tříd a disciplín.

Můžeme použít několik klasifikátorů:

- `fw` Využívá označení paketů firewallem.
- `u32` Využívá informací obsažených v hlavičce paketu.
- `route` Je založen na rozhodnutí, která routa bude pro paket použita při routování.

- `L7` Prozkoumává obsah paketů. Je schopen rozpoznat p2p toky.
- `rsvp`, `rsvp6` Používá se při routování paketů protokolem RSVP. Lze použít pouze u sítí, které máme pod kontrolou. Internet nerespektuje RSVP.
- `tcindex` Používá se ve spojení s disciplínou DSMARK.

.

Poznámka:

Identifikátory `1:` a `1:0` jsou shodné. Specifikace IP adresy konkrétní stanice například `10.0.0.1` se shoduje se zápisem `10.0.0.1/32` (/32 udává počet bitů v masce, je to zkracený zápis masky v dekadickém tvaru 255.255.255.255).

filter

Výpis ze souboru `iproute2/tc/tc_filter.c`

```
Usage: tc filter [ add | del | change | get ] dev STRING
       [ pref PRIO ] [ protocol PROTO ]
       [ estimator INTERVAL TIME_CONSTANT ]
       [ root | classid CLASSID ] [ handle FILTERID ]
       [ [ FILTER_TYPE ] [ help | OPTIONS ] ]

       tc filter show [ dev STRING ] [ root | parent CLASSID ]
Where:
FILTER_TYPE := { rsvp | u32 | fw | route | etc. }
FILTERID := ... format depends on classifier, see there
OPTIONS := ... try tc filter add <desired FILTER_KIND> help
```

filter fw

Výpis ze souboru `iproute2/tc/f_fw.c`

```
Usage: ... fw [ classid CLASSID ] [ police POLICE_SPEC ]
       POLICE_SPEC := ... look at TBF
       CLASSID := X:Y
```

filter u32

Výpis ze souboru `iproute2/tc/f_u32.c`

```
Usage: ...u32 [ match SELECTOR ... ] [ link HTID ] [ classid CLASSID ]
                [ police POLICE_SPEC ] [ offset OFFSET_SPEC ]
                [ ht HTID ] [ hashkey HASHKEY_SPEC ]
                [ sample SAMPLE ]
or        u32 divisor DIVISOR

Where: SELECTOR := SAMPLE SAMPLE ...
       SAMPLE := { ip | ip6 | udp | tcp | icmp | u{32|16|8} }
SAMPLE_ARGS
       FILTERID := X:Y:Z
```

filter route

Výpis ze souboru `iproute2/tc/f_route.c`

```
Usage: ... route [ from REALM | fromif TAG ] [ to REALM ]
                 [ flowid CLASSID ] [ police POLICE_SPEC ]
       POLICE_SPEC := ... look at TBF
       CLASSID := X:Y\n
```

filter rsvp

Výpis ze souboru `iproute2/tc/f_rsvp.c`

```
Usage: ... rsvp ipproto PROTOCOL session DST[/PORT | GPI ]
                [ sender SRC[/PORT | GPI ]
                [ classid CLASSID ] [ police POLICE_SPEC ]
                [ tunnelid ID ] [ tunnel ID skip NUMBER ]
Where: GPI := { flowlabel NUMBER | spi/ah SPI | spi/esp SPI |
                u{8|16|32} NUMBER mask MASK at OFFSET}
       POLICE_SPEC := ... look at TBF
       FILTERID := X:Y
```

police

Výpis ze souboru `iproute2/tc/m_police.c`

```
Usage: ... police rate BPS burst BYTES[/BYTES] [ mtu BYTES[/BYTES] ]
                [ peakrate BPS ] [ avrate BPS ]
                [ ACTION ]
Where: ACTION := reclassify | drop | continue
```

4 Koncové disciplíny

Každá z těchto disciplín (*Classless Queuing Disciplines*, zkráceně *qdisc*) může být použita jako primární disciplína síťového rozhraní, nebo jako disciplína koncové třídy (běžně nazývané jako *leaf class* neboli *list*) v hierarchických disciplínách (*Classfull Queuing Disciplines*, zkráceně opět *qdisc*). Z tohoto důvodu jsem tyto disciplíny nazval jako koncové.

Koncové disciplíny neobsahují jiné disciplíny. Určují pouze, jak bude s paketem naloženo, zda bude zařazen, zdržen nebo zahozen.

Poznamenejme, že výchozí disciplínou je `pfifo_fast`.

qdisc

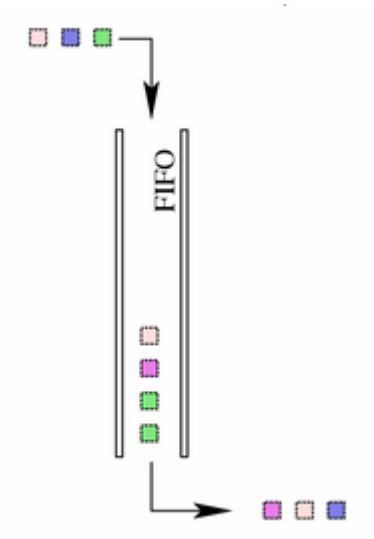
Výpis ze souboru `iproute2/tc/tc_qdisc.c`

```
Usage: tc qdisc [ add | del | replace | change | get ] dev STRING
       [ handle QHANDLE ] [ root | ingress | parent CLASSID ]
       [ estimator INTERVAL TIME_CONSTANT ]
       [ [ QDISC_KIND ] [ help | OPTIONS ] ]

       tc qdisc show [ dev STRING ] [ingress]
Where:
QDISC_KIND := { [p|b]fifo | tbf | prio | cbq | red | etc. }
OPTIONS := ... try tc qdisc add <desired QDISC_KIND> help
```

FIFO, First-In First-Out (`pfifo` a `bfifo`)

Algoritmus FIFO tvoří základ výchozí disciplíny síťových rozhraní v Linuxu `pfifo_fast`. Neovlivňuje ani nepřeuspořádává pakety. Jednoduše přidává příchozí pakety do fronty, kde se řadí za sebe podle toho, jak přišly.



Obrázek 4.1: Algoritmus FIFO

V reálných podmínkách musí mít fronta omezenou velikost (*buffer*), snažící se zabránit přetečení v případě, že pakety jsou z fronty odebírány pomaleji než přidávány. Velikost fronty se definuje parametrem `limit`. V Linuxu jsou implementovány dvě základní disciplíny: `pfifo` s velikostí fronty definovanou počtem paketů, a `bfifo` s velikostí fronty definovanou počtem bajtů.

Výpis ze souboru `iproute2/tc/q_fifo.c`

```
Usage: ... [p|b]fifo [ limit NUMBER ]
```

Vytvoření fronty `pfifo` (parametr `limit` je nastaven na 30 paketů):

```
tc qdisc add dev eth0 handle 2:0 parent 1:0 pfifo limit 30
```

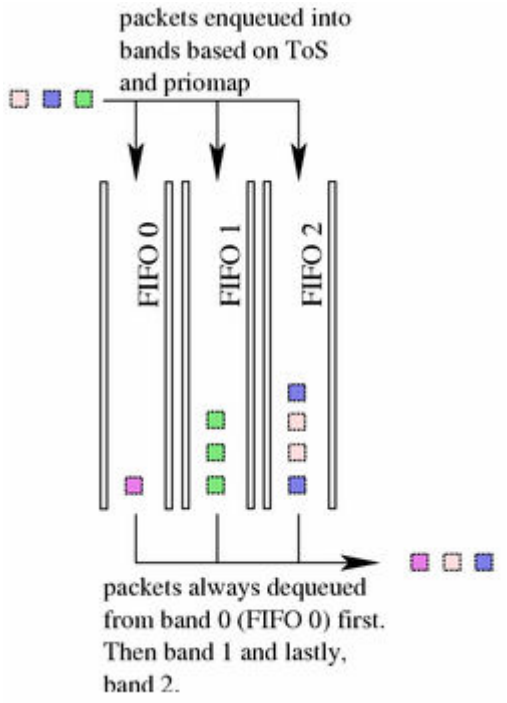
Vytvoření fronty `bfifo` (parametr `limit` je nastaven na 10 kilobajtů):

```
tc qdisc add dev eth0 handle 2:0 parent 1:0 bfifo limit 10240
```

Výchozí disciplína `pfifo_fast`

Disciplína pfifo_fast je výchozí disciplínou pro všechna síťová rozhraní v Linuxu. Jejím základem je disciplína FIFO, ovšem na rozdíl od ní umožňuje určitou priorizaci. Toky se dělí do třech skupin (individuální fronty FIFO). Toky s nejvyšší prioritou

(interaktivní) jsou ve skupině 0 a jsou vždy odbavovány jako první. Podobně tomu je ve skupině 1, která je vyprazdňována před skupinou 2.

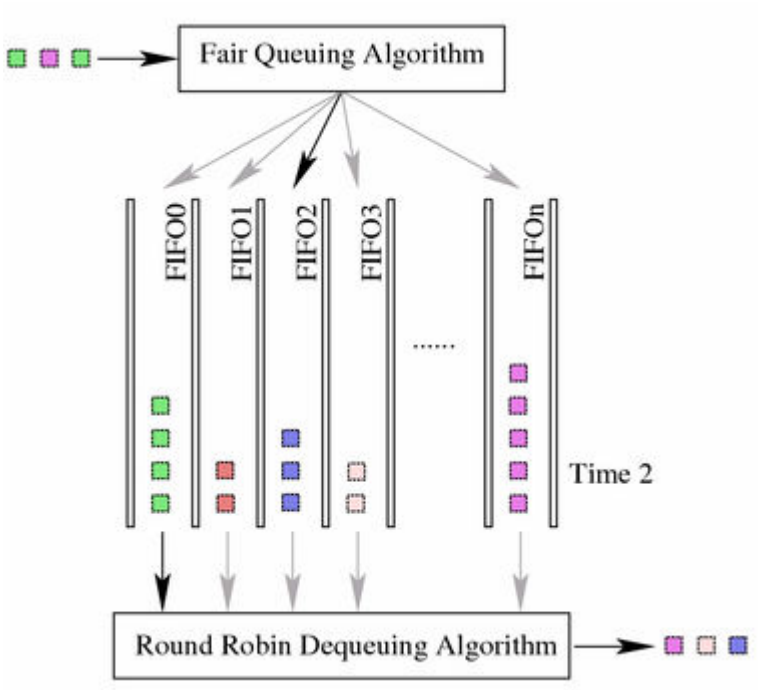


Obrázek 4.2: Algoritmus `pfifo_fast`

Pakety jsou rozdělovány podle nastavení TOS bitů ve své hlavičce. Administrátor nemá možnost disciplínu `pfifo_fast` jakkoli konfigurovat.

SFQ, Stochastic Fair Queuing

Discíplína SFQ umožňuje spravedlivé rozdělení šance na přenos dat libovolného počtu toků. Je to umožněno díky použité hash funkci, která rozděluje pakety (podle jejich hlavičky) do oddělených FIFO front, které jsou cyklicky vyprazdňovány (algoritmus round robin). Protože je tu možnost nespravedlivého rozdělování vyplývající z vybrané hash funkce, mění se funkce periodicky. Periodu určuje parametr `perturb`.



