VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ŘÍZENÍ PROVOZU VE VYSOKORYCHLOSTNÍCH SÍTÍCH V PROSTŘEDÍ DPDK

TRAFFIC SHAPING IN HIGH SPEED NETWORKS IN DPDK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PAVEL DOLEŽAL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ROMAN VRÁNA**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19786/2016/xdolez64

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Doležal Pavel**
Obor: Informační technologie
Téma: **Řízení provozu ve vysokorychlostních sítích v prostředí DPDK**
Traffic Shaping in High Speed Networks in DPDK
Kategorie: Počítačové sítě

Pokyny:

1. Nastudujte problematiku a možnosti řízení síťového provozu (traffic shaping)
2. Seznamte se s frameworkem DPDK pro vysokorychlostní zpracování síťového provozu
3. Navrhněte řešení pro řízení provozu v prostředí DPDK
4. Navržené řešení implementujte
5. Implementaci otestujte a změřte výkonnost
6. Dosažené výsledky diskutujte a navrhněte možná rozšíření

Literatura:

- Dle pokynů vedoucího

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vrána Roman, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Předmětem této bakalářské práce je plánování síťového provozu ve vysokorychlostních sítích. V práci je popsán framework DPDK, který lze využít pro rychlé zpracovávání paketů. Jsou popsány obecné mechanismy plánování síťového provozu a plánování provozu v Linuxu pomocí nástroje tc. Dále je představen návrh a implementace plánovače síťového provozu v prostředí DPDK pro sítě o šířce pásma 10 Gbps. Pro plánovač je použit komplexní mechanismus hierarchického modelu zásobníku žetonů. Systém byl otestován pomocí generátoru síťového provozu Spirent.

Abstract

This bachelor thesis is focused on traffic shaping in high speed networks. It presents framework DPDK, which can be used for fast packet processing. General traffic shaping mechanisms are described as well as traffic shaping in Linux using program tc. It also introduces a design and implementation of traffic shaper using DPDK framework for networks with 10 Gbps bandwidth. The traffic shaper uses a complex mechanism of hierarchical token bucket. The system was tested using high speed traffic generator Spirent.

Klíčová slova

plánovač síťového provozu, rozložení provozu, DPDK, model zásobníku žetonů, nástroj tc, vysokorychlostní sítě, kvalita služeb

Keywords

traffic shaper, traffic shaping, DPDK, token bucket, tc tool, high speed networks, quality of service

Citace

DOLEŽAL, Pavel. *Řízení provozu ve vysokorychlostních sítích v prostředí DPDK*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Roman Vrána.

Řízení provozu ve vysokorychlostních sítích v prostředí DPDK

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Romana Vrány. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Doležal
17. května 2017

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Romanu Vránovi za poskytnuté rady, ochotu a připomínky při tvorbě práce.

Obsah

Kapitola 1

Úvod

Žijeme v době internetu, kdy lidé často vlastní hned několik zařízení schopných komunikovat po síti. Každý samozřejmě chce svoje informace dostat ihned a jakékoliv zdržení na cestě po síti je nepřijatelné. Síťová infrastruktura ale není neomezená a úkolem síťových administrátorů je, aby každý uživatel dostal svůj díl přenosového pásma. K tomuto účelu slouží nástroje pro řízení síťového provozu.

S řízením síťového provozu se často setkáme i u těch nejmenších domácích sítí. Velkou důležitost ale má také ve vysokorychlostních sítích obsluhujících obrovské množství síťového provozu, kterému musí být zaručeno správné doručení. Tato práce se zaměřuje na tvorbu plánovače síťového provozu (traffic shaper), schopného plánovat pakety ve vysokorychlostních sítích s rychlostí síťového provozu až 10 Gbps.

Kapitola 2 představuje obecné principy plánování síťového provozu. Popisuje nejjednodušší principy, které používají všechna síťová zařízení i pokročilé techniky, které jsou potřeba pro komplexní plánování síťového provozu.

V kapitole 3 se čtenář dozví, jakým způsobem je řešeno plánování síťového provozu v systému Linux a jak si může každý uživatel plánování nastavit podle svých potřeb.

Kapitola 4 se věnuje představení frameworku DPDK, který poskytuje rozhraní pro implementaci aplikací schopných zpracovávat pakety ve vysokorychlostních sítích.

Kapitoly 5 a 6 představují návrh a implementaci plánovače síťového provozu v prostředí DPDK, který je schopen plánovat pakety v sítích s rychlostí provozu až 10 Gbps.

V kapitole 7 jsou uvedeny výsledky testování implementovaného plánovače a v kapitole 8 jsou navržena možná rozšíření plánovače a směr možného pokračování práce.

Kapitola 2

Mechanismy plánování

Rozložení provozu (traffic shaping) slouží k regulaci rychlosti a objemu provozu jednotlivých nebo agregovaných toků v síti. Snaží se přizpůsobit přenos paketů dané rychlosti a zmírnit zahlcení tím, že rozprostře v čase shluky paketů. Způsobí tím vyhlazení výstupní rychlosti provozu a tedy zajistí lepší využití přenosového pásma. Běžnými mechanismy rozložení provozu jsou model tekoucího vědra a model zásobníku žetonů. Hlavní rozdíl oproti ořezání provozu (policing) je v tom, že rozložení provozu nejenže omezuje rychlost toku, ale ukládá také přesahující provoz do paměti (bufferu). Je tedy potřeba dostatek místa v paměti pro ukládání přesahujícího provozu do front. Obvykle se rozložení provozu aplikuje na výstupní provoz ze sítě.

Při síťové komunikaci dochází na výstupech síťových prvků k vytváření front pro odchozí provoz. Řazení a vybírání paketů z front se nazývá plánování a procesu, který plánování provádí, se říká plánovač. Existuje mnoho způsobů, jak plánovat průchod paketů a obsluhu výstupních front. Poznatky v této kapitole vycházejí z těchto knih [19], [18].

2.1 FIFO fronty

FIFO fronta je nejběžnějším způsobem obsluhy paketů. Pakety frontu opouštějí podle principu First-In-First-Out, tedy v pořadí v jakém do fronty vstoupily. Pokud je výstupní fronta zaplněna, mechanismus pro zahazování paketů rozhodne, jestli se zahodí nově příchozí paket, nebo jiný paket z fronty. Jednou z výhod FIFO fronty je předvídatelné zpoždění paketu. Při rychlosti linky R (v bitech za sekundu) a maximální velikosti fronty B (v bitech), je zpoždění D možno spočítat tímto vztahem:

$$D \leq \frac{B}{R}$$

Nevýhodou FIFO fronty je, že nemá žádný mechanismus pro zacházení s pakety s různou prioritou. Pokud se ve frontě ocitne příliš velký paket, může zdržovat malé pakety ve frontě za ním. Pokud nějaký tok obsahuje velké množství paketů ve shlucích, může FIFO fronta způsobit odepření služby pro ostatní toky, dokud není tento tok obsloužen. FIFO fronta je využívána jako výchozí metoda pro správu linky, pokud není určeno jinak.

2.2 Prioritní fronty

Prioritní fronty klasifikují pakety do jedné nebo více prioritních tříd ve výstupní frontě. Způsoby klasifikace paketů jsou různé. Můžeme využít značek paketů (marking), jako například pole ToS (Type of Service) v hlavičce IP datagramu [1]. Toto pole se už dnes nazývá

DSCP (Differentiated services code point) [20] a obsahuje kódy definované diferenciovanými službami (DiffServ). Klasifikovat pakety ale můžeme také podle zdrojové či cílové IP adresy, čísel portů a dalších kritérií. Každá třída paketů má poté svou výstupní frontu. Při výběru dalšího paketu na výstup plánovač vybírá nejdříve pakety z tříd s vyšší prioritou. Pouze když je fronta s vyšší prioritou prázdná, začne plánovač obsluhovat fronty s nižšími prioritami. V rámci jedné prioritní fronty se pakety zpravidla obsluhují na principu FIFO.

Výhodou prioritních front je rozdělení toků do různých tříd a nastavení priorit pro jejich obsluhu. Například VoIP provoz a signalizace IP telefonie potřebují mít přednost před ostatním provozem. Nedokážeme ale garantovat konkrétní parametry přenosu ani využití přenosového pásma. Upřednostňujeme pouze pakety určtité třídy před jinými. Při použití prioritních front můžeme narazit na problém tzv. vyhladovění, kdy pakety patřící do třídy s vyšší prioritou budou neustále předbíhat pakety v třídách s nižší prioritou, které nikdy nebudou obslouženy. Tento jev se nazývá striktní prioritizace a je vhodné ho použít například při přenosu VoIP, kdy potřebujeme zaručit, že žadné pakety tohoto přenosu nebudou zahozeny.

2.3 Cyklické fronty (Round Robin)

Cyklické fronty řeší problém s vyhladověním paketů, na který jsme narazili u prioritních front. Zpracování paketů probíhá cyklickou obsluhou front, kdy je z každé fronty odebrán stejný počet paketů. Speciálním typem cyklických front jsou spravedlivé fronty (fair queues). Ty stejně jako cyklické fronty odebírají z front stejný počet paketů, ale tento počet je normalizován na délku paketů. Cyklické fronty zaručují, že nemůže být nějaká fronta opakovaně předbíhána. Pokud je fronta prázdná, obsluha pokračuje na nejbližší neprázdné frontě.

