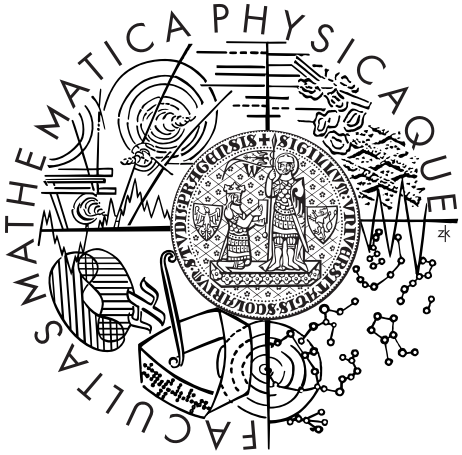
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Martin Pelikán

Optimalizace využití kapacity sítí na unixových systémech

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce:	RNDr. Leo Galamboš, Ph.D.
Studijní program:	informatika
Studijní obor:	správa počítačových systémů

Praha 2014

Vstřícným pedagogům, přátelům a především trpělivé rodině upřímně děkuji za podporu nejen během tvorby této práce, ale také v letech předešlých. Vedoucí práce RNDr. Leo Galamboš, Ph.D., dokázal podstatnými připomínkami významně zlepšit úroveň textu, za což rovněž děkuji. Dík patří též Mgr. Danielu Machovi za cestu do světa počítačových sítí a k pečlivosti při jejich studiu.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Optimalizace využití kapacity sítí na unixových systémech

Autor: Martin Pelikán

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Leo Galamboš, Ph. D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Moderní unixové systémy obsahují velmi výkonné síťové subsystémy s možnostmi konfigurace, jejichž pravý význam uživatelům nebývá zřejmý. Srovnání používaných algoritmů, dokumentace a vizualizace některých částí pomůže k jejich pochopení a racionalizaci výběru konkrétního nastavení. Vysvětlení síťového stacku od ovladačů po síťovou vrstvu ukáže problémy s měřením času, zpracováním provozu v burstech či velikostí front s ohledem na možnosti klasifikace, plánování či regulace provozu v Linuxu a OpenBSD. Text práce slouží jako přehled implementovaných algoritmů a doplněná dokumentace Linuxových akcí a filtrů, implementační část přináší nový portabilní nástroj k vizualizaci existující konfigurace vzdálených strojů bez nutnosti je upravovat.

Klíčová slova: síťový stack, traffic shaping, klasifikace provozu, fronty

Title: Optimizing the utilization of networks' capacity on unix-like systems

Author: Martin Pelikán

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Leo Galamboš, Ph. D., Department of Distributed and Dependable Systems

Abstract: Modern unix-like systems contain very powerful network stacks with configuration options often beyond operators' understanding. A comparison of available algorithms, documentation and visualization of certain components help their understanding, leading to better configuration choices. By explaining the network stack from the drivers up to the network layer will exhibit problems with timekeeping, burst traffic processing or queue management with regard to classification, scheduling or traffic regulation options in Linux or OpenBSD. The thesis works as an overview of implemented algorithms and updated documentation of Linux's actions and filters, while the implementation introduces a new portable tool to visualize existing configuration on remote machines without the need of modifying them.

Keywords: network internals, traffic shaping, traffic classification, queue management

Obsah

Úvod

Dosáhnout na dnes běžně používaných počítačových sítích přenosové rychlosti v řádu gigabitů za sekundu není ani pro relativně levný hardware problém. Nejen pro studium, ale i v reálném nasazení se v roli síťových směrovačů a serverů osvědčily open source operační systémy unixového typu (OS). Důvody jsou zřejmé; cena, dostupnost a rozsáhlá komunita. Nefunguje-li něco podle našich představ, máme možnost to změnit a případně zveřejnit výsledky své práce ke zhodnocení a kritice.

Z pohledu OS hraje zásadní roli efektivita předávání dat mezi pamětí typicky namapovanou do nějakého uživatelského procesu a příslušným síťovým adaptérem včetně jejich průběžného i následného zpracování. Při odesílání dat systém často rozhoduje nejen o tom, jaká data pošle a kam, ale též kolik, kdy a v jakém pořadí [Armit]. Správce takového systému má možnost rozdělovat prostředky mezi více uživatelů podle jejich potřeb či zásluh, přičemž způsob rozdělování může hrát významnou roli ve výsledné kvalitě poskytovaných služeb. Uživatel z nich potom chce reálně využít maximum a často při jeho dosahování chybuje.

Všechny dnešní sítě se skládají z několika propojených vrstev, které jsou z pohledu návrhu oddělené. Řešit problémy uživatelů jen na podmnožině použitých vrstev může zásadně poškodit chování jejího doplňku. Přitom téměř každá komponenta moderního síťového stacku má nastavitelné parametry,[1] jejichž jednotlivé změny mohou krátkodobě pomoci, ale chování celého systému zásadně zhoršit. U spousty parametrů je naprosto zásadní si nejdříve uvědomit, proč autoři daného systému vybrali dané výchozí hodnoty, co přesně nás vede k jejich změně a jak to ovlivní ostatní části systému. Existuje spousta publikací zevrubně popisujících, jak jsou jednotlivé části navrženy [TanWet] [Armit] či konkrétní implementace [LiNetInt] [SteWri], avšak nenalezl jsem shrnující publikaci s popisem jejich interakce a rozhodování mezi více přístupy k daným problémům.

Cílem práce je čtenáři přednést stručný popis jednotlivých částí, objasnit některá rozhodnutí autorů jednotlivých implementací, ukázat problémy vzniklé nesprávným užíváním a v implementační části představit srozumitelný pohled na nastavení libovolného systému, která se dosud prováděla jen velmi obtížně. Počítačové sítě jsou rozšířené zejména z důvodu své všestrannosti a, bohužel pro nás, různé strany mají různé představy nejen o optimálním využití, ale také o hranici použitelnosti a úrovni uživatelského komfortu.

Výhoda open source prostředí – možnost výběru – je zároveň jeho velkým omezením, protože různé systémy implementují různé mechanismy a složitost jejich konfigurace a implementace se zásadně liší. Cílem práce je srovnat existující implementace z Linuxu [LiSrc] a OpenBSD [ObSrc], upozornit na jejich nedostatky a vytvořit jednotný pohled na konfiguraci.

Známe-li relevantní vlastnosti fyzické vrstvy[2], můžeme je využít při rozhodování o odchozím síťovém provozu[3]. Nemáme-li v nějakém uzlu nebo cestě dostateč-

[1] Ať už uživatelsky přívětivě, například pomocí `sysctl(8)`, nebo jako části zdrojového kódu.

[2] Například poloviční resp. plný duplex (HDX resp. FDX), přenosové zpoždění a jeho variabilita, možnost výskytu, případně pravděpodobnost kolize.

[3] Pro odchozí resp. příchozí síťový provoz se často používají zkratky Tx resp. Rx pocházející z anglických slov *transmit* resp. *receive*, alternativně též *egress* resp. *ingress*.

nou kapacitu ke zpracování přijímaných dat, měli bychom být schopni rozlišovat důležitost jednotlivých úloh a informovat své okolí, aby zátěž zmírnilo na únosnou mez. Stejně tak na vyšších vrstvách máme možnost rozhodovat, tentokrát už i na základě uživatelských požadavků často abstrahovaných od technických možností, jakým způsobem bude vhodné se na dané síti chovat.

V oboru se ujal pojem dodržování kvality služeb, *Quality of Service* (QoS), znamenající nový význam pro slovní spojení *best effort*. Kromě rozhodnutí *kam* paket poslat je třeba rozhodnout *kdy* ho poslat [Armit], čímž se *best effort* může omezit jen na část síťového provozu, pro kterou nebyly explicitně žádné zdroje vyhrazeny [RFC2474]. Ačkoli mechanismy na zajištění bezztrátovosti paketové sítě existují[4], takový přístup vyžaduje obrovskou zátěž na všechny aktivní prvky použité infrastruktury a mnohonásobně tak zvyšuje náklady. Proto je velmi vhodné smířit se s tím, že na paketové síti nikdy nelze službu stoprocentně garantovat, a pod pojmem QoS se pak místo garance skrývá otázka „co pro zajištění požadované kvality udělal každý uzel po dané cestě“.

Zajišťování QoS lze považovat za formu maximalizace využití zdrojů dostupných pro nějakou část (třídu) provozu; provoz několika různých tříd paralelně na jedné infrastruktuře je další oblastí, kde unixové OS disponují spoustou více či méně dokumentovaných mechanismů, které na nepředvídatelně se chovající provoz mohou mít různý vliv. První kapitola pojednává o způsobu komunikace mezi OS a hardware, včetně problémů způsobených neuváženým využíváním prostředků. Druhá a třetí kapitola řeší problematiku linkové, resp. síťové vrstvy referenčního modelu ISO/OSI – využití dané linky, resp. informování routerů v rámci sítě je pro dosažení daných cílů mnohdy nezbytné. Práce z časových důvodů neřeší různé druhy reakce protokolů na vyšších vrstvách, ať již nastavování velikosti okna či optimalizace na provoz v reálném čase, neboť tam používaných algoritmů existují desítky a jejich srovnání by zcela jistě vydalo na samostatné dílo. Čtvrtá kapitola tedy popisuje existující možnosti klasifikace provozu a je většinou tvořena původním textem založeným pouze na stovkách kilobajtů zdrojových kódů s minimem komentářů jak záměru, tak funkcionality.

Ujasněme několik používaných konceptů a pojmů, které bývají přednášeny v základních kurzech o počítačových sítích nebo administraci unixových systémů a pro tuto práci jsou zásadní.

Kdy vlastně využiji QoS?

Stejně jako ve většině případů ladění nějaké konfigurace se ani zde neprojeví provedené změny za všech okolností. V sítích s velmi malým provozem jsou rozdíly v QoS téměř nepatrné a většina zde popsaných algoritmů začne mít efekt až při větším množství dat zdržených (*backlogged*) v daném stroji, případně jen u specifického druhu provozu. Ve většině sítí k nutnosti zasahovat do konfigurace QoS dochází jen velmi zřídka, na druhou stranu je nutné být na tyto situace připraven, znát dostupné možnosti a mít je otestované. V každém systému obsluhujícím

[4]Například *Data Center Bridging* (DCB) používající *Priority Flow Control* (PFC) popsaný standardy IEEE 802.1Qaz a 802.1Qbb [IEEE802.1], nebo protokol Infiniband vyznačující se především velmi nízkým přenosovým zpožděním či podporou pro RDMA, přímý přístup do paměti jiných strojů v síti [Infini].

aspoň desítky megabitů dat za sekundu *nějaké* fronty vždy jsou[5], pouze jsou při velmi malém vytížení nepatrné (např. délky 1) a algoritmy s nimi pracující nedostanou příležitost své rozdíly předvést. Stejně tak je snaha maximalizovat rychlost sítě zbytečná, nemáme-li na obou stranách pro takto rychlý tok dat podmínky, ať už omezené vlastnostmi hardware (rychlost disku, sběrnice či CPU) nebo software (uživatel se většinou nedívá na deset filmů zároveň).

Důležité je chápat QoS ve správném kontextu – přestože se na vyšších úrovních můžeme bavit o nastavení celé sítě, ve skutečnosti hrají roli pouze místa s úzkými hrdly. Grenville Armitage [Armit] doslova píše:

> Because one person's network is another person's link, the notion of end-to-end QoS must be generalized into one of edge-to-edge QoS.

Speciálně tedy platí, že vynechání požadovaného nastavení v jediném uzlu (resp. jeho chybná konfigurace) může poškodit chování všech cest přes něj procházejících. Dále je třeba vzít v potaz výpočetní náročnost vybrané konfigurace. Budeme-li několik chybějících procent výkonu směrovače řešit přidáním kódu vykonávaného během vlastního předávání paketů (tzv. *forwarding path* nebo též *hot path*), velmi pravděpodobně tím situaci pouze zhoršíme. Spousta uvedených mechanismů například „vymění“ dobu odezvy za propustnost, což je změna subjektivně velmi špatně rozpoznatelná, ale na kvalitu některých služeb má vysoce negativní vliv.

Architektura Classify, Queue, Schedule

Široce přijímaným a velmi obecným přístupem je tzv. *Classify, Queue, Schedule* (CQS), který odděleně definuje části systému starající se o klasifikaci provozu, resp. jeho rozdělování do nezávislých front, resp. plánování odesílání paketů z těchto front v pořadí odpovídajícímu požadavkům uživatele. Klasifikace obnáší identifikaci klíčových vlastností daného provozu (například zdroje, cíle, chování nebo objemu). Na základě sady takovýchto vlastností potom každý paket klasifikujeme, tj. zařadíme do nějaké třídy, v rámci které bude jedna nebo více front zajišťovat její co možná nejmenší vliv na třídy ostatní. Část systému zajišťující analýzu a formální rozdělení se nazývá *classifier* a jeho podobu ve zkoumaných systémech popisuje kapitola 4. Identifikátor vybrané třídy lze též zakódovat do některého z použitých protokolů, čímž můžeme usnadnit proces klasifikace zařízením dále po cestě. Na linkové vrstvě tuto klasifikaci popisuje kapitola 2.7, na síťové vrstvě potom kapitola 3.1. Plánovač (častěji *scheduler*) potom v nějakém pořadí z těchto front vybírá, který provoz vyšle na cestu nejdříve. Fronty slouží nejen k oddělení tříd od sebe, ale především k diferenciaci způsobů odesílání vyhovujícího provozu, případně zahazování[6] nadbytečného, nebo dokonce pravidla přímo porušujícího. [Armit] Plánovače typicky fungují těsně před vlastním odesláním paketu ze systému, aby se eliminoval vliv doby zpracování na kterékoli vrstvě, zejména šifrování. Proto jsou rozebrány v kapitole 2.3.

[5] O čemž jsem se přesvědčil při implementaci ovladače síťového adaptéru v rámci zápočtového projektu do předmětu Operační systémy.

[6] Nebo informování ostatních o nějakém zahazování; konkrétní akce je technický detail.

Linux a příslušné uživatelské nástroje [iproute2] tuto architekturu kopírují se vší její složitostí; vše se ovládá převážně programem *traffic control* (tc), který tedy zasahuje do několika kapitol. Čtenář tím doufejme lépe porozumí, jak se systém používá a jak je vhodné postupovat při čtení cizí konfigurace. Systémy rodiny BSD volí jednodušší a pragmatický přístup. V OpenBSD jako *classifier* funguje *Packet Filter* (pf) a provoz řadí do front dříve připravovaných subsystémem *Alternate Queueing* (ALTQ), který byl ve verzi 5.5 nahrazen osmi frontami s různou prioritou a možností zapnout algoritmus HFSC [AltqPlan]. Kapitola 2.3 rozebírá i tento subsystém. Ostatní BSD systémy ALTQ stále obsahují.

Architektura ALTQ, jak byla původně popsaná [Altq98] [Altq04], má fronty a plánovač umístěné v jedné komponentě – *queueing discipline*. V Linuxu může mít tento pojem (psaný *qdisc*, zde z pravopisných důvodů též *disciplína*) hlubší[7] význam. Složitější disciplíny umějí brát v potaz, že v rámci nich může být provoz dále členěn do dalších tříd – říkáme jim *classful*. Jednotlivým třídám (class) potom mohou být přiřazovány další, vnitřní, disciplíny. Ty jednodušší jsou potom označované jako *classless*; kromě předem nastavených parametrů je chování celé disciplíny jednotné.

Spravedlivé fronty

Už v roce 1985 zkoumal John Nagle problematiku ucpání paketové sítě a zřejmě jako první definoval koncept tzv. *spravedlivé fronty*, anglicky *fair queueing* [RFC970]. Namísto obsluhy dat ze sítě *First In, First Out* (FIFO) zavedl koncept spravedlivosti (*fairness*) jako schopnost všech zdrojů používat stejné množství kapacity sdílené linky realizovanou pomocí round-robin. Tehdy se zřejmě na všechny dostane, nebudou-li zdroje podvrhovat svoji skutečnou adresu. Článek však nijak nebere v úvahu velikost paketů. Proto v roce 1989 Alan Demers a další [DeKeSh89] analyzovali vlastní variantu fair queueing založenou na hypotetickém přidělování linky po bitech místo po paketech. Tato ze zřejmých důvodů v praxi nerealizovatelná idea je aproximována celou řadou algoritmů popsaných v kapitole 2.3, včetně alternativních rychlejších, leč ne vždy zcela přesných, metod.

Typografické konvence a použitý software

Manuálové stránky jsou psány jako `stránka(k)` kde k je číslo kapitoly a jsou relevantní k právě rozebíranému systému. Systémy s jádrem Linux často používají uživatelské programy, knihovny a manuálové stránky projektu GNU[8]. Systémy rodiny BSD mají své vlastní manuálové stránky rovněž prohledávatelné online[9]. Některé projekty (iproute2) mají své manuálové stránky jako součást instalačního balíku.

Zdrojové kódy se rovněž dají interaktivně procházet na více místech. Linux je dostupný na adrese `http://www.kernel.org/`, OpenBSD má jádro v podadresáří `src/sys` na `http://www.openbsd.org/cgi-bin/cvsweb/` K rych-

[7] A to doslova, neboť tímto prohlubujeme strom s uzly tvořenými samotnými disciplínami.

[8] `http://www.gnu.org/manual/`

[9] `http://www.openbsd.org/cgi-bin/man.cgi` `http://www.freebsd.org/cgi/man.cgi`

lému prohledávání zdrojových kódů některých systémů BSD poslouží též aplikace OpenGrok na adrese `http://www.bxr.su/`. Není-li uvedeno jinak, práce používá Linux i balík iproute2 verze 3.14.0[10] a OpenBSD 5.5.

Seznam použité literatury je z důvodu své velikosti rozdělen na kategorie a uspořádán abecedně dle použitých značek v textu. Čtenář tak rychleji najde požadovaný dokument, neboť může ignorovat sekce, které právě nečte. Technické detaily (měření času, fungování síťového hardware, obecně známé normy či odkazy na kód a dokumentaci) jsou tak odděleny od vědeckých publikací a dokumentů RFC.

[10] `http://git.kernel.org/cgit/linux/kernel/git/shemminger/iproute2.git`

Zběžné srovnání existujících implementací

	Linux	*OpenBSD*
cíle	množství funkcionality	jednoduchost, korektnost
výchozí stav	tři prioritní fronty na základě IP *Type of Service* (ToS)	osm prioritních front na základě *Type of Service* (ToS), upřednostňuje prázdná TCP potvrzení a protokoly CARP, STP a LACP
architektura	vysoce modulární	dvě disciplíny pro všechny uživatele
kontrola příchozího provozu	po aplikaci správných metod prakticky libovolná	pouze firewall
podpora více CPU	zámky na hranici disciplíny	pouze v uživatelském prostoru, *Big Kernel Lock*[a]
klasifikace	systém filtrů a akcí potenciálně využívající vstupy z jiných částí jádra	jednotné nastavení jako součást firewallu
ukládání konfigurace	v závislosti na systému	jednotné
oprávnění ke čtení konfigurace	lokální uživatel, bez varování jsou některé části (Netfilter) odepřeny	superuživatel
dokumentace	částečně manuálové stránky, částečně [LARTC], většinou neexistující	`pf.conf(5)`

[a]Některé ovladače a komponenty jsou již zamykané odděleně a jejich počet se zvyšuje; cílem projektu však není poskytovat výkon za každou cenu. Proces redukce takto zamykaných datových struktur neprobíhá pouhým přidáváním synchronizačních primitiv jako u ostatních systémů, ale změnami architektury vedoucích k čitelnějšímu a bezpečnějšímu paralelnímu kódu. Osobní konverzace s několika vývojáři naznačila, že komerční řešení postavené na OpenBSD mívají přepracované zámky na architektuře amd64, a jejich zkušenosti vedou k pečlivěji prováděným změnám do projektu samotného.

1. Interakce mezi hardware a OS

Tato kapitola stručně popíše možnosti, kterými nám s tokem paketů umožňují manipulovat výrobci hardwaru. Protože implementační detaily jednotlivých čipů nebo FPGA nejsou součástí studia a přesná implementace produkčních strojů nebývá veřejně známá, zaměří se tato kapitola na možnosti nastavení z pohledu administrátora a bude předpokládat bezchybnou implementaci s nulovým negativním dopadem na chování systému kromě těch vlastností, které jsme nastavili.

Tyto možnosti je vhodné umět využít také proto, že zpracování dat v hardware umožní podstatně snížit zátěž OS a ušetřený procesorový čas využít k něčemu užitečnějšímu. *Offloading*, jak se přesouvání některých výpočetně náročnějších operací do hardware nazývá, totiž kromě zátěže přesouvá také prostor pro potenciální bezpečnostní rizika [TrustNIC] a v případě vyšších síťových vrstev i některé možnosti ovlivňovat chování daného datového toku.

Několik takových možností nabízejí i spravovatelné switche už v nejnižších cenových kategoriích. Právě ty jsou určené do tzv. *access layer*, tedy části infrastruktury s koncovými zařízeními a jednotlivými uživateli. Protože zde pracujeme s daleko menšími objemy dat, můžeme si dovolit klasifikovat provoz na základě více kritérií a následně data označit tak, aby se ostatním strojům snáze odbavovala. Nutno zdůraznit, že na této úrovni je třeba nezanedbatelný důraz na bezpečnost, například nastavíme-li portům příliš vysokou prioritu a necháme je dostupné nedůvěryhodným nebo neautentizovaným uživatelům.

K uvedení čtenáře do kontextu nejdříve zopakujme několik principů operačních systémů. Díky obrovskému rozšíření počítačových sítí došlo k nárůstu požadavků na výkon OS, což objevilo několik problémů s naivním přístupem, jak jej popíšeme. Různé systémy je řeší různými způsoby, což nastíníme ve snaze zdůraznit rozdíly mezi nimi.

1.1 Jak funguje operační systém?

Unixové operační systémy umožňovaly běh několika procesů zároveň již od svého návrhu, takže je bezpečné předpokládat, že máme někde pro každý procesor uloženou informaci, který proces na něm naposledy běžel. Nejjednodušší pohled nám dá tři důvody, proč v daný okamžik na daném procesoru běží kód jádra:

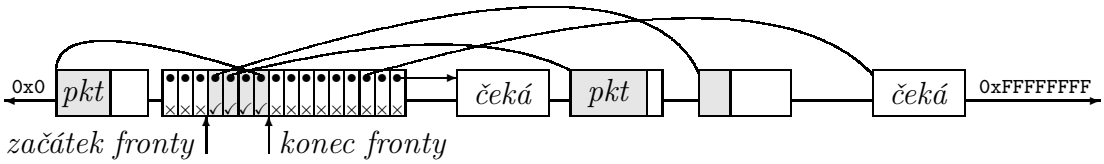
Uživatelský proces spustil systémové volání a říkáme, že jádro běží v *kontextu procesu*. Přenášená data překračují hranice adresových prostorů.

Byl naplánován nějaký proces patřící jádru, přičemž tyto procesy někdy nemají stejné vlastnosti jako uživatelské procesy; potom říkáme, že procesor běží v *kontextu procesu jádra*. Často jde o periodické události.

Daný procesor právě obsluhuje přerušení a potom říkáme, že procesor běží v *kontextu přerušení*. Jaký proces běžel naposledy je irelevantní.

Na různých systémech se tento pohled může v detailech lišit; místo procesů jsou vlákna, případně má některá část více podčástí.

V kontextu přerušení máme na daném procesoru přerušení *zamaskovaná*, aby nedocházelo k problémům se souběhem v situacích, kdy nějaké zařízení posílá



Obrázek 1.1: Ring buffer se šestnácti deskriptory.

přerušení velmi často. Tím dostaneme několik omezení; *nesmíme usnout*, protože nás nikdo nevzbudí[1] a *musíme rychle doběhnout*, jinak může systém pomalu reagovat při interaktivních úlohách jako přehrávání hudby nebo obsluha myši.

1.2 Jak funguje síťový adaptér?

Síťové adaptéry mají různě velká pole tzv. deskriptorů obsahujících ukazatele do fyzické paměti namapované do virtuálního adresového prostoru jádra. Na obrázku 1.1 je vodorovně znázorněný adresový prostor a kruhový buffer se šestnácti deskriptory obsahující čtyři přijaté pakety, resp. připravené k odeslání. Pro názornost zde nejsou všechny oblasti paměti (dále jen *buffery*), kam deskriptory ukazují; nějaké obsahují data a zbylé čekají na další použití. Struktura ring bufferu včetně deskriptorů je závislá čistě na hardware, který z nich čte ony adresy bufferů pro data; najdeme ji v dokumentaci k použitému čipu. [i217][i82599][BCM57711]

Odesílání dat je jednoduché: nastavíme deskriptory, aby ukazovaly na buffery s odesílanými daty, označíme je jako připravené (na obrázku ✓) a na adaptéru spustíme odesílání. Při některém z příštích přerušení nám adaptér řekne, které buffery již odeslal a ty můžeme znovu použít. Posílání více deskriptorů najednou je při rychlostech v řádu stovek megabitů za sekundu nutností, ale je třeba brát v potaz zpoždění případně zdržených dat; vizte dále *Byte Queue Limits* (BQL).

Po uložení přijatých dat do bufferů adaptér v příslušném deskriptoru ring bufferu pro přijatá data (nebo jinak, na obrázku ✓) signalizuje, že je může jádro zpracovat. Paměť spojená s takto neoznačenými deskriptory je prakticky plně v režii síťového adaptéru, který naopak nesmí používat deskriptory, které jsme mu nevrátili. I při příjmu mnoha paketů je nutné jedním přerušením přijmout více deskriptorů najednou a přitom neublížit zpoždění; vizte dále sekci 1.2.1.

Všechny datové buffery jsou ve skutečnosti celou dobu spravovány operačním systémem; tato paměť může být alokována speciálním způsobem[2] a pomocí API na manipulaci s mapováním paměti pro DMA je kód napsán tak, aby fungoval na všech architekturách dané zařízení podporujících. Detaily v tabulce 1.1.

Chce-li jádro deskriptor uvolnit pro další použití a zároveň neztratit přijatá data, může je buďto vykopírovat jinam (tzv. *bounce buffering*), nebo v deskriptoru vyměnit stávající buffer za jiný. Neudělá-li to včas, mohou teoreticky nastat dvě situace. Z pohledu výrobce hardware je nejvhodnější při obsazeném celém ring bufferu jednoduše přestat další data přijímat. O nová data tedy přijdeme, dokud si ta stávající nevyzvedneme; tato technika se označuje pojmem *tail-drop*. Teoreticky je možné v nevyřízených bufferech přepsat stará data novými; protože například

[1] Spaní typicky znamená odplánování procesu do fronty čekajících s ukazatelem na objekt, na který čeká; takto v kontextu přerušení ani usnout *nelze*.