Obrázek 4.3: Algoritmus SFQ

Výpis ze souboru `iproute2/tc/q_sfq.c`

```
Usage: ... sfq [ perturb SECS ] [ quantum BYTES ]
```

Vytvoření disciplíny SFQ:

```
tc qdisc add dev eth0 handle 2:0 parent 1:0 sfq perturb 10
```

SFQ nezajišťuje rovnoměrné rozdělení mezi uživatele či programy. Některé programy (například Kazaa, eMule, DC a dalších) otevírají tolik spojení, kolik jim je dovoleno, a tím férovost SFQ naprosto zlikvidují. V sítích se slušnými uživateli může SFQ přiměřeně rozdělovat síťové zdroje mezi jednotlivé toky, ale v jiném případě může dojít k zahlcení linky jedním p2p programem.

Každý tok má šanci na vyřízení a nestane se, aby jeden objemný tok zahltil většinu linky.

ESFQ, Extended Stochastic Fair Queuing

Koncepčně není disciplína ESFQ odlišná od SFQ, umožňuje ale nastavit více parametrů. Můžeme zvolit hash funkci pro rozdělení linky, a je možné dosáhnout opravdu rovnoměrného rozdělení mezi uživatele.

```
Usage: ... esfq [ perturb SECS ] [ quantum BYTES ] [ depth FLOWS ]
        [ divisor HASHBITS ] [ limit PKTS ] [ hash HASHTYPE]

Where:
HASHTYPE := { classic | src | dst }
```

RED, Random Early Detect

Algoritmus RED zamezuje zahlcení linky (využívá vlastnosti *slow start* protokolu TCP). Pokud celkový datový tok dosáhne stanovené hranice, je náhodně zahozen některý TCP paket, a tím je příslušný zdroj toku donucen ke snížení vysílací rychlosti. Každý paket je zahozen v jiném časovém okamžiku, tak aby nedošlo k situaci, kdy všechny zdroje sníží svou vysílací rychlost a kapacita zůstane nevyužita.

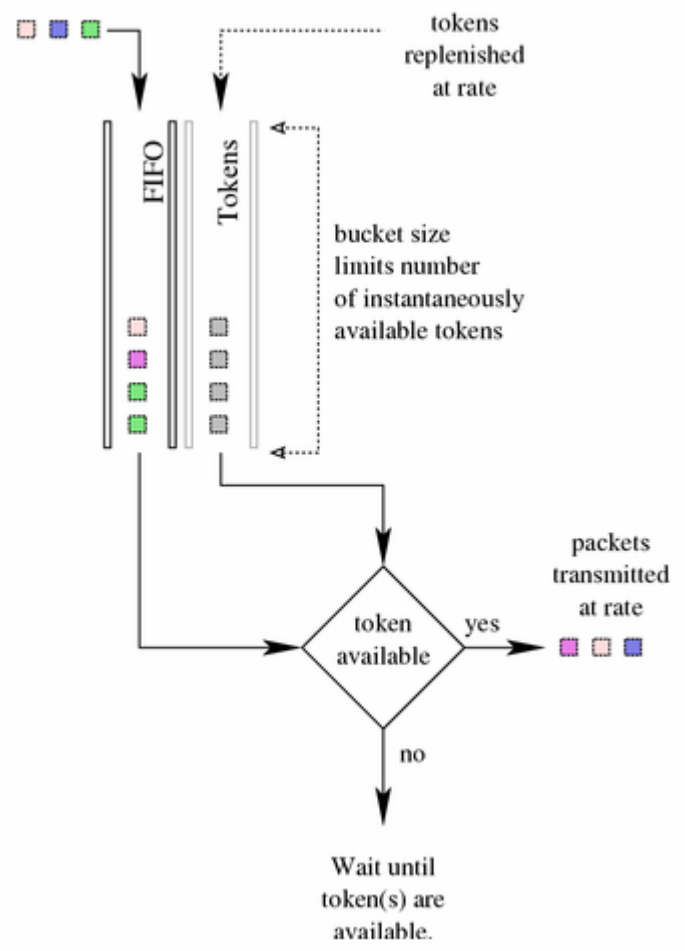
RED žádným způsobem nemění pořadí paketů, chová se ke všem paketům stejně. Má nízké výpočetní nároky a snaží se udržovat co největší propustnost a co nejnižší zpoždění linky.

Výpis ze souboru `iproute2/tc/q_red.c`

```
Usage: ... red limit BYTES min BYTES max BYTES avpkt BYTES burst
PACKETS probability PROBABILITY bandwidth KBPS
```

TBF, Token Bucket Filter

Tato disciplína funguje na principu *token* a *bucket* (česky kupon a vědro). Výborně se hodí k omezení rychlosti, kterou jsou pakety vybírány z fronty, a tím zpomaluje tok dat na požadovanou rychlost. Bucket je plněn tokeny. Pakety jsou propuštěny pouze pokud je k dispozici požadovaný token, jinak jsou pakety pozdrženy. Toto zdržení paketu má samozřejmě vliv na dobu odezvy. Algoritmus TBF je velice jednoduchý a nenáročný na výkon.



Obrázek 4.4: Algoritmus TBF

Výpis ze souboru `iproute2/tc/`

```
Usage: ... tbf limit BYTES burst BYTES[/BYTES] rate KBPS [ mtu
BYTES[/BYTES] ] [ peakrate KBPS ] [ latency TIME ]
```

Vytvoření disciplíny TBF s rychlostí 256kbit/s:

```
tc qdisc add dev eth0 handle 2:0 parent 1:0 tbf burst 20480 limit
20480 mtu 1514 rate 32000bps
```

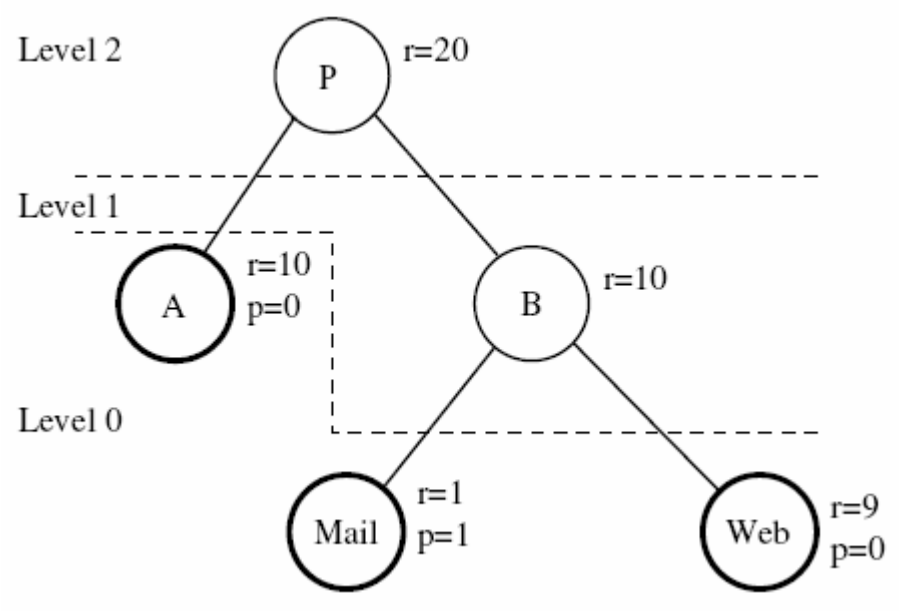
Přehled ostatních disciplín

Název	Popis
TEQL (Trafic Equalizer)	TEQL je pseudo device sdružující dva nebo více reálných rozhraní. Datový tok je pak rovnoměrně rozdělen mezi jednotlivé fyzické linky.
DS_MARK (Diffserv Marker)	Tato disciplína je určena pro implementaci v DiffServ (více o DiffServ naleznete v příloze na straně 72). Pracuje s oktetem TOS z IP záhlaví paketu a umožňuje dynamicky měnit jeho hodnotu (typ třídy podle výsledků značkování).
CSZ (Clark-Shenker-Zhang algoritmus)	CSZ algoritmus dovoluje garantovat datovým tokům minimální přenosové pásmo i maximální zpoždění. Současná implementace algoritmu je komplikovaná a s velkou režií. Ani její autoři ji nedoporučují používat.
ATM (Asynchronous Transfer Mode)	Pomocí této disciplíny je do modelu QoS integrována celá implementace ATM v Linuxu.

5 Hierarchické disciplíny

Hierarchické disciplíny (*Classfull Queuing Disciplines*, zkraceně *qdisc*) jsou velice užitečné v případě, kdy máme různé toky a chceme s nimi rozdílně naložit.

Jak to všechno funguje?



Obrázek 5.1: Sdílení linky

Obrázek 5.1 znázorňuje problém, kdy provider **P** potřebuje rozdělit internetové pásmo mezi firmy označené **A** a **B**. Každá firma má specifický požadavek na garantovaný tok *r* a maximální odezvy. Velikost odezvy přitom záleží na prioritě *p* (vyšší priorita má menší číselnou hodnotu). Firma **B** si navíc přeje oddělit datové toky pro mailové a webové služby. Zavedeme si základní terminologii:

Tabulka 5.1: Zavedená terminologie

Název	Popis
třída	je uzel sdílecího stromu a má definovaný požadovaný tok *r*
uzel	je pouze jiný název pro třídu
list	označuje uzel, který nemá žádné další následovníky, pouze listy mohou obsahovat pakety ve své interní frontě
vnitřní uzel	je uzel, který není listem
plný list	je list, který obsahuje nějaké pakety
plný vnitřní uzel	je uzel s potomkem typu plný list
podlimitní uzel	je uzel, jehož aktuální tok je menší než jeho *r*, o podlimitnosti rozhoduje *estimátor*
úroveň	uzlu (level) je hloubka uzlu v hierarchickém stromu, která je pro list definována jako 0. Způsob číslování je zřejmý z obrázku 5.1

Cílem hierarchických disciplín je pro každé vyvolání určit, který další paket má být odeslán. Obecně vzato, můžeme odeslat paket ze všech *nespokojených* listů, nejlépe v pořadí priorit. Pokud takový list není, pokračujeme plnými listy, které si mohou *půjčit* od svých předků.

V následujícím textu budeme sledovat, jak se která disciplína staví k problému „kdy si uzel může půjčit tok" a jakým způsobem se určí, zda je třída pod nebo nad svým limitem. Nejprve si ale něco povíme o klasifikaci datových toků.

Princip klasifikace toků

Paket vstoupí do disciplíny a postupně prochází hierarchií tříd, dokud se nedostane do koncové třídy, neboli do listu. K tomu, abychom určili, co s paketem uděláme (kam bude zařazen), slouží filtry.

Filtry jsou volány „ze vnitř“ disciplíny.

Filtry jsou spjaty s disciplínou a na jejich rozhodování záleží, do jaké třídy se daný paket zařadí. Každá podtřída zkusí jiný filtr a tím se určí další zařazení. Pokud paket žádnému fitru nevyhovuje, je zafrontován do koncové disciplíny, kterou třída obsahuje.

Většina hierarchických disciplín kromě toho, že mohou obsahovat koncové disciplíny, umožňuje také řízení rychlosti toku.