2.4 Váhové fronty WHQ

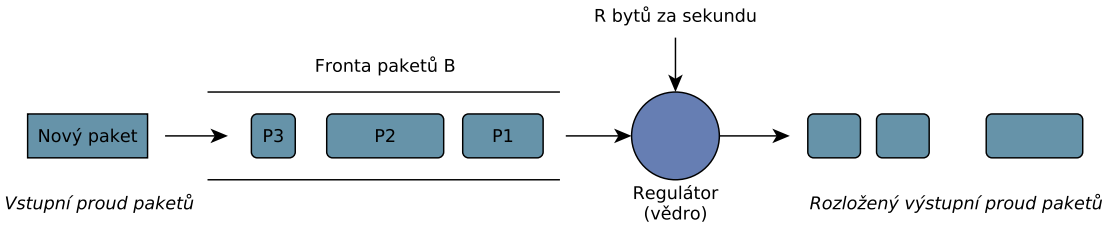
Váhové fronty rozšiřují vlastnosti cyklických front. Ve váhových frontách jsou přicházející pakety řazeny klasifikátorem do toků a každému toku je přiřazena jedna fronta. Identifikace toku probíhá na základně informací z hlaviček vrstev L3 a L4. Podle těchto informací se hashovací funkcí určí ukazatel do fronty. Pokud fronta pro daný tok již existuje, vloží se do ní paket. Pokud se jedná o první paket toku, je vytvořena nová fronta, do které se paket vloží. Každé frontě je přidělena váha, podle které se určuje, kolik paketů se z fronty odebere. Fronty se opět obsluhují cyklicky, z každé fronty je ale při průchodu odebrán různý počet paketů podle přidělené váhy. Počet front závisí na počtu aktivních toků, podle nichž jsou fronty dynamicky vytvářeny a rušeny. Kvůli tomu váhové fronty neumožňují přesné zajištění přenosového pásma. Pokud je systém front zaplněn, jsou přicházející pakety zahazovány.

2.5 Model tekoucího vědra (Leaky Bucket)

Model tekoucího vědra poskytuje mechanismus pro rozložení provozu tak, aby měl stálý výstup. Provozu, který je obvykle tvořen nepravidelně přicházejícími shluky dat, vnutí konstantní přenosovou rychlost. Pro představu uvažujme vědro s malou dírou na dně. Bez ohledu na rychlost přitékající vody do vědra je rychlost odtékající vody konstantní a záleží jen na velikosti díry na dně. Pokud je vědro plné a rychlost přitékající vody je větší než

rychlost odtékání, tak se další přitékající voda přelije přes okraj a neobjeví se na odtékajícím toku vody.

Tato myšlenka se používá pro rozložení datového provozu. Do vědra (fronty) přichází neregulovaný tok. Pakety odcházejí do sítě na dně vědra konstantní přenosovou rychlostí R bytů za sekundu. Velikost vědra je nastavena na určitou hodnotu bytů. Pokud je rychlost příchozích paketů větší než rychlost R na výstupu, vědro se časem naplní a další příchozí pakety jsou zahazovány. Model tekoucího vědra řídí tok v síti tak, že pakety nemohou být přeposílány větší než námi povolenou rychlostí R. Velikost fronty B určuje maximální zpoždění, které může paket při rozložení provozu získat. Grafické znázornění modelu si můžeme prohlédnout na obrázku 2.1.



Obrázek 2.1: Model tekoucího vědra [18]

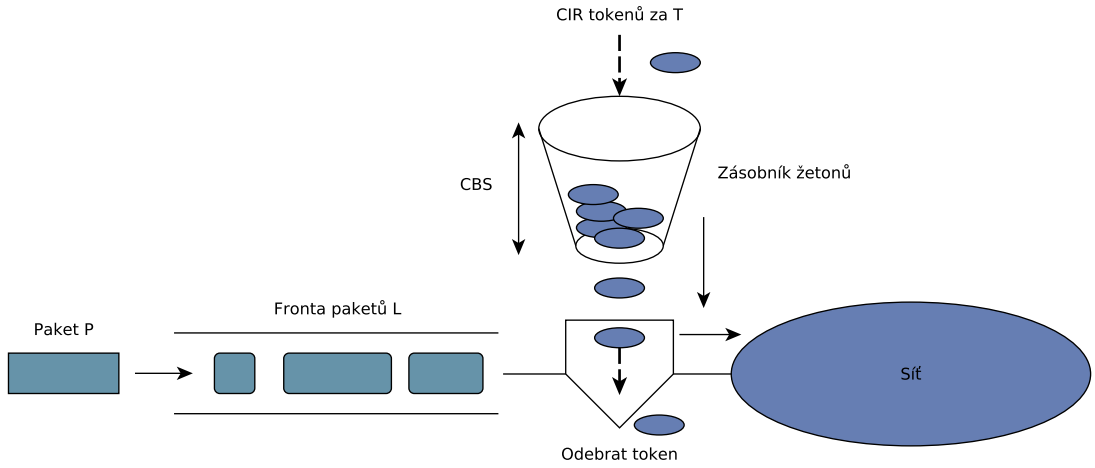
2.6 Model zásobníku žetonů (Token Bucket)

Model zásobníku žetonů umožňuje na rozdíl od modelu tekoucího vědra krátkodobě překročit definovanou výstupní rychlost tak, aby vstupní shluk dat prošel bez zahození. Tento model obsahuje zásobník, který je schopen uchovávat žetony. Jeden žeton odpovídá povolení odeslat jeden byte. Žetony se generují určitou rychlostí a ovlivňují tím průchod datového toku. Zásobník žetonů umožňuje odeslat shluky dat až do velikosti zásobníku B naráz a bez čekání. Krátkodobě tak může rychlost výstupu přesáhnout požadovanou průměrnou (střední) rychlost. Grafické znázornění modelu si můžeme prohlédnout na obrázku 2.2.

K popisu přenosu používá zásobník žetonů tyto veličiny:

Maximální velikost shluků CBS (Committed Burst Size) Definuje maximální počet bytů, které mohou být krátkodobě přeposlány po síti.

Průměrná rychlost CIR (Committed Information Rate) Určuje velikost dat, která se pošlou za daný časový interval. Reprezentuje dlouhodobou průměrnou rychlost odchozích paketů. Důležitý je interval, na kterém se CIR zjišťuje. Tok s rychlostí 100 paketů za sekundu je například více omezen než tok s rychlostí 6000 paketů za minutu, i když oba toky mají v delším časovém intervalu stejnou rychlost. Druhé omezení totiž umožňuje poslat například 1000 paketů za sekundu s tím, že na zbývajících 59 sekund zbývá 5000 paketů. U prvního omezení ale po dosažení limitu 100 paketů za sekundu dojde k zahazování paketů.



Obrázek 2.2: Model zásobníku žetonů [18]

Časový interval T Definuje dobu trvání shluku CBS.

Rychlost ve špičce PIR (Peak Information Rate) Definuje přenosovou rychlost zásobníku, kterou lze dosáhnout v kratším časovém intervalu, při posílání shluků dat rychlostí větší než CIR.

Průměrnou rychlost CIR můžeme vyjádřit tímto vztahem:

$$CIR = \frac{CBS}{T}$$

Tento vztah nám umožňuje vypočítat velikost zásobníku pro uložení shluků dat o délce trvání T. Platí také, že rychlost provozu nepřesáhne během celočíselného násobku časového intervalu T průměrnou rychlost *CIR*. Velikost zásobníku odpovídá hodnotě *CBS*, tedy maximálnímu shluku, který můžeme přenést do sítě. Nové žetony do zásobníku generujeme rychlostí *CIR* žetonů za časový interval. Pokud zásobník obsahuje méně žetonů než *CBS*, přidají se nově vygenerované žetony do zásobníku po hodnotu *CBS*, ostatní žetony se zahodí. Maximální rychlost přenosu shluku paketů ve špičce o dělce trvání t je dána vztahem:

$$PIR = \frac{CBS}{t} + CIR$$

2.7 Prevence zahlcení RED a WRED

Každá fronta v síťových zařízeních je konečná a může se tedy zahltit. Při zahlcení fronty dochází k zahazování paketů. Pakety patřící k toku TCP používají mechanismus řízení toku TCP, který při zahazování paketů daného toku sníží automaticky rychlost toku, aby zamezil zahazování. Po uvolnění front se rychlost toku postupně navyšuje, dokud se výstupní fronta opět nezahltí, což způsobí opětovné snížení rychlosti toku. Tento cyklus se nazývá *problém globální synchronizace TCP*.

Snížení rychlosti TCP toku ale může způsobit další problém. Pokud přenosové pásmo současně využívá tok paketů UDP, může si tento tok postupně přivlasnit celé přenosové pásmo. UDP totiž nemá žádný mechanismus pro snížení rychlosti toku. Když tedy TCP tok sníží svoji rychlost, aby zamezil zahazování paketů, tok UDP zabere uvolněnou část

přenosového pásma a omezí tím tok TCP paketů. V krajním případě mohou toky UDP úplně vytěsnit toky TCP z přenosového pásma. Tomuto problému se říká *vyhladovění TCP*.

Pro omezení výše popsaných problémů se implementuje mechanismus pro prevenci zahlcení RED (Random Early Detection). Tento mechanismus průběžně sleduje stav výstupních front a pokud velikost některé fronty přesáhne minimální práh Q_{min}, začnou se náhodně zahazovat pakety přícházející do této fronty. Pravděpodobnost zahazování příchozích paketů se zvyšuje spolu se zaplněním výstupní fronty. Čím více se fronta zaplní, tím větší je pravděpodobnost zahození příchozího paketu. Po dosažení maximálního prahu Q_{max}, který může být stejný jako skutečná velikost fronty, jsou zahazovány všechny příchozí pakety pro danou frontu. Pravděpodobnost zahazování příchozích paketů je definována vztahem

$$P_a = P_{max} \frac{Q_{avg} - Q_{min}}{Q_{max} - Q_{min}},$$

kde Q_{avg} je průměrný délka fronty a P_{max} je maximální pravděpodobnost zahození paketu, kterou je možno nastavit v intervalu $0 \leq P_{max} \leq 1$.

Mechanismus prevence zahlcení WRED (Weighted RED) pracuje na stejném principu jako mechanismus RED a rozšiřuje ho o přidání váh jednotlivým třídám paketů. Pravděpodobnost zahození příchozího paketu nezávisí jen na aktuální délce fronty, ale také na hodnotě IP precedence nebo DSCP, kterou získáme z IP hlavičky paketu. Čím vyšší prioritu má třída, do které paket patří, tím vyšší je minimální práh Q_{min} této třídy. Pakety třídy s vyšší prioritou se tedy začnou zahazovat později než pakety třídy s nižší prioritou.