[2] Neumí-li zařízení přístup do adresového prostoru CPU v plné šíři, musí mu být paměť naalokována s dostatečně nízkou adresou (typicky spodní 4 GB fyzické RAM).

	Linux	OpenBSD
datová struktura nesoucí pakety	`struct sk_buff`	`struct mbuf`
její dokumentace	`include/linux/skbuff.h`	`mbuf(9)` [SteWri]
alokace bufferů	`dma_alloc_*(GFP_KERNEL)`	`pool(9)` mcl2k: `MCLGETI`
práce s DMA	`Documentation/DMA-API*`	`bus_dma(9)`
při RX livelocku	vypínání RX přerušení	zmenšování RX ringu

Tabulka 1.1: Rozdíly v ovladačích síťových adadptérů.

protokolu TCP slouží nedoručený paket jako signál ucpání vedoucí k záměrnému zpomalení, bylo by z hlediska zahlcení sítě vhodnější „nedoručit“ ten dřívější. Protokol by potom dříve zareagoval a v ring bufferech by zbytečně nestála čekající data, protože by čekala už u odesilatele. Tyto techniky budeme analyzovat a používat u front na vyšších vrstvách; zde se jedná čistě o vlastnost hardware.

Přijme-li tedy síťová karta paket, pošle procesoru přerušení. Ten, může-li, pozastaví svoji činnost a začne jej vyřizovat – spustí patřičný *interrupt handler*[3]. Jak přesně kernel zjistí, kterou funkci zavolat, je závislé na použité architektuře a pro tuto práci nepodstatné; důležité je vědět, že od začátku obsluhy přerušení jsme v kontextu přerušení, takže bychom měli co nejdříve doběhnout. Naivně však ze síťové karty přečteme paket a zavoláme na něj funkci přijímající patřičný protokol na nejnižší vrstvě, která spustí funkce pro další vrstvy až do úplného doručení paketu do cíle.

1.2.1 Receive Livelock

Po dosažení určité zátěže předchozí systém nebude použitelně fungovat. Obsluha přerušení je rychlá, takže zvládne příjem velkého množství paketů a všechny doručí příslušným příjemcům. I za předpokladu, že nám při přijímání nedojde paměť, procesor neustále dostává nová data a naplánovaní konzumenti nestihnou data zpracovat, protože CPU neustále obsluhuje příjem nových. V systému se tak zvětšuje fronta starých dat, které není čas využít. Eventuelně vyprší jejich platnost [RFC970] a systém je zahodí. Veškerý čas strávený přijetím takových dat byl tedy zbytečně ztracen, protože se data mohla „zahodit sama“ už uvnitř síťového adaptéru. Co hůř, systém nemusí ani reagovat na příkazy administrátora.

Jeffrey Mogul a Kadangode Ramakrishnan [ReLi96][ReLi97] tento problém zaznamenali už v roce 1996. Síťové adaptéry od té doby mají řešení [IntModer] – zabudované časovače. Při přijetí paketu nedojde k přerušení hned, ale až po uplynutí nastavené doby, během které jich může přijít několik. V interrupt handleru se potom přijmou všechny pakety najednou, čímž se ušetří většina režie. Implementace takových časovačů je opět závislá na výrobci adaptéru a přerušení chodí buďto po nastaveném čase, nebo při zaplnění přijímacího ring bufferu. Nevýhoda je zřejmá: systém po nastavenou dobu neví, že nějaký paket přišel, tedy jej nemůže zpracovat a tím narůstá zpoždění. Proto je možné tyto časovače přizpůsobovat danému provozu; vizte například v Linuxu funkci `e1000_update_itr`. Některé adaptéry [i82599] tedy šly ještě dále – posílají tzv. *Low Latency Interrupt* (LLI), zmenší-li se počet volných deskriptorů pod nějakou úroveň.

[3]Můžeme se setkat s označením *Interrupt Service Routine* (ISR).

Aby nedošlo ke zmatkům v názvosloví, je vhodné zmínit, že časovače na síťových kartách se používají i pro detekci interních chyb (tzv. *watchdogs*); během provozu se pokaždé znovu natáhnou a po jejich vypršení se karta restartuje.

Toto řešení livelocků je však závislé na hardware; operační systémy proto nasazují další, univerzální mechanismus. V zájmu zachování interaktivity systému necháme část zpracování dat do přeplánovatelného kontextu procesu jádra.

V kontextu přerušení proběhne část příjmu dat a naplánování jiného procesu, který dodělá zbytek. Při odchodu se opět přerušení na daném procesoru povolí. V Linuxu se této části říká *top half*, přestože to polovina interrupt handleru rozhodně není[4].

V kontextu procesu jádra proběhne zpracování až po nejvyšší vrstvu, od kontrol všech datových struktur, firewallu, QoS po doručení na požadované místo. V Linuxu se této části říká *Bottom Half* (BH).

Linuxová verze se často označuje jako *New Application Programming Interface* (NAPI) a v interrupt handleru zablokuje na adaptéru přerušení pro příchozí pakety. Pomocí `__napi_schedule` a dalších funkcí naplánuje vlákno volající `netif_receive_skb` v cyklu pro všechny přijaté pakety. Na konci tohoto cyklu karta znovu přerušení zapne.

V OpenBSD se pomocí `schednetisr(proto)` naplánuje zpracování příslušného protokolu (IPv4, IPv6, MPLS) a pomocí `IF_INPUT_ENQUEUE` paket zařadí do příslušné fronty na vstupu. Zároveň je stále naplánovaný časovač `m_cltick_tmo` počítající jen uplynulé tiky. Úplně stejné tiky si počítá celý systém časovačů (globální proměnná `ticks` inkrementovaná v `timeout_hardclock_update`) napojený přímo na přerušení časovače. Pokud ovladač síťového adaptéru alokuje pomocí `MCLGETI`[5] buffer pro nová data, přičemž funkce `m_cldrop` zjistí, že plánovaný časovač nestihl tiknout a ten hardwarový mu utekl, jádro evidentně nestíhá obsluhovat, co si naplánovalo. Nastal tedy livelock. Na všech síťových adaptérech se multiplikativně zmenší počet dostupných bufferů, dokud se systém neuklidní. Statistiky tohoto procesu lze sledovat programem `systat(1)` na obrazovce `mbufs`: `LWM` (*low watermark*) je minimální počet dostupných bufferů pro přijímací ring buffer, `CWM` (*current watermark*) je aktuální stav, `HWM` (*high watermark*) je maximum, které se do daného adaptéru vejde a `LIVELOCKS` je informativní počítadlo, kolikrát musel tento systém zasáhnout. Je-li při alokaci dostatek prostředků a `CWM` je vyčerpaný, přidáním konstanty jej zvětšíme (srovnejte s ovládáním velikosti okna TCP).

Výrobci síťových adaptérů samozřejmě vydávají doporučení [SmPaOpt], jak je pro takové prostředí nejlepší daný ovladač nastavit.

1.2.2 Byte Queue Limits

Podobně jako OpenBSD omezuje velikost front na příjem paketů, začal Linux v roce 2011 omezovat odchozí fronty. Sada patchů zvaná BQL implementuje

[4] A už vůbec ne *horní*, chceme-li jako *vyšší* vrstvy síťového stacku označovat ty blíže k aplikacím a *nižší* ty blíže k hardware. Literatura psaná před rozšířením jádra Linux někdy používá tyto pojmy správně.

[5] Název znamená „mbuf cluster get on interface“; starší `MCLGET` počítá buffery pro celý systém dohromady a neumí tak řešit livelocky (stejně jako staré ovladače v Linuxu nepodporují NAPI).

API, kterým mohou ovladače umožnit systému regulovat velikost odchozích front v závislosti na předchozím stavu. Při odeslání zaznamenají, kolik dat odešlo a po dokončení spustí tzv. *completion event*, který na základě předchozího stavu fronty rozhodne o jejím limitu. Fronta může limit přečerpat pouze krátkodobě (příliš dlouhý paket) a ten se dynamicky mění (pomocí knihovny [LiSrc, `lib/dynamic_queue_limits.c`]); došlo-li během odesílání k „vyhladovění“, čili fronta limit porušila a nyní je prázdná, limit zvýšíme o tolik bajtů, kolik jí scházelo. Limit se snižuje právě tehdy, jsou-li ve frontě data a to na základě uplynulé doby mezi posledními *completion eventy*, z čehož plyne, že by měly být periodické. To může znamenat problémy při zavádění BQL do ovladačů, které posílají přerušení nepravidelně, nebo při volání `dql_completed` až z kontextu NAPI vlákna, což už může být pozdě. Při odesílání ovladače na frontě volají `netdev_tx_sent_queue` a při dokončení `netdev_tx_completed_queue`, která frontu pouze dočasně zapíná a vypíná.

1.2.3 Využití více procesorů zároveň

Předpokládejme nejvyšší možnou velikost rámce, *Maximum Transmission Unit* (MTU), rovnou 1500 B bez Ethernetové hlavičky. Zkusme naivně počítat: při plném vytížení adaptéru s d deskriptory za jedno přerušení obdržíme až $d \times MTU$ bajtů dat. Přijde-li tedy z 10 Gbit/s adaptéru používajícímu 256 deskriptorů 20000 přerušení za sekundu, stihneme přijmout pouze 7,5 Gbit/s! Sledujeme-li chvíli na vytížených systémech počet vykonaných přerušení v čase, ať už v Linuxu zpracováváním `/proc/interrupts` či v OpenBSD sledováním výpisu `systat(1)`, zjistíme, že 20000 za sekundu je opravdu vysoké číslo. Navýšit MTU je samozřejmě možné (rámcům větším než 1518 B se říká *jumbo frames*), ale není-li náš provoz tvořen převážně velkými pakety (např. *Storage Area Network*), budeme tím pouze plýtvat dostupnou pamětí na buffery. Máme-li však v systému více procesorových jader nebo jiných nezávislých jednotek schopných zpracovávat přerušení, můžeme je s moderními kartami zapojit do tohoto procesu.

Kruhových bufferů (pro jednoduchost dále označované jako fronty) může mít adaptér více; přiřadí je dostupným procesorům a přerušení doručuje nezávisle na ostatních. Hardware je naprogramovaný mechanismem *Receive-Side Scaling* (RSS) (`ethtool -x`) tak, aby podle nastavené hashovací funkce rozděloval zátěž mezi použité fronty; nemá-li více front, je stále možné v *top half* naplánovat zpracování každého paketu do BH na jiném procesoru. [RSS-NDIS] To se v Linuxu označuje jako *Receive Packet Steering* (RPS) a je nastavitelné v `/sys/class/net/<ROZHRANI>/queues/`.

OpenBSD má v době psaní práce stále pro většinu jádra jeden zámek, pročež tyto vlastnosti nelze implementovat. Verze 5.4 kromě několika málo „bezpečných“ systémových volání přinesla tzv. *mpsafe* interrupt handlery pro zvukové karty a ovladače dalších zařízení následovaly. V systému je však stále jedna fronta pro každý protokol (např. `ipintrq`) a firewall pf používá spoustu globálních proměnných. Ačkoli fronty a stavová tabulka by měly jít paralelizovat snadno, rozbíjení „velkého zámku“ nemá v projektu vysokou prioritu, neboť běžně přítomný počet gigabitových adaptérů v routeru obslouží systém bez problémů.

1.2.4 Virtualizace

Vícejádrové procesory se rozmohly zhruba ve stejné době, co začala být poptávka po jejich schopnostech hardwarově asistovat při virtualizaci, čímž se nabízí potenciál usnadnit hypervizorům i síťový provoz. *Peripheral Component Interconnect Special Interest Group* (PCI-SIG) zavedla do jedné ze svých PCI-Express specifikací funkci zvanou *Single Route Input/Output Virtualization* (SR-IOV). Prakticky má síťový adaptér v systému na svém zařízení několik *PCI funkcí*, které lze přiřadit virtuálním strojům. Každá funkce má samozřejmě vlastní *Base Address Register* (BAR), který virtuální stroj používá k jejímu nastavování a funkce se tedy tváří jako samostatný síťový adaptér. V systému přítomná *Input/Output Memory Management Unit* (IOMMU) potom překládá adresy mezi virtuálním strojem a danou funkcí síťového adaptéru, takže kromě počáteční konfigurace nemá hypervizor se sítí žádnou režii. Nevýhody tohoto přístupu se začnou projevovat, budeme-li chtít takovýto virtuální stroj migrovat.

1.3 Měření času

Rozhoduje-li QoS router *kdy* pošle který paket, je třeba ujasnit otázku dostatečně rychlého a přesného měření času, aby při dnes běžných desítkách tisíc paketů za vteřinu nedocházelo k technickým problémům porušujícím předpoklady používaných algoritmů. Poul-Henning Kamp, dlouholetý přispěvatel do systému FreeBSD, napsal článek vysvětlující některé základní principy ohledně měření času v jádře běžícím na více procesorech zároveň [PHKtime] a mimo jiné definoval:

Přesnost jako vzdálenost naměřených hodnot od uváděné frekvence časovače, korigovatelnou přičtením nebo odečtením nějaké konstanty.

Stabilitu jako velikost odchylky od uváděné hodnoty (což je nepředvídatelné).

Rozlišení jako délku jednoho kroku časovače (převrácená hodnota frekvence).

Roli hraje i rychlost přečtení aktuální hodnoty z hardware a u časovačů s nastavitelným intervalem vyvolávaného přerušení též rychlost jeho nastavení.

Čistě teoreticky, má-li síťové rozhraní kapacitu 10 gigabitů (10^{10} bitů) za sekundu (FDX), stihne takové zařízení za 1 μs přenést 10 000 bitů, tedy 1250 B. Pokud bychom tedy měli k dispozici měření času s rozlišením 1 μs, měly by naše algoritmy přidělující rychlost přesnost v řádu odeslaných kilobajtů. Rozlišení tedy hraje významnou roli; protože je však velmi závislé na použitém hardware[6], vysvětleme zde používané metody na architektuře *IA-32* a její 64bitové verzi, které jsou dnes nejrozšířenější.

Každý systém má čip řídící hodiny reálného času, *Real Time Clock* (RTC), který je pro naše potřeby nedostatečný a jím vrácený čas má význam pouze dlouhodobý. Historickým pozůstatkem je *Programmable Interrupt Timer* (PIT), nejčastěji viděný jako čip Intel 8254, používající hodinový signál o frekvenci 1,193182 MHz. Už od nepaměti je připojen k IRQ 0. Pomocí instrukce `outb` se na jeho I/O portu dá nastavit dělící registr (*divisor*), aby čip vyvolával přerušení po

[6] Není vhodné ke každému routeru kupovat přijímač GPS či atomové hodiny.

daném zlomku svého hodinového signálu. Kvůli nízkému rozlišení a složité[7] komunikaci pomocí I/O portů se dnes používá výhradně jako záloha, nemá-li daný systém žádný jiný spolehlivý zdroj času k dispozici.

Od procesorů řady Intel Pentium se objevila podstatně přesnější a rychlejší metoda získávání informace o uplynulém čase – jednoduché počítadlo procesorových tiků zvané *Time Stamp Counter* (TSC). Původní verze přinesla instrukci `RDTSC`, která do registrů `EDX:EAX` načetla 64bitový počet uplynulých tiků od restartu procesoru. Znal-li programátor frekvenci svého procesoru, mohl z ní odvodit vzorec převodu počtu tiků na dostatečně přesný údaj v nanosekundách.

TSC ale původně nebyl určen jako zdroj hodin reálného času, nýbrž k měření velmi krátkých kusů kódu za účelem optimalizace. Záměrně zcela postrádal možnost načasovat přeřušení, k čemuž zatím sloužil jen nevyhovující PIT.

První náhradu čipů i8254 přinesla norma *Advanced Configuration and Power Interface* (ACPI) a jmenovala se *Power Management Timer* [ACPI1.0]. Frekvence 3,579545 MHz a atomicky čitelné 24bitové, resp. 32bitové (podle tabulky `FACP`) čítače přinesly kolem roku 1996 znatelné vylepšení. Další náhrada PIT se jmenovala *multimedia timer*, nebo častěji *High Precision Event Timer* (HPET). Šlo o 32bitové časovače s minimální frekvencí 10 MHz. Až 32 nezávislých časovačů nabízelo své čtecí i nastavovací registry oznamované pomocí tabulky `HPET`. Uměly jak periodický režim („budík“, který se po obsluze přerušení opět natáhne na původní hodnotu) tak *one-shot* režim, kdy se po obsluze implicitně neděje nic. Toho mohl využívat tzv. *tickless kernel*, který si takto naplánoval přerušení dle své potřeby a neplýtval energií přerušeními, které by v mezičase neměly žádný význam. Posledním vylepšením je časovač umístěný na lokálním *Advanced Programmable Interrupt Controller* (APIC) každého procesoru, označovaný jako *LAPIC timer*. Frekvence některé ze sběrnic u daného procesoru je zde vydělena hodnotou dělícího registru. Kromě periodického a *one-shot* režimu umí na některých procesorech též tzv. *TSC-deadline* mód, kdy časovač posílá přerušení jakmile je hodnota TSC vyšší než nastavená hodnota (a procesor není v SMM). Pěkné shrnutí včetně příkladů je na stránce `http://wiki.osdev.org/APIC_timer`.

Nejnovější síťové čipy umějí vkládat informaci o čase přijetí paketu do svých deskriptorů. [i217, i82599] Linux toto umí používat; vizte `ethtool -T`.

1.3.1 Problémy s TSC

Jak bylo vysvětleno v kapitole 1.2, k přesné informaci o čase u události s paketem nepotřebujeme zdroj času schopný přerušovat CPU a čím rychlejší máme síť, tím vyšší vyžadujeme od hodin rozlišení. TSC tedy vypadá jako ideální kandidát, avšak práce s ním nebyla vždy jednoduchá. Uveďme zde jeho nejznámější nevýhody, jak se s nimi dá pracovat a jak na ně zareagovali výrobci procesorů.

Závisel na taktu procesoru, na kterém jsme `RDTSC` vykonali. Procesory poté začaly disponovat sofistikovanými mechanismy správy napájení (*power management*), které při nečinnosti snižovaly jejich takt a tím i spotřebu. Bohužel, s tím i naši předpokládanou frekvenci čítače instrukcí. Změna frekvence navíc nebyla okamžitá a zjistit kdy, jestli a na jak dlouho k ní došlo

[7] A pomalé; čip je připojen osmibitovou sběrnicí!

nebylo principiálně proveditelné. Zhruba[8] od uvedení řady Intel Core mívají procesory podporu tzv. invariantního TSC, což zaručuje (mimo jiné) rychlost tohoto čítače nezávislou na P-, C- ani T- stavech daného procesoru, které mohou ovlivňovat jeho frekvenci. [IA32sdm, svazek 3, kapitola 17.13.1][AMD-TSC, AMD64ref] Manuály doslova píší, že je možné používat invariantní TSC jako zdroj reálného času.

Každý procesor měl svůj, tedy ve víceprocesorových systémech reálně hrozilo, že se aplikaci po přeplánování na jiný procesor posunul čas dozadu. To mělo mnohdy fatální důsledky, zejména v na toto nepřipravených (*TSC-sensitive*) interaktivních aplikacích. Ačkoli to verze manuálu dostupná během psaní práce explicitně neuvádí, společnost Intel má patent [ITSCpat] na své řešení, kde je TSC synchronizované na všech jádrech daného procesoru a nedojde-li k přepisu tohoto čitače nebo k odpojení a připojení některého z procesorů, jsou tyto čítače stejné na všech procesorech. [IntelInit]

Superskalární procesory s jeho čtením hýbaly a bylo nutné používat serializační instrukce jako `CPUID`, které vytvářely bubliny v pipeline a narušovaly přirozený průběh měřeného programu, o rychlosti vykonávání nemluvě. Podrobně to popisuje článek od firmy Intel, vydaný krátce po uvedení procesoru Pentium II [RDTSC97]. Řešením byla nová serializující varianta původní instrukce: `RDTSCP`, která navíc atomicky načte číslo logického procesoru, takže je možné nejen sledovat změny mezi procesory, ale též detekovat přeplánování měřené aplikace na jiný procesor.

Frekvenci procesoru se ne vždy podaří přesně zjistit, pročež musejí systémy provést minimálně při spuštění kalibraci nějakým jiným zdrojem času. Oficiální návod od společnosti Intel [iCPUID] radí najít řetězec „Hz“ v modelu procesoru vráceném instrukcí `CPUID` a rozparsovat číslo s potenciální desetinnou čárkou před ním. Společnost VMware používá speciální „backdoor“ I/O port, kterým může virtualizovaný OS zjistit mimo jiné i frekvenci svého procesoru. [OpenVMT]

Pro úplnost zmiňme, že TSC lze také číst z *Machine Specific Register* (MSR), tudy jej lze i změnit a jeho čtení lze dokonce zakázat či emulovat.

1.3.2 Existující implementace

Linux

Důvody pro existenci několika zdánlivě různých API pro nízko- i vysokofrekvenční měření času najde čtenář v `Documentation/timers/hrtimers.txt` [LiSrc]. Subsystém zvaný `hrtimer` počítá čas v nanosekundách uložených v 64bitové hodnotě typu `ktime_t`. Jak jádro zajistí dostatečnou přesnost závisí na zvoleném `clocksource`, který se dozvíme v adresáři `/sys/devices/system/clocksource/` a v jádře jej popisuje stejnojmenná struktura. V kódu disciplín též najdeme funkci `psched_get_time`, která však interně používá `ktime_get`.

[8] Přesně to zjistíme instrukcí `CPUID` s `EAX=0x80000007`, má-li `EDX` nastavený bit 8 [IA32sdm].

OpenBSD

V minulosti se odstraněním volání `microtime(9)` nad každým paketem výrazně zvýšila propustnost některých zařízení, nicméně v částech ALTQ je čas stále potřeba. Správce může zvolit zdroj času nastavením `sysctl kern.timecounter` – každé zařízení schopné vydat informaci o čase má svoji strukturu `timecounter` a hodnotu, která čím vyšší, tím větší má daný zdroj přednost před ostatními.

Dizertační práce popisující implementaci ALTQ [Altq04] píše, že místo přepočítávání TSC na reálný čas pro každý paket škáluje subsystém svoje parametry na TSC. Protože však oba systémy musejí fungovat i na architekturách podobný mechanismus neobsahujících, toto se v nich neděje. Místo toho se obě API snaží vyhýbat drahým instrukcím dělení a aproximují převodní operace například bitovými posuny a přičítáním konstant.

Čas ve virtuálních strojích

Instrukce `RDTSC` vyvolává trap do hypervizora a jednoduchá měření potvrzují, že KVM jednoduše předá svoji hodnotu (což je o poznání rychlejší) [LiSrc, `arch/x86/kvm/emulate.c`], zatímco VMware (patrně kvůli determinizaci tohoto procesu pro *fault-tolerant* migraci procesem *vMotion*) provádí daleko náročnější operace, takže rozdíl oproti nabízeným ACPI hodinám není zvlášť výrazný. Výborný popis přímo od společnosti VMware [vmtime] rozebírá situaci v nejpoužívanějších systémech na trhu. Implementaci TSC v systému Xen popisuje Dan Magenheimer. [xentime]

2. Linková vrstva

Při odesílání paketů je vhodné se kromě prostého doplnění hlavičky zamyslet, jak dlouho bude odesílání trvat (kolik máme ještě dostupné kapacity), jaký provoz na odeslání čeká a jaké jsou na něj kladené požadavky. Přijímací strana zde má první možnosti klasifikace a kromě nepříliš časté regulace provozu[1] zde též figurují mechanizmy jako softwarový bridging (spojení více adaptérů do jednoho L2 segmentu a eliminace případných smyček) nebo trunking (též *link aggregation*, *bonding*, *teaming* apod.), kterými se tato práce nemusí zabývat.

Často zde vytváříme fronty uměle a jejich velikost je rovněž třeba pečlivě zvážit. Máme-li dostatečné prostředky na to udržet fronty v použitelném stavu, můžeme uvést různé algoritmy sloužící k plánování jejich provozu či přidělování kapacity linky a srovnat jejich možnosti, nároky a omezení.

2.1 Vsuvka k implementacím

OpenBSD

Z uživatelského hlediska je *výrazně* vhodnější OpenBSD, neboť má méně možností a nabízí k nastavení jen to, co většina uživatelů potřebuje. Předpokládá, že člověk dostatečně vzdělaný na ladění detailů bude umět aspoň trochu programovat a příslušné parametry si nastaví dopředu. Nový způsob konfigurace:

```
# vi /etc/pf.conf
queue root on { $IFACE1, $IFACE2 } bandwidth $KAPACITA
queue hlavni parent root bandwidth $KAPACITA_H default
queue vedlejsi on $IFACE1 parent root bandwidth $KAPACITA_V1
queue vedlejsi on $IFACE2 parent root bandwidth $KAPACITA_V2
queue lowdelay parent root bandwidth $KAPACITA_L $PARAMETRY_L

pass from (self) set queue vedlejsi
pass from <telefony> set queue (vedlejsi, lowdelay)
```

Výhoda této syntaxe je specifikace vztahu rodič-potomek u potomků a ne na řádku s jejich rodičem, z čehož plyne jednodušší správa velkého množství front. Původní implementace umožňovala pouze 64 front na rozhraní, nová verze toto omezení *de facto* odstraňuje.[2] Fronty jsou pojmenovány řetězci a není-li uvedené jejich rozhraní, vytvoří se na všech. Předáme-li konstrukci `set queue` dvě fronty jako ve druhém pravidle, jádro do druhé fronty zařadí prioritní provoz a prázdná TCP potvrzení, což zlepší interaktivní chování aplikací na špatných připojeních. Není-li na daném rozhraní fronta s daným názvem, použije se fronta s parametrem `default`. Vše je dokumentováno v `pf.conf(5)`.

Pro srovnání uveďme podobnou konfiguraci ALTQ, jak ji najdeme na ostatních systémech BSD. Tato verze uměla místo absolutních hodnot zpracovat i procentuelní část kapacity, což v nové verzi chybí.

[1] Tzv. *ingress shaping* je vhodný zejména na zmírnění potenciálně na CPU náročného provozu (například šifrované VPN), který by mohl svými nároky ohrozit chod celého systému.