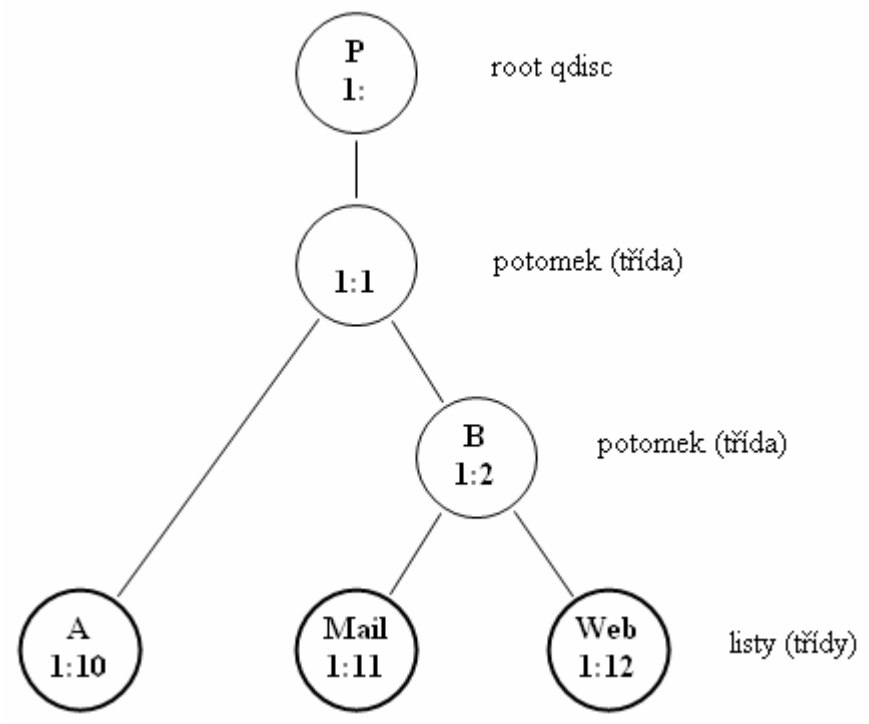
Rodina parametrů: `root`, `handle`, `parent`

Každé síťové rozhraní má jednu `root` disciplinu. Standardně je to disciplína `pfifo_fast`. Identifikátor disciplíny, parametr `handle`, se skládá ze dvou částí <hlavní číslo>:<vedlejší číslo>. Obě čísla jsou 16-bitová a jejich volba je na vás, identifikátor musí být ovšem jedinečný. Hlavní číslo zastupuje disciplínu, vedlejší potom jednotlivé třídy v disciplíně. `handle` všech tříd v jedné hierarchické disciplíně má hlavní číslo shodné s `root` disciplínou a vedlejší číslo jedinečné pro každou třídu uvnitř disciplíny.

> **Poznámka:**
>
> `handle` **třídy z různých** `root` **disciplín se nesmí shodovat v hlavním čísle, ale vedlejší čísla mohou být stejná. Vedlejší číslo identifikátoru disciplíny (koncové i hierarchické) je vždy 0. Lze ho také vynechat.**

Jak se používají filtry ke klasifikaci toků?

Mějme situaci jako je na obrázku 5.1, ale tentokrát zobrazíme její implementaci v hierarchické disciplíně, což znázorňuje obrázek 5.2.



Obrázek 5.2: Situace z obrázku 5.1 doplněná o identifikátory

Paket může být postupně klasifikován a procházet třídami například takto:

1: → 1:1 → 1:2 → 1:12

Paket se zařadí do fronty v listu 1:12. V tomto případě byly filtry spjaty se všemi uzly stromu a určovaly, jaká větev bude následovat.

Paket ovšem může být klasifikován přímo:

1: → 1:12

V tomto případě filtr spjatý s `root` pošle paket přímo do třídy 1:12.

CBQ

CBQ je nejznámější a zároveň co se týče konfigurovatelnosti nejsložitější hierarchická disciplína. Pracuje na základě vytíženosti linky. Pokud chceme například omezit rychlost na 1Mbit na 10Mbitové lince, pak tato linka musí být z 90% volná. Pro velký počet parametrů se stavá z konfigurace CBQ poměrně složitá záležitost. Každé třídě lze nastavit, zda si může půjčovat a zda půjčuje ostatním třídám. Na rozdíl od HTB nelze nastavit kolik si lze maximálně půjčit.

Půjčování

Jedním z pravidel, které je nutno respektovat, je:

Pravidlo 1 *Je-li X úroveň nejvyššího nespokojeného uzlu, není možné si půjčit od uzlu s úrovní > X.*

Pokud algoritmus nebude toto pravidlo respektovat, v modelu na obrazku 5.1 nastane situace, kdy listy s vyšší prioritou (**A** a **Web**) zahltí celou linku a **Mail** nedostane ani svůj garantovaný tok. Při respektování pravidla 1 toto nenastane, neboť **Mail** je *nespokojeným* uzlem a ostatní si nemohou půjčovat z úrovně jiné než 0 (neboli nemohou si půjčovat vůbec).

Algoritmus CBQ implementuje toto pravidlo zavedením globální proměnné *toplevel*, která odráží stav *X* z pravidla 1. Pří přidávání a odebírání paketu je proměnná přepočítávána. Obě tyto akce totiž ovlivní, zda je list plný či ne a definice nespokojeného uzlu (použitá v pravidle 1) závisí na plnosti listu.

Nespokojenost uzlu závisí také na aktuálním toku uzlu (zda je podlimitní). Uzel, který je nad limitem, se ovšem může stát podlimitním nejen během přidání či odebrání paketu, ale i po uplynutí dostatečně dlouhé doby, kdy uzel nebyl využíván. Tato asynchronní událost není v toplevel algoritmu zohledněná, a proto CBQ implementace, které toplevel využívají, jsou zatíženy jistou chybou.

Estimátor

Limitní stav je detekován měřením doby, která uplyne mezi odesláním dvou za sebou následujících paketů z měřené třídy. Čas se porovná s dobou vypočítanou z požadované rychlosti a znaménko získané odchylky určuje, zda je třída nad či pod limitem.

Z toho vyplývá, že tento algoritmus potřebuje ke své funkci znát přesnou rychlost fyzického rozhraní. Není problém určit rychlost ethernetového rozhraní, ale je to nemožné určit u zařízení jako je WiFi nebo u softwarových zařízení jako jsou tunely. Podobná zařízení totiž nemají pevnou rychlost odesílání dat, jelikož jejich aktuální rychlost kolísá a závisí na mnoha podmínkách (kvalita signálu, momentální zatížení routeru atd.).

cbq qdisc

Výpis ze souboru `iproute2/tc/q_cbq.c`

```
Usage: ... cbq bandwidth BPS rate BPS maxburst PKTS [ avpkt BYTES ]
               [ minburst PKTS ] [ bounded ] [ isolated ]
               [ allot BYTES ] [ mpu BYTES ] [ weight RATE ]
               [ prio NUMBER ] [ cell BYTES ] [ ewma LOG ]
               [ estimator INTERVAL TIME_CONSTANT ]
               [ split CLASSID ] [ defmap MASK/CHANGE ]
```

class

Výpis ze souboru `iproute2/tc/tc_class.c`

```
Usage: tc class [ add | del | change | get ] dev STRING
       [ classid CLASSID ] [ root | parent CLASSID ]
       [ [ QDISC_KIND ] [ help | OPTIONS ] ]

       tc class show [ dev STRING ] [ root | parent CLASSID ]
Where:
QDISC_KIND := { prio | cbq | etc. }
OPTIONS := ... try tc class add <desired QDISC_KIND> help
```

estimator

Výpis ze souboru `iproute2/tc/m_estimator.c`

```
Usage: ... estimator INTERVAL TIME-CONST
  INTERVAL is interval between measurements
  TIME-CONST is averaging time konstant
Example: ... est 1sec 8sec\
```

Tabulka 5.2: Parametry CBQ

Parametr	Význam
Konfigurace shapingu	
`avpkt`	Průměrná velikost paketu (udávaná v bajtech). Využívá se pro výpočet `maxidle`, který je odvozen z `maxburst` (udává se

	v paketech)
`bandwidth`	Fyzická propustnost rozhraní (například 100Mbit), potřebná pro výpočet idle time.
`cell`	Doba, po kterou je paket přenášen se může měnit po skocích, na základě velikosti paketů. To způsobuje „zrnitost“. Většinou je nastavena na 8 (musí být mocnina dvou).
`maxburst`	Počet paketů používaný pro výpočet `maxidle`. Pokud je `avgidle` na hodnotě `maxidle`, tento počet paketů může být odeslán (burst) před tím než `avgidle` klesne na 0. Vyšší hodnota zvyšuje toleranci k burst. Parametr `maxidle` nelze nastavit přímo, pouze prostřednictvím `maxburst`.
`minburst`	Podobné s `maxburst`. Vzhledem k tomu, že je nemožné zpracovávat události s periodou kratší než 10ms (dáno jádrem), je vhodné prodloužit periodu, a potom odeslat najednou určitý počet paketů, specifikovaný parametrem `minburst`.
`minidle`	Když je `avgidle` pod 0, znamená to, že tok je nad limitem a musí se počkat, dokud nebude `avgidle` na takové hodnotě, aby se mohl poslat jeden paket. V případě, že se čekací interval prodlužuje, je `avgidle` nastaven na `minidle`.
`mpu`	*Minimum packet size* (minimální velikost paketu). CBQ potřebuje tuto hodnotu k přesnému výpočtu idle time.
`rate`	Požadovaná rychlost (rovná se minimální rychlosti u tříd propůjčujících si kapacitu od nadřazených tříd).

Parametry ovlivňující chování CBQ – řídí WRR (Weighted Round Robin) proces

`allot`	Při požadavku na odeslání paketu se zkouší všechny koncové disciplíny (v listech) v pořadí jejich priorit. Každá z nich může odeslat omezené množství dat, jehož základní hodnotu určuje právě parametr `allot`.

`prio`	Nastavení priorit. Vychozí hodnota je 5.
`weight`	List s velkou propustností by měl mít šanci poslat během jednoho cyklu více dat než list s menší propustností, což umožňuje právě parametr `weight`. Často je nastavován na `rate/10`.

Sdílení a půjčování linky

`isolated/sharing`	Třída nastavená jako `isolated` nepůjčí své nevyužité pásmo jiným třídám. Opakem je `sharing`.
`bounded/borrow`	Třída nastavená jako `bounded` si nepůjčí nevyužité pásmo od nadřazené třídy. Opakem je `borrow`.

Inicializační skript CBQ.init

Z výše uvedeného je zřejmé, že konfigurace CBQ není zcela jednoduchá. Proto bylo vytvořeno několik skriptů, přívětivých pro lidi neznalé problematiky, které veškeré nastavení nástroje `tc` udělají za nás. Nejznámější a nejrozšířenější je skript cbq.init od Pavla Golebeva, který je volně k dispozici na adrese

`https://sourceforge.net/projects/cbqinit/`

HTB

Nepřehlednost příkazů CBQ a také fakt, že CBQ není úplně přesné, bylo podnětem pro vytvoření HTB (*Hierarchical Token Bucket*), jehož autorem je Martin Devera.

Implementace estimátoru využívá takzvaný *Leaky Bucke*. Výhodou je jeho intuitivní nastavení, nezávislost na fyzické rychlosti rozhraní a také to, že nevyžaduje přesný systémový časovač.