Kapitola 3

Plánování paketů v systému Linux

Rozložení provozu v Linuxu lze realizovat pomocí nástroje *tc* [15], který je součástí balíčku *iproute2* [12]. Tento plánovač je permanentně aktivní v linuxovém jádře. I když ho nechceme explicitně použít, běží na pozadí a provádí plánování paketů. Ve výchozím stavu tento plánovač udržuje základní frontu typu FIFO. Poznatky v této kapitole vycházejí z těchto zdrojů [3], [16].

Základní komponenty linuxové kontroly síťového provozu tvoří:

qdisc (plánovací disciplína) Qdisc je v podstatě plánovač. Každé výstupní rozhraní potřebuje nějaký plánovač, ve výchozím stavu jím je FIFO fronta. V Linuxu máme mnoho dalších typů plánování, které si můžeme nastavit. Qdisc je základní stavební blok, na kterém je postavena kontrola síťového provozu v Linuxu. Můžeme mu také říkat plánovací disciplína (queuing discipline). Plánovací disciplíny dělíme na beztřídní a třídní. Každé výstupní rozhraní má výchozí kořenový plánovač (root qdisc), přes který prochází všechen provoz poslaný na výstupní rozhraní. Tento kořenový plánovač se může dělit a obsahovat další plánovací disciplíny s třídami a třídními strukturami.

class (třída) Třída může existovat pouze uvnitř třídní plánovací disciplíny. Třídy jsou velice flexibilní a mohou obsahovat několik dceřiných tříd nebo i jedinou dceřinou plánovací disciplínu. Třída může sama obsahovat další třídní plánovací disciplínu, což umožňuje komplexní možnosti řízení provozu. Každá třída může mít libovolné množství filtrů na ni navázaných, které umožňují výběr vhodné dceřiné třídy, nebo reklasifikovat či zahodit paket vstupující do této třídy. Listová třída je konečná třída stromu plánovacích disciplín. Obsahuje plánovací disciplínu a nikdy neobsahuje dceřinné třídy.

filter (filtr) Filtr je nejkomplexnější komponenta linuxového řízení provozu. Poskytuje mechanismus pro provázání klíčových elementů řízení provozu. Nejjednodušší a nejzřejmější úlohou filtru je klasifikovat paket. Linuxové filtry umožňují uživateli klasifikovat pakety do výstupní fronty pomocí jednoho nebo několika různých pravidel. Filtry mohou být navázány na třídní plánovací disciplíny nebo na třídy, nicméně každý přichozí paket musí nejprve projít přes kořenovou plánovací disciplínu. Až po projití filtru navázaného na kořenovou plánovací disciplínu může být paket přesměrován do dalších dceřiných tříd, které mohou mít vlastní filtry a kde může být paket znovu klasifikován.

classifier (klasifikátor) Filtry, které můžeme nastavovat pomocí nástroje *tc*, mohou používat několik klasifikačních mechanismů. Nejběžnějším z nich je *u32 klasifikátor*. Tento klasifikátor umožňuje uživateli vybírat pakety na základě jejich atributů. Klasifikátory jsou nástroje, které mohou být použity jako součást filtru, k identifikaci vlastností paketu nebo jeho metadat.

handle Každá třída a třídní plánovací disciplína vyžaduje unikátní identifikátor ve struktuře plánování provozu. Tomuto unikátnímu identifikátoru se říká *handle* a skládá se ze dvou čísel – hlavního (major number) a vedlejšího (minor number). Tato čísla mohou být přiřazena uživatelem podle následujících pravidel. Hlavní číslo pro linuxové jádro nic neznamená a uživatel ho může libovolně nastavit. Všechny objekty řízení provozu se stejným rodičem nicméně musí mít stejné hlavní číslo. Dle konvencí mají objekty navázané přímo na kořenovou plánovací disciplínu hlavní číslo 1. Vedlejší číslo jednoznačně určuje objekt jako plánovací disciplínu, pokud je rovno 0. Jakákoliv jiná hodnota určuje objekt jako třídu. Všechny třídy sdílející stejného rodiče musí mít unikátní vedlejší číslo. *Handle* je externí identifikátor objektů použitelný pro uživatelské programy. Linuxové jádro udržuje svůj interní identifikátor pro každý objekt.

3.1 Beztřídní plánovací disciplíny

Jsou to elementární plánovače používané v Linuxu. Tyto plánovací disciplíny mohou být použity jako primární plánovací disciplína na rozhraní, nebo uvnitř listové třídy třídní plánovací disciplíny. Beztřídní plánovací disciplína *pfifo_fast* je výchozím plánovačem používaným v Linuxu.

3.1.1 FIFO (pfifo, bfifo)

FIFO algoritmus tvoří základ pro výchozí plánovací disciplínu na všech Linuxových síťových rozhraních. Neprovádí rozložení provozu ani přeskupování paketů. Pouze přeposílá pakety jak nejrychleji může po jejich přijetí a zařazení do fronty. Udržuje seznam paketů, kdy příchozí paket je vždy zařazen na konec seznamu. Když je vyžádán paket k odeslání do sítě, je vybrán paket ze začátku seznamu.

Reálná plánovací disciplína FIFO musí mít nicméně pevně stanovený limit velikosti fronty, aby nedošlo k přetečení, pokud jsou pakety do fronty přijímány rychleji, než mohou být odesílány. Linux implementuje dvě základní plánovací disciplíny FIFO podle způsobu omezení velikosti fronty – fronta omezená počtem bytů (*bfifo*) a fronta omezená počtem paketů (*pfifo*). Nehledě na použitý typ fronty je velikost fronty definována parametrem *limit*. Všechny nově vytvořené třídy používají plánovací disciplínu *pfifo*, dokud nejsou nastaveny jinak.

3.1.2 pfifo_fast

Toto je výchozí plánovací disciplína pro všechna rozhraní v Linuxu. Tato plánovací disciplína je založena na *pfifo*, ale umožňuje do určité míry nastavení priorit. Obsahuje tři fronty FIFO pro oddělení provozu. Provoz s nejvyšší prioritou (interaktivní toky) je umístěn do fronty 0 a je vždy obsloužen první. Podobně je provoz ve frontě 1 vždy obsloužen před provozem ve frontě 2. Pakety jsou řazeny do jedné z front na základě hodnoty bitů ToS (Type of Service) z IP hlavičky paketu. Každá ze tří front může mít maximální velikost

podle parametru *limit*. Jakmile je jedna z front plná, další příchozí pakety, které patří do této fronty, jsou zahazovány.

3.1.3 SFQ (Stochastic Fair Queuing)

SFQ neumožňuje rozložení provozu, ale pouze plánuje přenos paketů na základě toků. Cílem je spravedlivé posílání paketů tak, že všechny toky se cyklicky střídají v posílání paketů, čímž se zamezí upřednostnění určitého toku před ostatními. Je toho dosaženo pomocí hashovací funkce, která roztřídí provoz do samostatných FIFO front, které jsou cyklicky obsluhovány. Protože existuje možnost, že hashovací funkce přidělí dvěma tokům stejnou frontu, je tato funkce periodicky upravována. Délku této periody lze nastavit pomocí parametru *perturb*, výchozí hodnotou je 10 sekund.

Hodnota hashovací funkce může být také získána z externího klasifikátoru toků nastaveného pomocí filtru. Pokud je použit výchozí interní klasifikátor, SFQ používá k výpočtu hodnoty hashovací funkce zdrojovou a cílovou IP adresu a zdrojový a cílový port z hlavičky paketu, pokud jsou k dispozici.

Dalšími důležitými parametry, které můžeme nastavit jsou *divisor* – určuje maximální počet současně vytvořených FIFO front a *depth* – určuje maximální velikost FIFO front.

3.1.4 RED (Random Early Drop)

RED je plánovací disciplína, která se snaží chytře regulovat velikost své fronty. Většina front jednoduše zahazuje příchozí pakety, pokud se zaplní. RED se snaží o postupné zahazování paketů. Jakmile fronta dosáhne určité průměrné délky, příchozí pakety mají nastavitelnou pravděpodobnost, že budou zahozeny. Tato pravděpodobnost se lineárně zvětšuje do hodnoty maximální průměrné délky fronty, která nemusí být rovna maximální velikosti fronty.

Mezi nastavitelné parametry patří:

min Průměrná velikost fronty, při jejímž dosažení začne zahazování paketů s určitou pravděpodobností.

max Průměrná velikost fronty, při jejímž dosažení je pravděpodobnost zahazování paketů na maximu. Měla by být alespoň dvakrát větší než hodnota *min*. Výchozí hodnota je *limit*/4.

limit Skutečná maximální velikost fronty v bytech.

probability Maximální pravděpodobnost zahazování paketů. Určena jako celé číslo od 0,0 do 1,0. Doporučené hodnoty jsou 0,01 a 0,02, výchozí hodnotou je 0,02.

3.1.5 TBF (Token Bucket Filter)

Tato plánovací disciplína je založena na modelu zásobníku žetonů. Omezuje tedy rychlost přenosu paketů na určenou hodnotu s možným krátkodobým nárůstem rychlosti. Pokud není v zásobníku dostatek žetonů k přenesení paketu, je paket pozdržen ve frontě, dokud není zásobník doplněn dostatkem žetonů. TBF v Linuxu umožňuje také nastavit maximální rychlost při krátkodobém nárůstu rychlosti. Tato maximální rychlost je implementována jako druhý TBF s menším zásobníkem.

Mezi nastavitelné parametry patří:

limit nebo latency *Limit* je maximální hodnota bytů, které mohou čekat ve frontě na tokeny. Lze také nastavit parametr *latency*, což je maximální čas, který může paket strávit čekáním ve frontě na tokeny.

burst Velikost zásobníku žetonů v bytech. Určuje také maximální velikost shluku k odeslání krátkodobě zvýšenou rychlostí.

mpu (minimum packet unit) Minimální velikost paketu. Určuje minimální spotřebu tokenů pro odeslání jednoho paketu. Výchozí hodnota je 0.

rate Průměrná rychlost odesílání paketů.