[2] Autorem této změny je autor této práce.

```
# vi /etc/pf.conf
altq on { $IFACE1, $IFACE2 } $DISCIPLINA bandwidth $KAPACITA ↙
→queue { hlavni vedlejsi lowdelay }
queue hlavni bandwidth $KAPACITA_H $PARAM_H
queue vedlejsi on $IFACE1 bandwidth $KAPACITA_V1 $PARAM_V1
queue vedlejsi on $IFACE2 bandwidth $KAPACITA_V2 $PARAM_V2
queue lowdelay bandwidth $KAPACITA_L $PARAMETRY_L

pass from (self) set queue vedlejsi
pass from <telefony> set queue (vedlejsi, lowdelay)
```

Linux a traffic control

Neexistence uživatelsky přívětivého manuálu odradila nejednoho správce od konfigurace QoS, protože na rozdíl od OpenBSD nestačí pouze uvést jména a kapacity front a ke stávajícímu pravidlu třemi slovy dopsat, kam patří. Pokusím se zde vysvětlit způsob práce s utilitou tc i s jednotlivými algoritmy v práci rozebranými.

Linux kopíruje architekturu CQS se vší složitostí; dokonce ji přidává, neboť na rozdíl od OpenBSD nedochází k jednotné klasifikaci na vstupu, ale (kromě *ingress* disciplín) až při odchodu z konkrétního rozhraní a navíc částečně závisle na použité disciplíně. Více o klasifikaci je v kapitole 4. Základním stavebním kamenem je *qdisc* a za úkol má především plánovat provoz k odeslání. Zobrazme všechny disciplíny v systému přítomné:

```
$ /sbin/tc qdisc show
qdisc mq 0: dev enp11s0 root
```

`mq` označuje typ disciplíny; tato je výchozí na adaptérech s více hardwarovými frontami. `0:` je identifikátor disciplíny, 32bitové číslo zapsané v šestnáctkové soustavě jako dvě 16bitová oddělená dvojtečkou. `root` říká, že je na daném rozhraní disciplína nejvýše postavená.

`mq` je classful; má třídu pro každou zapnutou odchozí frontu. Při výpisu tříd ale musíme uvést název rozhraní.

```
$ /sbin/tc class show dev enp11s0
class mq :1 root
class mq :2 root
```

Tyto třídy pochopitelně nelze smazat ani k nim přidat nové, ale můžeme pod ně „pověsit“ jiné disciplíny:

```
# tc qdisc add dev enp11s0 parent :1 handle 1111: netem delay↙
→ 111ms
$ /sbin/tc qdisc show dev enp11s0
qdisc mq 0: root
qdisc netem 1111: parent :1 limit 1000 delay 111.0ms
```

Netem je jednoduchá disciplína, která uměle přidává nežádoucí chování za účelem testování; zde jsme na rozhraní uměle zvýšili dobu odezvy. Ač nemůže mít

žádné třídy, můžeme specifikovat jako její dítě jinou disciplínu, do které provoz přijde potom[3]. Její možnosti popisuje `tc-netem(8)`.

```
# tc qdisc add dev enp11s0 parent 1111: fq_codel
$ /sbin/tc qdisc show
qdisc mq 0: dev enp11s0 root
qdisc netem 1111: dev enp11s0 parent :1 limit 1000 delay ↙
→111.0ms
qdisc fq_codel 8018: dev enp11s0 parent 1111: limit 10240p ↙
→flows 1024 quantum 1514 target 5.0ms interval 100.0ms ecn
```

Nezadáme-li `handle`, systém přidělí automatický identifikátor, na který ale záměrně nelze ve složitějších disciplínách nastavit třídy. Složitější funkcionalitu popíše kapitola 2.4.5, kdy si ukážeme první skutečně classful qdisc.

Vymažeme-li kořen, Linux automaticky vrátí výchozí disciplínu (`pfifo_fast` či `mq`). Vymazat `mq` nelze, třeba ji nejdříve něčím nahradit a to potom smazat.

```
# tc qdisc del dev enp11s0 root
```

Jednotky se v OpenBSD specifikují pomocí `Kb`, `Mb`, resp. `Gb`; jde o bity za vteřinu, násobky 1000. Utilita tc má pro tyto jednotky syntax `kbit`, `mbit` a `gbit`, přičemž `kbps`, `mbps` a `gbps` reprezentují bajty za vteřinu násobené 1000 a předponami `ki-`, `mi-`, resp. `gi-` dostaneme násobky 1024.

2.2 Active Queue Management (AQM)

Význam technik v této kategorii je dán problémem dnes nazývaným *buffer bloat*, kdy protokoly vyšších vrstev ve snaze maximalizovat svoji propustnost posílají stále více dat, které se při ucpání sítě namísto zahození ukládají do front. Protože síť stále absorbuje nová data, z pohledu koncové stanice je prostor ke zrychlení. Kdyby však včas dorazila potvrzení o doručených datech a chyběla ta ostatní, odesilatel by tím dostal podnět ke zpomalení. Místo „odložení na později“ tedy *něco* zahodíme[4], což nám ušetří a odesilateli poradí. Nejčastěji používaná technika, *Random Early Detection* (RED), využívá pozorování, že datové toky konzumující velkou část kapacity zaberou úměrně velké místo ve frontách. Pravděpodobnost, že se náhodným výběrem trefíme do velkého toku je tedy větší než do nějakého menšího (který má v celé frontě třeba jen 1 paket, ale je interaktivní a tedy nevhodný k zahození). Novějším algoritmem je *controlled delay* (CoDel, vysl. [kodl]), který místo délky fronty kontroluje nejmenší dobu, jakou byl za daný interval nějaký paket zdržen uvnitř našeho stroje, z čehož plyne nutnost pro každý paket měřit čas příchodu a odchodu. Vzrostla-li nejmenší doba námi způsobeného zpoždění nad požadovanou mez, nestíháme odesílat dostatečně rychle. Popišme si některé používané algoritmy podrobněji.

[3] Mezi commity `02201464` a `10f6dfcf` byla disciplína `netem` classless, zde je jen jako příklad.

[4] Místo zahazování můžeme pakety pouze značkovat; více v dalších kapitolách.

2.2.1 Random Early Detection (RED)

Sally Floydová a Van Jacobson [RED93] v roce 1993 navrhli jako míru ucpání používat průměrnou délku vzniklé fronty[5]. Překročí-li min_{th} paketů, přejde algoritmus do fáze *congestion avoidance* a začne zahazovat s pravděpodobností p_a závislou na průměrné délce avg a počtu paketů od posledního zahození. Frontě o průměrné délce víc než max_{th} paketů algoritmus zahodí všechno. V článku zvolená funkce p_a nám umožňuje tu *reálnou* pravděpodobnost upravovat nepřímo: nechť p_b označuje „jak špatně na tom jsme“ vztahem

$$p_b = max_p \times \frac{avg - min_{th}}{max_{th} - min_{th}}$$

kde $max_p \in (0; 1]$ je uživatelem nastavená nejvyšší hodnota dosažená těsně před tím, než průměr překročí max_{th}. Skutečná pravděpobnost p_a je diskrétní náhodná proměnná s rovnoměrným rozdělením $R(n; \lfloor 1/p_b \rfloor)$, kde n je počet paketů od posledního zahození. Pravděpodobnost zahození se časem zvyšuje a její horní mez je ovlivněná průměrnou délkou fronty s uživatelským parametrem.

$$R(n; 1/p_b) = \frac{1}{1/p_b - n} = \frac{p_b}{1 - n \times p_b}$$

Doporučovaná hodnota max_p byla 0.02, což na horní hranici znamenalo zahodit každý třicátý paket s pravděpodobností 5 % a každý čtyřicátý osmý s pravděpodobností $\frac{1}{2}$. Vizte též `http://www.icir.org/floyd/red.html#notes`.

Ona průměrná délka fronty je exponenciální klouzavý průměr, anglicky *Exponentially Weighted Moving Average* (EWMA), s nastavitelnou vahou $w_q \in (0, 1]$. Nechť d označuje aktuální délku fronty a avg označuje předchozí průměr. Potom je nová průměrná délka fronty rovna

$$(1 - w_q) \times avg + w_q \times d$$

Extrémní váha $w_q = 1$ tedy kompletně ignoruje minulost, což není vhodné[5]. Příliš malý vliv aktuální délky fronty na hodnotu průměru má za následek, že se při zvětšení provozu průměr špatně přizpůsobuje očekávaným výsledkům; pozdní nástup stavu zahazujícího provoz dá prostor k vytvoření persistentní fronty, která prvních několik vteřin škodí interaktivnímu provozu a než se jeho uživatel rozhodne nahlásit problém, vše „se vyřeší“. Naopak po skončení dostatečně dlouhého stahování algoritmus škodí těm spojením, která ve frontách zůstala.

Horní závora w_q

Předpokládejme, že je na začátku fronta prázdná, tedy $avg = 0$, a přijde burst L paketů. Po L-tém paketu bude hodnota avg_L rovna:

[5] Aktuální délka fronty se v závislosti na burstiness provozu mění tak rychle, že její sledování nemá dlouhodobý význam; mechanizmus zahazující při každém burstu by plýtval kapacitou.

$$
\begin{aligned}
avg_L &= \sum_{i=1}^{L} i w_q (1-w_q)^{L-i} && (2.1)\\
&= w_q(1-w_q)^L \times \sum_{i=1}^{L} i(\frac{1}{1-w_q})^i && (2.2)\\
&= w_q(1-w_q)^L \times \frac{\frac{1}{1-w_q} + (L\frac{1}{1-w_q} - L - 1)\frac{1}{(1-w_q)^{L+1}}}{(1-\frac{1}{1-w_q})^2} && (2.3)\\
&= w_q(1-w_q)^L \times \frac{\frac{1}{1-w_q} + (\frac{L}{(1-w_q)^{L+2}} - \frac{L+1}{(1-w_q)^{L+1}})}{(1-\frac{1}{1-w_q})^2} && (2.4)\\
&= w_q(1-w_q)^L \times \frac{\frac{1}{1-w_q} + \frac{L-(L+1)(1-w_q)}{(1-w_q)^{L+2}}}{(1-\frac{1}{1-w_q})^2} && (2.5)\\
&= \frac{\frac{w_q(1-w_q)^{L+1}+w_qL-(L+1)w_q(1-w_q)}{(1-w_q)^2}}{\frac{(1-w_q)^2-2(1-w_q)+1}{(1-w_q)^2}} && (2.6)\\
&= \frac{(1-w_q)^{L+1} + L - (L+1)(1-w_q)}{w_q} && (2.7)\\
&= \frac{(1-w_q)^{L+1} + L - L - 1 + w_qL + w_q)}{w_q} && (2.8)\\
&= L + 1 + \frac{(1-w_q)^{L+1} - 1}{w_q} && (2.9)
\end{aligned}
$$

Krok 1 je součet konečné řady, krok 2 z distributivity sčítání vzhledem k násobení, krok 3 podle

$$\sum_{i=1}^{L} i x^i = \frac{x + (Lx - L - 1)x^{L+1}}{(1-x)^2}$$

Kroky 4 a 5 upravují jen čitatele zlomku z nahrazené sumy, v dalších krocích je použitá binomická věta. Tento vztah je vhodné použít k nastavení w_q.

Existující implementace

V ALTQ jsou v souboru `altq_red.c` pevně definované doporučené hodnoty, v Linuxu se podle popisu v manuálové stránce `tc-red(8)` dají všechny parametry nastavit v bajtech. To vnáší do očekávaných výsledků další nepřesnosti, neboť původní článek nespecifikuje, v jakých jednotkách by měla být délka měřena[6]. Linux se uživatele ptá na `avpkt`, průměrnou velikost paketu na jeho síti, a `burst`, očekávaný počet paketů v burstu (v rovnici výše L).

V souboru `tc_red.c` z balíku iproute2 najdeme výše odvozený vzorec používající bajty namísto paketů; chceme mít min_{th} nad hranicí námi požadovaného průměru:

$$burst + 1 - \frac{qmin}{avpkt} < \frac{1-(1-w_q)^{burst}}{w_q}$$

[6]Několik poznámek najde čtenář na `http://www.icir.org/floyd/REDaveraging.txt`.

Klíčové vlastnosti

Nevyžaduje informace o stavu jednotlivých toků (*per-flow state*), což značně šetří paměť, ale zůstává šance, že se trefí do nevhodného provozu.

Průměrná délka fronty jako relevantní ukazatel je metrika přinejmenším zavádějící. Používáním průměrné velikosti paketu ve výpočtech přestává být algoritmus použitelný pro laiky, kteří nechápou skutečnou (aproximační) roli takovýchto konstant.

Rozšířený nejen v unixových OS, ale i v komerčních řešeních. Zejména proto je důležité algoritmus chápat a umět použít, přestože existují lepší.

Snaží se řešit globální synchronizaci, tj. problém, kdy se vlivem ucpání zahodí část provozu všem datovým tokům zároveň, ty se vzpamatují souběžně a příští burst přijde ode všech najednou. Náhodná proměnná by se měla postarat o „rozptýlení“ zahazování do širšího časového intervalu.

Velké množství variant bylo publikováno a implementováno, což celou konfiguraci značně zesložiťuje. Zmiňme například *Generic RED* (GRED) v Linuxu, umožňující vybírat kandidáty k zahození s pomocí spodních bitů proměnné `tc_index`; vizte kapitolu 3.1.3.

Choose and Keep or Kill (CHOKe)

Algoritmus na první pohled velmi podobný RED náhodně vybere paket z fronty a porovná jej s právě přišlým. Patří-li do stejného toku, zahodí oba. [CHOKe] Varianta implementovaná v Linuxu má několik limitů, po kterých je daleko agresivnější: vizte `tc-choke(8)`.

2.2.2 Blue, Stochastic Fair Blue (SFB)

V roce 1998 publikovala IETF dokument nazývaný *RED manifesto*, [RFC2309] silně doporučující nasazení nějaké formy AQM, přestože jediná rozšířená metoda byla RED a široká veřejnost neměla zkušenosti s nastavováním jeho parametrů. Feng, Kandlur, Saha a Shin [Blue99] tedy přišli s jiným přístupem, konkrétně místo průměrné délky fronty používali pouze její krajní stavy (prázdná nebo plná). Pravděpodobností p_m se zahazuje provoz, který se nevešel do nějakého limitu hloubky fronty L. Při každém takovém zahození paketu p_m zvýší o d_1; je-li linka chvíli prázdná, p_m sníží o d_2. Algoritmus se tedy „naučí“ vhodnou míru zahazování paketů na dané síti.

Jedna z variant, nyní zařazená do Linuxového kernelu (`tc-sfb(8)`), se jmenuje *Stochastic Fair Blue* (SFB), protože stochasticky zohledňuje oddělené datové toky. Disciplína obsahuje Bloomův filtr o L úrovních a N přihrádkách. Příchozí paket se zahashuje a hash najde na všech úrovních, kde v příslušných příhrádkách poupraví pravděpodobnosti p_m podle kritérií výše. Jsou-li ve všech rovny 1 (pro ukládání desetinných čísel je použit formát Q0.16), algoritmus spustí *rate-limiting*. Jinak se zahodí s nejmenší pravděpodobností p_m z přihrádek svého hashe.

Toky, které nereagují na zahazování tak ve svých přihrádkách rychle zvednou pravděpodobnosti na 1 a díky N^L kombinacím je velmi malá šance, že zahodíme

nějaký další tok spolu s nereagujícím. To však nezastaví DDoS útoky s podvrženými zdrojovými adresami, na což reaguje varianta *Resilient Stochastic Fair Blue* (RSFB), která má za SFB umístěný ještě *whitelist* automaticky tvořený z regulérních toků na špatně detekované pokusy o útok.

2.2.3 Controlled Delay (CoDel)

Spoluautor algoritmu RED Van Jacobson začal mít pochybnosti o svém rozhodnutí používat průměrnou délku fronty jako indikátor zahlcení sítě a v roce 1999 se snažil publikovat článek nazvaný *RED in a different light*. [RED-DL99] Článek údajně nebyl přijat kvůli kapitole vysvětlující princip samoregulujících se systémů, [RED-DL10] ale ukázal nové možnosti omezování ucpání sítě. Kathleen Nichols a Van Jacobson navrhli místo délky fronty počítat čas, jaký v ní její stávající obsah strávil [CoDel]. Komunita se zájmem o problém *buffer bloat*[7] se mimo jiné zabývá testováním tohoto algoritmu v různém prostředí.

Stanovili si požadované maximum doby zpoždění, kterou fronty na daném stroji vytvoří, a označili jej jako `target`. Bylo-li po pevně daný časový `interval` zpoždění *všeho* ve frontě větší než `target`, algoritmus se po uplynutí intervalu přepne do zahazovacího stavu. To znamená, že první paket zahodí ihned a další za $\frac{1}{\sqrt{d}}$, kde d je počet zahození od vstupu do zahazovacího stavu. Jakmile začneme target zase splňovat, algoritmus zahazovací stav opustí a provoz teče bez újmy[8].

Výchozí hodnoty jsou 5 ms target hlídaný po 100 ms intervalech a není doporučováno je měnit; algoritmus je doslova navržen jako „one code, no knobs, any link rate“ [CoDel]. Target tedy není pevný limit, ale přání, od kterého když se systém oddaluje, tak nás algoritmus „trestá“ zvyšující se frekvencí zahazování. Důležité je omezení z druhé strany; na 56 kbps modemu stihneme za 5 ms přenést pouze 35 B. Proto se tedy zahazuje pouze, je-li ve frontě víc než MTU bajtů dat. To je ekvivalentní targetu 215 ms; chceme-li jej snížit bez hýbání s targetem, snížíme MTU. Na opačném konci rychlosti dostupného hardware, v řádech gigabitů za sekundu, potom může za 5 ms fronta nabýt až několika MB; ucpání takhle velkých linek z důvodu zahlcení však na unixových systémech v dohledné době řešit nebudeme, protože nás začne tížit propustnost datových sběrnic a paměti.

V Linuxu se měření času ve frontě provádí pouze v rámci odchozí disciplíny, takže je vhodné algoritmus používat s ovladači podporujícími BQL, minimálně pro účely *měření* odezvy.

2.2.4 Proportional Integral controller Enhanced (PIE)

Argumentuje náročností implementace měření času pro každý odbavený paket reagoval na algoritmus CoDel kolektiv lidí ze společnosti Cisco Systems vlastním návrhem nazvaným *Proportional Integral controller Enhanced* (PIE) a publikovaným jako koncept pro organizaci IETF. [PIE12] Namísto použití průměru délky fronty algoritmus používá odhad zpoždění spočtený jako délka fronty dělená odhadem odchozí rychlosti[9]. Pravděpodobnost zahození paketu se upravuje nejen

[7] `http://www.bufferbloat.net/projects/codel/wiki`

[8] Snadná aproximace převrácené druhé odmocniny jednoduchými strojovými operacemi umožňuje rychlé provedení jak na konvenčních procesorech, tak v FPGA. Není ani třeba používat FPU, což v jádře OS znamená režii na ukládání stavu koprocesoru.

[9] Littlův zákon, angl. *Little's Law*.

na základě odchylky odhadovaného zpoždění od cílové hodnoty (*target* jako u algoritmu CoDel), ale též podle rostoucí či klesající tendence tohoto odhadu:

$$p = p + \alpha \times (delay_{estimate} - delay_{target}) + \beta \times (delay_{estimate} - delay_{old\_estimate})$$

Odhad odchozí rychlosti je kvůli výkonu měřen po dostatečně velkých blocích (použitá hodnota 10 kB v makru `QUEUE_THRESHOLD`). Do proměnné `dq_count` se průběžně počítá množství odeslaných dat a po každém bloku se z času od posledního měření nastaví odhad rychlosti do `avg_dq_rate`, což je onen dělitel aktuální délky fronty určující odhad zpoždění.

Krátkodobé bursty nemají na chování algoritmu zásadní vliv, neboť malé množství přepočtů nezvýší zahazovací pravděpodobnost dostatečně vysoko. Přesto má algoritmus proměnnou `burst_time` určující délku nekontrolovaného burstu. Je-li na síti klid (nulová zahazovací pravděpodobnost, stávající i minulý odhad odezvy menší než polovina požadovaného), povolují se bursty délky až 100 ms. Každý přepočet zahazovací pravděpodobnosti od povolené délky odečte uplynulou dobu (stále využíváme pouze jedinou informaci o čase získanou jednou za 10 kB) a zůstane-li tato doba kladná, provoz není omezen (vizte `drop_early()`). Podobný princip, leč operující s velikostí dat místo času, uvidíme v disciplíně *Deficit Round Robin* (DRR). Použití PIE je dokumentované v `tc-pie(8)`.

2.3 Plánovače nepodporující omezování

Tyto plánovače mají být rychlé a jednoduché na použití. Neumějí složité možnosti nastavení, ale v komplexní hierarchii disciplín provádějí ty nejzákladnější funkce.

2.3.1 First In, First Out (FIFO) a prioritní fronty

Linux má několik classless disciplín řadících pakety triviálním algoritmem FIFO. Disciplíny `pfifo` resp. `bfifo` jsou omezené daným počtem paketů resp. bajtů (zadaných parametrem `limit`) a stejně jako většina ostatních disciplín mají dokumentaci v manuálových stránkách `tc-pfifo(8)` resp. `tc-bfifo(8)`. Ve výchozím nastavení má Linux na všech rozhraních disciplínu `pfifo_fast`[10], což jsou tři `pfifo` fronty (*bands*) omezené na `qlen` příslušného rozhraní[11], přičemž dokud je nějaký provoz ve frontě s nižším číslem, na vyšší se nedostane. Tento typ plánování lze nalézt i ve spoustě spravovatelných switchů pod názvem *Strict Priority* (SP). Mapování do těchto tří front probíhá na základě *Type of Service* (ToS) a je nastavitelné poněkud zmateným parametrem `priomap`; Linux převádí ToS na svoje vlastní číslo *lp* „Linux Priority“ a *lp*-té číslo v poli `priomap` je z $\{0, 1, 2\}$, neboť reprezentuje příslušný *band*. *lp* lze nastavit i z uživatelského prostoru (pomocí `setsockopt(2) SO_PRIORITY`) nebo upravit `tc(8)` filtrem; více dále.

Existuje totiž ještě classful qdisc s názvem `prio`, aby měl správce možnost pomocí podřízených disciplín omezit množství provozu na jednotlivých prioritách. Lze nastavit až 16 bandů (konstanta `TCQ_PRIO_BANDS`), nicméně po každé změně jejich počtu je třeba znovu vyplnit `priomap`, popsaný v `tc-prio(8)`.

[10] S výjimkou adaptérů s více hardwarovými frontami, u kterých automaticky nasadí `mq`.

[11] Maximální délku fronty na rozhraní nastavíte pomocí parametru `txqueuelen` programem `ip(8)`, více v `ip-link(8)`.

Jako příklad uveďme interaktivní sezení programu OpenSSH. To je označené ToS hodnotou *lowdelay* (`0x10`) a podle tabulky v `tc-prio(8)` mu tedy kernel přiřadí $lp = 6$. Výchozí nastavení vypadá takto:

```
$ /sbin/tc qdisc show
qdisc pfifo_fast 0: dev enp11s0 root refcnt 2 bands 3 priomap↙
→ 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

Běžný provoz má $lp = 0$ a je tedy v prostřední frontě; `priomap[6]` $= 0$ a tak má SSH sezení nejvyšší prioritu.

OpenBSD od verze 5.1 používá systém podobný spravovatelným switchům či `pfifo_fast`, protože disciplínu `priq` v ALTQ málokdo dobrovolně používal. Kvůli nutnosti zapínat ALTQ na rozhraních znamenalo toto nastavení zvýšenou režii ve *forwarding path*, snižující propustnost v embedded zařízeních s nízkým výkonem CPU. Systém má nyní standardně na každém rozhraní osm front, do kterých můžeme pomocí pf řadit provoz přidáním „`set prio` n“ k pravidlu, kde $n \in [0; 7]$. Výchozí provoz má $n = 3$ (konstanta `IFQ_DEFPRIO`) a speciálně provoz s IPv4 ToS nastaveným na *lowdelay*, STP BPDU, LACPDU a CARP announcementy mají $n = 6$. Na rozdíl od tc zde vyšší n znamená vyšší prioritu. Navíc lze pravidlo napsat jako „`set prio (`n`,` m`)`“ a stavy z takto nastavených pravidel navíc dávají *lowdelay* provozu a prázdným TCP paketům s flagem ACK prioritu m. Toto ocení zejména uživatelé asymetrických připojení, kde snadno ucpaný *egress* nepustí včas potvrzení o právě stažené části souboru, na což TCP zareaguje snížením rychlosti i při dostatku volné kapacity směrem „dovnitř“.

Tento systém je velmi jednoduchý na implementaci[12], velikostí režie téměř k nerozeznání od obyčejné FIFO a protože některý provoz chceme preferovat téměř vždy (ne-li, pf nám jej dovolí překlasifikovat), nelze tyto fronty odstranit.

2.3.2 Deficit Round Robin (DRR)

Nejjednodušší forma plánování *round-robin* v Linuxu je popsaná v `tc-drr(8)`. Správce nastaví jednotlivé fronty a *kvantum* bajtů, o které se zvyšuje *deficit* každé fronty. Při odbavení paketu se deficit sníží o jeho délku, což zaznamená jeho poměr vůči ostatním frontám (kvantum je pro všechny stejné, ve výchozím stavu MTU).