Každá třída má kromě garantovaného toku *r* i maximální tok *c* (anglicky ceil, neboli strop). Stav třídy z hlediska aktuální velikosti toku budeme označovat barvami semaforu. Barvu třídy nadefinujeme pro daný časový okamžik *t* a aktuální tok *a(t)* jako:

zelenou pokud platí $a(t) < r$, tedy třída je pod limitem,

žlutou pokud $r \leq a(t) \leq c$, taková třída je sama pod limitem, ale nedosáhla stropu, tudíž si může zkusit půjčit od rodiče, nebo

červenou pro $a(t) > c$. Červená třída přesáhla strop, a proto již nemůže odeslat žádný paket.

Hierarchie a půjčování

Definujme algoritmus pro výběr dalšího paketu k odeslaní:

Algoritmus 1 *Ze všech plných listů volme takový, který by si při odesílání paketu mohl půjčit tok od rodiče na nejnižší úrovni. Pokud je takových listů více, vybereme ten s nejvyšší prioritou. Pokud máme stále více listů, střídáme je pravidelně podle poměru jejich* ***r****.*

List si může korektně půjčit sám od sebe. Algoritmus 1 splňuje následující očekávání:

- Garantované toky jsou vždy splněny.
- Toky tříd se stejnou prioritou si půjčují tok od společného rodiče v poměru jejich *r*.
- Třída s vyšší prioritou získá tok od společného rodiče přednostně.
- Třída s vyšší prioritou má kratší odezvy.

Výběr paketu

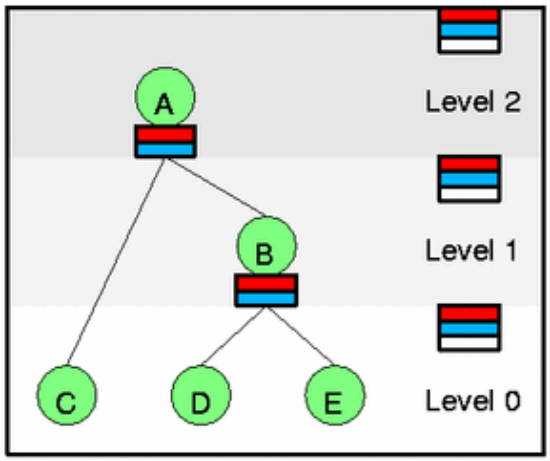
Během výběru paketu musíme najít zelenou třídu s nejnižší úrovní, která má potomka plný list a k potomkovi vede cesta se žlutými třídami (včetně listů). Pokud bychom toto hledání realizovali procházením všech tříd během výběru paketu, složitost by byla $O(N)$, kde N je počet tříd, což je neúnosné.

Budeme tedy muset udržovat informace o aktuálním stavu. Každý vnitřní uzel si udržuje seznam potomků, kteří by si od něj rádi půjčili nějaký tok. Podle definice to mohou být pouze plné žluté třídy. Tento seznam (takzvaný *interní půjčovací seznam*)

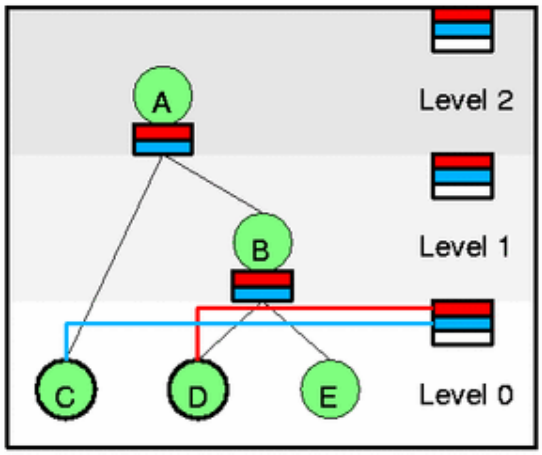
musí být veden zvlášť pro jednotlivé priority, neboť uzel může být předkem listu s libovolnou prioritou. Naše příklady mají pouze dvě priority.

V pravé části obrázků 5.3 až 5.8 vidíme podobné obdélníčky. Ignorujme prozatím nejspodnější bílý. Dva horní tvoří takzvaný *globální půjčovací seznam*. Seznam obsahuje třídy určité úrovně a priority, které jsou plné a zároveň zelené (tedy nespokojené). Jinými slovy, třídy, které jsou na jakémkoli globálním seznamu, jsou připraveny odeslat paket. Nyní tedy v souladu s algoritmem 1 stačí vybrat nejspodnější úroveň s neprázdným globálním seznamem, zde seznam s nejvyšší prioritou, a sledovat interní seznamy, které nás dovedou až k listu, který má odeslat další paket.

Obrázek 5.3 ukazuje stav, kdy v HTB není žádný paket a všechny třídy jsou zelené. Pokud přijdou pakety listům **C** a **D**, listy se změní na plné, a protože jsou zatím i podlimitem (neboli jsou stále ještě zelené), můžeme je připojit do globálních seznamů pod patřičné priority, což znázorňuje obázek 5.4. V tomto okamžiku by se pakety odesílaly z listu **D**, a po jeho vyprázdnění případné další pakety z listu **C**.



Obrázek 5.3: Žádné pakety



Obrázek 5.4: Pakety v C a D

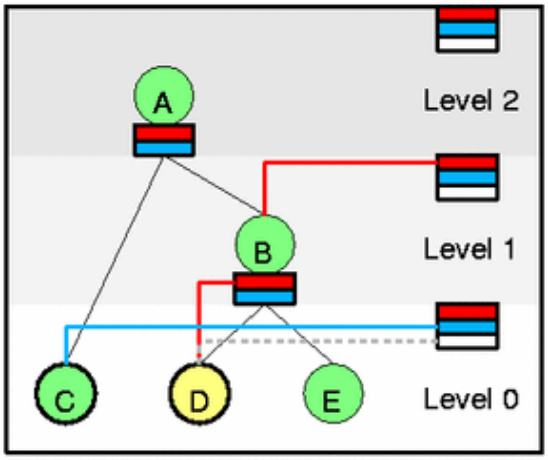
Pokud pošleme paket z **D** a estimátor rozhodne, že je **D** žluté (došlo k přetečení **r**), odpojí se **D** od globálního listu, neboť již nemůže odesílat bez půjčky, a zařadí se na vysokoprioritní interní seznam třídy **B**. Aby **D** mohlo nadále posílat pakety výpůjčkami od **B**, je samo **B** zařazeno na globální list úrovně 1 (vysoké priority). Tento stav vidíme na obrázku 5.5.

Pokud **D** nějakou dobu neodešle paket, změní se jeho barva v určitém čase opět na zelenou. Proto zařadíme **D** na speciální *čekací seznam*, který má každá úroveň. Toto

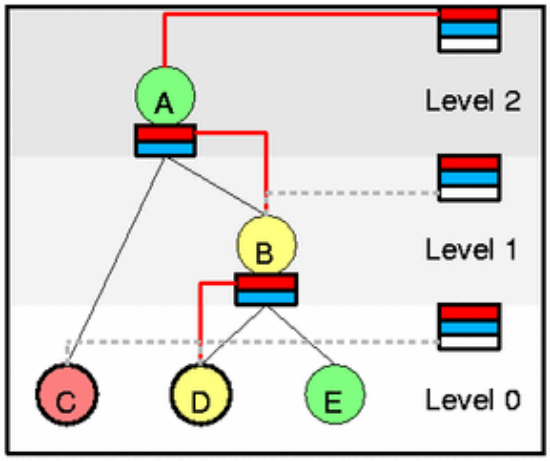
zařazení je naznačeno čárkovanou čarou. Po uplynutí předem spočítaného času se změní barva třídy. Čekací seznam řeší problém CBQ.

Při žádosti na odeslání dalšího paketu bude vybráno **C** nikoli **D**, a to i přes to, že **D** má vyšší prioritu. Je to proto, že v globálním seznamu začíná cesta k **C** na nižší úrovni. **C** má nárok na svůj garantovaný tok, tudíž dostane přednost před **D**.

Obrázek 5.6 zobrazuje situaci po odeslání paketu z **C**. Řekněme, že tato třída překročí **c** a je tudíž červená, navíc **B** je nyní žlutá. **B** se připojí do interního seznamu **A**, a místo **B** je v globálním seznamu nyní **A**. Červené **C** je pouze na čekacím seznamu (nemůže sama poslat paket ani si půjčit). Z globálního seznamu je zřejmé, že jediná třída, která může poslat, je **D**.



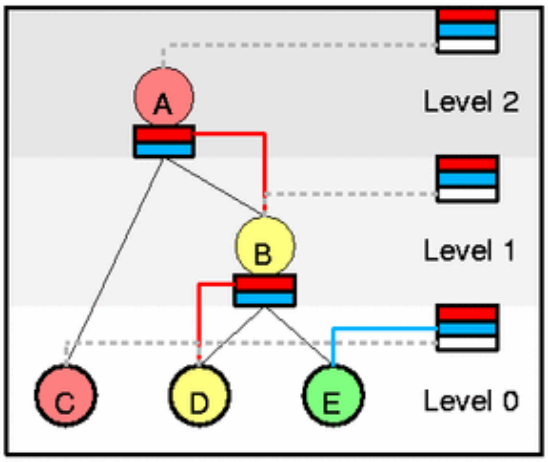
Obrázek 5.5: D přesáhlo svého rate



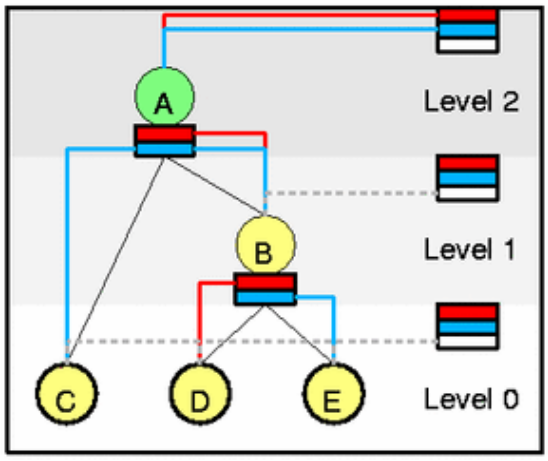
Obrázek 5.6: C dosáhlo svého ceil, B dosáhlo svého rate

Na obrázku 5.7 vidíme situaci, kdy **A** je červené a **E** je plné zelené. Je vidět, že cesta po interních seznamech existuje, i když nejvyšší třída nemá žádný spoj na globální seznam. Pakety budou odesílány z **E**.

Obrázek 5.8 poukazuje na možnost napojení více tříd na společné interní seznamy. Nebýt vyšší priority **D**, byli bychom nuceni střídat odebírání paketu mezi všemi listy v poměru jejich *r*. Jelikož má **D** vyšší prioritu, bude z globálního seznamu vybíráno pouze **D**, a po jeho vyprázdnění se dostane na ostatní.



Obrázek 5.7: A dosáhlo svého ceil



Obrázek 5.8: Všechny třídy si půjčují od A

Inicializační skript HTB.init

HTB.init je shellový skript odvozený od CBQ.init, který umožňuje jednoduché nastavení traffic control založené na HTB. K dispozici je na adrese

```
https://sourceforge.net/projects/htbinit/
```

Nyní si uvedeme několik příkladů, na nichž bude vše demonstrováno.

Příklady a simulace

Příklad 1

Mějme situaci z obrázku 5.2.

Nejprve přidáme na eth0 disciplínu HTB s identifikátorem 1:.

```
tc qdisc add dev eth0 root handle 1: htb default 10
```

Potom vytvoříme jednotlive třídy.