Další parametry jsou k dispozici, pokud chceme omezit maximální rychlost při krátkodobém nárůstu rychlosti:

peakrate Maximální rychlost vyprazdňování zásobníku.

mtu/minburst Velikost menšího zásobníku. Pro největší přesnost by měl být nastaven na velikost MTU rozhraní.

3.2 Třídní plánovací disciplíny

Třídní plánovací disciplíny jsou velmi užitečné, pokud máme několik druhů síťového provozu, které vyžadují specifické zacházení, protože umožňují větvení. Větvím této stromové struktury se říká třídy. Každá třída v nástroji *tc* musí mít jméno. K tomu slouží parametr *classid*. Parametr *parent* ukazuje na rodiče třídy. Všechna jména by měla být ve tvaru *x:y*, kde x je název rootu (kořen síťového zařízení) a y je jméno třídy. Pokud nastavíme síťovému zařízení třídní plánovací disciplínu, je potřeba nastavit filtry, které určí, které pakety budou zpracovány kterou třídou.

3.2.1 HTB (Hierarchical Token Bucket)

HTB používá model zásobníku žetonů spolu s třídním systémem a filtry, aby umožnil komplexní kontrolu síťového provozu. Díky modelu půjčování nabízí HTB mnoho sofistikovaných technik kontroly provozu, mezi které patří například rozložení provozu. HTB je rozšířením plánovací disciplíny TBF, umožňuje uživateli zanořovat různě do sebe zásobníky žetonů a definovat jednotlivě jejich charakteristiky.

Veškeré rozložení provozu probíhá pouze v listových třídách plánovací disciplíny. Vnitřní a kořenové třídy existují pouze k tomu, aby určovaly, jak má model půjčování rozdělovat dostupné tokeny. Model půjčování je důležitou součástí HTB. Dceřiné třídy se snaží půjčovat tokeny od svých rodičů, jakmile překročí danou přenosovou rychlost *rate*. Vypůjčit tokeny se snaží do dosažení maximální rychlosti *ceil*, po jejímž dosažení začne řadit příchozí pakety do fronty, dokud není k dispozici více tokenů. Aby model půjčování fungoval, každá třída musí udržovat čítač tokenů používaných sebou a svými dceřinými třídami. Z tohoto důvodu je jakýkoliv token použitý dceřinou nebo listovou třídou započítán každé rodičovské třídě, dokud není dosažena kořenová třída. Každá dceřiná třída, která si chce vypůjčit token, si ho vyžádá od rodičovské třídy. Pokud rodičovská třída přesáhne svoji hodnotu *rate*, pokusí

se také vypůjčit token od své rodičovské třídy a tak dále, dokud není nalezen volný token nebo je dosaženo kořenové třídy.

Mezi nastavitelné parametry patří:

default Volitelný parametr každého objektu HTB. Výchozí hodnota je 0, která způsobí, že každý neklasifikovaný paket je odeslán hardwarovou rychlostí a přeskočí všechny třídy nabalené na kořenovou plánovací disciplínu.

rate Nastavuje minimální požadovanou rychlost provozu. Ekvivalent hodnoty CIR v modelu zásobníku žetonů, neboli garantovaná rychlost pro danou listovou třídu.

ceil Nastavuje maximální požadovanou rychlost provozu. Může být považován za ekvivalent PIR v modelu zásobníku žetonů.

burst Velikost zásobníku žetonů pro rychlost *rate*.

cburst Velikost pomocného zásobníku žetonů pro rychlost *ceil*.

quantum Klíčový parametr modelu půjčování. Jednotka, po které jsou půjčovány tokeny mezi třidami.

prio Nastavuje priority jednotlivých tříd na stejné úrovni. Třídy s nejnižší prioritou jsou v cyklické obsluze tříd obslouženy první. Povinný parametr.

3.2.2 PRIO (prioritní plánovač)

Plánovací disciplína PRIO funguje na jednoduchém principu. Když je připravena odeslat paket, zkontroluje první třídu v pořadí. Pokud tato třída obsahuje paket, tak je odeslán. Pokud je fronta této třídy prázdná, je zkontrolována další třída v pořadí, až je nalezen paket nebo jsou zkontrolovány všechny třídy.

Při tvorbě této plánovací disciplíny pomocí nástroje *tc* je vytvořen fixní počet pásem. Každé pásmo obsahuje třídu, další třídy už poté nelze přidávat. Při odesílání paketu je vždy jako první zkontrolováno pásmo 0. Dále jsou na řadě pásma 1, 2, atd. Pakety, které jsou potřeba poslat přednostně, bychom tedy měli zařadit do pásma 0.

Klasifikaci paketů do pásem je možné provést třemi způsoby:

z uživatelského prostoru Proces s dostatečným oprávněním může přímo určit třídu paketu pomocí parametru SO_PRIORITY.

programově (přes nástroj tc) Použitím filtru navázaného na kořenovou plánovací disciplínu.

s odkazem do priomapy Nejčastěji používaný způsob, kdy priorita je odvozena od pole ToS (Type of Service) z IP hlavičky příslušného paketu.

ToS bity mohou mít různou definici. RFC 2474 [20] popisuje novější a preferovanou verzi mapování, ale některé programy mohou stále používat starší definici uvedenou v RFC 791 [22].

Mezi nastavitelné parametry patří:

bands Nastavuje množství vytvořených pásem, výchozí hodnota je 3.

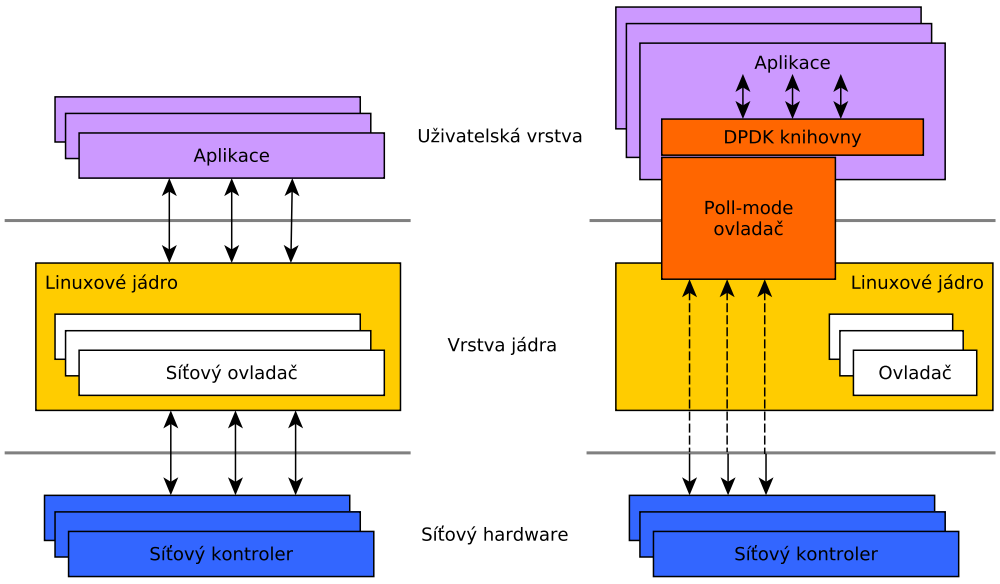
priomap *tc* filtr navázaný na kořenovou plánovací disciplínu. Specifikuje, jak plánovací disciplína přiděluje pakety do pásma.

Kapitola 4

DPDK

DPDK (Data Plane Development Kit) je sada knihoven a ovladačů síťových karet určených pro rychlé zpracovávání paketů v uživatelské vrstvě systému. Mezi funkce frameworku patří podpora víceprocesového zpracování, poskytnutí poll-mode ovladačů, knihovny s paměťovými strukturami podporujícími přístup ke sdílené paměti bez nutnosti zámků, knihovny pro klasifikaci a plánování paketů, hash knihovna ad. Spojením funkcí těchto knihoven můžeme vytvořit komplexní aplikace pro rychlé zpracování paketů v uživatelské vrstvě systému [7].

V současnosti je DPDK plně kompatibilní s Linuxem a existuje částečný port na FreeBSD. DPDK se snaží o co nejlepší využití výkonu procesorů a síťových karet zejména společnosti Intel. Seznam podporovaných síťových karet jiných výrobců se ale neustále rozrůstá [11]. DPDK podporuje rovnoměrné rozložení provozu aplikací na všechna jádra procesoru, abychom dosáhli co největšího výkonu. Hlavní výhodou DPDK je minimalizování režie linuxového jádra při komunikaci aplikace v uživatelské vrstvě se síťovým hardwarem a jeho ovladači, jak je znázorněno na obrázku 4.1. Tímto postupem dosáhneme velké úspory výkonu při zpracovávání paketů.



Obrázek 4.1: Zmenšení režie zpracování paketu s frameworkem DPDK

4.1 Poll-mode ovladače

Linux většinou využívá k obsluze hardware ovladače založené na obsluze přerušení. Při příchodu paketu na síťovou kartu vyvolá ovladač přerušení. Jádro systému musí pozastavit svoji další aktivitu a zavolat obsluhu přerušení pro danou událost. V případě příchodu paketu je paket zkopírován do vstupní fronty jádra. Jádro se poté postará o zpracování paketu. Kód pro obsluhu přerušení má přednost před kódem zpracovávajícím paket. Při vysoké rychlosti příchozího toku paketů se může stát, že kód obsluhy přerušení bude stále dostávat přednost před zpracováním paketu. Vstupní fronta jádra se tak zaplní a systém selže. Tomuto problému se v literatuře říká *receive-livelock* [2].

Framework DPDK využívá k obsluze síťových karet svoje tzv. poll-mode ovladače, které běží v uživatelském prostoru systému. Tyto ovladače používají k obsluze síťových karet metodu pollingu. Tato metoda spočívá v opakovaném aktivním dotazování síťové karty ohledně jejího stavu. Ovladač přistupuje k deskriptorům vstupních a výstupních front síťové karty bez jakýchkoliv přerušení systému, což umožňuje rychlé přijímání, zpracování a odesílání paketů aplikací v uživatelském prostoru [10].