```
# tc qdisc add dev enp19s0 root handle F: drr
# tc class add dev enp19s0 parent F: classid F:1 drr
# tc class add dev enp19s0 parent F: classid F:2 drr
```

2.3.3 Stochastic Fairness Queueing (SFQ)

Paul McKenney v roce 1990 zaznamenal, že zajistit spravedlivost mezi jednotlivými datovými toky přestává být ve větším počtu výpočetně únosné[13] a navrhl stochastickou variantu [SFQ90] hashující datové toky do pevně daného počtu front. Kvůli neustálé možnosti kolize se ale hashovací funkce periodicky mění (jednou za `perturb` sekund), což by mělo kolize učinit méně předvídatelné a krátkodobé.

[12] Vizte makro `IF_DEQUEUE`, ze kterého rovněž plyne plánování typu Strict Priority.

[13] Propustnost pamětí od té doby nevzrostla úměrně využívání sítí.

Implementace v Linuxu má standardně 1024 front (parametr `divisor`) a v případě přetečení se zahazuje vždy z fronty s nejvíce daty. Jelikož při výměně hashovací funkce docházelo ke změnám pořadí paketů v rámci jednoho toku, což zbytečně mate například implementace TCP (Jak dostatečně rychle poznat, jestli paket vynechaný přeuspořádáním někdy přijde, nebo se ztratil?), ve výchozím nastavení je výměna vypnuta (`perturb 0`).

Často používaná varianta dostala název *Enhanced Stochastic Fairness Queueing* (ESFQ), neboť původní SFQ rozšiřovala o možnost změnit hashovací funkci a některé další parametry. [ESFQ] Některé parametry se dostaly do hlavní řady a funkční ekvivalent změny hashovací funkce zajistíme například pomocí DRR a filtru `flow`, více v kapitole 4.2.4.

Pro svoji rychlost je tato disciplína často používaná, ačkoli trpí zásadními nedostatky. Kromě přeuspořádávání paketů jsou problémem aplikace, které ke stažení jednoho datového toku otevřou paralelně n (až stovky) spojení. Jedním ze vstupů hashovací funkce jsou čísla portů, což této aplikaci výrazně zvýší šanci dostat svá spojení do různých front a při zahazování, fronty nejen že neodpovídají skutečným uživatelům, ale zahození jednoho paketu „uškodí“ n-krát méně.

Z těchto důvodů je přinejmenším sporné označovat algoritmus jako *spravedlivý* a není-li nasazen v rámci dostatečně omezené třídy provozu (například vyhrazené pro agresivně se chovající aplikace), je lepší se mu vyhnout.

2.3.4 Quick Fair Queueing (QFQ)

Problémem *round-robin* plánování je, že s rostoucím počtem datových toků roste čas mezi bursty každého z nich. (Plánovač musí projít $\mathcal{O}(n)$ front, než se na stávající zase dostane řada.) V konstantním čase vybrat paket k odeslání a zároveň moc nezhoršit tuto dobu (časový rozdíl odpovídající přenosu zhruba 5×MTU) umí algoritmus *Quick Fair Queueing* (QFQ). [QFQ09] Podobně jako algoritmus *Hierarchical Fair Service Curve* (HFSC) představený dále počítá pro každý přijatý paket *virtuální čas* začátku a konce odesílání v „ideálním systému“ a podle něj pakety posílá na linku. Třídu k přiřadí do *skupiny* i podle nastavené váhy ϕ_k a jejího nejdelšího paketu L_k tak, že $i = \lceil \log_2 \frac{L_k}{\phi_k} \rceil$. Jsou-li váhy od 10^{-6} do 1 a pakety mezi 64 B a 16 kB, dostaneme takto méně než 32 skupin. Číslo skupiny tedy slouží jako index do bitových polí, na základě kterých se plánuje; instrukcí `ffs` (*find first bit set*) vybereme nejvhodnější skupinu, ve které jsou pakety *bucket-sortem*[14] seřazené podle virtuálního času začátku odesílání. Bitová pole se skupinami jsou celkem čtyři; plánuje se výhradně z pole *eligible, ready* (ER) s pakety, které v ideálním systému již měly odejít, a neblokují je toky s vyšším indexem skupiny, které již měly *dojít*. Pole *eligible, blocked* (EB) obsahuje bity u skupin, které již měly odejít, ale je před nimi důležitější provoz, *ineligible* varianty těchto polí jsou udržovány tak, aby ve vhodný virtuální čas mohly být v konstantním čase „povýšeny“ do stavu *eligible*.

```
# tc qdisc add dev enp19s0 root handle A: qfq
# tc class add dev enp19s0 parent A: classid A:1 qfq weight 1↙
→ maxpkt 4088
```

[14] Přesněji jeho aproximací sloužící k omezení univerza časů a tudíž zajištění konstantní složitosti.

```
# tc class add dev enp19s0 parent A: classid A:2 qfq weight 5
```

2.3.5 fq_codel, návrh na nový výchozí algoritmus

Komunita se zájmem o problém buffer bloat se snaží navrhnout a prosadit algoritmus vhodný pro masové rozšíření, zajišťující běžným uživatelům rychlejší odezvy v interaktivních aplikacích (včetně prohlížení webu) pokud možno za všech okolností. Při odbavování front používá DRR a CoDel na zajištění odezvy, zatímco do nich provoz řadí minimalistická varianta SFQ rozdělující různé toky ve výchozím nastavení do 1024 `flows` pomocí Jenkinsovy hashovací funkce. Teoreticky máme tedy šanci 1:1024, že naši interaktivní aplikaci ohrozí v jedné frontě vysoký provoz jiných uživatelů, přičemž na výstupu takováto fronta s vyčerpaným deficitem před sebe pustí ty ostatní, po čemž následuje CoDel. Pro svoji jednoduchost qdisc neobsahuje žádnou podporu priorit; k dosažení efektu `pfifo_fast` nad ním můžeme použít classful qdisc `prio`, vizte sekci 2.3.1.

Každý *flow* má záznam maximálně 64 bajtů dlouhý a operace s frontou jsou pouze na začátku (*dequeue*) a na konci (*enqueue*), což minimalizuje počty výpadků cache ve *forwarding path*. Dojde-li na zahazování paketů, projde se všech 1024 front a z té nejobsazenější se zepředu zahodí paket. Obsazenost je rovněž uložena efektivně, ve 4 kB souvislé paměti, takže je na moderních procesorech s hardwarovým prefetchingem ono maximum nalezené rychle.

Nasazení této classless disciplíny je záměrně nejjednodušší možné; případné parametry konzultujte s `tc-fq_codel(8)`. V době psaní práce již měly tuto disciplínu jako výchozí vybrány některé Linuxové distribuce, například OpenWRT (`https://openwrt.org/`) určená do domácích routerů.

```
# /sbin/tc qdisc add dev enp11s0 root fq_codel
```

2.3.6 Heavy-Hitter Filter (HHF)

Patrně nejnovějším[15] algoritmem na spravedlivější plánování paketů je *Heavy-Hitter Filter* (HHF) používající tzv. *multi-stage* filtr. [HHF13] Klasifikace paketů probíhá pomocí k hashovacích funkcí do pole k čítačů počtu dat do nich zahashovaných. Toky s velkým objemem dat (angl. *heavy-hitters*, zkr. HH[16]) své čítače posunou výše než ostatní a pokud pro jakýkoli tok *všechny* čítače přesahují `admit_bytes`, je automaticky klasifikován jako HH. Čítače se nulují s periodou `reset_timeout`; máme tedy dva parametry určující minimální rychlost přenosu HH. Algoritmus si standardně pamatuje až 2048 HH toků (`hh_limit`) ve spojovém seznamu procházeném při každé klasifikaci a vyhazuje záznamy neaktivní déle než 1 s (`evict_timeout`). Je-li tok již na seznamu, neinkrementují se jeho čítače ve filtru, aby se uvolnil prostor ostatním tokům. Plánovačem je vážený DRR zkopírovaný z disciplíny `fq_codel` (HH mají váhu 1, malé toky `non_hh_weight`).

Parametry jako 16 kB filtr (přestože se periodicky maže pouze 512-bytová bitmapa), funkcí `kzalloc` alokovaný spojový seznam délky až 2048 a stoprocentní

[15] Podpora v programu `tc(8)` byla do balíku iproute2 začleněna několik dní před odevzdáním práce (commit `ac74bd2a`) a manuál ještě není napsaný.

[16] Přehlédněme slangový význam slovního spojení; v populární literatuře se datové toky občas přirovnávají ke zvířatům. Vizte `http://en.wikipedia.org/wiki/Elephant_flow`.

jistota chytit požadované toky s možností občas ublížit nějakému dobře se chovajícímu — takovýto algoritmus bude místo poskytovatelů internetu se stovkami tisíc drobných spojení v tísících front určen spíše do datových center, kde jsou rozdíly v rychlostech řádově větší, počty toků řádově menší a provoz s vysokými nároky na dobu odezvy je pravděpodobně odbaven přednostně.

2.4 Rozdělování kapacity linky

V architektuře CQS zatím vynechme klasifikační fázi (které se věnuje až kapitola 4, protože často potřebuje informace od vyšších vrstev) a ukažme implementované mechanizmy, které s našimi třídami provozu umějí pracovat. Připomeňme, že minimálně od uvedení prvního článku o systému ALTQ v roce 1998 [Altq98] se této komponentě říká *queueing discipline* (qdisc) a má za úkol primárně provoz z dané třídy přijmout (operace *enqueue*) a až je požádána, jejím kritériím vyhovující provoz naplánuje k odeslání (operace *dequeue*).

Z pohledu nepříliš vzdělaného správce je *traffic shaping* metoda na „omezení rychlosti“, ale ve skutečnosti má daleko důležitější význam: zmenšovat *burstiness* odchozího provozu. [17] Tedy i když my zrovna máme kapacity dostatek a zdánlivě nemáme důvod upřednostňovat prioritní provoz před velkým množstvím méně důležitého, někde za námi může existovat hloupý uzel, který té kapacity tolik nemá a dojde mu burst nedůležitých paketů. Nezbyde mu ve frontě místo pro ty prioritní a zahodí je. Na cestě od koncového uživatele k *Internet Service Provider* (ISP) k tomu typicky dojde v místech, kde má ISP nainstalované zařízení s velmi vysokým průtokem a z technických důvodů může použít pouze *policing*, který žádný provoz nezdrží.

2.4.1 Leaky Bucket

Technika „děravého kbelíku“ (*Leaky Bucket*) byla popsaná už v roce 1986 jako nejjednodušší řešení alokace kapacity. [NewDir] Data lijeme do kbelíku[18], v jehož dně je díra o velikosti dle naší požadované maximální rychlosti. Úroveň hladiny se dá naprogramovat periodicky snižovaným[19] čítačem, který zvýšíme s každým přijatým paketem. Přeteče-li kbelík (čítač dosáhne nastaveného horního limitu), nová data zahodíme (*tail-drop*). Velikost kbelíku tedy reguluje, jak velký burst náš systém udrží.

2.4.2 VirtualClock

Lixia Zhangová v roce 1989 představila návrh nové architektury sítí s přepínáním paketů nazvané *Flow Network*, která měla přinést možnost garantovat službě propustnost a dobu odezvy. Jednou z myšlenek v této technické zprávě [FlowNet] byl algoritmus *VirtualClock* sloužící k měření a plánování provozu podobně jako *Time Division Multiplexing* (TDM), ale s výhodami statistického multiplexingu jako například možnost za běhu půjčit nevyužité pásmo jiným žadatelům.

[17] Někdy též „vyhlazení“ průběhu datového toku.

[18] Prioritní data si představme jako kapalinu o vyšší hustotě.

[19] Plýtvat na tohle časovači je zbytečné, stačí při operaci *dequeue* odečíst, kolik času uplynulo.

Systémy TDM hodinami reálného času rozdělí dostupnou kapacitu a přidělují kanály svým uživatelům, aniž by se tito mezi sebou ovlivňovali. Nemá-li nějaký uživatel co poslat, jeho časové kvantum zůstane nevyužité, což má přesně řešit statistický multiplexing. Libovolný datový tok si v něm můžeme představit, jak plyne konstantní rychlostí v nějakém virtuálním čase – tik virtuálních hodin je určen průměrnou mezerou mezi jednotlivými pakety. Požadovanou rychlost r_i nám zadá uživatel, t je aktuální stav hodin reálného času.

Při inicializaci zaveďme pro třídu i časovou jednotku $vtick_i = \frac{1}{r_i}$

První přijatý paket dosadí t do proměnných $VirtualClock_i$ a $auxVC_i$

Po přijetí paketu nastavíme $auxVC_i$ na $max(t, auxVC_i)$ a toto bude čas, kdy paket odejde z našeho systému. Potom zvýšíme obě proměnné o $vtick_i$.

Proměnná $auxVC_i$ zaručí požadovanou maximální rychlost a $VirtualClock$ pouze nese přehled o celkově přenesených datech; jakmile překročí t, přijali jsme víc paketů, než byl dovolený limit. Měřící proces může nastavením $VirtualClock = t$ resetovat měřící interval.

Tento algoritmus je na první pohled hodně podobný děravému kbelíku, ale takto napsaný má nekonečnou paměť. Hlavní rozdíl je ale v tom, že u každého přijatého paketu máme definovaný čas, kdy má odejít. Děravý kbelík má pouze frontu, kterou umí hýbat a udržovat ji. Informace o (virtuálním) čase odeslání se totiž může hodit plánovači rozhodujícím se mezi více frontami.

2.4.3 Token Bucket Filter (TBF)

Linux má *Token Bucket Filter* (TBF) jako asi nejjednodušší algoritmus na *traffic shaping* a jde o velmi minimalistickou implementaci děravého kbelíku. Povinné parametry `rate`, resp. `burst`, resp. `latency` nastavují rychlost odtoku dat, resp. množství dat k okamžitému odeslání v bajtech (Pozor na granularitu časovače!), resp. velikost kbelíku určující, kolik dat disciplína smí zdržet (měřeno v sekundách způsobeného zpoždění). Volitelný parametr `peakrate` nastavuje druhý kbelík omezený na jeden paket, který má za úkol omezovat rychlost „vypouštění“ burstu.

Dokumentace `tc-tbf(8)` i kód jsou plné upozornění na problémy s nedostatečným rozlišením časovače a používá-li hardware offloading spojující menší pakety do větších bloků, *Generic Segmentation Offload* (GSO), algoritmus je ve funkci `tbf_segment` opět rozdělí. Kvůli těmto faktům bych se osobně raději v praxi bez detailního měření tomuto algoritmu vyhnul.

```
# tc qdisc add dev enp19s0 root tbf rate 15Mbit burst 30Kbit ↙
→latency 10ms
```

2.4.4 Class-Based Queueing (CBQ)

Sally Floydová a Van Jacobson [CBQ95] zdůraznili, že mít možnost sdílet nevyužitou kapacitu mezi jednotlivými třídami hraje zásadní roli, a tak na základě předchozích publikovaných [SCZ93] i nepublikovaných článků popsali strukturovaný mechanismus zvaný *Class-Based Queueing* (CBQ). Na ukládání politiky

použili strom, jehož uzly tvoří jednotlivé třídy, a pakety jsou spojené pouze se třídami v listech. Součet kapacit přidělených dětem daného uzlu nesmí překročit jeho vlastní kapacitu. Úroveň (*level*) uzlů je definovaná jako 0 pro listy a nejvyšší úroveň dítěte +1 pro ostatní uzly. *Obecný* plánovač (*general scheduler*) z listů vybírá naplánovaný provoz a stará se o jeho požadavky na odezvu. Může také počítat jejich množství a provádět shaping; třída je *vyčerpaná*, použila-li v poslední době více než vyhrazené kapacity. O vnitřní uzly stromu se stará *rozdělovací* plánovač (*link-sharing scheduler*), který může vyčerpaným listům půjčit nevyužitou kapacitu z jiné části podstromu. Toto se přitom nesmí dít zcela libovolně, ale na základě nějakého rozhodnutí (Chceme, aby nejméně platící uživatelé měli někdy stejné podmínky jako ostatní?). Řekneme, že třída je *neregulovaná*, obsluhuje-li ji obecný plánovač; jakmile musí nastoupit plánovač rozdělovací, třída se označuje jako *regulovaná*. Ve skutečnosti je mechanizmus tvořen jedním plánovačem kombinujícím obě části a komponenta zvaná *estimátor* pro každou třídu měří, která část plánovače se má použít.

Třídy označené jako *bounded* si nesmějí půjčovat kapacitu linky od svých sourozenců; *izolované* třídy nesmějí navíc ani půjčovat svoji nevyužitou kapacitu ostatním. Článek požaduje, aby třída zůstala neregulovaná, právě když platí nějaká z následujících možností:

Není vyčerpaná, nebo

Má nevyčerpaného předka na úrovni i a zároveň na úrovních nižších než i žádný provoz nečeká na odbavení.

Kontrolovat druhou podmínku pro libovolný podstrom může být výpočetně náročné, pročež navrhuje článek dvě aproximace: *ancestor-only* a *top-level*. První aproximace má ke konci článku příklad, který při velkém množství tříd a správném provozu dovolí dřívějším třídám nechat vyhladovět ty poslední. Top-level aproximace upravuje druhý bod na:

Má předka s volnou kapacitou na úrovni aspoň *toplevel* (v ALTQ „cutoff“). Tato proměnná se řídí heuristikou při příchodu paketu nastavující nejnižší úroveň po cestě od kořene do aktuální třídy, na které má uzel volnou kapacitu. Pokud při odbavení paketu vyprázdníme frontu jeho třídy nebo třída přestane být neregulovaná, *toplevel* nastavíme na ∞.

Operace *dequeue* používá *Weighted Round Robin* (WRR) na priority a v rámci nich vybírá provoz ze tříd (`cbq_dequeue_prio` v Linuxu). Pro danou linku a požadovanou rychlost spočítá, jak dlouho která třída *nesmí nic dělat*, aby na lince zůstalo místo pro ostatní.

Estimátor: buď v klidu, ale ne moc dlouho

Disciplíně zadáme jako jeden z parametrů celkovou dostupnou kapacitu. Při každé operaci *dequeue* změříme, jak dlouho byla linka v klidu a aktualizujeme průběžně ukládaný dlouhodobý průměr (EWMA, uložený ve stavové proměnné `avgidle`). Rozdíl skutečné a průměrné doby klidu je kladný, dodržuje-li třída svůj limit; záporný rozdíl říká, že třída dočasně přečerpala svůj limit a plánovač ji nějakou dobu nechá být.[20] Průměrná doba klidu musí být samozřejmě limitována, aby

[20]Tyto výpočty provádí v Linuxu `cbq_update` a v OpenBSD `rmc_update_class_util`.

třída nemohla nekonečně „šetřit“ kapacitu a následně se ji pokusit překročit; tehdy místo průměru dosadíme konstantu `maxidle` spočtenou z požadovaného počtu paketů `maxburst` (jejich velikost je `avpkt`), který bychom chtěli při prázdné frontě bez problémů odbavit. Důvodem pro nastavování `maxidle` pomocí odhadu „kolik průměrných paketů budeme chtít“ namísto přímého zápisu je, že program tc konstantu uloží tak, aby se jádro obešlo bez FPU a instrukcí dělení při výpočtu EWMA (dělení číslem 2^{ewma_log} je bitový posun o $ewma\_log$ doprava).

Maxburst dovnitř, minburst ven a čekat offtime

Protože někdy není vhodné, aby algoritmus poslal pouze jeden paket a čekal do dalšího naplánování odesílajícího vlákna na CPU, parametrem `minburst` lze zvýšit granularitu celého procesu: algoritmus odešle tolikrát více dat a třídy potom tolikrát déle čekají. Výchozí nastavení je 0, ale na systémech s pomalým CPU nebo delším kvantem plánovače (např. 100 Hz) můžeme s touto hodnotou experimentovat; spočítá se z ní konstanta `offtime`, kterou se penalizují přečerpané třídy. Pokud adekvátně nezvýšíme `maxburst`, pouze tím zvýšíme dobu čekání!

ALTQ automaticky přizpůsobí `offtime` kapacitě příslušné linky. tc nabízí třídám namísto převzetí `offtime` a `maxidle` od rodiče vypočíst jejich vlastní; k úpravě „představy o dostupné kapacitě“ slouží matoucí parametr `bandwidth`.

Protože tato třída nemá v classless použití velký význam (stejný efekt dosáhneme pomocí jednoduššího TBF), ukažme si nyní práci s classful třídami pomocí tc.

2.4.5 Vsuvka k implementacím: *classful qdiscs*

Zatímco v ALTQ stačí jednoduše přidávat řádky začínající `queue`, tc má na první pohled matoucí syntaxi velmi podobnou definici qdiscu, lišící se podstatnými detaily. `cbq` je v komunitě známá především svojí složitostí, zdánlivou nesmyslností požadovaných parametrů a implementací, která nevrací jasné chybové zprávy (často vidíme jen výstup `strerror(3)`).

```
# tc qdisc add dev enp3s0 root handle 1: cbq bandwidth 42Mbit↙
→ avpkt 512
# tc class add dev enp3s0 parent 1:0 classid 1:DEAD cbq allot↙
→ 3200 rate 2Mbit bounded
# tc class add dev enp3s0 parent 1:0 classid 1:BEEF cbq avpkt↙
→ 1000 rate 4Mbit
```

První řádek pověsí qdisc do stromu pod interface `enp3s0`, pro názornost přímo do kořene. Všimněme si explicitního pojmenování uzlu číslem `0x00010000`, ke kterému jsou vztaženy jeho potomci – dvě různé třídy.

Pokud jsme četli `tc-cbq(8)` či `tc-cbq-details(8)`, může nás zmást toto:

`allot` je údajně pro třídy povinný; určuje totiž počet bajtů při jedné operaci *dequeue* z dané třídy odeslaných[21]. Specifikujeme-li však `rate` (maximální rychlost) a `avpkt`, nastaví se `allot` na minimální hodnotu $\frac{3 \times avpkt}{2}$.

[21] Stejnojmenný parametr u disciplíny ale slouží k něčemu úplně jinému, a to je výpočet tabulky `rtab[l]`, kde je pro danou rychlost uložená doba odesílání paketu o délce l. Z prostorových důvodů se potom l zarovná na násobek `cell`, kterou pokud nedefinujeme, bude z `allot` spočtena tak, aby měla tabulka nejvíce 255 řádků. Více v `tc_calc_rtable`.

`bandwidth` u disciplíny nastaví její maximální kapacitu, což je vhodné na pomalejší lince připojené do rychlejšího rozhraní ke zmenšení bufferů. U třídy jej naopak nepotřebujeme, protože slouží jen k ladění `maxidle` a `offtime`.

`prio` ve skutečnosti povinný není; nezadáme-li jej, nastaví se nejnižší (numericky nejvyšší) priorita a nejmenší váha pro WRR.

2.4.6 Hierarchical Fair Service Curve (HFSC)

Formální přístup nezávisle přidělující kapacitu, odezvu i schopnost sdílet kapacitu s ostatními třídami zvolili Ion Stoica, Hui Zhang a T. S. Eugene Ng [HFSC99]. V této kapitole budeme písmenem i označovat třídu a písmenem t aktuální čas. *Křivka* (angl. *service curve*) je neklesající funkce říkající množství dat v daném čase – kolik bylo třídou obslouženo resp. kolik je pro ni rezervováno značí $w_i(t)$ resp. $S_i(t)$. Značme $w(t_1, t_2)$ počet dat odeslaných v době mezi t_1 a t_2; tedy $w(t) = w(0, t)$. Uživatel má možnost každé třídě specifikovat křivku `realtime` popisující průběh přidělování garantované kapacity, případně křivku `linkshare` určující relativní kapacitu k půjčení sousedům. Strop určuje křivka `upperlimit` a nad rámec původního článku ji dopsal Oleg Cherevko.

Z těchto tříd potom opět sestavíme strom, kde listy budou garantovat své křivky a vnitřní uzly jen kontrolovat proveditelnost našich požadavků. Buď $B_i(t)$ množina všech časů před t, kdy třídě i vznikla dočasná fronta dat čekajících na odbavení; „*beginnings of backlogged periods*“. Pro jednoduchost nechť čas 0 je v každé B_i. Křivka S_i je *garantovaná*, pokud pro každý t_2, kdy provoz dané třídy čeká na odbavení, existuje $t_1 \in B_i(t_2)$, že $S_i(t_2 - t_1) \leq w_i(t_1, t_2)$. Speciálně začátek křivky S_i musí zajistit, aby se od t_1 do t_2 třídě i dostalo požadované rychlosti. Vycházeje z algoritmu *Earliest Deadline First* počítáme deadline křivku $D_i(t)$ jako nejmenší množství dat, které by do času t mohla třída i mít zaručeně odeslané.

$$D_i(t) = min_{s \in B_i(t)}(S_i(t - s) + w_i(s))$$

Prakticky není nutné ukládat celou množinu B_i, stačí dva největší prvky b_1 a b_2.

$$D_i(b_2, t) = min(D_i(b_1, t), S_i(t - b_2) + w_i(b_2))$$

Pokud jsme tedy od času b_2 zrychlili nad křivku S_i, v čase t dostaneme minimálně tolik, co bylo spočtené v čase b_1. Analogicky zpomalení nárůstu w_i (tj. aktuální rychlosti) vede ke snížení požadavků v čase t, abychom nemohli kapacitu „spořit“.

Nás ale typicky nezajímá, kolik dat by mohlo být kdy přeneseno, ale *kdy bude odeslán příští paket*. Potřebujeme inverzní funkci: buď $D_i^{-1}(y)$ nejmenší čas x, že $D_i(x) = y$. Právě příchozí paket o délce l tedy bude odeslán nejpozději ve:

$$d_i = D_i^{-1}(w_i(t) + l)$$

Způsob zajištění odezvy

Tradiční algoritmy jako FQ nebo VirtualClock zřejmě nabízejí křivku shodnou s lineární funkcí, jejíž sklon – přírůstek dat v čase – je její přenosová rychlost. Mějme tedy s takovouto křivkou dvě třídy, f pro FTP a v pro real-time video. Na video rezervujeme 3 Mbit/s, na FTP 30 Mbit/s a uvažujme, že právě přišel

burst o velikosti 64 kB. d_v je nejmenší čas x, že $D_v(x) = 64$ kB a protože $w_v(t)$ je zatím 0 a přenést takovýto burst videa bude podle rezervace trvat 170 ms, to je také doba, kterou algoritmus nastaví jako deadline. Přitom ve třídě f, kterou doba odezvy principiálně nezajímá, bude deadline desetkrát menší!

Autoři kromě přenosové rychlosti chtěli zohlednit podobné nároky na odezvu a proto zvolili křivku po dvou částech lineární: prvních d milisekund má křivka sklon m_1 a potom m_2. Real-time provoz tedy využije krátkou deadline, což pustí bursty videa rychle do cíle, ale dlouhodobě nemá třída o moc víc, než potřebuje. (Mezi bursty je totiž dostatečná prodleva pro ostatní provoz.)

Křivka je *konvexní* resp. *konkávní*, má-li druhou derivaci kladnou resp. zápornou. Implementace se zabývají pouze konkávními křivkami, neboť provoz nevyžadující garantovanou dobu odezvy je typu *best effort* a svoji rychlost tak přizpůsobí dostupné kapacitě.

Plánovač

Deadline křivka říká počet dat *už odeslaných* a plánovač HFSC má třídy podle času d_i seřazené, aby odešel vždy ten provoz, který má podle garantovaných křivek dojít nejdřív. Čas začátku odesílání těchto dat e_i se označuje jako *eligible time* a počítá se z inverze deadline křivky, pochopitelně bez délky odesílaného paketu. $d_i - e_i$ je tedy doba, jakou bude paket cestovat při rychlosti plynoucí z jeho křivky a minulého chování dané třídy.

Třida i je *vhodná* (*eligible*), když $e_i < t$. Vhodné třídy jsou na seznamu seřazeném podle d_i připravené k odeslání. Pokud není třeba stíhat něčí deadline, můžeme zbylou kapacitu rozdělovat ostatním; virtuální čas v_i reprezentuje normalizované množství služby, kterou třída i dostala.

Plánovač je podobně jako u CBQ rozdělen na dvě části: ze seznamu vhodných kandidátů vybírá *real-time* kritérium ty třídy, které potřebují garantovanou službu. Nejsou-li takové, zbylou kapacitu rozdělí *rozdělovací* kritérium podle virtuálního času: na každé úrovni vezme nejmenší a jakmile narazí na list, naplánuje jej. Z toho speciálně plyne, že třídy půjdou za sebou nikoli průchodem do šířky, ale do hloubky, takže hloubka stromu sama o sobě na prioritu dané třídy nemá vliv. Díky stálému řazení seznamů aktivních dětí resp. seznamu vhodných tříd plánovač vždy vybere tu třídu, která to nejvíc potřebuje, nezávisle na jejím umístění ve stromové struktuře.

Existující implementace

Kód HFSC v Linuxu i v ALTQ (ze kterého stávající subsystém v OpenBSD kód převzal) je většinou shodný, ale zatímco ALTQ má seznamy aktivních dětí uložené jako spojové seznamy a ručně je udržuje seřazené podle virtuálního času, Linux má místo toho dva černo-červené stromy, podle virtuálního času a podle tzv. *fit-time* používaného k výpočtu upperlimit křivky. Aby nedocházelo k velkým rozdílům mezi sourozenci, je po aktivování třídy jako virtuální čas dosazen aritmetický průměr první a poslední třídy na daném seznamu aktivních. Tedy se dá předpokládat, že `actlist_insert` půjde relativně daleko po spojovém seznamu a černo-červený strom najde využití. Rovněž seznam vhodných tříd je v Linuxu černo-červený strom; teoretická výhoda asymptoticky logaritmických operací je nesporná, ale není-li v systému příliš velký rozptyl v požadavcích na real-time,

většina přidávání vede přímo na konec (na což je `ellist_insert` optimalizovaná) a rovněž čtení je výhradně ze začátku.

Příklady