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps \
   ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:2 htb rate 50kbps \
   ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 50kbps \
   ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:11 htb rate 10kbps \
   ceil 100kbps
```

```
tc class add dev eth0 parent 1:2 classid 1:12 htb rate 40kbps \
   ceil 100kbps
```

Prvním příkazem vytvoříme „root“ třídu 1:1, s rodičem qdisc 1:. Root třída, stejně jako ostatní třídy v HTB disciplíně, umožňuje svým potomkům půjčovat si jeden od druhého, ale jedna root třída si nemůže půjčit od druhé. Pokud bychom další tři třídy vytvořili přímo pod disciplínou, nemohly by si navzájem půjčovat nevyužité pásmo. Proto třídy musí být pod jednou root třídou, čehož je dosaženo zbylými příkazy.

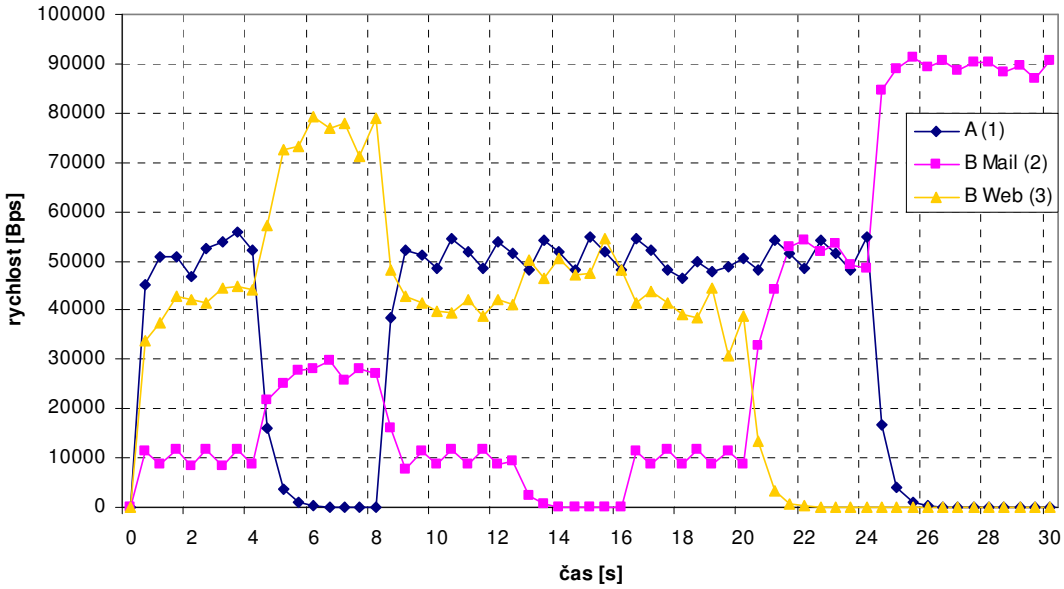
Dále musíme určit jaké pakety budou patřit do které třídy. K tomu použijeme například filtr u32. Klasifikujeme pouze toky uživatele B, veškeré ostatní toky (tedy toky uživatele A) jsou směrovány do třídy 1:12 díky parametru `default` v definici disciplíny.

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 match ip dport 80 0xffff flowid 1:12
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 flowid 1:11
```

Toto jsou všechny příkazy, které potřebujeme. Můžeme ještě případně specifikovat koncové disciplíny v listech.

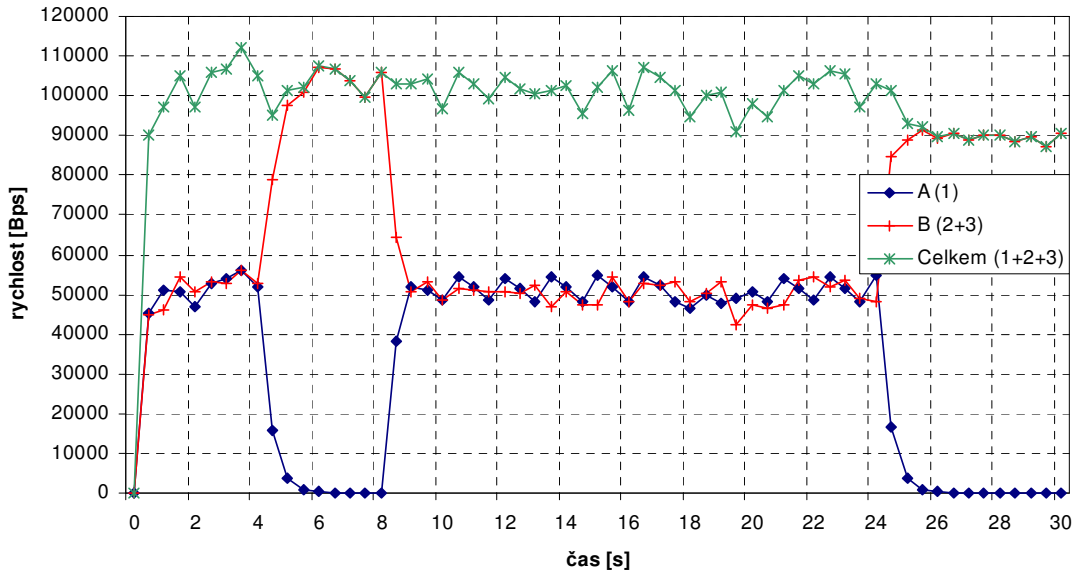
```
tc qdisc add dev eth0 parent 1:10 handle 10: pfifo limit 5
tc qdisc add dev eth0 parent 1:11 handle 20: pfifo limit 5
tc qdisc add dev eth0 parent 1:12 handle 30: sfq perturb 10
```

Podívejme se, co se stane, když do každé třídy pošleme tok 90kbps. Možné průběhy toků 1 (představující uživatele A), 2 a 3 (představující uživatele B) ukazuje graf 5.1. V čase 4s tok 1 klesne na 0kbps. Jeho pásmo si rozdělí toky 2 a 3 v poměru svých `rate`. V 8s opět pustíme tok 1 s požadavkem na 90kbps (návrat k předchozí situaci). V momentě 12s klesne tok 2 na 0kbps. Jak je vidět, jeho pásmo připadne pouze toku 3, a to z toho důvodu, že tok 1 již nabyl svého `rate` ale třída 1:2 (pod kterou jsou toky 2 a 3) byla pod urovní svého `rate`. V 16s opět pustíme tok 2, ve 20s klesne tok 3 na 0kbps a ve 24s klesne tok 1 na 0kbps. Proč ke konci tok 2 nedosáhl svého `rate`? Je to jednoduše proto, že má požadavek na 90kbps.



Graf 5.1: Simulované toky k příkladu 1

Graf 5.2 zobrazuje jak se uživatelé A a B podělili o linku a celkové vytížení linky (součet všech tří toků).



Graf 5.2: Vytížení linky

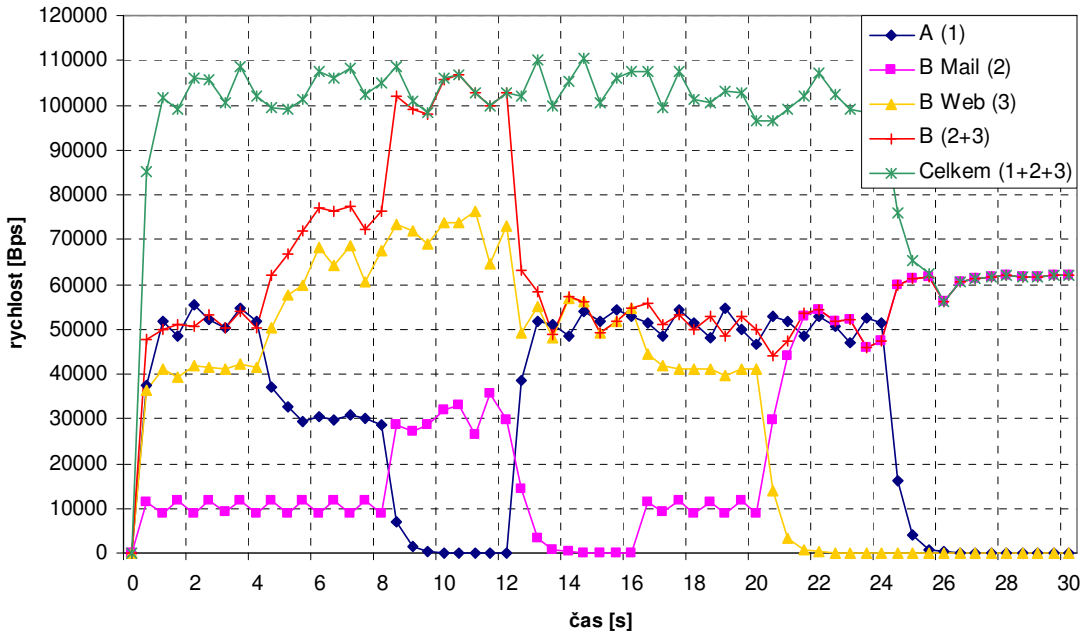
Příklad 2

Příklad 1 rozšíříme o priorizaci a změníme maximální rychlosti toků.

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps \
```

```
   ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:2 htb rate 50kbps \
   ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 50kbps \
   ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:11 htb rate 10kbps \
   ceil 60kbps prio 1
tc class add dev eth0 parent 1:2 classid 1:12 htb rate 40kbps \
   ceil 70kbps prio 0
```

Tentokrát tok 1 v čase 4s změníme na 30kbps. Jak je vidět na grafu 5.3, volné pásmo si vzal pouze tok 3, a to díky vyšší prioritě než má tok 2. V 8s snížíme tok 1 na 0kbps a co se stane? Uvolněné pásmo si vezme nejprve tok 3, který ale dosáhne svého `ceil`, zbytek potom připadne toku 2. V momentě 12s opět pustíme tok 1 a zároveň snížíme tok 2 na 0kbps. V 16s pustíme tok 2. Potom postupně vypínáme toky 3 a 1, tok 2 má linku celou pro sebe dosáhne svého `ceil`.



Graf 5.3: Prio a ceil v akci

Příklad 3

Disciplínu HTB v příkladě 2 nahradíme za CBQ.

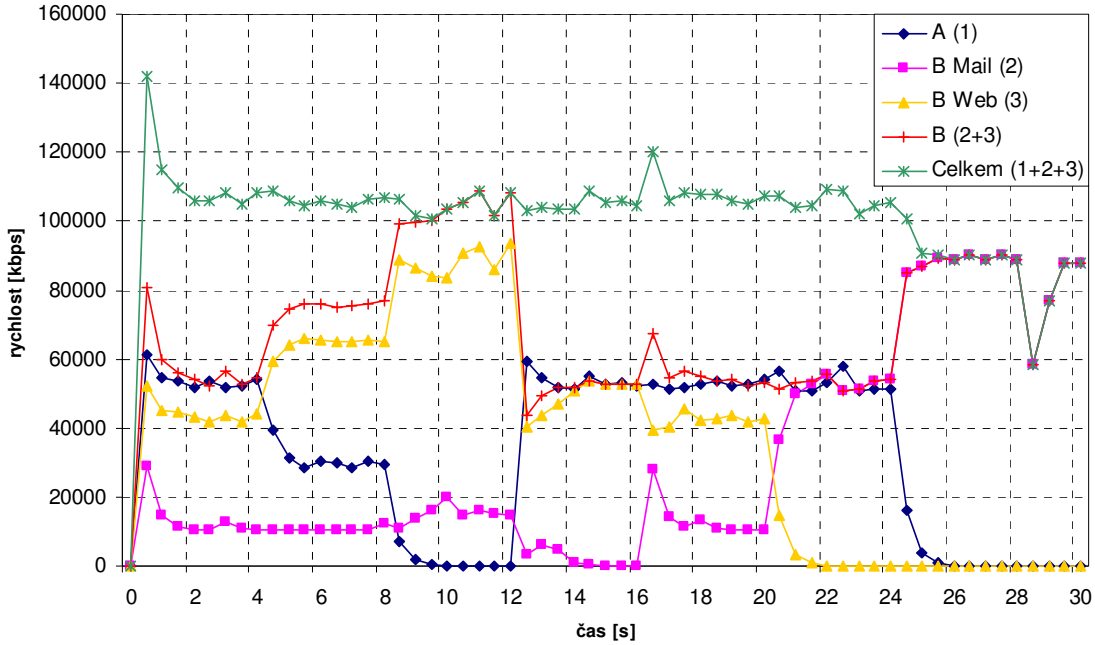
Nejprve přidáme na eth0 disciplínu CBQ s identifikátorem 1:.