4.2 Velké paměťové stránky

Kdykoliv proces používá operační paměť, procesor musí vyznačit část RAM, kterou tento proces využívá. Kvůli efektivnosti je paměť RAM alokována po úsecích určité velikosti, kterým se říká stránky. Tyto stránky mají zpravidla 4096 B. Procesor a operační systém si musí pamatovat, které stránky patří kterému procesu a kde v paměti jsou uložené. Při požadavku procesu na přístup ke svým datům se musí projít všechny stránky, dokud nenajdeme stránky patřící dánému procesu. Čím více operační paměti procesy naalokují, tím déle trvá, než najdeme stránky patřící danému procesu, což může mít vliv na výkon aplikace.

Mnoho současných architektur podporuje alokování větších stránek než obvyklých 4096 B. Těmto paměťovým stránkám se říká různě podle systému, v němž se alokují – *huge pages* v Linuxu, *super pages* v BSD nebo *large pages* ve Windows [21]. Větší velikost pamětových stránek znamená jejich menší počet a větší velikost souvislých bloků paměti, což urychluje jejich vyhledávání pro konkrétní proces. Framework DPDK vyžaduje použití velkých paměťových stránek pro alokaci paměťových struktur používáných pro ukládání paketů do mezipaměti. Velikost těchto stránek je většinou 2 MB nebo 1 GB podle nastavení uživatele.

4.3 EAL (Environment Abstraction Layer)

Vrstva EAL je zodpovědná za přístup DPDK aplikací k nízkoúrovňovým zdrojům jako hardware a paměťový prostor. Poskytuje aplikacím obecné rozhraní, které skrývá specifika daného prostředí a zajišťuje inicializaci zdrojů jako paměťový prostor, PCIe zařízení, časovače atd. EAL rovněž zajišťuje rozdělení běhu aplikace mezi jednotlivá jádra procesoru. Informace o PCI zařízeních a paměťovém prostoru získává EAL z rozhraní jádra */sys* a z modulů jádra *uio_pci_generic* nebo *igb_uio*.

Pro inicializaci jednotlivých procesů DPDK aplikace je použita standardní knihovna *pthread*. Při spuštění aplikace specifikujeme, která jádra procesoru chceme využít pro běh aplikace pomocí parametru *-c COREMASK*, kde *COREMASK* je hexadecimální bitová maska jader procesoru. Po spuštění aplikace se nejprve spustí funkce *main()*. V ní by měla být volána funkce *rte_eal_init()*, která provede inicializaci jader přidělených pro běh

aplikace v masce jader procesoru. Tato inicializace je provedena voláními funkcí knihovny *pthread*. Pokud není nastaveno jinak, je každému jádru procesoru přiděleno jedno vlákno vytvořené knihovnou *pthread*.

EAL již v inicializační fázi aplikace provede alokaci fyzické paměti aplikace pomocí funkce *mmap()* a velkých paměťových stránek. Alokace probíhá s využitím souborového systému linuxového jádra *hugetlbfs*. Tato paměť je poté přístupná knihovnám DPDK pro práci s pamětí, které si v ní mohou rezervovat pojmenované úseky pro svou potřebu [8].

4.3.1 Spouštěcí parametry

Při spouštění DPDK aplikací musíme nastavit dva druhy parametrů – EAL parametry a aplikační parametry. Vzorové spuštění DPDK aplikace vypadá takto:

```
./DPDK-aplikace <EAL parametry> -- <aplikační parametry>
```

Aplikační parametry jsou specifické pro každou aplikaci. EAL parametry jsou ale jednotné pro všechny DPDK aplikace a lze jimi nastavit mnoho důležitých vlastností aplikace [9]:

-c COREMASK nebo -l CORELIST *COREMASK* je hexadecimální bitová maska jader procesoru. *CORELIST* je seznam jader procesoru. Určují, která jádra procesoru může DPDK aplikace využít. Jediné povinné parametry EAL vrstvy.

-n NUM Počet paměťových kanálů na soket procesoru. Nastavuje podle počtu povolených jader procesoru pro aplikaci.

-b <domain:bus:devid.func> Zamezuje EAL vrstvě v použití daného portu PCIe zařízení.

--use-device <domain:bus:devid.func> Říká EAL vrstvě, že může použít pouze tyto porty PCIe zařízení. Nemůže být použito zároveň s přepínačem *-b*.

--socket-mem MB Určuje paměť v MB, kterou má EAL vrstva alokovat z velkých paměťových stránek pro dané sokety.

-d Přidání ovladače nebo složky s ovladači k načtení. Aplikace by měla tento přepínač využít k načtení ovladačů, které jsou nastaveny jako sdílené knihovny.

-m MB Určuje paměť v MB, kterou má EAL vrstva alokovat z velkých paměťových stránek nehledě na sokety. Je doporučeno místo toho používat přepínač *--socket-mem*.

-r NUM Určuj počet paměťových stupňů.

-v Zobrazí informace o verzi DPDK při spuštění aplikace.

--huge-dir Specifikuje adresář, kde jsou připojené velké paměťové stránky.

--file-prefix Prefix použitý při pojmenování velkých paměťových stránek pro danou aplikaci. Hodí se při běhu několika DPDK aplikací naráz.

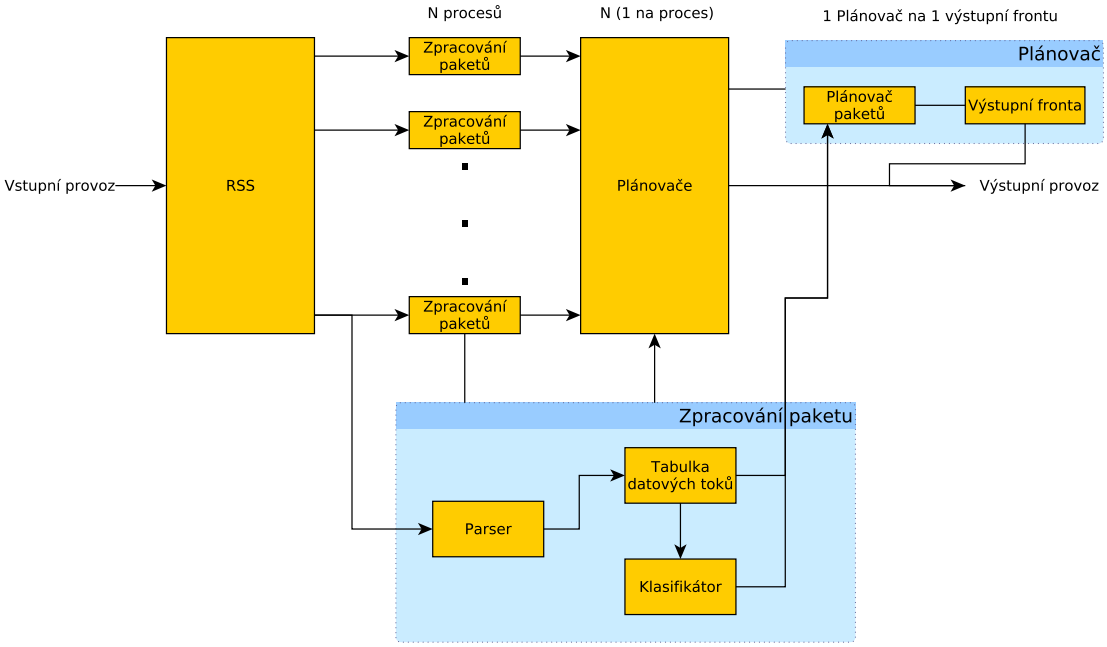
--proc-type Typ instance procesu dané aplikace. Využívá se ve víceprocesových aplikacích. Může být primární nebo sekundární.

--xen-dom0 Podpora pro běh aplikace na Xen Domain0 bez velkých paměťových stránek. Xen Domain0 je platforma pro virtualizaci hardware.

Kapitola 5

Návrh systému

Vytvoření plánovače síťového provozu nespočívá jen ve výběru plánovacího mechanismu. Plánovač musí zvládnout celé zpracování paketu od přijmutí paketu z jednoho rozhraní po odeslání paketu na druhé rozhraní. Po přijmutí paketu ho musíme zpracovat pomocí parseru a poté klasifikovat, abychom dostali parametry nutné pro zvolený plánovací mechanismus. Poté paket zařadíme do front plánovače, který obstará nakonfigurované rozložení provozu. Nakonec si vyžádáme další paket z front plánovače a ten odešleme na výstupní rozhraní. Vypracovaný návrh kompletního plánovače síťového provozu můžeme vidět na obrázku 5.1.



Obrázek 5.1: Návrh plánovače řízení provozu v DPDK

5.1 RSS (Receive Side Scaling)

Abychom byli schopni zpracovávat síťový provoz o rychlosti až 10 Gbps, je nutné rozložit provoz aplikace do několika procesů. V našem návrhu každý proces vykonává celé zpracování a plánování paketu. Jednotlivé procesy se tedy liší jen v tom, které pakety zpracovávají. Pro co nejvyšší výkon této metody je potřeba zajistit rovnoměrné rozložení příchozích paketů mezi všechny procesy aplikace.

Pro rovnoměrné rozložení paketů mezi jednotlivé procesy jsme využili RSS. Tato technologie spočívá ve vypočítání hash hodnoty pro každý paket již na síťovém rozhraní. Tato hash hodnota je poté použita jako index do tzv. Indirection table, což je tabulka, ve které je zvoleno jádro procesoru ke zpracování paketu. Hash hodnoty metody RSS se zpravidla počítají podle určitých polí v hlavičkách paketu jako například zdrojová a cílová IP adresa. Tím je zajištěno, že pakety patřící jednomu datovému toku jsou zpracovány stejným jádrem procesoru [14].

5.2 Parsování a klasifikace paketu

Po přidělení paketu danému procesu a jeho načtení ze vstupní fronty je potřeba získat z paketu informace potřebné pro jeho klasifikaci. Klasifikaci provádíme na základě hodnot z IP hlavičky a hlavičky transportního protokolu daného paketu.