```
# vi /etc/pf.conf
queue root on vmx0 bandwidth 300Mb
queue lowdelay parent root bandwidth 4Mb min 5Mb burst 40Mb ↙
→for 500ms, max 14Mb qlimit 200
queue servery parent root bandwidth 10Mb, max 200Mb
queue misc parent root bandwidth 64Kb max 2Mb default

queue zakaznik1 parent root bandwidth 1Mb burst 3Mb for 5000↙
→ms, max 20Mb
queue z1a parent zakaznik1 bandwidth 64Kb min 512Kb max 1Mb
queue z1b parent zakaznik1 bandwidth 64Kb max 20Mb
queue zakaznik2 parent root bandwidth 2Mb burst 6Mb for 5000↙
→ms, min 512Kb, max 30Mb
```

Klíčová slova `bandwidth` (povinné), resp. `min`, resp. `max` znamenají křivky `linkshare`, resp. `realtime`, resp. `upperlimit`.

Třída `lowdelay` má tedy za každých okolností povolen krátkodobý burst a zákazník 1 má svoji třídu na real-time provoz `z1a`. Servery budou mít desetkrát resp. pětkrát více kapacity z dostupné volné (až do příslušného horního limitu, samozřejmě) než zákazníci 1 resp. 2. Někteří ISP prvních několik (zde 5) vteřin povolí větší podíl na sdílené kapacitě, což uspokojí zejména uživatele webových stránek nestahující velké objemy dat.

Ekvivalent pomocí tc:

```
# tc qdisc add dev enp3s0 root handle f:0 hfsc default 0xDEFA
# tc class add dev enp3s0 parent F:0 classid F hfsc sc rate ↙
→300Mbit ul rate 300Mbit
# tc class add dev enp3s0 parent F:F classid DE1A hfsc sc m1 ↙
→40Mbit d .5 m2 5Mbit ul m2 14Mbit
# tc class add dev enp3s0 parent F:F classid 5E8B hfsc ls m2 ↙
→10Mbit ul m2 200Mbit
# tc class add dev enp3s0 parent F:F classid DEFA hfsc sc m2 ↙
→64Kbit ul m2 2Mbit

# tc class add dev enp3s0 parent F:F classid C001 hfsc rt ↙
→rate 512Kbit ls m1 3Mbit d 5 m2 1Mbit ul m2 20Mbit
# tc class add dev enp3s0 parent F:C001 classid CA01 hfsc sc ↙
→rate 512Kbit ul rate 1Mbit
# tc class add dev enp3s0 parent F:C001 classid CB01 hfsc sc ↙
→rate 20Mbit ul rate 20Mbit

# tc class add dev enp3s0 parent F:F classid C002 hfsc rt m2 ↙
→512Kbit ls m1 6Mbit d 5 m2 2Mbit ul m2 30Mbit
```

OpenBSD	Linux	poznámka ke frontě
root	`F:F`	prohloubení stromu, qdisc neumí omezit kapacitu
lowdelay	`F:DE1A`	tc nás nenechá nastavit `ul` bez `ls`
misc	`F:DEFA`	výchozí fronta pro neklasifikovaný provoz
zákazníci	`F:Cxxx`	ve jménech tříd daného qdiscu stejné první 2 B

2.4.7 Hierarchical Token Bucket (HTB)

Velmi populární algoritmus ukládající třídy rovněž do stromu, navržený a v několika verzích implementovaný Martinem Deverou [HTB03], si dává za cíl odstranit složitost a neefektivitu Linuxové implementace CBQ. Každá třída v má:

Zaručenou rychlost (*assured rate*) $ar(v)$ a je-li třída nějakou dobu v klidu, může k ní dostat až *burst* bajtů; to je zároveň způsob, jak vyrovnat nepřesnosti při měření času[22]. Aktuálně naměřenou rychlost značme $r(v)$.

Strop $ceil(v)$; rychlost, kterou nesmí nikdy překročit (analogicky nad ni *cburst*).

Úroveň $level(v)$; listy mají 0, kořen 7 (`TC_HTB_MAXDEPTH - 1`, není-li to list) a ostatní uzly o 1 menší než jejich rodič.

Prioritu $p(v)$ (opět čím menší hodnota, tím vyšší priorita), kterou můžeme upřednostnit na dané úrovni tuto třídu před ostatními.

Barvu *červenou* resp. *žlutou* resp. *zelenou*, je-li $r(v) > ceil(v)$ resp. $r(v) \geq ar(v)$ resp. $r(v) < ar(v)$.

Děti daného uzlu značme $K(v)$.

Kvantum $q(v)$ je jednotka dat, kterou rodič půjčí danému dítěti, vyčerpá-li svůj příděl kapacity. Poměr rychlosti a velikosti kvanta se ovládá parametrem `r2q`, s limity hlášenými do `dmesg(8)`.

Zájemci $D(v)$ jsou listy z $K(v)$, do kterých vede cesta složená ze žlutých uzlů (včetně listů samotných). Mohli by si chtít půjčit kapacitu od v.

Vnitřní uzly stromu mají definované $r(v) = \sum_{d \in K(v)} r(d)$ a síťový provoz mohou obsluhovat jen listy. Půjčená (*borrowed*) rychlost uzlu v od rodiče p je

$$b(v) = \begin{cases} \frac{q(v) \times r(p)}{\sum_{i \in D(p), p(i) = p(v)} q(i)} & \text{pokud } p(v) \leq \min_{i \in D(p)} p(i) \\ 0 & \text{jinak} \end{cases}$$

Mělo by platit, že si uzel mohl půjčit až po svůj *ceil*:

$$r(v) = \min\{ceil(v), ar(v) + b(v)\}$$

[22] Program tc standardně spočítá $burst = \frac{ar}{HZ} + MTU$, aby byla ar opravdu zaručená.

Plánovač

Třída se spodními 16 bity nulovými je označována jako *direct* a její provoz má přednost přede všemi ostatními. Jakmile je v zeleném uzlu provoz připravený k odbavení, uzel se napojí na tzv. *globální seznam*, což je ve skutečnosti množina seznamů – pro každou kombinaci $(level, prio)$ jeden (`htb_prio::row`) – sloužící k rychlému nalezení čekajícího provozu v celém stromě. Zežloutne-li list, přepojí se na tzv. *interní seznam* svého rodiče, od kterého si půjčí. Jeho rodič se potom zachová stejně – přidá se na globální nebo prarodičův interní seznam podle svého stavu. Interní seznamy mají právě vnitřní uzly pro každou prioritu (`htb_class::un.inner.clprio[p].feed`).

Červené a žluté uzly jsou zařazeny do prioritní *čekací fronty* (značené *wait tree*, *event queue* nebo `wait_pq`) seřazené podle času nejbližší změny barvy; každá úroveň má vlastní frontu. Tímto algoritmus okamžitě pozná uzly čerstvě připravené k odesílání, které se vrátí na příslušný seznam podle své nové barvy. Speciálně připomeňme, že červené uzly pouze čekají, žluté čekají a jsou na interním seznamu svého rodiče a zelené jsou na globálním seznamu právě tehdy, čeká-li v nich (nebo v jejich následnících) nějaký provoz.

Příklady

```
# tc qdisc add dev enp3s0 root handle F: htb default 0xDEFA
# tc class add dev enp3s0 parent F: classid F:1 htb rate 5↙
→Mbit ceil 7Mbit
# tc class add dev enp3s0 parent F: classid F:2 htb rate 5↙
→Mbit ceil 7Mbit
# tc class add dev enp3s0 parent F:2 classid F:2A htb rate 1↙
→Mbit
```

2.5 Méně časté situace

Disciplíny a jejich třídy se „vyhodnocují“ při odchodu provozu z příslušného rozhraní. To přináší dvě otázky – lze takto omezovat i příchozí traffic? Co když budeme chtít počítat přidělenou kapacitu na více rozhraních zároveň?

2.5.1 Disciplína `ingress`

Již několikrát bylo zmíněno, že ačkoli v době příchodu paketu principielně nedává smysl provádět jakákoli omezení (prostředky již byly spotřebovány), v praxi se to může hodit. Přidáním disciplíny `ingress` na rozhraní nenahradíme tu stávající, ale přidáme novou, kterou bude „protékat“ příchozí provoz. Přestože je classless, můžeme na ni nastavit filtry; kromě *policeru* na šifrovaný provoz na slabém stroji bývá často `ingress` používán k přesměrování dat na jiné rozhraní. Více v kapitole 4.2.2, která ukazuje časté použití této disciplíny k dosažení stejného efektu jako u odchozího provozu. Disciplína nastavuje výsledek klasifikace do proměnné `skb->tc_index`, což se bude hodit například v kapitole 4.2.7.

2.5.2 Intermediate Functional Block (IFB)

Linux má speciální rozhraní, `dummy`, které podobně jako `vether(4)` v OpenBSD všechny pakety pouze zahazuje. Toto rozhraní se hodí zejména jako cíl přesměrování, který se dá sledovat například pomocí *Berkeley Packet Filter* (BPF) knihovnou `pcap(3)`. O něco „chytřejší“ je zařízení *Intermediate Functional Block* (IFB): pošleme-li tudy paket, zavolá se na něj znovu `netif_receive_skb` a data tak proudí dále systémem. Jde na něj nastavit libovolná disciplína jako na kterýkoli jiný a pomocí tc do něj můžeme přesměrovat příchozí provoz[23]. Takováto virtuální rozhraní je tedy možné použít ke sdílení konfigurace mezi více fyzickými.

2.6 Ethernet Flow Control (IEEE 802.3x)

Jakýkoli přechod mezi technologiemi s různou přenosovou rychlostí, přestože disponují rozhraním uvádějícím rychlost daleko větší[24], nutně znamená, že při saturaci rychlejší části musí ta pomalejší nějaká data zahodit, aby se rychlost přizpůsobila úzkému hrdlu. Totéž platí, teče-li jedním portem switche provoz tolika ostatních, že jej přetíží; této situaci se říká *head-of-line blocking*. V takových případech je možné explicitně oznámit zahlcení již na Ethernetu a nastavit rychlejší stranu tak, aby tato oznámení akceptovala.[25] Zahlcená strana pošle tzv. *pause frame* obsahující časové kvantum, po které nechce být tímto portem rušena. Použijeme-li však toto na portu sloužícím jako tranzitní, zdržíme nejen toky způsobující zahlcení, ale i ty neškodné, potenciálně vyžadující interaktivitu. Zároveň tato technika pomalu zvyšuje v tranzitním uzlu délku fronty (Jde o mechanizmus pouze na linkové vrstvě!) způsobem podporujícím globální synchronizaci. V těchto situacích je velmi vhodné Flow Control vypnout.

Operační systémy umějí nastavit zvlášť posílání a akceptování těchto rámců, v Linuxu se pomocí `ethtool -A enp3s0`, v OpenBSD pomocí `ifconfig em0 txpause,-rxpause`.

Objevuje se rozšíření *Priority Flow Control* (PFC) omezující jen vybrané třídy provozu. [IEEE802.1]

2.7 VLAN (IEEE 802.1Q)

Je-li v síti nasazena technologie VLAN, čtyřbajtová hlavička obsahuje kromě jiného i 3 bity specifikující tzv. *Class of Service* (CoS). VoIP telefony často umějí posílat telefonní provoz takto otagovaný vyšší prioritou a provoz zařízení připojených za telefonem prochází bez tagování. V OpenBSD tyto tři bity korespondují s hodnotou `set prio` v pf (vizte `vlan(4)`), v Linuxu se dají nastavovat programem `vconfig(8)`. Díky jednoduchosti tohoto mechanizmu jej podporují téměř všechny spravovatelné switche.

[23] Původem IFB pochází z nezařazeného *Intermediate Queueing Device* (IMQ), do kterého putoval provoz přes speciální Netfilter target.

[24] Například bezdrátový spoj omezený zakoupenou licencí na využívání VF pásma

[25] Koncept explicitního oznamování zahlcení vidíme různě implementovaný na všech vyšších vrstvách.

3. Síťová vrstva

Na hranici linky jde o její kapacitu. Na hranici transportní vrstvy jde o efektivitu odesílání dat z aplikace do nijak nespecifikovaného prostředí při zajištění spolehlivosti a ohleduplnosti k ostatním. Třetí vrstva modelu ISO/OSI však rozděluje síť pouze logicky a jako taková s sebou nenese žádnou oblast pro optimalizaci[1].

Přenášíme zde však informace o QoS mezi vrstvami okolními. V minulosti existoval pokus o explicitní žádosti na přidělení prostředků, konkrétně architektura *Integrated Services* (IntServ) s *Resource Reservation Protocol* (RSVP), ale pro svoji složitost, špatnou podporu a škálovatelnost se téměř nikde neujala.

3.1 Klasifikace paketů pro další uzly

Druhý bajt v hlavičce IPv4 je znám pod několika názvy. Původně se jmenoval *Type of Service* (ToS) a byl rozdělen na dvě části: *precedence* tvořila první tři bity a měla indikovat prioritu daného paketu; zbylé tři[2] bity určovaly, jakou kvalitu služeb by si odesilatel daného paketu představoval. Socketové API umožňovalo aplikacím si tuto hodnotu nastavovat pomocí `setsockopt(2) IP_TOS` popsaného v Linuxových systémech v manuálové stránce `ip(7)`, v OpenBSD `ip(4)`.

Novější architektura *Differentiated Services* (DiffServ) [RFC2474] naposledy změnila význam celého bajtu označením prvních šesti bitů jako *Differentiated Services Code Point* (DSCP) a ponecháním zbylých dvou nevyužitých (*currently unused*, `CU`). DiffServ rozděluje služby do tříd nikoli podle požadavků konkrétního uživatele, ale podle charakteru služby samotné. Na rozdíl od ToS se nesnaží třídy předem definovat, ale nechává místo správci lokální sítě (*DS domain*), který podle konkrétních nároků sestaví požadavky na příslušné DSCP a jejich chování (*per-hop behavior*) nastaví na všech aktivních prvcích. Principiálně tedy neexistuje jednotné nastavení v celém internetu, ale v rámci zachování kompatibility jsou na některé hodnoty kladené speciální nároky. DSCP = 0 je vyhrazené pro výchozí provoz a musí být konfigurovatelné odděleně od všech ostatních tříd; *Expedited Forwarding* (EF) s hodnotou `101110|CU`, někdy též 46 či `0xB8` (pozor!), slouží k provozu citlivému na dobu odezvy či její výkyvy (*jitter*) a nastavují ji typicky stolní VoIP telefony. Dosud bylo publikováno několik doporučení, jak s tímto prostorem hodnot nakládat, včetně částečného shrnutí. [RFC4594]

[1] Výjimku tvoří efektivita a velikost routovací tabulky, což skvěle řeší například komprimovaná hashovací trie od Henryho Tzhenga [RTblTrie] schopná pojmout stovky tisíc IPv4 rout do cache moderního procesoru a najít cestu v malých jednotkách přístupů.

[2] Poslední dva bity měly význam dokonce třikrát změněn. [RFC3168, sekce 22]

minimize delay	`000|1000|0`	`0x10`
maximize throughput	`000|0100|0`	`0x08`
maximize reliability	`000|0010|0`	`0x04`
minimize monetary cost	`000|0001|0`	`0x02`

Tabulka 3.1: Často používané hodnoty ToS.

drop-pref	Class 1	Class 2	Class 3	Class 4
low (1)	`001|010` (10)	`010|010` (18)	`011|010` (26)	`100|010` (34)
medium (2)	`001|100` (12)	`010|100` (20)	`011|100` (28)	`100|100` (36)
high (3)	`001|110` (14)	`010|110` (22)	`011|110` (30)	`100|110` (38)

Tabulka 3.2: DSCP hodnoty tříd *Assured Forwarding* (AF) binárně a dekadicky; AFij je v i-tém sloupci, j-tém řádku.

3.1.1 CS: Kompatibilita s *IP Precedence*

Starší vybavení používající první tři bity ve významu *IP Precedence*, tedy oktet tvaru `xxx000|CU`, mají část kompatibility zachované pomocí kódů *Class Selector* (CS). Bity `xxx` nastavené na 110 či 111 u prioritního provozu údajně byly často používané routovacími protokoly, pročež je vyžadováno, aby takovéto pakety měly vyšší prioritu než standardní.

3.1.2 AF: Třídy a zahazování v rámci nich

Část kódů se jmenuje *Assured Forwarding* (AF) a definuje třídy provozu od jedné do čtyř zapsané ve vyšších třech bitech DSCP; vyšší číslo zapsané znamená vyšší prioritu. V rámci *každé* třídy může aplikace v nižších třech bitech definovat, jak moc je který *paket* důležitý, pomocí *drop precedence*: protože šestý bit nastavený na 1 znamenal „definováno lokálním uživatelem“ a hodnota `000` je obsazená, spodní tři bity doporučené pro AF vypadají takto:

010 znamená *low drop precedence*, čili nejdůležitější paket

100 znamená *medium drop precedence*

110 znamená *high drop precedence*, čili nejméně důležitý paket

Názvy konkrétních hodnot zde kromě dekadického zápisu šesti bitů a nejednoznačného hexadecimálního používají ještě jeden formát: AFij, kde i je číslo třídy a j je $drop\_precedence >> 1$; vizte tabulku 3.2. Provoz v rámci jedné třídy je zakázáno přeuspořádávat.

3.1.3 Disciplína `dsmark`

Routery na hranici mezi několika administrativními (DS) doménami někdy potřebují přemapovat jednu sadu DSCP na jinou, například podle nasmlouvaných garancí na různé třídy provozu od jiného poskytovatele. V Linuxu má takovýto přechod usnadnit classful disciplína `dsmark`, která tvoří tabulku o `indices` řádcích. Disciplína provádí následující kroky začínajíc ve funkci `dsmark_enqueue`:

Nastavíme-li disciplíně `set_tc_index`, DSCP (ToS se zamaskovanými ECN bity, více v kapitole 3.2) se nastaví do `skb->tc_index`.

Je-li horních 16 bitů `skb->priority` stejných s `handle` aktuální disciplíny, do `tc_index` se uloží *spodních* 16 bitů a pokračuje se další disciplínou v pořadí. Takto lze explicitně obejít klasifikaci.

Jinak se provádí klasifikace (typicky filtry `tcindex` popsané v kapitole 4.2.7) a při jejím úspěchu se spodních 16 bitů z jejího výsledku uloží do `tc_index`. Při neúspěchu se uloží, byl-li definován, parametr `default_index`.

Existuje tedy právě jeden průchod disciplínou, kdy `tc_index` zůstane.

Přestože je `tc class show` neukáže, disciplína už od vytvoření obsahuje třídy s označením 0 až `indices`−1. Funkce `dsmark_dequeue` zařadí provoz do třídy `tc_index & (indices-1)`. To mimo jiné znamená, že jediné důvody nastavovat `indices` na jinou hodnotu než 256 jsou, chceme-li používat starší význam bajtu ToS, případně mít disciplínu rozlišující pouze AF *drop precedence*.

Třída s parametry `mask` a `value` říká, které bity původního DS bajtu se mají zachovat, resp. nastavit. Příklad použití najdeme v kapitole 4.2.7.

3.1.4 Identifikace jednotlivých toků

Protokol IPv6 nabízí ještě jednu položku – dvacetibitové pole *flow* sloužící jako pomoc pro klasifikátory při identifikaci příslušnosti toku do nějaké třídy. V době psaní práce bylo aktuální RFC 6437; spíše než jako doporučení pro volbu hodnot těchto polí (náhodná čísla dobře padající do hashovacích funkcí routerů po cestě) řeší bezpečnostní rizika (skrytý komunikační kanál, přetížení hashovací funkce při útocích, autenticita) a nedefinuje, má-li aplikace nějakým konkrétním způsobem bránit kolizím mezi ostatními, má-li kernel proaktivně vymýšlet kombinace i když o to aplikace nepožádala, nebo může-li například webový browser značit více spojení jako jeden *flow*. Uživatel může pomocí `setsockopt(2) IPV6_FLOWINFO` tuto hodnotu ovlivnit; více v `ipv6(7)` v Linuxu, resp. `ip6(4)` na BSD.

3.2 Explicit Congestion Notification (ECN)

Celý svět si rychle zvykl na fakt, že protokoly přenášející proud dat berou ztrátu paketu jako signál ke zpomalení. Koncem roku 1999 začalo IETF vymýšlet dobrovolný mechanizmus [RFC3168] využívající některé dosud rezervované položky hlaviček k explicitní notifikaci koncových uzlů o vzniklém ucpání části sítě. Jakmile prvek po cestě změří ucpávající se fronty, například uvnitř AQM, může nastavit *Congestion Experienced* (CE) doufaje, že druhá strana zareaguje.

Protokol IPv4 k tomuto účelu využívá poslední dva bity druhého bajtu, které mají následující význam:

00 značí, že toto spojení *Explicit Congestion Notification* (ECN) nepoužívá. Výhradně z důvodu kompatibility se starším vybavením.

11 značí *Congestion Experienced*.

01 nebo 10 značí, že tento tok ECN podporuje, ale ucpání zatím nenastalo. Dvě hodnoty jsou zvolené kvůli potenciálním starším implementacím ToS, které by jeden bit mohly omylem poškodit.

IPv6 má nezarovnaných 8 bitů označených *Traffic Class*, majících stejný význam jako DSCP a ECN.

4. Klasifikace

K závěru se dostáváme do fáze, která je v Linuxu složitější, než si leckdo dokáže představit. Je to také důvod, proč jsme dosud nemohli naše nastavení použít. Obecně vzato může klasifikace rozhodovat na základě čehokoli, což ji neřadí ani na jednu síťovou vrstvu – dostala tedy vlastní kapitolu.

4.1 *Packet filter* v OpenBSD

Umístění klasifikátoru je v OpenBSD jiné než v Linuxu. Jako klasifikátor slouží pf, jež není třeba zmiňovat, neboť se probírá v předmětu Administrace UNIXu a je výborně dokumentovaný v `pf.conf(5)`. Pravidla obsahující `set queue` nastaví každému matchlému mbufu do pole `qid` číslo fronty, které se následně použije na odchozím rozhraní. Není-li tam taková fronta definovaná, použije se výchozí. Nastavit klasifikátor je tedy triviální.

4.2 Filtry *traffic control* a pomocné moduly

Ke klasifikaci se v Linuxu výhradně používají filtry nastavené utilitou tc. Některé (jednodušší) umějí použít i informace z jiných subsystémů[1] a jiné lze nastavit na jednodušší politiku a nevyužité subsystémy potom vyřadit z provozu.

Uveďme pro ilustraci nejjednodušší použití, a to přenechání klasifikace na firewallu Netfilter s mapováním jeho značek na naše identifikátory tříd a disciplín:

```
# tc qdisc add dev enp19s0 parent F:AA handle F:80AA fq_codel

# iptables -t mangle -A FORWARD -s 10.0.0.1 -j MARK --set-↙
→mark 0xDEDA
# tc filter add dev enp19s0 parent F: handle 0xDEDA fw ↙
→classid F:80AA
```

Vytvořili jsme potomka uzlu `F:AA` s disciplínou `fq_codel` a do něj přesměrovali pakety, kterým Netfilter přidělil značku `0xDEDA`. Ukázka `iptables` potom tuto značku nastaví procházejícím paketům se zdrojovou adresou 10.0.0.1. Obě 16bitové konstanty jsou pochopitelně různé pouze z didaktických důvodů; `0xDEDA` se objeví v `skb->mark` a `80AA` je explicně zvolený název disciplíny, aby nedocházelo k nejasnostem při opakovaném načítání konfigurace. Filtr je nastavený na uzlu `F:`, do kterého se pochopitelně musí paket nejdříve dostat; vhodné je mít filtry co nejvýše ve stromu a pokud možno na jednom místě.

Neexistence dokumentace bohužel znamená, že mohu jen ztěží odhadnout důležitost jednotlivých částí pro typického uživatele (Kdo je to vlastně *typický uživatel*?) a proto se zde pokusím celý framework popsat „zevnitř“ s referencemi do kódu jádra i utility tc. Znát názvy funkcí pochopitelně není důležité, ale žádný mně známý dokument ([LARTC, TLDP-tc] či `README.iproute2+tc` z balíku

[1] Například filtr `fw` používá značky subsystému Netfilter a hodí se tak na router provádějící NAT, neboť ke klasifikaci dochází na odchodu z rozhraní, kde už je původní adresa přepsaná.

iproute2 [iproute2]) nepopisuje fungování filtrů dostatečně pro úroveň bakalářské práce, což se zde pokusím napravit možná příliš složitým, ale přesným způsobem.

Každý qdisc či třída s sebou může mít nastavený řetěz filtrů (`tcf_chain`), což jsou pravidla ne nepodobná pravidlům firewallu; každé pravidlo obsahuje *selektor* („pro jaký provoz platí“) a *akci* („co se s ním stane“).[2] Řetěz filtrů je ve skutečnosti spojový seznam struktur `tcf_proto` seřazený podle parametru `preference`, který uvidíme v konfiguraci. Nový filtr zařadí na správné místo funkce `tc_ctl_tfilter`, která též zaručí, že každá úroveň preference obsluhuje nejvýše jeden *protokol*[3]. Každá úroveň preference používá jeden *typ* filtru.

Přestože mají disciplíny při *enqueue* vlastní klasifikační funkci, většina nakonec stejně volá `tc_classify`, která prochází řetěz filtrů od nejvyšší (numericky opět nejnižší) preference a volá každý přiřazený klasifikátor. Nespecifikujeme-li preferenci, systém přiřazuje nižší dostupná čísla počínaje 49152 (`0xC000`) nebo vyšším použitým; nezadaný protokol odpovídá `protocol all`[4].

Syntaxi jednotlivých typů filtrů po zadání `tc filter add <TYP> help` program hrubě nastíní, ale přesný popis chybí. Pro usnadnění orientace v textu bude na začátku každé sekce seznam klíčových parametrů, které popisuje.

4.2.1 Základní syntaxe

Parametry: `dev`, `estimator`, `parent`, `preference`, `priority`, `protocol`

```
# tc filter show dev <ROZHRANI>
# tc filter {add|change|replace|delete} \
  dev <ROZHRANI> parent <RODICOVSKY-QDISC-NEBO-CLASS> \
  [protocol <PROTO>] [{priority,preference} <PREFERENCE>] \
  [estimator <INTERVAL> <KONSTANTA>] \
  <TYP-FILTRU> <PARAMETRY-FILTRU>
```

Nutnost specifikovat rozhraní je stejná jako u `tc class`. Manuálová stránka `tc(8)` popisuje možnost místo rodiče uvést `root`, ale nenašel jsem ani funkční příklad, ani smysl takovéto volby. Klíčová slova `priority` a `preference` jsou si funkčně ekvivalentní. Všimněme si, že zde není parametr `handle` jako jednoznačný identifikátor filtru; tím je jeho umístění ve stromě, protokol a preference. `handle` má pouze lokální význam, v rámci každého typu filtru jiný.

4.2.2 Filtr `basic`, akce a *ematch*

Parametry: `action`, `classid`, `flowid`, `match`, `police`.

Překvapivě, filtr s názvem `basic` nemá dokumentaci žádnou. Zaslouží si zmínku kvůli vlastnostem společným s ostatními typy, konkrétně podpoře *akcí* a *ematchů*

[2] Pro srovnání, systém Windows Filtering Platform akcím volaným z filtrů říká *callouts* a implementuje jimi i funkce v unixových systémech řešené samostatně, například IPsec či TCP offloading; více na `http://msdn.microsoft.com/en-us/library/ff571066.aspx`.

[3] To je další parametr předávaný z uživatelského prostoru; umí filtrovat jen konkrétní `skb->protocol`. Ve výchozím stavu použitá konstanta `ETH_P_ALL` filtr zkusí použít vždy.

[4] Během testování jsem zjistil, že CentOS verze 6.5 nemá správně implementované `protocol all`, vypisuje `protocol [768]` a takové filtry nefungují.

(z angl. *extended match*, vizte `tc-ematch(8)`). Klíčová slova `classid` a `flowid` jsou si funkčně ekvivalentní.