```
tc qdisc add dev eth0 root handle 1: cbq bandwidth 100Mbit \
   cell 8 avpkt 1000
```

Potom vytvoříme jednotlive třídy.

```
tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit \
   rate 100kbps weight 100kbit prio 5 allot 1514 cell 8 maxburst 20 \
   avpkt 1000 bounded
tc class add dev eth0 parent 1:1 classid 1:2 cbq bandwidth 100Mbit \
   rate 50kbps weight 50kbit prio 5 allot 1514 cell 8 maxburst 20 \
   avpkt 1000
tc class add dev eth0 parent 1:1 classid 1:10 cbq bandwidth 100Mbit \
   rate 50kbps weight 50kbit prio 5 allot 1514 cell 8 maxburst 20 \
   avpkt 1000
tc class add dev eth0 parent 1:2 classid 1:11 cbq bandwidth 100Mbit \
   rate 10kbps weight 10kbit prio 5 allot 1514 cell 8 maxburst 20 \
   avpkt 1000
tc class add dev eth0 parent 1:2 classid 1:12 cbq bandwidth 100Mbit \
   rate 40kbps weight 40kbit prio 1 allot 1514 cell 8 maxburst 20 \
   avpkt 1000
```

Jak je vidět, CBQ má mnohem více parametrů než HTB. Všimněte si parametru `bounded` v definici třídy 1:1. CBQ neumožňuje nastavit strop, který by omezil třídu v půjčování nevyužitého pásma (postrádá parametr `ceil`, který má HTB), což je názorně vidět na grafu 5.4. V momentě 8s si uvolněné pásmo tokem 1 vezme nejprve tok 3, který dosáhne svého maxima 90kbps, a zbytek potom připadne toku 2. Opět je to způsobeno vyšší prioritou toku 3.



Graf 5.4: CBQ a priorizace

Z porovnání grafů 5.3 a 5.4 je patrné, že CBQ má až na špičky při skokových změnách hladší průběh než HTB.

PRIO

Disciplína `prio` neumožňuje nastavit maximální či garantovanou šířku pásma, dělí pouze datové toky na základě konfigurace filtrů. Disciplína prio není nepodobná koncové disciplíně pfifo_fast, ovšem zde každá skupina představuje třídu místo jednoduché FIFO frotny.

Standardně je tvořena třemi třídami, jež obsahují disciplínu FIFO, která může být nahrazena libovolnou jinou dostupnou koncovou disciplínou.

Když je paket přidáván do disciplíny, vybere se příslušná třída pomocí filtrů.

Pokud je paket naopak odebírán, zkouší se napřed třída :1. Paket z vyšších tříd se odešle pouze v případě, že jsou nižší třídy prázdné (neobsahují ani jeden paket).

Disciplína prio je velice vhodná v případě, že požadujeme priorizaci jistých druhů toků klasifikovaných pomocí filtrů, nejen pouze příznaky TOS.

Výpis ze souboru `iproute2/tc/q_prio.c`

```
Usage: ... prio bands NUMBER priomap P1 P2...
```

Tabulka 5.3: Parametry disciplíny prio

Parametr	Význam
`bands`	Počet skupin, které se mají vytvořit. Každá skupina představuje třídu. Pokud se tento počet změní, musí se změnit i následující parametr `priomap`.
`priomap`	Pokud se ke klasifikaci toků nepoužijí filtry, disciplína `prio` rozhodne o zařazení toku na základě priorit v TC_PRIO.

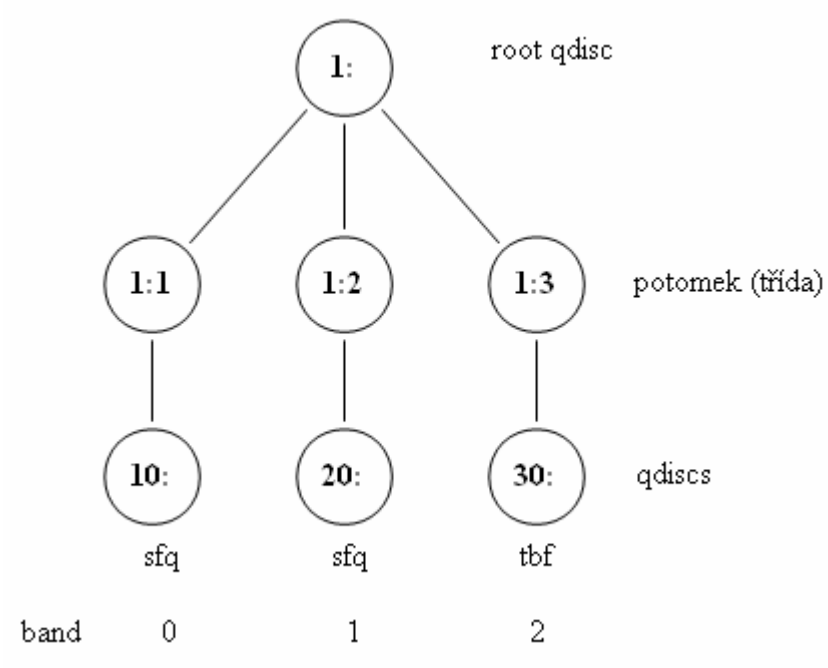
Tabulka 5.4: Definice TC_PRIO bitů

TC_PRIO	Číslo	Odpovídá TOS
`BESTEFFORT`	0	`Maximize Reliability`
`FILLER`	1	`Minimize Cost`
`BULK`	2	`Maximize Throughput (0x8)`
`INTERACTIVE_BULK`	4	
`INTERACTIVE`	6	`Minimize Delay (0x10)`
`CONTROL`	7	

Skupina 0 je klasifikována do třídy *major*:1, skupina 1 do třídy *major*:2 atd.

Příklad 4

Vytvoříme strom jako je na následujícím obrázku 5.9.



Obrázek 5.9: Stromová struktura, kterou chceme vytvořit

Tímto příkazem vytvoříme zároveň i třídy 1:1, 1:2 a 1:3.

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

Nyní přidáme koncové disciplíny.

```
tc qdisc add dev eth0 parent 1:1 handle 10: sfq
tc qdisc add dev eth0 parent 1:2 handle 20: sfq
tc qdisc add dev eth0 parent 1:3 handle 30: tbf rate 200kbit \
   buffer 1600 limit 3000
```

Interaktivní toky půjdou do disciplín 10: nebo 20:, objemné toky (bulk) jsou omezeny a řadí se do 30:.

Závěr

Pokud člověk po přečtení této diplomové práce získa ucelený přehled o řízení datových toků na počítačové síti v prostředí operačního systému GNU/Linux, bylo dosaženo stanoveného cíle.

S rozvojem internetu nachází nástroje traffic control stále větší uplatnění. Častým podnětem k nasazení traffic control je nový software, především stahovací p2p programy, na druhé straně pak interaktivní datové toky jako jsou on-line audio a video přenosy, SSH spojení atd.

Z praxe mohu výhody řízení toků dat jenom potvrdit. Jsem členem neziskového občanského sdružení *ITF-Sdružení pro svobodný přístup k informačním technologiím*, které je provozovatelem počítačové sítě *FreeNet Liberec*. V této době sdružení čítá přibližně 300 členů. Jsem členem Rady sdružení a jedním ze tří hlavních spravců sítě.

Použité zdroje

Následující seznam není zcela uplný, jsou v něm obsaženy pouze nejdůležitější zdroje informacích, z kterých jsem čerpal. Velká většina návodů je dostupná v elektronické podobě na internetu. Vhodným zdrojem informací jsou také diskuzní fóra, kde jsou řešeny konkrétní problémy.

Linux Advanced Routing & Traffic Control HOWTO

`http://lartc.org/howto/`

Ostatní návody na traffic control

`http://www.docum.org/docum.org/kptd`

`http://linux-ip.net/articles/Traffic-Control-HOWTO/`

`http://www.trekweb.com/~jasonb/articles/traffic_shaping/`

Několik okazů s vhodnými informacemi

`http://www.netfilter.org/`

`http://www.digriz.org.uk/jdg-qos-script/`

`http://www.siliconvalleyccie.com/linux-hn/iptables-intro.htm`

`http://www.cesnet.cz/doc/techzpravy/2001/20/`

Přehled TCP a UDP portů

`http://en.wikipedia.org/wiki/TCP_and_UDP_port_numbers`

Domovská stránka HTB

`http://luxik.cdi.cz/~devik/qos/htb/`

Stránka věnovaná CBQ

`http://www.icir.org/floyd/cbq.html`

Linux Dokumentační projekt, 3. aktualizované vydání; kolektiv autorů; Překlad oficiální linuxové dokumentace; Computer Press, a.s.; ISBN 80-7226-761-2; pdf soubor

Používáme Linux, 3. aktualizované vydání; Matt Welch, Matthias Kalle Dalheimer, Terry Dawson, Lar Kaufman; nakladatelství Computer Press, a.s.; ISBN 80-7226-698-5

Linux síťové servery; Craig Hunt; nakladatelství SoftPress s.r.o.; ISBN-80-86497-59-3

Ethloop – generátor paketů a měřič datových toků; simulační nástroj použitý k získání dat potřebných pro grafy

`http://luxik.cdi.cz/~devik/qos/ethloop/`

Rejstřík

Příloha

DiffServ (*Differentiated Service*)

Motivací pro hledání alternativ k *integrovaným službám* byly oprávněné obavy ze špatné škálovatelnosti protokolu RSVP, zejména pokud by se měly rezervace prostředků routerů realizovat napříč celým Internetem. Pro velmi zatížené routery přenášející statisíce toků by totiž nároky na paměť (uchování stavových dat) a výpočetní kapacitu (klasifikace paketů) byly skutečně extrémní.

Pracovní skupina IETF pro *diferencované služby (DiffServ)* proto vytvořila podstatně jednodušší model QoS, který je založen na agregaci datových toků do malého počtu tříd *(Class of Service - CoS)* a jim odpovídajících kvalitativních typů služeb. K vyznačení příslušnosti paketu k dané třídě se využívá oktet TOS v hlavičce IPv4 anebo oktet Traffic Class v hlavičce IPv6 (prozatím se využívá pouze šest bitů z těchto oktetů).

Model DiffServ se v současné době ještě vyvíjí. Vychází se z pojmu DS-domény, což je administrativní jednotka analogická autonomnímu systému. Uvnitř DS-domény je zavedena vůči diferencovaným službám jednotná administrativní strategie - vyhodnocování oprávněnosti požadavků, přiřazení toků do tříd, označení paketů a odpovídající diferencované chování aktivních prvků.

V DS-doméně se rozlišují tři typy uzlů (routerů):

- **Okrajový uzel** Leží na rozhraní DS-domény a části sítě, která nepoužívá značkované pakety. Tento uzel má nejobtížnější úlohu, neboť musí provádět klasifikaci vstupních toků a na základě administrativní strategie, stupně vytížení sítě apod. označkovat všechny pakety, které pak dále posílá do své DS-domény.
- **Vnitřní uzel** Jeho úloha je naopak velmi jednoduchá, neboť neprovádí ani žádnou klasifikaci paketů, pouze s paketem určitým definovaným způsobem naloží podle značky v jeho hlavičce.
- **Hraniční uzel** Leží na rozhraní dvou DS-domén, které mohou mít rozdílná klasifikační a jiná pravidla. Způsob, jímž se nakládá s pakety přicházejícími z jiné

DS-domény záleží na dohodě mezi těmito doménami. V běžném případě se provede pouze jednoduchá reklasifikace na základě značky v příchozím paketu a cílové adrese a paketu se přiřadí nová značka.

Velmi zajímavou platformou pro experimenty s DiffServ je také operační systém Linux, který od verze 2.2 obsahuje několik variant dispečinku paketů (například CBQ). K dispozici je také samostatné rozšíření Linuxu o DiffServ:

`http://diffserv.sourceforge.net/`

Výpisy ze zdrojových souborů

qdisc

Výpis ze souboru `linux/net/sched/sch_api.c`