Abychom nemuseli klasifikovat všechny příchozí pakety zvlášť a zlepšili výkon aplikace, přidali jsme do návrhu aplikace tabulku datových toků. Do tabulky datových toků (flow cache) se ukládají toky, jejichž pakety už plánovač klasifikoval. Každý příchozí paket nejprve zkontroluje tuto tabulku, jestli už se v minulosti neklasifikoval paket stejného datového toku. Pokud je v tabulce nalezen záznam odpovídající aktuálnímu paketu, jsou pro vstupní parametry plánovače použity hodnoty vyčtené z tabulky a paket nemusí projít klasifikačním procesem. V opačném případě je nutné zavolat klasifikátor a vyčíst vstupní parametry pro plánovač z hlaviček paketu. Každý proces aplikace má svoji vlastní tabulku datových toků.

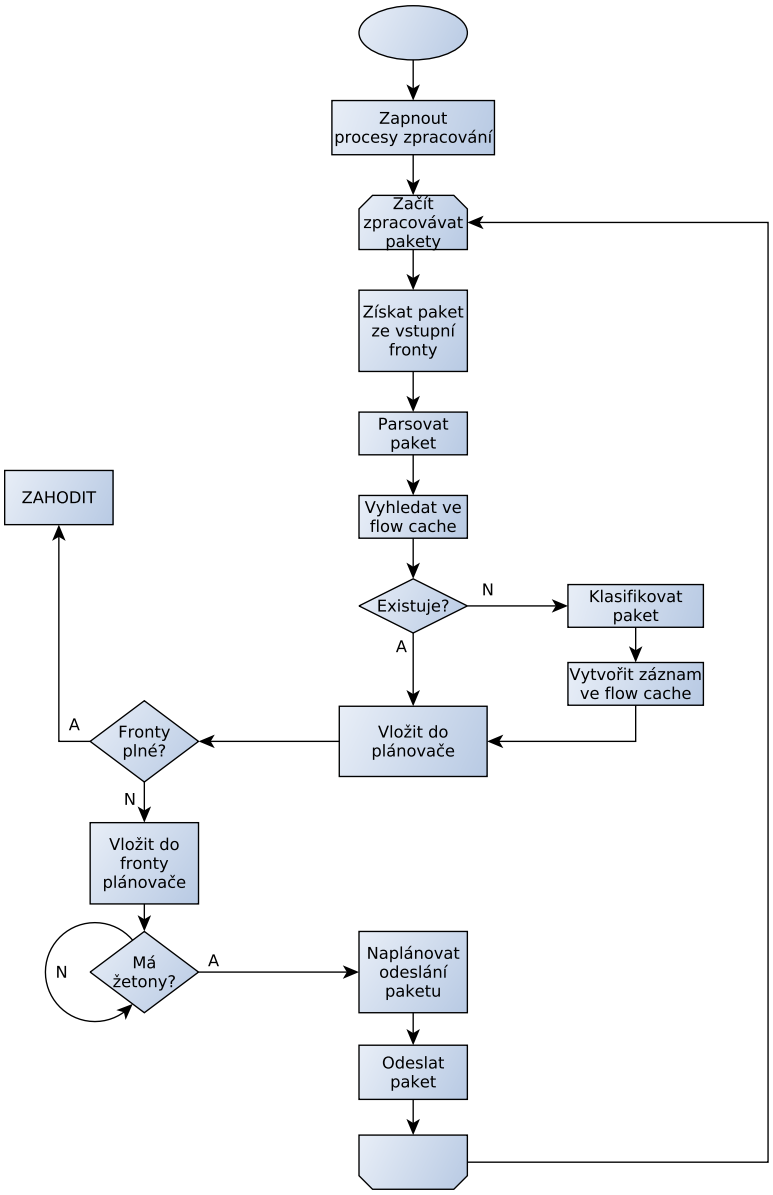
5.3 Plánovač paketů a rozložení provozu

Klasifikovaný paket je zařazen do plánovače. Ideální plánovač poskytuje pokud možno co největší možnosti nastavení rozložení provozu pro různé typy provozu a pro různé datové toky. Nejpoužívanějším mechanismem pro komplexní rozložení provozu je model zásobníku žetonů představený v kapitole 2.6. V kapitole 3.2 je představen princip zanořování jednotlivých plánovacích mechanismů do sebe, který umožňuje dosažení velkých možností nastavení rozložení provozu.

Jako vhodný plánovač byl zvolen mechanismus postavený na principu Linuxového *HTB*, který byl představen v kapitole 3.2.1. Tento mechanismus umožňuje velké možnosti nastavení a také umožňuje použití různých mechanismů plánování ve vnitřních vrstvách plánovače. Každý proces aplikace má svůj vlastní plánovač. Všechny tyto plánovače mají stejnou konfiguraci, kterou zvolíme při startu aplikace.

5.4 Výstup aplikace

Každý proces aplikace si vyžádává paket k odeslání ze svého plánovače. Plánovač zkontroluje svoje fronty a vybere paket k odeslání. Tento paket je poté odeslán do příslušné výstupní fronty. Počet výstupních front aplikace je stejný jako počet vstupních front a počet procesů aplikace. Výstupní rozhraní cyklicky vybírá pakety z těchto výstupních front a paket tím opouští aplikace. Celý postup zpracování paketu v jednom procesu aplikace můžeme vidět v diagramu na obrázku 5.2.



Obrázek 5.2: Diagram zpracování paketu

Kapitola 6

Implementace plánovače řížení provozu

Implementace plánovače síťového provozu vychází z návrhu představeného v předchozí kapitole. Pro implementaci byl zvolen programovací jazyk C a knihovny frameworku DPDK. Program byl implementován a testován na verzi frameworku DPDK 17.02. Kostra plánovače vychází z příkladu jednoduchého plánovače ve frameworku DPDK, který je dostupný v instalačním adresáři DPDK [6] pod licencí BSD stejně jako zbytek DPDK. Tento příklad byl modifikován a rozšířen, aby vyhovoval našemu návrhu.

6.1 Vstupní parametry programu

Program na vstupu očekává jednu nebo více dvojic vstupní port a výstupní port a většinou konfigurační soubor s nastavením rozložení provozu pro plánovač. Zadání konfiguračního souboru není povinné, plánovač se může inicializovat s výchozími hodnotami, které jsou uzpůsobené pro průchod veškerého provozu rychlostí 10 Gbps. Detailní popis všech vstupních parametrů programu je uveden v příloze B. Zpracování vstupních parametrů je provedeno pomocí knihovny `getopt`.

6.2 Rozdělení na procesy

Abychom dosáhli požadovaného výkonu aplikace, je nutné zpracování paketů rozdělit na více procesů. Dle návrhu v předchozí kapitole má každý proces na starosti celé zpracování a plánování paketů a procesy se liší jen pakety, které zpracovávají. Naše aplikace mapuje jeden proces na jedno jádro procesoru. Počet použitých jader a tedy i počet procesů aplikace se určuje pomocí EAL parametru *-c COREMASK*, kterým určíme, která jádra procesoru chceme v aplikaci použít. Jako speciální případ lze aplikaci povolit jen jedno jádro procesoru a aplikace tedy poběží v jednoprocesovém režimu, který ale značně omezí její výkon.

Pro rovnoměrné rozložení příchozích paketů a jejich odeslání na výstupní port musíme inicializovat počet vstupních front vstupního rozhraní a počet výstupních front výstupního rozhraní podle počtu procesů aplikace. Tento úkon se děje v hlavním procesu aplikace ještě před spuštěním smyčky zpracování paketů na všech ostatních procesech. Hlavní proces před aktivací ostatních procesů provede inicializaci EAL vrstvy pomocí funkce frameworku DPDK `rte_eal_init()`, která uvede ostatní dostupná jádra procesoru do vyčkávacího stavu. Poté hlavní proces provede inicializaci samotných vstupních a výstup-

ních portů, jejich front a provede rezervaci paměťových struktur pro dané dvojice portů ve velkých paměťových stránkách zavoláním funkce `app_init()`. Po dokončení všech nutných inicializací je spuštěna hlavní zpracovávácí smyčka aplikace na všech dostupných jádrech procesoru včetně jádra s hlavním procesem pomocí volání funkce frameworku DPDK `rte_eal_mp_remote_launch(app_main_loop, NULL, CALL_MASTER)`. Funkce `app_main_loop` provádí cyklické zpracovávání a plánování paketů a parametr `CALL_MASTER` specifikuje, aby se tato funkce zavolala i na hlavním procesu.