```
# tc filter add dev enp19s0 parent F: basic classid F:4444
```

Tento příklad téměř nic nedělá, pouze všechno zařazuje do třídy `F:4444`.

Akce jsou v balíku iproute2 staticky nebo dynamicky linkované moduly (cesta bývá `/lib/tc/m_*.so`, nastavitelná proměnnou prostředí `TC_LIB_DIR`), aby se na jejich vývoji mohlo snáze podílet více vývojářů. Každá akce zde má vlastní funkci parsující parametry z příkazové řádky (`action_util::parse_aopt`), které do Netlinkové zprávy[5] zabalí a odešle do kernelu. Přesnou syntax vybraných akcí popíšeme níže. Za zmínku též stojí adresář `doc/actions` [iproute2] s popisem některých částí přímo od jejich autorů.

V jádře existuje od roku 2004 *tc filter extension API* (struktura `tcf_exts`), které má za úkol oddělit funkcionalitu akce od kódu selektoru filtru a tím umožnit psát akce univerzálně. [tcf-exts] Potom se akce ujme *packet action API* (struktura `tc_action`) ze souboru `net/sched/act_api.c`, kde se inicializuje pomocí `tc_action_ops::init`.

Utilitou tc lze všechny používané akce daného typu vypsat a konkrétní akci smazat: (Z mně nepochopitelných důvodů se parametr `x` nikde nepoužívá.)

```
# tc actions list x <TYP-AKCE>
# tc actions delete x <TYP-AKCE> index <I>
```

`<I>` danou akci identifikuje *v rámci svého typu*. Přidat či změnit akci umí příkaz `tc actions change <TYP-AKCE> index <I> <PARAMETRY-AKCE>`, zadáme-li index v desítkové soustavě (přestože se vypisuje v šestnáctkové). Smysl explicitního uvádění parametru `index` spočívá v možnosti přiřadit jednu akci více filtrům, dokonce i na více rozhraních zároveň. Vizte druhý řádek příkladu níže.

Daleko příjemnější je specifikovat akce na stejném řádku jako filtry, přičemž většina akcí má po zadání parametru `help` aspoň náznaky příkladů použití.

```
# tc filter add dev enp19s0 parent F: basic classid F:4444 ↙
→action <TYP-AKCE> <PARAMETRY-AKCE>
# tc filter add dev enp4s0 parent F: basic action <TYP-AKCE> ↙
→index <I>
```

Generické akce (`gact`)

Nejjednodušší akce, zahodit paket či pokračovat v klasifikaci jiným filtrem, umí modul `act_gact`. Program tc nabízí tyto akce bez nutnosti explicitně psát `gact`. Zadáme-li jako typ akce `drop`, modul paket zahodí; `pass` jej odešle, `continue` pokračuje, jako by tento řádek filtru neexistoval, a `reclassify` spustí uvnitř `tc_classify` (tedy až *za* klasifikátorem specifickým pro disciplínu) znovu celou klasifikaci. Zajímavá akce je `pipe`. Sama o sobě není užitečná, ale jelikož některé akce mají nastavitelnou generickou akci podmíněnou výsledkem jejich kódu (např. *policer*), můžeme takto nevyhovující provoz poslat „dál“ v seznamu akcí na daném filtru, aniž bychom museli znovu procházet klasifikací. Akce `continue` tedy pokračuje dalším filtrem, zatímco `pipe` pokračuje další akcí na *tomto* filtru.

[5] Zprávy `RTM_{NEW,GET,DEL}ACTION` mají dnes už prázdné TLV `TCA_ACT_OPTIONS`, za kterým má akce svoje data; přesný formát není zdokumentován.

```
# tc filter add dev enp19s0 parent F: basic action drop
```

Spustit Netfilter *target* (ipt, xt)

Jádro jednoduše zavolá `xt_target::target` na právě klasifikovaný `skb`. Podle výsledku subsystému Netfilter paket zahodí, povolí nebo nechá dále klasifikovat. Je vhodné zde zmínit dva výchozí Netfilter cíle sloužící ke klasifikaci: `CLASSIFY` resp. `MARK` nastavují v `skb` pole `priority` resp. `mark`, které lze z uživatelského prostoru nastavovat pomocí `setsockopt(2)` `SO_PRIORITY` resp. `SO_MARK`. Zatímco `MARK` lze použít odkudkoli, `CLASSIFY` pouze v chainech `OUTPUT`, `FORWARD` a `POSTROUTING`, protože priorita se přiřazuje automaticky funkcí `rt_tos2priority` a došlo by tak k přepsání nastavené hodnoty.

Tato akce může tedy sloužit jako „druhá strana“ pro filtr `fw`:

```
# tc filter add dev enp19s0 parent F: basic action xt -j MARK↙
→ --set-mark 0xAB
```

Mirroring a přesměrování na jiné rozhraní (mirred)

Potřebujeme-li paket těsně před odchodem z daného rozhraní přesměrovat či zkopírovat na jiné, modul `act_mirred` na daný `skb` zavolá `dev_queue_xmit` s novým odchozím rozhraním.

```
# tc action add mirred help
# tc filter add dev enp19s0 parent F: basic action mirred <↙
→SMER> <AKCE> dev <ROZHRANI>
```

Často uváděný příklad použití varianty `redirect` je přesměrování příchozího provozu z `ingress` disciplíny na rozhraní IFB, kde můžeme provádět jeho shaping stejně jako na odchozích rozhraních.

```
# tc qdisc add dev enp19s0 ingress
# tc filter add dev enp19s0 parent FFFF: basic action mirred ↙
→egress redirect dev ifb0
# tc qdisc add dev ifb0 root hfsc
```

Při testování na živém provozu doporučuji vyměnit `redirect` za `mirror` a používat `tcpdump(8)` na `ifb0`. Směr `ingress` není utilitou tc podporován. Následující příklad spolehlivě „zamrzne“ stroj při prvním paketu odcházejícím z rozhraní `enp19s0`. Podle manuálu v `doc/actions/mirred-usage` to není chyba.

```
# tc filter add dev enp19s0 parent F: basic action mirred ↙
→egress mirror dev enp19s0
```

Bezstavový překlad adres (nat)

Je-li první parametr `egress` resp. `ingress`, rozhoduje se podle zdrojové resp. cílové adresy; nemá tedy nic společného s umístěním filtru v rámci průchodu systémem. Je-li to adresa z druhého parametru (`OLD`), přepíše se na třetí (`NEW`);

zadáme-li `OLD` jako rozsah adres, spodní bity nové adresy zůstanou zachované (příklad přepíše 10.0.0.33 na 192.168.144.33). Modul podporuje pouze IPv4.

```
# tc action add nat help
# tc filter add dev enp19s0 parent F: basic action nat egress↙
→ 10.0.0.0/24 192.168.144.0
```

Délka prefixu se v Netlinkové zprávě předává jako bitová maska a při výpisu a převodu zpět na délku nedochází k ošetření endianity ([iproute2, `print_nat`])

Úprava paketů (`pedit`) a bufferů (`skbedit`)

Tyto moduly mají relativně slušně popsanou nápovědu; `pedit` z příkazové řádky rozparsuje offset od začátku síťové vrstvy (i záporný), velikost měněného pole (`u8|u16|u32`) a požadovanou změnu (`clear|invert|set <HODNOTA>|retain <MASKA>`), což dohromady tvoří *klíč* (struktura `tc_pedit_key` v jádře) a na každý paket tato akce aplikuje postupně všechny klíče. Podobně jako správa filtrů v iproute2 dynamicky linkuje některé filtry ze souborů `m_*.so`, tento modul má soubory `p_*.so` obsahující syntaktický cukr na tvorbu klíčů. Většina distribucí je ale ve výchozím stavu neinstaluje.

Nepovinná část parametrů začínající `at` není vypisovaná, takže ji není možné vizualizovat bez zásahu do klientského software. Parser parametrů také bere pouze vstup bez klíčových slov `offmask` a `shift`. Z těchto důvodů jsem se rozhodl nepovinnou část vynechat.

Akce `skbedit` umožňuje kromě editace již zmíněných `mark` a `priority` také mapování paketů do hardwarových front, jsou-li na rozhraní přítomné. Tyto akce nelze z pochopitelných důvodů používat na více rozhraních zároveň. Akce `csum` umožňuje nechat jádro přepočítat kontrolní součty vybraných hlaviček.

```
# tc action add csum help
# tc action add pedit help
# tc action add skbedit help
# tc filter add dev enp19s0 parent F: basic action pedit ↙
→munge offset 12 u32 invert munge offset -8 u16 set 0x4141 ↙
→csum ip4h and tcp or udp
# tc filter add dev enp19s0 parent F: basic action skbedit ↙
→queue_mapping 1
```

Policing (`police`)

Toto byla v minulosti jediná možná akce, o čemž svědčí stále funkční stará syntaxe filtru `basic` s vynechaným klíčovým slovem `action`. Nápověda filtru `basic` tuto možnost stále nabízí, ale uvnitř jádra se obě varianty převedou na stejný kód vedoucí do `tcf_exts` API. Existuje též popis její funkcionality [LARTC, sekce 12.3] ukazující na dvě možnosti implementace policeru. Parametrem `avrate` šlo implementovat policer pomocí periodických časovačů z `net/core/gen_estimator.c` (horní podmínka v `tcf_act_police` [LiSrc, `act_police.c`]), nicméně nyní jádro vrátí `EINVAL` a poslední zprávy u commitů píší o blíže nespecifikovaném „zachování zpětné kompatibility se staršími verzemi iproute2“. Pomocí děravého kbelíku parametry `rate` a `burst` policer nastavit lze. Akce můžeme této akci

nastavit dvě; první je standardně `reclassify` pro nadlimitní provoz, často nahrazována dalšími akcemi v řetězu (přesměrování, jiná třída, ...). Druhá akce – „výsledek“ – je standardně `pass` či `ok` a vykoná se pro provoz policeru *vyhovující*. Parser tyto dvě akce oddělí lomítkem (jsou pouze `gact`, takže složitější operace je potřeba řešit pomocí `pipe`) a jsou součástí parametru `conform-exceed`. Vizte přemarkování nadlimitního provozu a jeho následné přesměrování ven jiným rozhraním.

```
# tc action add police help
# tc filter add dev enp19s0 parent A: basic classid A:EEEE ↙
→action police rate 25Mbit burst 4k conform-exceed drop/pass
# tc filter add dev enp19s0 parent F: basic classid F:600D ↙
→action police rate 3Mbit burst 1k conform-exceed pipe/pass ↙
→skbedit mark 0xBAD mirred egress redirect dev enp11s0
```

4.2.3 Extended matches

Některé starší klasifikátory (*incoming device*, `fw`, `tcindex`) se vývojáři snaží nahradit obecným mechanizmem *ematch*. Jde o „malé klasifikační nástroje, které je zbytečné psát jako samostatné klasifikátory“ [LiSrc, `net/sched/ematch.c`] a lze z nich skládat logické výrazy se spojkami AND a OR. Výsledný strom tedy má očíslované (`matchid`) uzly tvořené jednotlivými „voláními“ matchovacích funkcí s parametry (struktura `tcf_ematch`), linearizovaný programem tc a zvalidovaný jádrem: `tcf_em_tree_validate` volá pro každý uzel s implementovanou nějakou formou validace jeho `tcf_ematch::ops::change`. Hlavní výhodou tohoto mechanizmu je, že je v `tc-ematch(8)` relativně dobře zdokumentován. Představme si některé typy matchovacích funkcí:

cmp slouží k porovnávání dat v paketech s konstantami; `trans` převede endianitu dle aktuální architektury a `mask` zamaskuje čtená data před porovnáním

```
# tc filter add [...] basic match 'cmp(u32 at 16 trans ↙
→gt 42) and not cmp(u8 at 11 mask 0x7F eq 0x0A)' [...]
```

ipset testuje proti obsahu daného `ipsetu`, což je zhruba ekvivalent tabulky ve firewallu pf Čtení takovéhoto nastavení vyžaduje práva superuživatele, ačkoli to tc nedá najevo jinak než výpisem naprosto nečekaným způsobem deformovaného výstupu tohoto i následujících *ematchů*.

meta má v jádře spoustu funkcí s „metadaty“, které můžeme při klasifikaci využít. Jde například o *load average*, `mark` systému Netfilter nebo velikosti bufferů. Seznam aktuálně podporovaných vrátí příkaz:

```
# tc filter add basic match 'meta(list)'
```

nbyte porovnává řetězce od zadaného offsetu; přestože nápověda naznačuje možnost *c-escape-sequence*, nenašel jsem kód zpracovávající první parametr

u32 by měl být funkčně ekvivalentní staršímu `u32` filtru; zde jej však můžeme kombinovat s ostatními typy *ematchů*. Funkčně shodný s ematchem `cmp`.

4.2.4 Filtr `flow`

Parametry: `baseclass`, `divisor`, `hash`, `key`, `keys`, `map`, `perturb`.

Podle zprávy v commitu[6] `e5dfb815` od Patricka McHardyho [LiSrc] slouží tento filtr ke zpřesnění hashovací funkce uvnitř *Stochastic Fairness Queueing* (SFQ), což může pomoct spravedlivému oddělování jednotlivých datových toků. Filtr má dva režimy: `hash` umí specifikovat klíče k rozdělování do front pomocí (parametrem `perturb` v sekundách potenciálně periodicky randomizované) Jenkinsovy hashovací funkce a režim `map` umí přesměrovat provoz do fronty na základě konkrétní hodnoty některého z klíčů (například posledního bajtu IP adresy).

Režim `map` umožňuje mapovací klíč kombinovat pomocí jednoduchých operací s konstantami; ve třetím příkladu se od klíče `dst` odečítá konstanta `0xC0A80000` zapsaná jako IPv4 adresa, čímž zůstane jen posledních 16 bitů.

```
# tc filter add flow help
# tc filter add dev enp11s0 protocol ip pref 42 parent F: ↙
→handle 0x2 flow hash keys src,dst,proto,proto-src,proto-dst↙
→ divisor 1024
# tc filter add dev enp11s0 protocol ip pref 42 parent F: ↙
→handle 0xA flow map key dst addend -192.168.0.0 divisor 256
```

Výsledná třída bude `baseclass` + výsledek % `divisor` (pokud není roven 0).

4.2.5 Filtr `fw`

Parametry: `classid`, `handle`, `indev`.

Nejjednodušší použití filtru `fw` bylo na začátku kapitoly: mapování značky systému Netfilter na konkrétní třídu či disciplínu. Za zmínku zde stojí role parametru `handle` – je povinný (jako například u filtru `flow`), je třeba jej uvést ještě před názvem filtru (v Netlinkové zprávě `tcmsg` to není TLV ale součást společné hlavičky) a namísto pouhé identifikace pravidla zde slouží jako očekávaná hodnota `skb->mark`.

Lze jej zapsat ve tvaru `0xDEAD/0xBEEF`, kde druhé číslo slouží jako bitová maska značky klasifikovaného paketu. Parametrem `indev` lze rozpoznat pakety přijaté na daném rozhraní (filtr nepodporuje *ematche*).

```
# tc filter add dev enp1s0f1 parent F: handle 0xDEDA/0xFEED ↙
→fw classid 0x80AA indev enp4s6
```

4.2.6 Filtr `route`

Parametry: `from`, `fromif`, `to`.

Je-li možné rozhodnout o zařazení do tříd během vyhledávání ve směrovací tabulce, tento modul vytvoří filtr takovéto zařazení rozpoznávající. Každý záznam v routovací tabulce má pole *realm* standardně nastavené na 0; příkazem

[6] Což je asi nejvhodnější zdroj příkladů, neboť dokumentace opět chybí.

`ip route ... realm <N>` jej u dané cesty nastavíte, kde ji následně najdou parametry `from` nebo `to`. Filtr je vhodné použít, máme-li několik cest do dané sítě a chcete mít pro každou cestu jiná pravidla.

```
# tc filter add route help
```

4.2.7 Filtr `tcindex`

V kapitole 3.1.3 jsme si ukázali, jak disciplína `dsmark` umí před přepsáním bajtu DS uložit jeho původní hodnotu do proměnné `skb->tc_index`. Tento filtr podle ní klasifikuje: nejdříve jej zamaskuje svým parametrem `mask`[7] a výsledek posune doprava o `shift` bitů. Takto vzniklé číslo najde jako spodních 16 bitů třídy a do té paket zařadí. Filtr má parametr `fall_through`, který v případě neexistence dané třídy použije další filtr v pořadí. Výchozí `pass_on` tedy z filtru vrátí „neúspěch“ (a `dsmark` pak použije `default_index`, je-li uveden).

Následující ukázka extrahuje DSCP posunutím uloženého `tc_index` o dva ECN bity, zachovává EF a ToS *lowdelay* pakety, přečísluje DSCP AF3x na ToS *lowdelay* a pro ostatní provoz tento bajt vynuluje.

```
# tc qdisc add dev ifb0 root handle F: dsmark indices 256 ↙
→set_tc_index default_index 0xFF
# tc filter add dev ifb0 parent F: protocol ip pref 7 tcindex↙
→ mask 0xFC shift 2 pass_on

# export CLASSIFY="tc filter add dev ifb0 protocol ip pref 7"
# ${CLASSIFY} parent F: handle 4 tcindex classid F:10
# ${CLASSIFY} parent F: handle 26 tcindex classid F:10
# ${CLASSIFY} parent F: handle 28 tcindex classid F:10
# ${CLASSIFY} parent F: handle 30 tcindex classid F:10
# ${CLASSIFY} parent F: handle 0x2E tcindex classid F:B8

# export REMARK="tc class change dev ifb0 parent F:"
# ${REMARK} classid F:10 dsmark mask 0x0 value 0x10
# ${REMARK} classid F:B8 dsmark
# ${REMARK} classid F:FF dsmark mask 0x0 value 0x0
```

Program `tc(8)` neumožňuje specifikovat jiné akce než `police`, přičemž použití `pipe` jako jednoho z výsledků policeru automaticky vrátí neúspěch. Aktuální verze jádra obsahuje chybu, kdy `tcf_exts_validate` přeskočí validaci u filtrů bez akce a jádro během `tcf_action_dump` dereferencuje neinicializovaný ukazatel na spojový seznam akcí přiřazených filtru. Stejný průchod nedokumentovanou funkcí `tcf_exts_validate` provádí i jiné filtry a není tedy jasné, na čí straně je chyba. Aktuální verze systému CentOS tyto chyby nevykazuje.

[7] Neplést s parametrem tříd `dsmark` používaným k úpravě výsledného paketu!

4.2.8 Filtr u32

Parametry: `divisor`, `hashkey`, `ht`, `link`, `match`, `offset`, `police`, `sample`.

Tento filtr[8] se jmenuje `u32`, protože jako klíče používá 32bitové hodnoty uvnitř klasifikovaného paketu; klíč se skládá z pozice[9], požadované hodnoty a bitové masky přes klasifikovaná data. Každý řádek (`struct tc_u_knode`) tohoto filtru může obsahovat libovolné množství klíčů, které musejí vyhovět všechny (logické *and*) a na dané úrovni preference celý filtr může mít takto zařazených řádků více, čímž mezi nimi dosáhne efektu logického operátoru *or*.

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 \
       match ip src 10.0.0.4  \
       match ip dport 5001 0xFFFF \
       classid F00:BAR
```

Oba klíče použily syntaktický cukr, který program tc převede na:

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 \
       match u32 0x0A000004 0xFFFFFFFF at 12 \
       match u32 0x00001389 0x0000FFFF at 20 \
       classid F00:BAR
```

Syntaxe `ip dport` využívá faktu, že protokoly TCP a UDP mají čísla portů na stejném offsetu. Srovnejme s následujícím (nefunkčním) příkladem:

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 \
       match ip src 10.0.0.4  \
       match tcp dst 5001 0xFFFF \
       classid F00:BAR
```

Při výpisu filtrů[10] si všimněme jiného offsetu – místo čísla 20 je tam `nexthdr+0`. Offset můžeme totiž specifikovat i pomocí dat v přijatém paketu (například hlavičky předchozího protokolu). Ten „hlavní“ offset, na začátku matchování nastavený na začátek hlavičky síťové vrstvy, totiž platí jednotně pro celou *tabulku* řádků. Připomeňme, že uvádět na každém řádku `protocol ip` je nutné, neboť jiné protokoly mají na těchto offsetech jiná data!

Hashovací tabulka uvnitř

Parametry: `divisor`, `handle`, `hashkey`, `ht`, `link`, `sample`.

Na výpisu filtrů si všimněme pole `fh` – to je identifikátor daného řádku v rámci daného filtru. Ačkoli je to 32bitové číslo, skládá se ze tří dvojtečkami oddělených částí: 12bitového identifikátoru hashovací tabulky (`HTID`), 8bitového indexu v té

[8] Neplést s *ematchem* u32!

[9] Typicky jde o offset od začátku, přesněji od hodnoty `offset` daného selektoru; více dále.

[10] Při nastavování vřele doporučuji si otevřít okno vypisující statistiky: `tc -s filter show dev enp19s0`.

tabulce a 12bitového identifikátoru řádku. K zadání těchto parametrů se opět používá lokální parametr `handle`.

Vytvořme hashovací tabulku `A:` s 256 pozicemi; kvůli způsobu, jakým je modulární parser stavěn je třeba `handle` uvést před typem filtru `u32`. Jednotlivé řádky lze do tabulky přidat parametrem `ht <HTID>:`.

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→handle A: u32 divisor 256
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 ht A: \
       match ip src 10.0.0.4 \
       match ip dport 5001 0xFFFF \
       classid F00:BAR
```

Rovněž si všimněme, že při vytvoření takovéto tabulky se automaticky vytvořila i tabulka `800:` s jedním slotem[11] – v této tabulce se začíná matchovat. Matchne-li někde řádek s parametrem `link <HTID>:`, filtr začne okamžitě vyhodnocovat danou tabulku. Takto lze tabulky mezi sebou hierarchicky propojovat.

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 \
       link A: \
       match ip src 10.0.0.0/24
```

Parametrem `hashkey` můžeme takto skočit přímo do konkrétního slotu dané tabulky, kde index je libovolná hodnota z přijatého paketu. Tento parametr má dva „podparametry“ `mask` a `at` a protože slotů může být nejvýše 256, klíč se automaticky posune tak, aby nejnižší nastavený bit masky byl první.

Upravme předchozí příklad, aby hashoval podle třetího bajtu zdrojové IP adresy; zde žádný syntaktický cukr neexistuje:

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 \
       link A: hashkey mask 0x0000FF00 at 12 \
       match ip src 10.0.0.0/16
```

Nyní provoz z 10.0.3.0/24 půjde do tabulky `A:3:` a zde začne pravidlem s nejnižším identifikátorem (řádek níže[12] tedy bude mít `fh` rovno `A:3:111`).

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→handle 0x111 u32 ht A:3: \
       match ip src 10.0.3.4 \
       match ip dport 5001 0xFFFF \
       classid F00:BAR
```

[11] Čísla `0x800` a výš typicky znamenají, že identifikátor přidělilo jádro.