```
Short review.
-------------

This file consists of two interrelated parts:

1. queueing disciplines manager frontend.
2. traffic classes manager frontend.

Generally, queueing discipline ("qdisc") is a black box,
which is able to enqueue packets and to dequeue them (when
device is ready to send something) in order and at times
determined by algorithm hidden in it.

qdisc's are divided to two categories:
- "queues", which have no internal structure visible from outside.
- "schedulers", which split all the packets to "traffic classes",
  using "packet classifiers" (look at cls_api.c)

In turn, classes may have child qdiscs (as rule, queues)
attached to them etc. etc. etc.

The goal of the routines in this file is to translate
information supplied by user in the form of handles
to more intelligible for kernel form, to make some sanity
```

checks and part of work, which is common to all qdiscs
and to provide rtnetlink notifications.

All real intelligent work is done inside qdisc modules.

Every discipline has two major routines: enqueue and dequeue.

---dequeue

dequeue usually returns a skb to send. It is allowed to return NULL,
but it does not mean that queue is empty, it just means that
discipline does not want to send anything this time.
Queue is really empty if q->q.qlen == 0.
For complicated disciplines with multiple queues q->q is not
real packet queue, but however q->q.qlen must be valid.

---enqueue

enqueue returns number of enqueued packets i.e. this number is 1,
if packet was enqueued successfully and <1 if something (not
necessary THIS packet) was dropped.

Auxiliary routines:

---requeue

requeues once dequeued packet. It is used for non-standard or
just buggy devices, which can defer output even if dev->tbusy=0.

---reset

returns qdisc to initial state: purge all buffers, clear all
timers, counters (except for statistics) etc.

---init

initializes newly created qdisc.

```
   ---destroy

   destroys resources allocated by init and during lifetime of qdisc.

   ---change

   changes qdisc parameters.
```

qdisc [p|b]fifo

Výpis ze souboru `linux/net/sched/sch_fifo.c`

```
/* 1 band FIFO pseudo-"scheduler" */
```

qdisc tbf

Výpis ze souboru `linux/net/sched/sch_tbf.c`

```
/*      Simple Token Bucket Filter.
        ======================================

        SOURCE.
        -------

        None.

        Description.
        ------------

        A data flow obeys TBF with rate R and depth B, if for any
        time interval t_i...t_f the number of transmitted bits
        does not exceed B + R*(t_f-t_i).

        Packetized version of this definition:
        The sequence of packets of sizes s_i served at moments t_i
        obeys TBF, if for any i<=k:

        s_i+....+s_k <= B + R*(t_k - t_i)

        Algorithm.
```

Let N(t_i) be B/R initially and N(t) grow continuously with time as:

N(t+delta) = min{B/R, N(t) + delta}

If the first packet in queue has length S, it may be transmited only at the time t_* when S/R <= N(t_*), and in this case N(t) jumps:

N(t_* + 0) = N(t_* - 0) - S/R.

Actually, QoS requires two TBF to be applied to a data stream. One of them controls steady state burst size, another one with rate P (peak rate) and depth M (equal to link MTU) limits bursts at a smaller time scale.

It is easy to see that P>R, and B>M. If P is infinity, this double TBF is equivalent to a single one.

When TBF works in reshaping mode, latency is estimated as:

lat = max ((L-B)/R, (L-M)/P)

NOTES.

If TBF throttles, it starts a watchdog timer, which will wake it up when it is ready to transmit.
Note that the minimal timer resolution is 1/HZ.
If no new packets arrive during this period, or if the device is not awaken by EOI for some previous packet, TBF can stop its activity for 1/HZ.

```
        This means, that with depth B, the maximal rate is

        R_crit = B*HZ

        F.e. for 10Mbit ethernet and HZ=100 the minimal allowed B is
~10Kbytes.

        Note that the peak rate TBF is much more tough: with MTU 1500
        P_crit = 150Kbytes/sec. So, if you need greater peak
        rates, use alpha with HZ=1000 :-)
*/
```

---

qdisc cbq

Výpis ze souboru `linux/net/sched/sch_cbq.c`

```
/*      Class-Based Queueing (CBQ) algorithm.
        =======================================

        Sources: [1] Sally Floyd and Van Jacobson, "Link-sharing and
Resource
                 Management Models for Packet Networks",
                 IEEE/ACM Transactions on Networking, Vol.3, No.4, 1995

                 [2] Sally Floyd, "Notes on CBQ and Guaranted Service",
1995

                 [3] Sally Floyd, "Notes on Class-Based Queueing:
Setting
                 Parameters", 1996

                 [4] Sally Floyd and Michael Speer, "Experimental
Results
                 for Class-Based Queueing", 1998, not published.

        ---------------------------------------------------------------

        Algorithm skeleton was taken from NS simulator cbq.cc.
        If someone wants to check this code against the LBL version,
```

```
        he should take into account that ONLY the skeleton was
borrowed,
        the implementation is different. Particularly:

        --- The WRR algorithm is different. Our version looks more
        reasonable (I hope) and works when quanta are allowed to be
        less than MTU, which is always the case when real time classes
        have small rates. Note, that the statement of [3] is
        incomplete, delay may actually be estimated even if class
        per-round allotment is less than MTU. Namely, if per-round
        allotment is W*r_i, and r_1+...+r_k = r < 1

        delay_i <= ([MTU/(W*r_i)]*W*r + W*r + k*MTU)/B

        In the worst case we have IntServ estimate with D = W*r+k*MTU
        and C = MTU*r. The proof (if correct at all) is trivial.


        --- It seems that cbq-2.0 is not very accurate. At least, I
cannot
        interpret some places, which look like wrong translations
        from NS. Anyone is advised to find these differences
        and explain to me, why I am wrong 8).

        --- Linux has no EOI event, so that we cannot estimate true
class
        idle time. Workaround is to consider the next dequeue event
        as sign that previous packet is finished. This is wrong because
of
        internal device queueing, but on a permanently loaded link it
is true.
        Moreover, combined with clock integrator, this scheme looks
        very close to an ideal solution.  */
```

qdisc red

Výpis ze souboru `linux/net/sched/sch_red.c`

```
/*      Random Early Detection (RED) algorithm.
        =======================================
```

```
        Source: Sally Floyd and Van Jacobson, "Random Early Detection
Gateways
        for Congestion Avoidance", 1993, IEEE/ACM Transactions on
Networking.

        This file codes a "divisionless" version of RED algorithm
        as written down in Fig.17 of the paper.

Short description.
------------------

        When a new packet arrives we calculate the average queue
length:

        avg = (1-W)*avg + W*current_queue_len,

        W is the filter time constant (choosen as 2^(-Wlog)), it
controls
        the inertia of the algorithm. To allow larger bursts, W should
be
        decreased.

        if (avg > th_max) -> packet marked (dropped).
        if (avg < th_min) -> packet passes.
        if (th_min < avg < th_max) we calculate probability:

        Pb = max_P * (avg - th_min)/(th_max-th_min)

        and mark (drop) packet with this probability.
        Pb changes from 0 (at avg==th_min) to max_P (avg==th_max).
        max_P should be small (not 1), usually 0.01..0.02 is good
value.

        max_P is chosen as a number, so that max_P/(th_max-th_min)
        is a negative power of two in order arithmetics to contain
        only shifts.


        Parameters, settable by user:
        -----------------------------
```

```
        limit           - bytes (must be > qth_max + burst)

        Hard limit on queue length, should be chosen >qth_max
        to allow packet bursts. This parameter does not
        affect the algorithms behaviour and can be chosen
        arbitrarily high (well, less than ram size)
        Really, this limit will never be reached
        if RED works correctly.

        qth_min         - bytes (should be < qth_max/2)
        qth_max         - bytes (should be at least 2*qth_min and less
limit)
        Wlog            - bits (<32) log(1/W).
        Plog            - bits (<32)

        Plog is related to max_P by formula:

        max_P = (qth_max-qth_min)/2^Plog;

        F.e. if qth_max=128K and qth_min=32K, then Plog=22
        corresponds to max_P=0.02

        Scell_log
        Stab

        Lookup table for log((1-W)^(t/t_ave).


NOTES:

Upper bound on W.
-----------------

        If you want to allow bursts of L packets of size S,
        you should choose W:

        L + 1 - th_min/S < (1-(1-W)^L)/W

        th_min/S = 32         th_min/S = 4
```

```
        log(W)  L
        -1      33
        -2      35
        -3      39
        -4      46
        -5      57
        -6      75
        -7      101
        -8      135
        -9      190
        etc.
 */
```

qdisc sfq

Výpis ze souboru `linux/net/sched/`

```
/*      Stochastic Fairness Queuing algorithm.
        =======================================

        Source:
        Paul E. McKenney "Stochastic Fairness Queuing",
        IEEE INFOCOMM'90 Proceedings, San Francisco, 1990.

        Paul E. McKenney "Stochastic Fairness Queuing",
        "Interworking: Research and Experience", v.2, 1991, p.113-131.


        See also:
        M. Shreedhar and George Varghese "Efficient Fair
        Queuing using Deficit Round Robin", Proc. SIGCOMM 95.


        This is not the thing that is usually called (W)FQ nowadays.
        It does not use any timestamp mechanism, but instead
        processes queues in round-robin order.

        ADVANTAGE:
```

```
        - It is very cheap. Both CPU and memory requirements are
minimal.

        DRAWBACKS:

        - "Stochastic" -> It is not 100% fair.
        When hash collisions occur, several flows are considered as
one.

        - "Round-robin" -> It introduces larger delays than virtual
clock
        based schemes, and should not be used for isolating interactive
        traffic from non-interactive. It means, that this scheduler
        should be used as leaf of CBQ or P3, which put interactive
traffic
        to higher priority band.

        We still need true WFQ for top level CSZ, but using WFQ
        for the best effort traffic is absolutely pointless:
        SFQ is superior for this purpose.

        IMPLEMENTATION:
        This implementation limits maximal queue length to 128;
        maximal mtu to 2^15-1; number of hash buckets to 1024.
        The only goal of this restrictions was that all data
        fit into one 4K page :-). Struct sfq_sched_data is
        organized in anti-cache manner: all the data for a bucket
        are scattered over different locations. This is not good,
        but it allowed me to put it into 4K.

        It is easy to increase these values, but not in flight.  */
```

---

qdisc prio

Výpis ze souboru `linux/net/sched/sch_prio.c`

```
* net/sched/sch_prio.c  Simple 3-band priority "scheduler".
```

qdisc csz

Výpis ze souboru `linux/net/sched/sch_csz.c`

```
/*      Clark-Shenker-Zhang algorithm.
        =======================================

        SOURCE.

        David D. Clark, Scott Shenker and Lixia Zhang
        "Supporting Real-Time Applications in an Integrated Services
Packet
        Network: Architecture and Mechanism".

        CBQ presents a flexible universal algorithm for packet
scheduling,
        but it has pretty poor delay characteristics.
        Round-robin scheduling and link-sharing goals
        apparently contradict minimization of network delay and jitter.
        Moreover, correct handling of predictive flows seems to be
        impossible in CBQ.

        CSZ presents a more precise but less flexible and less
efficient
        approach. As I understand it, the main idea is to create
        WFQ flows for each guaranteed service and to allocate
        the rest of bandwith to dummy flow-0. Flow-0 comprises
        the predictive services and the best effort traffic;
        it is handled by a priority scheduler with the highest
        priority band allocated       for predictive services, and the
rest ---
        to the best effort packets.

        Note that in CSZ flows are NOT limited to their bandwidth.  It
        is supposed that the flow passed admission control at the edge
        of the QoS network and it doesn't need further shaping. Any
        attempt to improve the flow or to shape it to a token bucket
        at intermediate hops will introduce undesired delays and raise
        jitter.

        At the moment CSZ is the only scheduler that provides
```

true guaranteed service. Another schemes (including CBQ)
do not provide guaranteed delay and randomize jitter.
There is a proof (Sally Floyd), that delay
can be estimated by a IntServ compliant formula.
This result is true formally, but it is wrong in principle.
It takes into account only round-robin delays,
ignoring delays introduced by link sharing i.e. overlimiting.
Note that temporary overlimits are inevitable because
real links are not ideal, and the real algorithm must take this
into account.

ALGORITHM.

--- Notations.

B is link bandwidth (bits/sec).

I is set of all flows, including flow 0.
Every flow $a \in I$ has associated bandwidth slice $r_a < 1$
and
$\sum_{a \in I} r_a = 1$.

--- Flow model.

Let m_a is the number of backlogged bits in flow a.
The flow is {\em active}, if $m_a > 0$.
This number is a discontinuous function of time;
when a packet i arrives:
\[
m_a(t_i+0) - m_a(t_i-0) = L^i,
\]
where L^i is the length of the arrived packet.
The flow queue is drained continuously until $m_a == 0$:
\[
{d m_a \over dt} = - { B r_a \over \sum_{b \in A} r_b}.
\]
I.e. flow rates are their allocated rates proportionally
scaled to take all available link bandwidth. Apparently,
it is not the only possible policy. F.e. CBQ classes
without borrowing would be modelled by:
\[

{d m_a \over dt} = - B r_a .
\]
More complicated hierarchical bandwidth allocation
policies are possible, but unfortunately, the basic
flow equations have a simple solution only for proportional
scaling.

--- Departure times.

We calculate the time until the last bit of packet is sent:
\[
E_a^i(t) = { m_a(t_i) - \delta_a(t) \over r_a },
\]
where $\delta_a(t)$ is number of bits drained since t_i.
We have to evaluate E_a^i for all queued packets,
then find the packet with minimal E_a^i and send it.

This sounds good, but direct implementation of the algorithm
is absolutely infeasible. Luckily, if flow rates
are scaled proportionally, the equations have a simple
solution.

The differential equation for E_a^i is
\[
{d E_a^i (t) \over dt } = - { d \delta_a(t) \over dt} { 1 \over r_a} =
{ B \over \sum_{b \in A} r_b}
\]
with initial condition
\[
E_a^i (t_i) = { m_a(t_i) \over r_a } .
\]

Let's introduce an auxiliary function $R(t)$:

--- Round number.

Consider the following model: we rotate over active flows,
sending $r_a B$ bits from every flow, so that we send
$B \sum_{a \in A} r_a$ bits per round, that takes
$\sum_{a \in A} r_a$ seconds.

Hence, $R(t)$ (round number) is a monotonically increasing
linear functionof time when A is not changed
\[
{ d R(t) \over dt } = { 1 \over \sum_{a \in A} r_a }
\]
and it is continuous when A changes.

The central observation is that the quantity
$F_a^i = R(t) + E_a^i(t)/B$ does not depend on time at all!
$R(t)$ does not depend on flow, so that F_a^i can be
calculated only once on packet arrival, and we need not
recalculate E numbers and resorting queues.
The number F_a^i is called finish number of the packet.
It is just the value of $R(t)$ when the last bit of packet
is sent out.

Maximal finish number on flow is called finish number of flow
and minimal one is "start number of flow".
Apparently, flow is active if and only if $F_a \leq R$.

When a packet of length L_i bit arrives to flow a at time
t_i,
we calculate F_a^i as:

If flow was inactive ($F_a < R$):
$F_a^i = R(t) + {L_i \over B r_a}$
otherwise
$F_a^i = F_a + {L_i \over B r_a}$

These equations complete the algorithm specification.

It looks pretty hairy, but there is a simple
procedure for solving these equations.
See procedure csz_update(), that is a generalization of
the algorithm from S. Keshav's thesis Chapter 3
"Efficient Implementation of Fair Queeing".

NOTES.

* We implement only the simplest variant of CSZ,

```
        when flow-0 is a explicit 4band priority fifo.
        This is bad, but we need a "peek" operation in addition
        to "dequeue" to implement complete CSZ.
        I do not want to do that, unless it is absolutely
        necessary.

        * A primitive support for token bucket filtering
        presents itself too. It directly contradicts CSZ, but
        even though the Internet is on the globe ... :-)
        "the edges of the network" really exist.

        BUGS.

        * Fixed point arithmetic is overcomplicated, suboptimal and
even
        wrong. Check it later.  */


/* This number is arbitrary */
```

---

estimator

Výpis ze souboru `linux/net/sched/estimator.c`

```
This code is NOT intended to be used for statistics collection,
   its purpose is to provide a base for statistical multiplexing
   for controlled load service.
   If you need only statistics, run a user level daemon which
   periodically reads byte counters.

   Unfortunately, rate estimation is not a very easy task.
   F.e. I did not find a simple way to estimate the current peak rate
   and even failed to formulate the problem 8)8)

   So I preferred not to built an estimator into the scheduler,
   but run this task separately.
   Ideally, it should be kernel thread(s), but for now it runs
   from timers, which puts apparent top bounds on the number of rated
   flows, has minimal overhead on small, but is enough
   to handle controlled load service, sets of aggregates.
```

```
   We measure rate over A=(1<<interval) seconds and evaluate EWMA:

   avrate = avrate*(1-W) + rate*W

   where W is chosen as negative power of 2: W = 2^(-ewma_log)

   The resulting time constant is:

   T = A/(-ln(1-W))


   NOTES.

   * The stored value for avbps is scaled by 2^5, so that maximal
     rate is ~1Gbit, avpps is scaled by 2^10.

   * Minimal interval is HZ/4=250msec (it is the greatest common
divisor
     for HZ=100 and HZ=1024 8)), maximal interval
     is (HZ/4)*2^EST_MAX_INTERVAL = 8sec. Shorter intervals
     are too expensive, longer ones can be implemented
```

at user level painlessly.

---

filter fw

Výpis ze souboru `linux/net/sched/cls_fw.c`

net/sched/cls_fw.c Classifier mapping ipchains' fwmark to traffic class.

filter u32

Výpis ze souboru `linux/net/sched/cls_u32.c`

```
        The filters are packed to hash tables of key nodes
        with a set of 32bit key/mask pairs at every node.
        Nodes reference next level hash tables etc.

        This scheme is the best universal classifier I managed to
```

```
   invent; it is not super-fast, but it is not slow (provided you
   program it correctly), and general enough.  And its relative
   speed grows as the number of rules becomes larger.

   It seems that it represents the best middle point between
   speed and manageability both by human and by machine.

   It is especially useful for link sharing combined with QoS;
   pure RSVP doesn't need such a general approach and can use
   much simpler (and faster) schemes, sort of cls_rsvp.c.
```

filter route

Výpis ze souboru `linux/net/sched/cls_route.c`

```
   1. For now we assume that route tags < 256.
      It allows to use direct table lookups, instead of hash tables.
   2. For now we assume that "from TAG" and "fromdev DEV" statements
      are mutually  exclusive.
   3. "to TAG from ANY" has higher priority, than "to ANY from XXX"
```

filter rsvp

Výpis ze souboru `linux/net/sched/cls_rsvp.h`

```
/* Comparing to general packet classification problem,
   RSVP needs only sevaral relatively simple rules:

   * (dst, protocol) are always specified,
     so that we are able to hash them.
   * src may be exact, or may be wildcard, so that
     we can keep a hash table plus one wildcard entry.
   * source port (or flow label) is important only if src is given.

   IMPLEMENTATION.

   We use a two level hash table: The top level is keyed by
   destination address and protocol ID, every bucket contains a list
   of "rsvp sessions", identified by destination address, protocol and
   DPI(="Destination Port ID"): triple (key, mask, offset).
```

```
   Every bucket has a smaller hash table keyed by source address
   (cf. RSVP flowspec) and one wildcard entry for wildcard
reservations.
   Every bucket is again a list of "RSVP flows", selected by
   source address and SPI(="Source Port ID" here rather than
   "security parameter index"): triple (key, mask, offset).

   NOTE 1. All the packets with IPv6 extension headers (but AH and
ESP)
   and all fragmented packets go to the best-effort traffic class.

   NOTE 2. Two "port id"'s seems to be redundant, rfc2207 requires
   only one "Generalized Port Identifier". So that for classic
   ah, esp (and udp,tcp) both *pi should coincide or one of them
   should be wildcard.

   At first sight, this redundancy is just a waste of CPU
   resources. But DPI and SPI add the possibility to assign different
   priorities to GPIs. Look also at note 4 about tunnels below.

   NOTE 3. One complication is the case of tunneled packets.
   We implement it as following: if the first lookup
   matches a special session with "tunnelhdr" value not zero,
   flowid doesn't contain the true flow ID, but the tunnel ID
(1...255).
   In this case, we pull tunnelhdr bytes and restart lookup
   with tunnel ID added to the list of keys. Simple and stupid 8)8)
   It's enough for PIMREG and IPIP.

   NOTE 4. Two GPIs make it possible to parse even GRE packets.
   F.e. DPI can select ETH_P_IP (and necessary flags to make
   tunnelhdr correct) in GRE protocol field and SPI matches
   GRE key. Is it not nice? 8)8)

   Well, as result, despite its simplicity, we get a pretty
```

powerful classification engine. */