[12] Neuvedeme-li `pref`, jádro vygeneruje prázdnou `u32` hashovací tabulku s dalším automaticky přiděleným číslem preference (stejně jako u ostatních filtrů), ale protože hashovací tabulky jsou v rámci disciplíny sdílené, přidávání záznamů do nich není závislé na hodnotách preferencí tabulku používajících pravidel. Nově vytvořená tabulka tedy zůstane připojená k nové úrovni (nade všemi *dříve* automaticky přidělenými hodnotami preference) prázdná.

Parser příkazové řádky umí rozpoznávat parametr `sample`, za kterým je právě jeden klíč (`sel2`, opět lze použít syntaktický cukr), ze kterého spočítá proměnnou `hash` následovně: [iproute2, `tc/f_u32.c`]

```
hash = sel2.sel.keys[0].val&sel2.sel.keys[0].mask;
hash ^= hash>>16;
hash ^= hash>>8;
```

Jelikož jádro počítá slot v rámci tabulky funkcí `u32_hash_fold`, která funguje úplně jinak (posun masky k prvnímu nastavenému bitu), hodnota spočtená parametrem `sample` zcela jistě není ten správný index do hashovací tabulky. Skutečný význam tohoto parametru je mi neznámý. Následující příklad přiřadí naše pravidlo do špatného slotu `0xD`.

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→handle 0x111 u32 ht A: sample ip src 10.0.3.4 \
        match ip src 10.0.3.4 \
        match ip dport 5001 0xFFFF \
        classid F00:BAR
```

Parametrem `indev` lze filtrovat podle rozhraní, na kterém paket přišel, podobně jako *ematchem* `meta(rt_iif)`.

Způsoby specifikace offsetů

Parametry: `at`, `mask`, `plus`, `shift`, `eat`, `nexthdr+`.

Protože mají některé protokoly proměnnou délku svých hlaviček, není často možné najít požadovaný klíč na pevně zadaném místě. Filtr má parametr `offset`, který s pomocí *svých parametrů* dokáže spočítat skutečný offset zevnitř paketu.

`at` n specifikuje odkud (od stávajícího offsetu) se má nový offset vzít.

`mask` *hex* vymaskuje z daného slova požadovanou hodnotu

`shift` n tuto hodnotu posune doprava o n bitů

`plus` n k ní přičte n

`eat` je klíčový, chceme-li propojit několik tabulek mezi sebou. Pokud jej neuvedeme, právě spočtený offset bude v jádře figurovat pouze v proměnné `off2`, čili `nexthdr+` bude fungovat správně a „hlavní“ offset zůstane (na začátku hlavičky síťového protokolu). Uvedeme-li jej, tabulka nový offset „sní“, tj. přičte ke „hlavnímu“, což posune i pevné offsety řádků linkované tabulky.

Často uváděný příklad extrahuje délku IPv4 hlavičky, aby začal fungovat syntaktický cukr `tcp dst` ze začátku této sekce. Je vhodné si jej vyzkoušet s i bez parametru `eat` a pozorovat, funguje-li `match ip`.

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 \
        link A: offset at 0 mask 0x0F00 shift 6 plus 0 \
        match protocol 6 0xFF
```

```
# tc filter add dev enp3s0 protocol ip pref 42 parent F00: ↙
→u32 ht A: \
       match ip src 10.0.0.4 match tcp dst 5001 0xFFFF \
       classid F00:BAR
```

4.2.9 Filtr `bpf`

V říjnu 2013 napsal Daniel Borkmann filtr umožňující klasifikaci na základě výsledku běhu programu pro *Berkeley Packet Filter* (BPF), [BPF93] jehož výsledkem[13] může být 0 při neúspěchu, −1 pro zachování specifikované `classid` a kladné číslo specifikující novou třídu. [cls_bpf] Kód filtru je možné nechat vygenerovat programem `tcpdump(8)` (popis jazyka v `pcap-filter(7)`), nebo jej překládat a ladit ručně nástroji `bpf_asm` či `bpf_dbg`. [LiSrc, `tools/net`] Protože Linux na některých architekturách umožňuje překládat BPF do nativního kódu, má tento filtr vysokou šanci nahradit všechny ostatní. Prozatím jej nelze kombinovat s *ematchi*, ale čtení z vyhrazených záporných adres umožňuje rozšířit klasifikační kritéria podobným způsobem.

```
# tcpdump -ddd ip6 and tcp port 22 | tr '\n' ',' > ssh-v6.bpf
# tc filter add dev enp1s0f1 parent F: bpf run bytecode-file ↙
→ssh-v6.bpf flowid F:22 action police rate 256Kbit burst 3Kb
```

Součástí balíku netsniff-ng (`http://netsniff-ng.org/`) je program `bpfc(8)` schopný překládat BPF instrukce zapsané v textovém formátu. Implementační část práce rovněž obsahuje disassembler, neboť program `tc(8)` vrací bajtkód.

```
# cat ssh-v6.ops
ldh [12]
jne #0x86dd, L9
ldb [20]
jne #0x6, L9
ldh [54]
jeq #0x16, L8
ldh [56]
jne #0x16, L9
L8: ret #0xffff
L9: ret #0x0
# tc filter add dev enp1s0f1 protocol ipv6 pref 622 parent F:↙
→ bpf run bytecode "`bpfc -f tc -i ssh-v6.ops`" flowid F:22
```

Použití tohoto filtru může být programátorskou výzvou zejména v kombinaci s protokolem IPv6, neboť je k dispozici pouze pole `M` šestnácti 32bitových registrů, akumulátor a registr `X` a skoky lze provádět jen dopředu přes nejvýše 255 instrukcí.

4.3 Shrnutí

V rámci časových možností tato kapitola představila schopnosti Linuxu ovládané příkazem `tc filter`, ke kterému dosud neexistuje ucelená dokumentace.

[13] Na rozdíl od jiných použití jako selekce provozu v síťových ovladačích nebo v Netfilter modulu `xt_bpf` či kontrola argumentů systémových volání mechanismem SECCOMP.

5. Implementační část

5.1 Požadavky

Program by měl přehledně vizualizovat použité nastavení na vzdálených strojích. Kód by měl být přenosný (pro případ, že uživatel bude donucen použít jiný systém, než na který je zvyklý) a především snadno udržovatelný (pro případ, že se rozšíří nějaká nová či nepopsaná forma nastavení). Důležitý požadavek je nemuset na routery instalovat žádný nový software, protože mohou být potenciálně chráněné proti zápisu, mít staré verze knihoven nebo běžet na méně častých architekturách.

5.2 Vybraný software

Použil jsem programovací jazyk Python a balík Paramiko, protože v nich minimum řádků kódu přehledně zajistí funkcionalitu, kterou potřebuji. Stávající frontend zajišťuje modul `tkinter`, který je součástí distribuce Python. Program je napsaný pro Python 3 a testovaný na verzi Python 3.3.5, neboť pro verzi 3.4.1 zatím není k dispozici binární instalátor balíku PyCrypto. Program po jeho doinstalování a manuální instalaci `py-ecdsa` a `paramiko` běží i na systému Microsoft Windows. Postup je v uživatelské dokumentaci níže.

Na vzdálených strojích program vyžaduje pouze standardní nástroje, čímž odpadá nutnost instalovat vlastního agenta do celé sítě. Linuxové systémy by měly mít balík iproute2 obsahovat; programy `ifconfig(8)` a `pfctl(8)` jsou součástí instalace OpenBSD. Některé části vyžadují zvýšená oprávnění; v Linuxu se standardně spouštějí utility pod přihlášeným uživatelem[1] a zaškrtnutím „Use sudo“ program spouští všechny příkazy utilitou `sudo(8)`, přičemž standardní chybový výstup je používán na detekci dotazů na heslo, zobrazení dialogového okna a případného ohlášení chyby.[2] Protože OpenBSD nepovolí číst nastavení pf ani ALTQ bez práv superuživatele, `sudo(8)` se pro tyto příkazy používá vždy.

5.3 Existující řešení

Kromě generátorů pravidel (například *tcng* z `http://tcng.sourceforge.net/`) mi není znám žádný software, který by práci s tc usnadňoval, s výjimkou nástroje *ktctool* `http://developer.berlios.de/projects/ktctool/` trpícího následujícími vadami:

Od roku 2006 není udržovaný, pročež závisí na velmi starých verzích knihoven (Qt verze 3) a je tak obtížné jej vůbec zkompilovat. Navíc nepodporuje aktuální škálu prostředků v jádře.

[1] Soubor `software/Linux.py` má na začátku proměnné `IP_PATH` resp. `TC_PATH` definující cestu k těmto utilitám, neboť se tyto mezi distribucemi liší.

[2] Z čehož plyne, že musíme mít v `sudoers(5)` vypnuté `requiretty`, neboť po zavolání funkce `Transport::get_pty` balík Paramiko sjednotí oba výstupové kanály, čímž v podstatě znemožní spolehlivé oddělení výpisů `sudo` od vlastních dat spouštěného programu.

Neumožňuje se připojit na vzdálený stroj, což nutí administrátora instalovat na router jinak nepotřebný software.

Nezobrazuje filtry hierarchicky a jeho nápověda není obsáhlejší než existující zdroje.

Je obrovský co do velikosti (většinou nepotřebného) kódu. Částečně za to mohou zbytečné nástroje jako GNU autotools[3] a částečně dlouhé kusy textu „zabudované“ do kódu programu, tvořící jej podstatně méně čitelným.

Je pouze pro Linux a nepomůže se správou OpenBSD či ALTQ, přestože je tato jednodušší na pochopení z příkazové řádky.

Jeho vzhled z dostupných snímků obrazovky však toto dílo inspiroval. Ktctool umí také konfiguraci měnit. V této práci jsem se rozhodl psát vizualizaci pouze pro čtení již od začátku, z následujících důvodů:

Konfiguraci ukládá každý jinak. Každý správce, každá distribuce, každá varianta embedded systému má naprostou volnost co do uložení dané konfigurace, ať už jako lépe či hůře formátovaný shellový skript (proměnné substituující části příkazů nebo obsahující různé části politiky), či jako generátor pravidel z nějakého jiného jazyka. Principiálně není možné a eticky není vhodné do takovéto konfigurace strojově zasahovat.

Ke změně nastavení unixových systémů sloužila vždy výhradně příkazová řádka. Tím systém vyžadoval od uživatele jistou úroveň zkušeností a neposkytoval abstrakce, které mohou zastínit některé podstatné detaily takto vytvořené konfigurace. Zdrojem ke korektní konfiguraci by měla být dokumentace, což tato práce do jisté míry napravuje popisem některých nezdokumentovaných částí.

Množství dostupných mechanizmů je tak obrovské, že riziko implementační chyby potenciálně způsobující pád vzdáleného systému není vhodné podstupovat. Nekompatibilita plynoucí z potenciálně opravených chyb v utilitě tc by šanci na vznik chyby pouze zvýšila.

5.4 Uživatelská dokumentace

5.4.1 Instalace

Pro běh programu je třeba nainstalovat interpret Pythonu a následující tři knihovny. Popišme instalaci na Microsoft Windows, neboť v unixových OS je podobná, pouze se o balíky stará balíčkovací systém.

1. `https://www.python.org/download/windows/` obsahuje odkaz na 32bitovou binární verzi Pythonu 3.3.5. Při instalaci můžeme nechat zaškrtnuté všechny volby.

[3]Program je od návrhu tvořen pouze pro Linux a není tedy třeba zajišťovat portabilitu na jiné systémy. V současném stavu kompilaci programu více škodí než pomáhají.

2. `http://www.voidspace.org.uk/python/modules.shtml#pycrypto` obsahuje 32bitovou binární verzi balíku PyCrypto 2.6 pro Python verze 3.3.

3. `https://github.com/warner/python-ecdsa/archive/master.zip` obsahuje balík py-ecdsa

4. `https://github.com/paramiko/paramiko/archive/master.zip` obsahuje balík Paramiko

Oba stažené balíky někam rozbalíme a nainstalujeme například příkazem:

```
C:\Python3.3\python.exe setup.py install
```

Spustíme-li `setup.py bdist_wininst`, balík `distutils` do adresáře `dist` automaticky přípraví spustitelný instalační balík pro všechny nalezené verze Pythonu. Hlavní výhoda instalace pomocí připravených instačních balíků namísto přímého `install` je možnost odinstalování pomocí ovládacího panelu *Programs and Features* místo ručního odstraňování souborů.

Aktuální verze balíku Paramiko nepodporuje metody výměny klíčů a HMAC vyžadované RFC 4253 a novější verze OpenSSH mohou zamítnout všechny starší (DH group 1, SHA1 či MD5). Toto řeší například verze na CD, neboť potřebné patche obsahuje jak odděleně, tak zahrnuté do připraveného instalačního balíku.[4]

Spustíme-li nyní interpret Pythonu, po zadání `import paramiko` by se neměly objevit žádné chybové hlášky. Program je připraven ke spuštění; nainstalovat jej lze stejným postupem, spouští se skript `qviz.py`, jak je uvedeno v `setup.py`.

Unixové systémy občas potřebují Tkinter doinstalovat, balík v OpenBSD se jmenuje `python-tkinter`, v Ubuntu `python-tk`, v Gentoo je třeba USE flag `tk`.

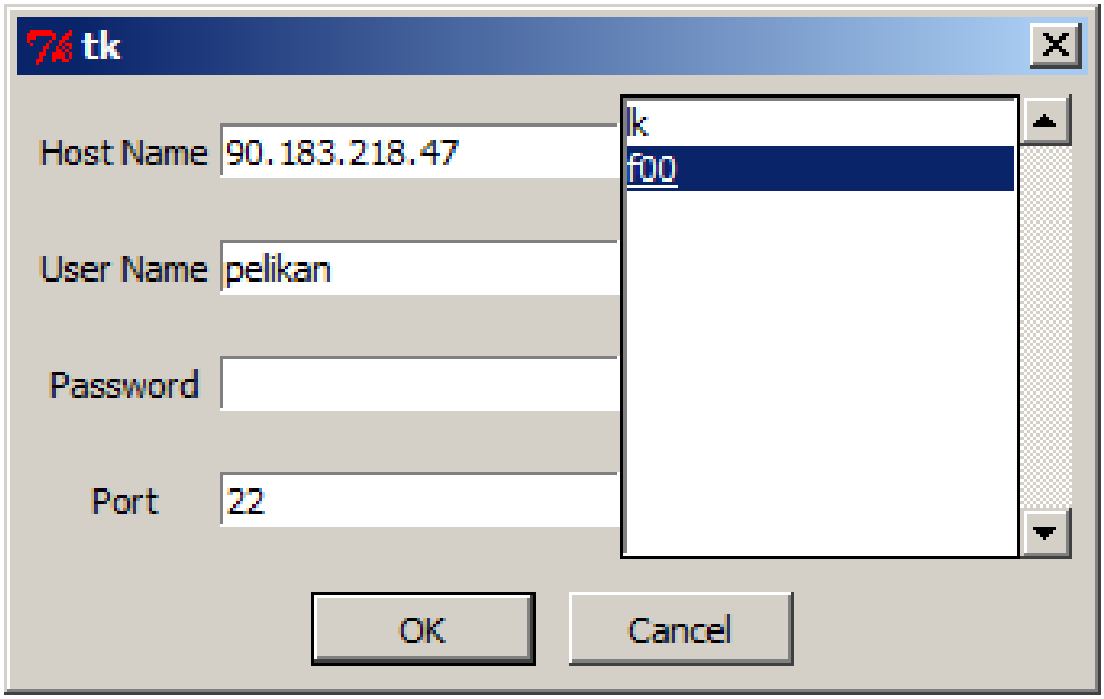
5.4.2 Používání programu

Program spustíme skriptem `qviz.py`. Horní menu[5] obsahuje tři možnosti: ukončit program (*Ctrl+Q*), připojit se k novému stroji (*Ctrl+A*) a znovu načíst nastavení právě vybraného stroje (*Ctrl+R*). Začít lze pouze novým připojením: v následujícím dialogu lze vyplnit adresu cílového stroje, uživatelské jméno a heslo a port, na kterém běží SSH démon, případně vybrat uložené nastavení ze seznamu. Balík Paramiko zajišťuje čtení standardního nastavení z balíku OpenSSH popsaného v manuálové stránce `ssh_config(5)` [ObSrc]. Připojovací dialog tuto konfiguraci přečte z adresáře `~/.ssh/config`[6] a vytvoří seznam s jednotlivými cíli (`Host`), na něž se dvojitým kliknutím program začne připojovat. Uvedeme-li do konfiguračního souboru klauzuli `IdentityFile`, program bude ignorovat vyplněné heslo a použije soukromý klíč z daného souboru. Je-li klíč zašifrovaný, dialogovým oknem se zeptá na heslo, avšak rozšifrování klíče trvá typicky až 10 vteřin, během kterých UI nereaguje. Implicitně umí využít běžící instanci `ssh-agent(1)` (netestováno

[4] Projekt Paramiko již delší dobu trpí nedostatkem aktivity, několikrát změnil svého správce a zapracovat změny jako podporu ECDSA či Pythonu 3 trvalo několik let; změna převzata z: `https://github.com/Bluehorn/paramiko/commit/ecdda18191616d87fd9f233e878bd278d744405a`

[5] Ve Windows se místo nápisů zobrazují klávesové zkratky, což je nejspíš chyba `Tkinter`.

[6] Což se ve Windows rozepíše jako `%USERPROFILE%\.ssh\config`, kam můžeme bez problému zkopírovat své nastavení z unixového systému.



Obrázek 5.1: Přihlašovací okno s přednastavenými cílovými stroji.

ve Windows), což správci umožní připojovat se prakticky pouze dvojitým kliknutím, bez zadávání hesla. Veřejné klíče vzdálených strojů program načítá ze souboru `~/.ssh/known_hosts`. Balík Paramiko nepodporuje mechanismus ED25519, pročež Váš systém nemusí znát RSA klíče vzdálených strojů a bude se ptát na jejich potvrzení. Konfigurační data začne program stahovat až po stisknutí tlačítka „Refresh host“, resp. *Ctrl+R*. Na obou typech systémů se tak naplánují a postupně vykonají čtení dostupných síťových rozhraní, jejich front, tříd a disciplín a nastavení klasifikátoru, v tomto pořadí. Prováděné příkazy jsou vypisovány v pravé spodní části okna, pro informaci uživatele o rychlosti celého procesu.

Výběrem příslušných polí ve stromě vlevo se v pravé horní části zobrazí její vizualizace, v závislosti na dostupné implementaci daného typu dat.

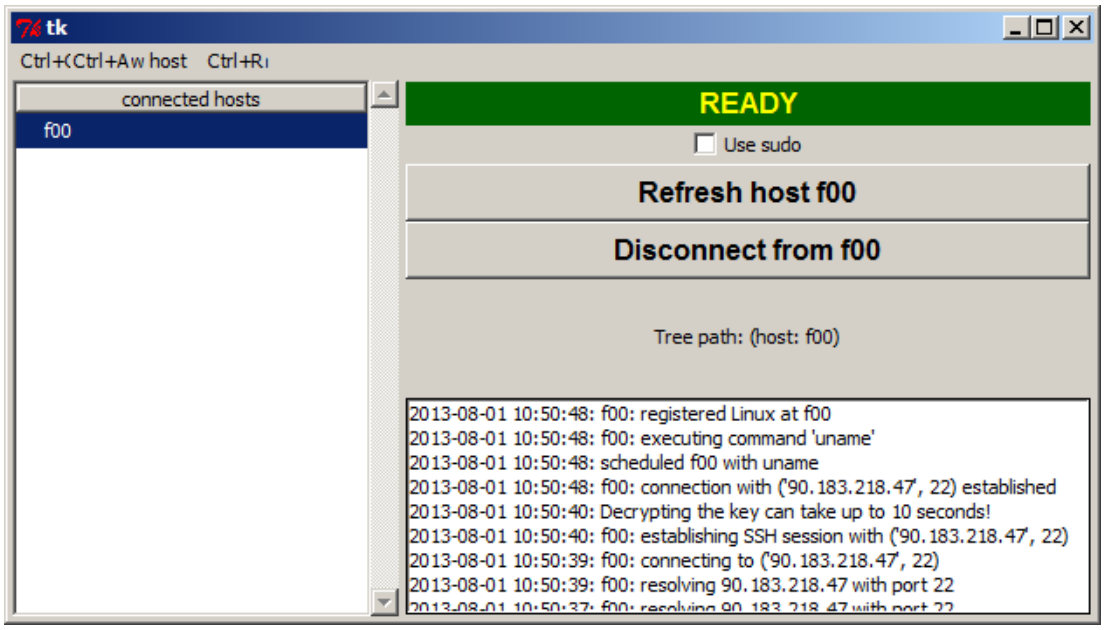
5.5 Návrh programu

Udržovatelnost kódu zajišťuje rozdělení programu na následující části:

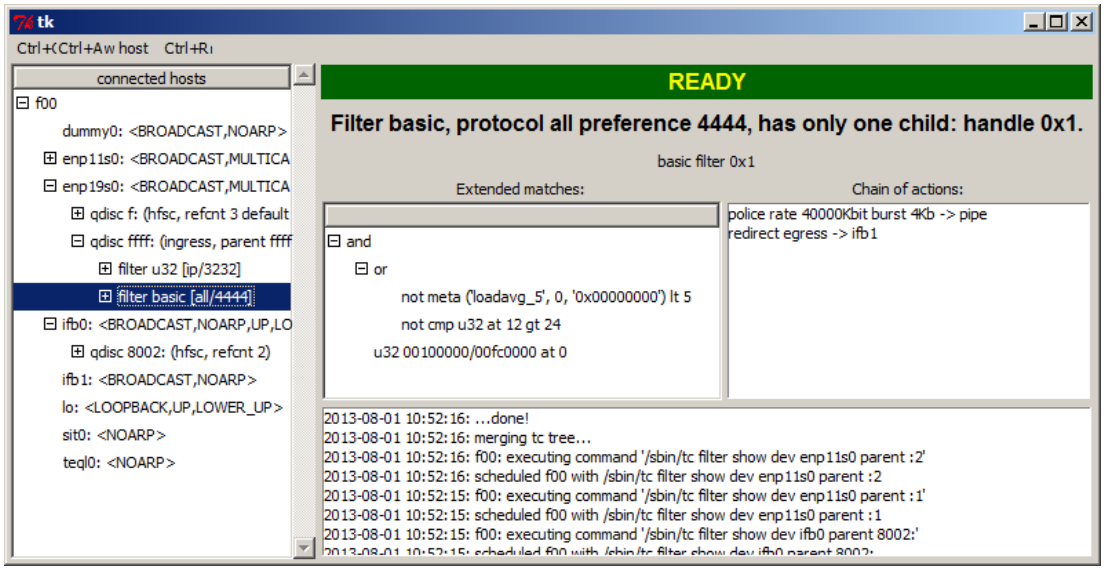
Popis cílového systému třídami v adresáři `software/` obsahuje metody na stažení a zpracování aktuální konfigurace ze vzdáleného stroje.

Parsery výpisu jednotlivých utilit jsou funkce a třídy v adresáři `parsers/` poskytující strukturovaná data k aktualizaci stávajícího popisu systému.

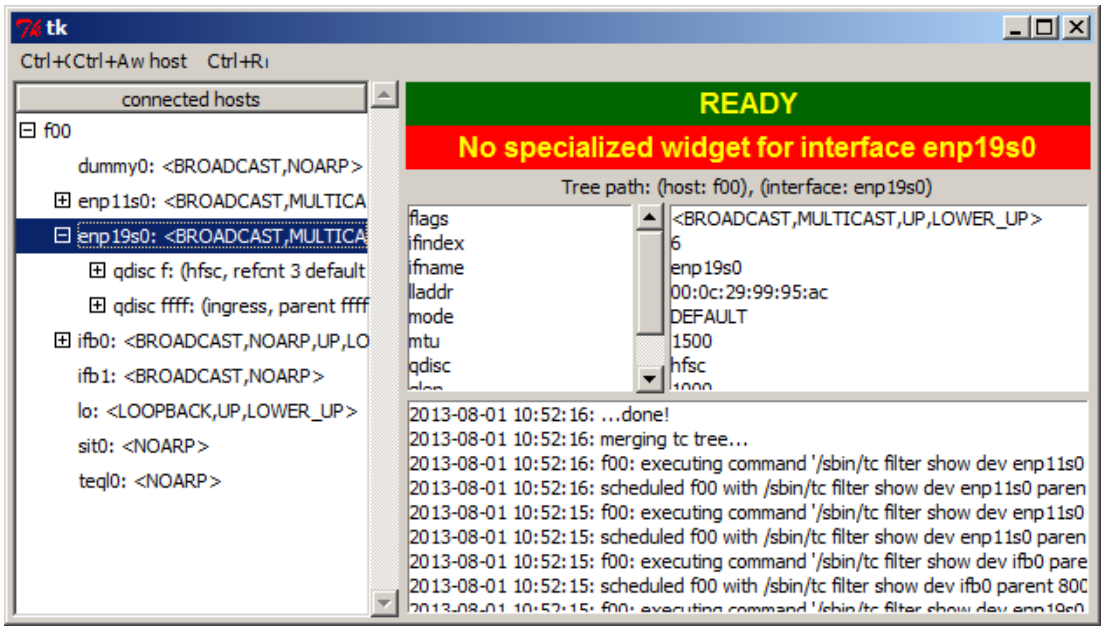
Frontend má své metody v třídě `QVizTkinter` a ostatní metody v adresáři `frontend/` slouží jednotlivým konfiguračním prvkům k vizualizaci jejich detailů. Univerzální metoda `display_dict_table` zobrazí přesněji nespecifikované prvky jako dvojice *(klíč, hodnota)*, což většině prvků k vizualizaci stačí.



Obrázek 5.2: Hlavní okno po připojení ke vzdálenému stroji.



Obrázek 5.3: Ukázka filtru `basic`.



Obrázek 5.4: Většině zobrazení k porozumění stačí seznam klíčů a hodnot.

Hlavní smyčka je metoda `run` frontendu volající `process_events` třídy `QViz`. Kombinací těchto dvou metod program zajišťuje interaktivní obsluhu událostí uživatelského rozhraní (UI) při stahování dat ze vzdálených strojů.

Kombinace předchozích prvků tvoří systém vhodný nejen k vizualizaci konfigurace disciplín a klasifikačních prostředků, ale též rozšiřitelný o případné další prvky jako například procházení vzdáleným souborovým systémem, spouštění služeb, přesměrování portů, nastavováni sítě, degradace CARP a další.

Každý připojený stroj má vlastní frontu příkazů (FIFO), a `process_events` před každým voláním `select(2)` tyto fronty projde (*round-robin*) a potřebují-li dodělat či začít naplánovanou práci, zaregistruje příslušné sokety.

Po připojení ke vzdálenému stroji (`SSH_Host`) na něm QViz automaticky naplánuje `uname` a podle jeho výstupu příslušnému stroji přiřadí instanci třídy z adresáře `software/`. Úplně první kanál `SSH_Host::schroedinger` je spuštěný program `cat(1)` používaný výhradně k ověření, je-li dané spojení živé.[7] Toto sice zvyšuje časové nároky na jednotlivé operace (minimálně jeden *Round-Trip Time* (RTT) navíc za každý příkaz), ale slouží to k obejití následujícího problému. Funkce `paramiko.Transport::open_session` nemá volitelný časový limit a přeruší-li se SSH spojení (např. z důvodu hibernace notebooku či špatného připojení), program v tomto volání visí i několik minut, než dojde ke vrácení výjimky. Takováto kontrola spojení není stoprocentní, ale s podstatně vyšší pravděpodobností (doběhne totiž těsně před voláním `open_session`) zajistí interaktivitu programu nezávislou na vlastnostech sítě.[8]

[7] Posláním řetězce `alive\n` druhá strana řetězec zopakuje a až po jeho úspěšném přečtení může QViz pokračovat.

[8] Na adrese `https://github.com/paramiko/paramiko/issues/86` existuje patch údajně tuto chybu opravující.

5.5.1 Obnovení konfigurace – průchod programem

1. Uživatel vyvolá akci obnovení, například stiskem *Ctrl+R*.
2. `frontend.__cmd_refresh_host` v použitém frontendu obslouží událost a z levého *Treeview* `frontend.ltv` zjistí požadovaný stroj k obnově.
3. `QViz::refresh_host` je hlavní část programu zkoušející nepřipojená spojení znovu navázat.
4. `QVizSoftware::refresh` jsou metody pro obnovení konkrétní instance software zjištěného na daném stroji.
5. `QVizSoftware::async_fetch_*` naplánuje stažení požadovaných dat a zaregistruje si callback na jejich zpracování.
6. `Host::push_task` toto přidání do fronty provádí.
7. Návrat z obsluhy akce (stisku tlačítka či klávesy).

V této chvíli čekáme, dokud hlavní smyčka programu (`frontend.run`) nezačne zpracovávat jednotlivé požadavky:

1. `QViz::process_events` projde všechny připojené stroje a naplánuje ty potřebné; protože knihovna `Tkinter` volá obsluhu událostí i z některých nečekaných míst,[9]) implementoval jsem okolo seznamu strojů `QViz::hosts` zamykací mechanizmus popsaný na začátku této funkce.
2. `Host::pop_task` vybere naplánovanou úlohu z fronty.
3. `Host::cat_alive` otestuje spojení těsně před voláním `open_session`, která je zatím pouze blokující.
4. `Host::run/Host::sudo` spustí daný příkaz na vzdáleném stroji.
5. `Host::move_forward` je volaný několikrát po sobě, oznámí-li nám volání `select(2)`[10] dostupná data.
6. `Host::check_if_exited` z doběhlého programu přečte návratový kód.
7. `Host::finish_fetch_*` bývají typicky callbacky zaregistrované funkcemi `async_fetch_*` zajišťující konkrétní funkcionalitu.
8. Příslušný parser z adresáře `parsers/*` vytvoří třídu nebo `dict` z načteného řetězce.
9. Na konci posloupnosti příkazů čtoucích konfiguraci (typicky implementováno počítadlem referencí a proměnnou `Host::locked`) nějaká funkce slije vytvořené seznamy do výsledné datové struktury.
10. `frontend.ltv_{add,delete,update}_node` uzly takto slitých stromů postupně vizualizují.

[9]Například uprostřed zjišťovacího dialogu, což vede na podobný problém dokumentovaný na `http://bugs.python.org/issue11387`.

[10]Funkce `poll(2)` není dostupná ve verzi Python pro Windows.

5.5.2 Zobrazení vizualizace – průchod programem

1. Uživatel označí uzel ve stromě vlevo (`frontend.ltv`).
2. `frontend.__cmd_ltv_select` v použitém frontendu najde cestu z uzlu do kořene stromu
3. `Host::whatis` vrátí podle cesty dvojici *(typ, handle)*
4. `frontend.visualize_type` najde nejvhodnější funkci pro vizualizaci daného typu. Tyto funkce se registrují v globálním `dictu` `tkinter_type2fn`, jež je upravován na konci souborů s vizualizacemi a definován na začátku `tkinter.py`.
5. Frontend smaže `frontend.stuff` s předchozí vizualizací a *handle* předá s cestou nové funkci.
6. Je vykresleno.

Závěr

Práce detailně popsala průchod síťového provozu jádry Linux a OpenBSD a na každé vrstvě odpovídající referenčnímu modelu ISO/OSI ukázala některé nabízené možnosti konfigurace, včetně rozboru příslušných algoritmů. Implementační část přináší vizualizaci nepřehledného systému *traffic control*, který je v práci popsán vysoce nad rámec své vlastní dokumentace. Z časových důvodů však bylo třeba vynechat například analýzu různých mechanizmů *congestion control* v protokolu TCP či popis datových struktur protokolu Netlink, jímž systém tc komunikuje.

Z dosud vybraných algoritmů na *traffic shaping* práce ukázala, že není třeba podporovat všechny, protože při vhodném nastavení mohou dávat podobné výsledky. Rovněž rozebrala různé pohledy na *Active Queue Management*, kde nové a podstatně jednodušší algoritmy podávají výsledky blíže očekáváním uživatelů.

Nápady na vylepšení, plány do budoucna

Program byl kromě vizualizace QoS otestován též na monitorování okolo 130 routerů za pomoci spouštění příkazů přes SSH na proprietárním firmware. Kromě frontendu `tkinter` existuje ještě frontend vracející kešovaná data na dotazy pomocí JSON RPC soketem, na který je napojen informační systém data zobrazující. (Tento frontend zatím nebyl otestován na Pythonu 3 a není pochopitelně třeba k této práci.) Testy ukázaly, že při větším množství konkurentních operací[11] (u kopírování souborů kolem 4 spojení, u periodického stahování textových dat okolo 100 spojení) výkon dramaticky klesá, neboť knihovna Paramiko vytvoří pro každé spojení zvláštní vlákno a kvůli *Global Interpreter Lock* (GIL) v Pythonu tato vlákna tvoří efekt konvoje. Python neumožňuje měnit pořadí vláken čekajících na GIL, což nechá vlákna kopírující data čekat nepřiměřeně dlouho, než ostatní vlákna dodělají drobnosti jako keepalive či periodické stahování monitorovaných dat. Implementace čekání na podmínkovou proměnnou použitou uvnitř knihovny Paramiko v Pythonu 2.7 používá čekání proměnlivé délky funkcí `sleep`[12] a není tedy možné vlákno ani včas probudit, pročež ze statistik systémových volání nereagující a vytížený program vypadá, že celou dobu spí.

Začal jsem s vývojem škálovatelného SSH multiplexeru v C++11 pomocí knihoven libevent (`http://libevent.org/`) a libssh2 (`http://libssh2.org/`), ale implementace zatím není dostatečně stabilní na použití v bakalářské práci.

Možnosti vizualizace mohou vypadat značně omezené, ale přidání grafiky specifické pro konkrétní typ objektu spočívá pouze v zaregistrování příslušné funkce (vizte například `display_filter`) do globálního mapování mezi typy dat a jejich vizualizací. Nabízí se možnost parsovat a vykreslovat statistiky.

Potenciál tohoto programu lze podstatně rozšířit o jakákoli data ve stromové struktuře přenesitelná protokolem SSH (souborový systém, databáze LDAP) a za úvahu jistě stojí i možnost spouštět příkazy, editovat části konfigurace či její export.

[11] Upgrade firmware několika strojů zároveň vyžaduje data nejdřív nakopírovat pomocí SCP.
[12] `https://github.com/python/cpython/blob/2.7/Lib/threading.py`, `Condition.wait`

Seznam použité literatury

Knihy

[Armit] Armitage, Grenville. *Quality of Service in IP Networks*. 1st ed. Pearson Education, 2000. ISBN 978-1-578-70189-6. Zkontrolovat.

[LiNetInt] Benvenuti, Christian. *Understanding Linux Network Internals*. 1st ed. O'Reilly, 2005. ISBN 978-0-596-00255-8.

[SteWri] Stevens, W. Richard a Gary R. Wright. *TCP/IP Illustrated, Vol. 2: The Implementation*. 1st ed. Addison-Wesley, 1995. ISBN 978-0-201-63354-2.

[TanWet] Tanenbaum, Andrew S. a David J. Wetherall. *Computer networks*. 5th ed. Boston: Pearson Prentice Hall, 2011. ISBN 978-0-13-212695-3.

Traffic shaping a AQM

[Altq98] Cho, Kenjiro. *A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers*. In: *Proceedings of USENIX 1998 Annual Technical Conference*. [online], červen 1998. Dostupné z: `http://www.sonycsl.co.jp/person/kjc/kjc/papers/usenix98/`

[Altq04] Cho, Kenjiro. *ALTQ: Alternate Queueing for BSD UNIX*. [online], 2004. Dostupné z: `http://www.sonycsl.co.jp/person/kjc/kjc/software.html#ALTQ`

[Blue99] Feng, Wu-chang, Dilip D. Kandlur, Debanjan Saha, Kang G. Shin. Blue*: A New Class of Active Queue Management Algorithms*. [online], 1999. Dostupné z: `http://www.eecs.umich.edu/techreports/cse/99/CSE-TR-387-99.pdf`

[BPF93] McCanne, Steve, Van Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. In: *Proceedings of the USENIX 1993 Annual Technical Conference*. [online], leden 1993. Dostupné z: `https://www.usenix.org/legacy/publications/library/proceedings/sd93/mccanne.pdf`

[CBQ95] Floyd, Sally a Van Jacobson. *Link-sharing and Resource Management Models for Packet Networks*. In: *IEEE/ACM Transactions on Networking, Vol. 3 No. 4*. [online], 1995. Dostupné z: `http://ee.lbl.gov/papers/link.pdf`

[CHOKe] Pan, Rong, Balaji Prabhakar a Konstantinos Psounis. *CHOKe – a stateless active queue management scheme for approximating fair bandwidth allocation*. In: *Proceedings of IEEE INFOCOM Conference on Computer Communications, pp.942-951* [online], březen 2000. Dostupné z: `http://www.stanford.edu/~balaji/papers/00chokea.pdf`

[CoDel] NICHOLS, Kathleen a Van JACOBSON. *Controlling Queue Delay*. In: *ACM Queue, Vol. 10 No. 5*. [online], 2012. Dostupné z: http://queue.acm.org/detail.cfm?id=2209336

[ConAv88] JACOBSON, Van. *Congestion Avoidance and Control*. In: *SIGCOMM '88 Symposium proceedings on Communications architectures and protocols*, s. 314–329. [online], 1988. Dostupné z: http://dl.acm.org/citation.cfm?id=52356

[DeKeSh89] DEMERS, Alan, Srinivasan KESHAV a Scott SHENKER. *Analysis and Simulation of a Fair Queueing Algorithm*. [online], 1989. Dostupné z: http://dl.acm.org/citation.cfm?id=75248

[ESFQ] HICKEY, Corey. *ESFQ for Linux 2.6*. [online], 2008. Dostupné z: http://fatooh.org/esfq-2.6

[FlowNet] ZHANG, Lixia. *A New Architecture for Packet Switching Network Protocols*. [online], 1989. Dostupné z: http://www.dtic.mil/dtic/tr/fulltext/u2/a216292.pdf

[HFSC99] ZHANG, Hui et al. *Hierarchical Packet Schedulers*. [online], 1999. Dostupné z: http://www.cs.cmu.edu/~hzhang/HFSC/main.html

[HHF13] LAM, Terry. *net-qdisc-hhf: Heavy-Hitter Filter (HHF) qdisc [LWN.net]*. [online], 2013. Dostupné z: http://lwn.net/Articles/577208/

[HTB03] DEVERA, Martin. *HTB home*. [online], 2003. Dostupné z: http://luxik.cdi.cz/~devik/qos/htb/

[IW10harm] GETTYS, Jim. *IW10 Considered Harmful*. [online], 2011. Dostupné z: http://tools.ietf.org/html/draft-gettys-iw10-considered-harmful-00

[NewDir] TURNER, Jonathan S. *New Directions in Communications (Or Which Way to the Information Age?)*. In: *IEEE Communications Magazine, Vol. 24, Issue 10*. [online], 1986. Dostupné z: http://dl.acm.org/citation.cfm?id=2291592

[QFQ09] CHECCONI, Fabio, Luigi RIZZO a Paolo VALENTE. *QFQ: Efficient Packet Scheduling with Tight Guarantees*. [online], 2012. Dostupné z: http://info.iet.unipi.it/~luigi/papers/20120309-qfq.pdf

[PIE12] PAN, Rong, Preethi NATARAJAN, Chiara PIGLIONE, Mythili S. PRABHU, Vijay SUBRAMANIAN, Fred BAKER a Bill VER STEEG. *PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem*. [online], 2012. Dostupné z: http://tools.ietf.org/html/draft-pan-tsvwg-pie-00

[RED93] FLOYD, Sally a Van JACOBSON. *Random Early Detection Gateways for Congestion Avoidance*. In: *IEEE/ACM Transactions on Networking, Vol. 1 No. 4*. [online], 1993. Dostupné z: http://www.icir.org/floyd/papers/early.pdf

[RED-DL99] Jacobson, Van, Kathleen Nichols a Kedarnath Poduri. *RED in a Different Light*. Koncept. [online], 1999.

[RED-DL10] Gettys, Jim. *RED in a Different Light | jg's Ramblings*. [online], 2010. Dostupné z: http://gettys.wordpress.com/2010/12/17/red-in-a-different-light/

[SCZ93] Shenker, Scott, David D. Clark, Lixia Zhang. *A Scheduling Service Model and a Scheduling Architecture for an Integrated Services Packet Network*. [online], 1993. Dostupné z: http://groups.csail.mit.edu/ana/Publications/PubPDFs/A%20Scheduling%20Service%20Model.pdf

[SFQ90] McKenney, Paul E. *Stochastic Fairness Queueing*. In: *INFOCOM'90. Ninth Annual Joint Conference of the IEEE Computer and Communication Societies.'The Multiple Facets of Integration'. Proceedings., IEEE*. IEEE, 1990. p. 733-740. Dostupné z: http://www.cs.duke.edu/courses/spring09/cps214/papers/sfq.pdf

Měření času

[ACPI1.0] Intel, Microsoft, Toshiba *Advanced Configuration and Power Interface Specification*. Verze 1.0. [online], 1996. Dostupné z: http://www.acpi.info/spec10.htm

[AMD64ref] AMD. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Verze 3.21. [online], říjen 2013. Dostupné z: http://support.amd.com/TechDocs/24594.pdf

[AMD-TSC] Brunner, Richard. *TSC and Power Management Event on AMD Processors*. [online], listopad 2005. Dostupné z: https://lkml.org/lkml/2005/11/4/173

[IA32sdm] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Obj. č. 325462-047US. [online], červen 2013. Dostupné z: http://download.intel.com/products/processor/manual/325462.pdf

[iCPUID] Intel. *Intel Processor Identification and the CPUID Instruction*. Application Note 485. [online], 2012. Dostupné z: http://www.intel.com/content/dam/www/public/us/en/documents/application-notes/processor-identification-cpuid-instruction-note.pdf

[IntelInit] Levicki, Igor, *P-State invariant TSC on Nehalem platforms with multi-packages*. [online], červen 2010. Dostupné z: http://software.intel.com/en-us/forums/topic/289368

[ITSCpat] Dixon, Martin G., Rajesh S. Parthasarathy, Jeremy J. Shrall, *Controlling Time Stamp Counter (TSC) Offsets For Mulitple Cores And Threads*[13]. US Patent 20110154090 A1 [online], červen 2011. Dostupné z: http://www.google.com/patents/US20110154090

[13] Překlep je opravdu součástí názvu patentu.

[PHKtime] Kamp, Poul-Henning. *Timecounters: Efficient and precise timekeeping in SMP kernels*. [online]. Dostupné z: http://phk.freebsd.dk/pubs/timecounter.pdf

[RDTSC97] Intel. *Using the RDTSC Instruction for Performance Monitoring*. [online], 1997. Dostupné z: http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf, původně z: http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

[vmtime] *Timekeeping in VMware Virtual Machines*. Information Guide. [online]. Dostupné z: http://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf

[xentime] Magenheimer, Dan. *TSC_MODE HOW-TO*. [online]. Dostupné z: http://xenbits.xen.org/docs/4.2-testing/misc/tscmode.txt

Síťové adaptéry

[BCM57711] Broadcom. *BCM57711 Programmer's Guide*. [online] Dostupné z: http://www.broadcom.com/collateral/pg/57710_57711-PG200-R.pdf

[i217] Intel. *Intel Ethernet Controller I217 Datasheet*. [online], verze 2.0, duben 2013. Dostupné z: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/i217-ethernet-controller-datasheet.pdf

[i82599] Intel. *Intel 82599 10 GbE Controller Datasheet*. [online], verze 2.76, září 2012. Dostupné z: http://www.intel.com/content/dam/doc/datasheet/82599-10-gbe-controller-datasheet.pdf

[IntModer] Intel. *Interrupt Moderation Using Intel Gigabit Ethernet Controllers*. Application Note 450, rev. 1.1. [online], 2003. Dostupné z: http://www.intel.com/design/network/applnots/ap450.htm

[OpenVMT] VMware. *Open Virtual Machine Tools*. [online]. Dostupné z: http://open-vm-tools.sourceforge.net/

[ReLi96] Mogul, Jeffrey a Kadangode Ramakrishnan. *Eliminating Receive Livelock in an Interrupt-Driven Kernel*. In: *Proceedings of the USENIX 1996 Annual Technical Conference*. San Diego, CA: USENIX Association, 1996. Dostupné z: http://www.stanford.edu/class/cs240/readings/mogul.pdf

[ReLi97] Mogul, Jeffrey a Kadangode Ramakrishnan. *Eliminating Receive Livelock in an Interrupt-Driven Kernel*. In: *ACM Transactions on Computer Systems*. New York: Association for Computing Machinery, 1997, č. 3, s. 217–252. ISSN 0734-2071. Dostupné z: http://www2.research.att.com/~kkrama/papers/kk-mogul-TOCS.pdf

[RSS-NDIS] Microsoft. *Scalable Networking: Eliminating the Receive Processing Bottleneck — Introducing RSS*. [online], 2004.

Dostupné z: http://download.microsoft.com/download/5/d/6/5d6eaf2b-7ddf-476b-93dc-7cf0072878e6/ndis_rss.doc

[SmPaOpt] Intel. *Small Packet Traffic Performance Optimization for 8255x and 8254x Ethernet Controllers*. Application Note 453, rev. 1.0. [online], 2003. Dostupné z: http://www.intel.com/design/network/applnots/ap453.htm

[TrustNIC] Duflot, Loïc, Yves-Alexis Perez. *Can you still trust your network card?* [online], březen 2010. Dostupné z: http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf

Normy

[IEEE802.1] *IEEE 802.1: Bridging & Management*. [online] Dostupné z: http://standards.ieee.org/about/get/802/802.1.html

[IEEE802.3] *IEEE 802.3: Ethernet*. [online] Dostupné z: http://standards.ieee.org/about/get/802/802.3.html

[Infini] InfiniBand Trade Association. *InfiniBand Architecture Specification*, release 1.2.1. [online], 2007. Dostupné z: https://cw.infinibandta.org/document/dl/7143 (po registraci)

Requests for Comments

[RFC970] Nagle, John B. *On Packet Switches with Infinite Storage*. RFC 970. In: *IEEE Transactions on Communications, Vol. Com-35, No. 4* [online], 1985. Dostupné z: http://tools.ietf.org/html/rfc970

[RFC1350] Sollins, K. *The TFTP Protocol (Revision 2)*. RFC 1350. [online], 1992. Dostupné z: http://tools.ietf.org/html/rfc1350

[RFC2309] Braden, B., D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, L. Zhang. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. RFC 2309. [online], 1998. Dostupné z: http://tools.ietf.org/html/rfc2309

[RFC2474] Nichols, K., S. Blake, F. Baker a D. Black. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. [online], 1998. Dostupné z: http://tools.ietf.org/html/rfc2474

[RFC3168] Ramakrishnan, K., S. Floyd, a D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. [online], 2001. Dostupné z: http://tools.ietf.org/html/rfc3168

[RFC3549] Salim, J., H. Khosravi, A. Kleen a A. Kuznetsov. *Linux Netlink as an IP Services Protocol*. RFC 3549. [online], 2003. Dostupné z: http://tools.ietf.org/html/rfc3549

[RFC4594] Babiarz, J., K. Chan, a F. Baker. *Configuration Guidelines for DiffServ Service Classes*. RFC 4594. [online], 2006. Dostupné z: http://tools.ietf.org/html/rfc4594

[RFC5681] Allman, M., V. Paxson a E. Blanton. *TCP Congestion Control*. RFC 5681. [online], 2009. Dostupné z: http://tools.ietf.org/html/rfc5681

Kód, dokumentace a neoficiální návody

[AltqPlan] Brauer Henning. *t2k13 Hackathon Report – checksums, newqueue*. [online], 2013. Dostupné z: http://www.undeadly.org/cgi?action=article&sid=20130606101818

[cls_bpf] Borkmann, Daniel. *[PATCH net-next v2] net: sched: cls_bpf: add BPF-based classifier*. [online], 2013. Dostupné z: http://www.spinics.net/lists/netdev/msg255395.html

[iproute2] *iproute2 / Linux Foundation*. [online]. Dostupné z: http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2

[LARTC] kolektiv autorů. *Linux Advanced Routing & Traffic Control HOWTO*. [online], 2003. Dostupné z: http://www.lartc.org/

[LiSrc] *The Linux Kernel Archives*. [online]. Dostupné z: http://kernel.org/

[ObSrc] *OpenBSD*. [online]. Dostupné z: http://www.openbsd.org/

[RTblTrie] Tzeng, Henry Hong-Yi. *Longest Prefix Search Using Compressed Trees*. In: *Proceedings of IEEE Globecom*. [online], 1998. Dostupné z: http://ece.ut.ac.ir/classpages/F83/Advanced%20Computer%20Networks/PAPERS/LOOKUP/tzeng98longest.pdf

[tcf-exts] Graf, Thomas. *[PATCH 2/9] PKT SCHED: tc filter extension API*. In: *linux-netdev* mailing list. [online], 2004. Dostupné z: http://marc.info/?l=linux-netdev&m=110442875424745&w=2

[TLDP-tc] Brown, Martin A. *Traffic Control HOWTO*. [online], verze 1.0.2, říjen 2006. Dostupné z: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/

[UVM99] Cranor, Charles a Gurudatta Parulkar. The UVM Virtual Memory System. In: *Proceedings of the USENIX 1999 Annual Technical Conference*. Monterey, CA: USENIX Association, 1999. Dostupné z: http://www.usenix.org/event/usenix99/full_papers/cranor/cranor.pdf

Seznam použitých zkratek

ACPI *Advanced Configuration and Power Interface* 15

AF *Assured Forwarding* 42

ALTQ *Alternate Queueing* 6

APIC *Advanced Programmable Interrupt Controller* 15

AQM *Active Queue Management*

BH *Bottom Half* 12

BPF *Berkeley Packet Filter* 39

BQL *Byte Queue Limits* 10

CBQ *Class-Based Queueing* 31

CE *Congestion Experienced* 43

CoS *Class of Service* 39

CQS *Classify, Queue, Schedule* 5

CS *Class Selector* 42

DCB *Data Center Bridging* 4

DiffServ *Differentiated Services* 41

DRR *Deficit Round Robin* 26

DSCP *Differentiated Services Code Point* 41

ECN *Explicit Congestion Notification* 43

EF *Expedited Forwarding* 41

EWMA *Exponentially Weighted Moving Average* 22

FDX *Full Duplex*

FIFO *First In, First Out* 6

HDX *Half Duplex*

HFSC *Hierarchical Fair Service Curve* 28

HHF *Heavy-Hitter Filter* 29

HPET *High Precision Event Timer* 15

IFB *Intermediate Functional Block* 39

ISP *Internet Service Provider* 30

ISR *Interrupt Service Routine* 11

IMQ *Intermediate Queueing Device* 39

LLI *Low Latency Interrupt* 11

MSR *Machine Specific Register* 16

MTU *Maximum Transmission Unit* 13

NAPI *New Application Programming Interface* 12

OS operační systém unixového typu 3

pf *Packet Filter* 6

PFC *Priority Flow Control* 4

PIE *Proportional Integral controller Enhanced* 25

PIT *Programmable Interrupt Timer* 14

QFQ *Quick Fair Queueing* 28

QoS *Quality of Service* 4

RED *Random Early Detection* 21

RSFB *Resilient Stochastic Fair Blue* 25

RSS *Receive-Side Scaling* 13

RTC *Real Time Clock* 14

RTT *Round-Trip Time* 62

Rx příchozí provoz

SFB *Stochastic Fair Blue* 24

SFQ *Stochastic Fairness Queueing* 51

SP *Strict Priority* 26

SR-IOV *Single Route Input/Output Virtualization* 14

TBF *Token Bucket Filter* 31

tc *traffic control* 6

TDM *Time Division Multiplexing* 30

ToS *Type of Service* 8

TSC *Time Stamp Counter* 15

Tx odchozí provoz

WRR *Weighted Round Robin* 32

Přílohy

Na přiloženém CD se nacházejí následující soubory:

`pelikanm_bakalarska_prace.tgz` zdrojový text bakalářské práce

`pelikanm_bakalarska_prace.pdf` přeložená práce ve formátu PDF

`qviz-1.0.tgz` program QViz (implementační část)

`paramiko_patches.tgz` použité patche proti standardní verzi balíku Paramiko

`windows_binaries.zip` připravené instalátory pro Microsoft Windows