Pakety na jednotlivá jádra rozřazujeme podle toho, do kterého datového toku patří. Datový tok paketu určujeme podle pětice následující údajů – zdrojová IP adresa, cílová IP adresa, transportní protokol, zdrojový port a cílový port. Pro rovnoměrné rozřazení datových toků mezi jednotlivá jádra používáme technologii RSS představenou v kapitole 5.1, která již na síťovém rozhraní vypočte hash hodnotu pro daný paket a přes tzv. Indirection table určí, na kterou vstupní frontu paket poslat. Nastavení výpočtu hash hodnoty přes RSS provedeme při inicializaci vstupního rozhraní ve funkci `app_init()`. Ve struktuře frameworku DPDK `rte_eth_conf` povolíme technologii RSS zadáním následujících údajů:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .mq_mode = ETH_MQ_RX_RSS,
    },
    .rx_adv_conf = {
        .rss_conf = {
            .rss_key = hash_key,
            .rss_hf = ETH_RSS_PROTO_MASK,
        },
    },
};
```

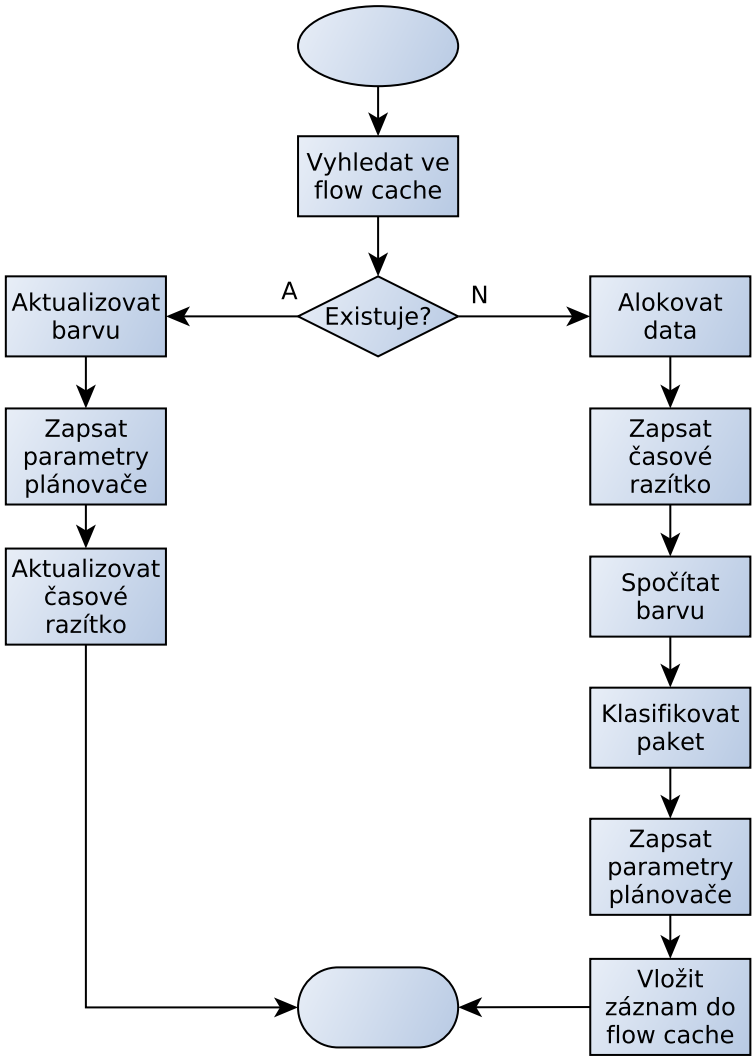
Nastavení `mq_mode` aktivuje na vstupním rozhraní technologii RSS. Nastavení `rss_hf` řekne technologii RSS, aby počítala hash hodnotu paketu podle těchto polí v hlavičce paketu – zdrojová IP adresa, cílová IP adresa, transportní protokol, zdrojový port a cílový port. Dále toto nastavení zajistí počítání hash hodnoty pro pakety IPv4 i IPv6.

Nevýhodou výchozího nastavení RSS je, že spočítaná hash hodnota paketu není stejná pro opačné toky. To jsou toky, které mají zrcadlově prohozené zdrojové a cílové IP adresy a zdrojové a cílové porty. Jsou to tedy toky jednoho spojení v opačném směru. Abychom dosáhli počítání stejných hash hodnot pro opačné toky, čemuž říkáme symetrické RSS, musíme upravit vstupní klíč funkce na výpočet hash hodnot. S řešením přišli S. Woo a K. Park ve svém článku o symetrickém RSS [24]. Jejich vstupní klíč zajišťuje počítání stejných hash hodnot pro opačné toky a zároveň zachovává rovnoměrné rozložení datových toků mezi jednotlivá jádra procesoru. Hodnota `hash_key` je následující:

```
#define RSS_HASH_KEY_LENGTH 40
static uint8_t hash_key[RSS_HASH_KEY_LENGTH] = {
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A,
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A,
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A,
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A,
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A,
};
```

6.3 Klasifikace paketu

Pakety jsou jednotlivými procesy ze vstupních front vybírány po dávkách, jejichž velikost je možné nastavit aplikačním parametrem `--bsz`. Výchozí hodnotou je 64 paketů. Manipulace s pakety je možná díky struktuře DPDK frameworku `rte_mbuf`. Tato struktura obsahuje samotný paket, k jehož částem můžeme přistupovat pomocí ukazatele `buf_addr` a offsetu do dané části paketu. Obsahuje ale také dodatečné informace pro zpracování paketu frameworkem DPDK jako například vstupní parametry pro plánovač nebo hash hodnotu vygenerovanou technologií RSS pro daný paket. Každý paket dávky je zpracován a klasifikován aplikační funkcí `process_packet`, jejíž průchod je znázorněn v diagramu na obrázku 6.1.



Obrázek 6.1: Diagram průchodu paketu funkcí `process_packet`

6.3.1 Tabulka datových toků

Tabulku datových toků (flow cache) jsme si představili v kapitole 5.2. Je implementována jako hash tabulka pomocí knihovny `rte_hash` frameworku DPDK. Jako klíč pro výpočet hash hodnoty do tabulky je použita hash hodnota paketu vypočítaná technologií RSS, ke které můžeme pro daný paket přistoupit přes položku `.hash.rss` struktury `rte_mbuf`. Pro výpočet hashe jsme vybrali hashovací funkci *jhash* implementovanou v DPDK. Ke každému klíči je do tabulky datových toků přidán ukazatel na datovou strukturu `pkt_sched_data`, která obsahuje informace potřebné pro zařazení paketu do plánovače. Tuto strukturu si popíšeme v kapitole 6.3.3.

Na začátku zpracování každého paketu zkusíme pomocí DPDK funkce `rte_hash_lookup_data` vyhledat odpovídající záznam v tabulce datových toků. Pokud je nalezen záznam odpovídající danému paketu, jsou pro vstupní parametry plánovače, které musíme zapsat do struktury `rte_mbuf` daného paketu, použity hodnoty ze záznamu tabulky. I přes nalezení záznamu v tabulce datových toků musíme aktualizovat barvu paketu a časové razítko. O obarvování paketu se dočteme více v kapitole 6.3.2. Časové razítko paketu značí, kdy jsme naposledy zpracovávali paket určitého datového toku. Toto razítko slouží jako prevence zahlcení tabulky datových toků v průběhu času. Každých 5 minut běhu aplikace všechny procesy zavolají funkci `clear_flow_cache` na svoji tabulku datových toků, která tabulku projde a vymaže z ní záznamy, které nebyly aktualizovány více než 5 minut. Časové údaje získáváme pomocí DPDK funkce `rte_rdtsc`, která čte TSC registr procesoru a vrací počet cyklů procesoru od startu systému.

V případě, že v tabulce datových toků není nalezen záznam odpovídající zpracovávanému paketu, musí paket kromě obarvení a zápisu časového razítka projít klasifikačním procesem, který určí vstupní parametry paketu pro plánovač. Nakonec je pro datový tok tohoto paketu vytvořen nový záznam v tabulce datových toků, aby se urychlilo zpracování dalších paketů stejného datového toku.

6.3.2 Barva paketu

Barvu paketu určujeme kvůli prevenci zahlcení plánovače. Plánovač frameworku DPDK podporuje mechanismus prevence zahlcení WRED, který jsme popisovali v kapitole 2.7. Tento mechanismus způsobuje preventivní zahazování paketů podle aktuální délky fronty dříve, než dojde k jejímu úplnému zahlcení. Aby byl nicméně tento mechanismus ve frameworku DPDK aktivní, je potřeba nastavit parametr *CONFIG_RTE_SCHED_RED* v konfiguračním souboru frameworku *.config* již před jeho překladem. Postup tohoto nastavení můžeme najít v dokumentaci DPDK [5].

Barvení paketu je implementováno pomocí modelu *srTCM color blind* [13]. Tento model definuje 3 možné barvy paketu jako semafor – zelenou, žlutou a červenou. Každý datový tok si udržuje stav svojí fronty v plánovači a obarvuje podle něj příchozí pakety. Stav fronty je udržován pomocí struktury `rte_meter_srtcm` z DPDK knihovny `rte_meter`. Funkce této knihovny `rte_meter_srtcm_color_blind_check` poté určí barvu každého příchozího paketu pro daný datový tok. Zelenou barvu paket obdrží, pokud zahlcení fronty nepřesáhlo nastavenou minimální úroveň. Žlutou barvu paket obdrží, když se délka fronty pohybuje mezi minimální a kritickou úrovní zahlcení. Červeně se pakety obarvují po přesáhnutí kritické úrovně zahlcení fronty. Tyto pakety plánovač vždy zahazuje.

6.3.3 Klasifikace

Pokud není pro paket nalezen záznam v tabulce datových toků, musí projít procesem klasifikace. Během tohoto procesu musejí být inicializovány položky v následující struktuře:

```
struct pkt_sched_data {
    uint32_t subport;
    uint32_t pipe;
    uint32_t traffic_class;
    uint32_t queue;
    uint32_t color;
    uint64_t timestamp;
    struct rte_meter_srtcm meter;
};
```

Položky `color` a `meter` slouží k nastavení barvy paketu, které jsme popsali v kapitole 6.3.2. Položka `timestamp` slouží k nastavení časového razítka, které jsme popsali v kapitole 6.3.1. Zbylé položky struktury slouží k nastavení nutných vstupních parametrů paketu pro plánovač. Položka `subport` definuje v plánovači námi určenou skupinu uživatelů. Položkou `pipe` poté v plánovači určíme konkrétního uživatele. Položka `traffic_class` rozlišuje v plánovači mezi různými typy síťového provozu. A nakonec položka `queue` určuje výslednou frontu paketu. Tyto 4 položky a hodnotu položky `color` je nutné zapsat do DPDK struktury paketu `rte_mbuf` ještě předtím, než předáme paket plánovači. Detailnější popis významu vstupních parametrů plánovače najdeme v kapitole 6.4.

Hodnoty vstupních parametrů plánovače určujeme podle údajů v hlavičkách L3 a L4 vrstvy paketu. K těmto údajům se dostáváme pomocí položky `buf_addr` struktury `rte_mbuf` pro daný paket, ve které se pohybujeme pomocí offsetů. Aplikace si musí být schopna poradit s pakety IPv4 i IPv6. Klasifikátor tedy nejdříve určí IP verzi paketu a podle ní poté vyhledává nutné údaje v hlavičkách paketu. Musíme totiž mít na paměti, že L3 hlavičky IPv4 a IPv6 paketů se liší [22] [4].

Položku `subport` určíme podle začátku cílové IP adresy paketu. Konkrétně vybereme prvních 16 bitů adresy a provedeme logický součin s maximálním počtem subportů plánovače změněným o jedničku. Podobně určíme i položku `pipe` definující konkrétního uživatele. Zde ale vybereme koncových 16 bitů cílové IP adresy paketu. S těmito bity poté obdobně provedeme logický součin s maximálním počtem uživatelů plánovače zmenšeným o jedničku.

Nejkomplexnější částí klasifikace je určení položky `traffic_class`. Určení typu síťového provozu provedeme pomocí šestibitového pole DSCP v hlavičce L3 vrstvy. RFC 2474 rozlišuje 8 tříd typu provozu [20]. Plánovač frameworku DPDK má ale napevno nastaveny pouze 4 třídy provozu [5]. Museli jsme tedy některé třídy provozu definované v RFC 2474 spojit dohromady. Výsledné rozdělení do tříd plánovače je uvedeno v tabulce 6.1.

Položku `queue` určíme podle cílového portu paketu z hlavičky L4 vrstvy. S hodnotou portu opět provedeme logický součin s počtem front na třídu provozu zmenšeným o jedničku.

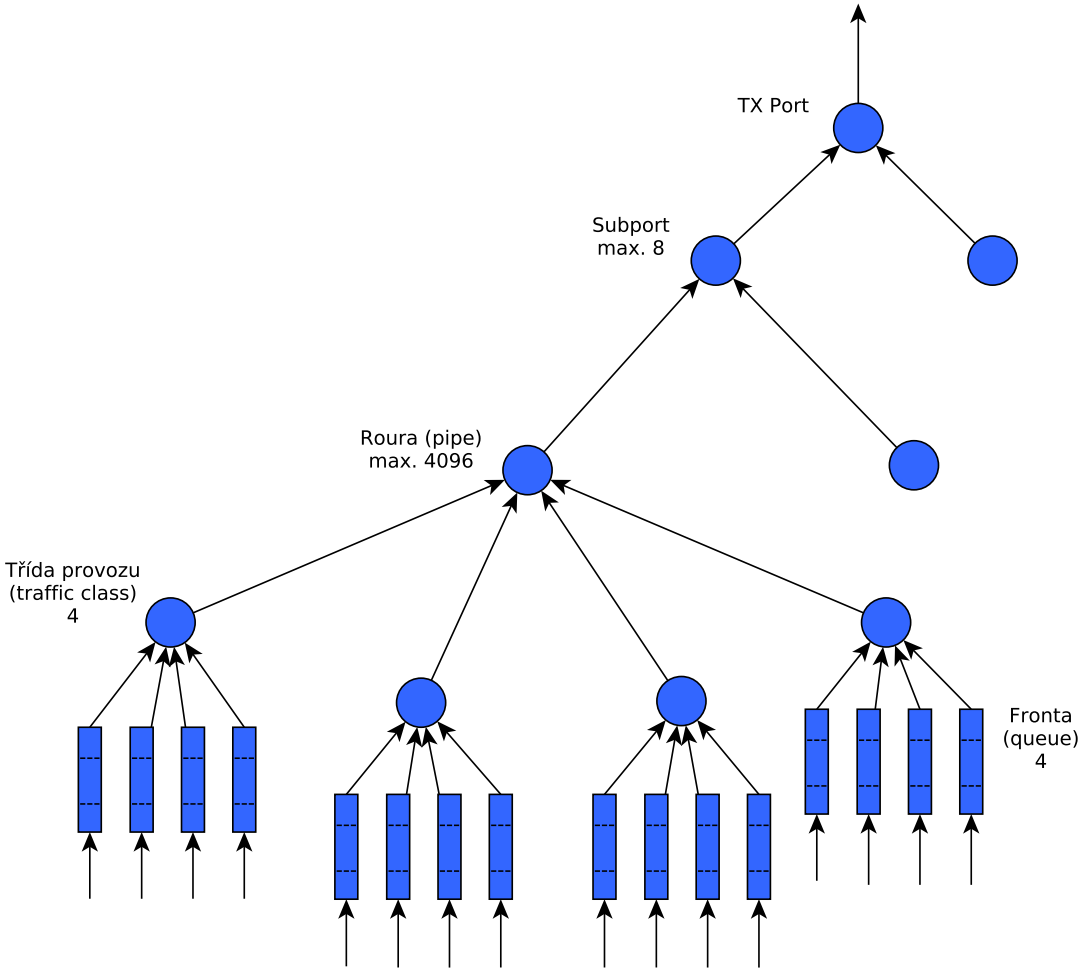
6.4 Plánovač

Jako plánovač aplikace jsme použili plánovač dostupný v knihovně `rte_sched` frameworku DPDK. Tento plánovač je implementován jako hierarchický model zásobníku žetonů podobný linuxové plánovací disciplíně HTB představené v kapitole 3.2.1. Pro každé výstupní rozhraní implementuje plánovač 4 plánovací vrstvy. Každá vrstva obsahuje jiný plánovací

DSCP	Binární hodnota	Třída plánovače
CS0 (nejnižší priorita)	000 xxx	3
CS1	001 xxx	2
CS2	010 xxx	2
CS3	011 xxx	1
CS4	100 xxx	1
CS5	101 xxx	0
CS6	110 xxx	0
CS7 (nejvyšší priorita)	111 xxx	0

Tabulka 6.1: Třídy síťového provozu

mechanismus a je ji možné samostatně nastavit. Paket se do plánovače zařazuje zavoláním funkce frameworku DPDK `rte_sched_port_enqueue`. Schéma plánovače je znázorněno na obrázku 6.2.



Obrázek 6.2: Schéma plánovače provozu v DPDK [5]

Nejvyšší vrstvou plánovače je *subport*, který reprezentuje skupinu uživatelů sítě. V plánovači lze nastavit maximálně 8 různých subportů. Tato vrstva je implementována jako zásobník žetonů. Druhou vrstvou plánovače je *pipe* neboli roura. Roury reprezentují konkrétního uživatele sítě a stejně jako vrstva *subport* jsou implementovány jako zásobník žetonů. Maximální počet rour na jeden subport je 4096.

Třetí vrstvu tvoří *traffic class* neboli třídy provozu. Každá roura plánovače má přesně 4 třídy provozu. Protože některé třídy provozu by měly mít vždy přednost před ostatními, je vrstva *traffic class* implementována jako striktní prioritní fronty popsané v kapitole 2.2. Fronta s vyšší prioritou má tedy vždy přednost před frontou s nižší prioritou. Implementován je model půjčování, kdy třídy s nižší prioritou mohou využít část pásma fronty s vyšší prioritou, pokud ji tato fronta zrovna nepoužívá.

Nejnižší vrstvou plánovače je *queue* neboli fronta. Každá třída provozu má přesně 4 fronty, které jsou obsluhovány mechanismem váhových front popsaným v kapitole 2.4. Ve výchozím stavu mají všechny 4 fronty nastaveny stejnou váhu a pakety jsou z nich odebírány rovnoměrně.

Aplikace ve výchozím stavu nastaví plánovač na průchod veškerého síťového provozu maximální rychlostí 10 Gbps. Uživatel si může nastavení plánovače přizpůsobit zadáním konfiguračního souboru při spuštění aplikace přes aplikační parametr `--cfg`. V tomto konfiguračním souboru je možné podrobně nastavit všechny vrstvy plánovače. Pro vrstvy *subport* a *pipe* můžeme nastavit parametry zásobníku žetonů jako velikost zásobníku nebo obnovovací frekvenci žetonů. Ve vrstvě *traffic class* můžeme nastavit různou šířku pásma pro jednotlivé třídy provozu a povolit, nebo zakázat model půjčování. Vrstvě *queue* můžeme nastavit různé váhy pro jednotlivé fronty. V konfiguračním souboru také můžeme podrobně nastavit parametry pro mechanismus prevence zahlcení WRED. Podrobný popis nastavení konfiguračního souboru najdeme v dokumentaci frameworku DPDK [6].

6.5 Odeslání paketu

Pakety k odeslání na výstupní rozhraní získáme z plánovače pomocí funkce frameworku DPDK `rte_sched_port_dequeue`. Pakety jsou z plánovače získávány po dávkách. Výchozí velikost dávky je 64 paketů. Její velikost můžeme nastavit při spuštění aplikace pomocí aplikačního parametru `--bsz`. Následně jsou pakety zařazeny do výstupní fronty rozhraní. V této fázi již nemůžeme pakety zahodit. Dávka paketů je tedy vkládána do výstupní fronty, dokud nejsou přijaty všechny její pakety. Každý proces aplikace má svoji výstupní frontu. Výstupní síťové rozhraní si cyklicky vybírá pakety ze všech svých front.

Kapitola 7

Testování

Tato kapitola se zabývá výkonnostními testy naší aplikace. Cílem této práce bylo vytvořit plánovač síťového provozu, který bude schopen plánovat provoz o rychlosti až 10 Gbps. Jako testovací prostředí posloužila síťová laboratoř L311 na Fakultě informačních technologií VUT v Brně. Síťový provoz byl simulován pomocí generátoru síťového provozu Spirent TestCenter SPT-2000A. Toto zařízení umožňuje generování síťového provozu o námi požadované rychlosti 10 Gpbs. Můžeme pomocí něho také zaznamenávat statistky o příchozím síťovém provozu [23]. Zařízení Spirent tedy bylo použito jak pro generování síťového provozu, tak pro měření výstupních statisti naší aplikace.

Samotná aplikace byla spuštěna na serveru *young-lady*, na kterém byl nainstalován framework DPDK ve verzi 17.02. Konfigurace tohoto serveru byla následující:

CPU: Intel Xeon E5-2620@2.00 GHz

RAM: 32 GB DDR3@1600 MHz

Operační systém: Debian 8.0 "Jessie", jádro Linux 4.9.13

NIC: Intel X-520-2, 2x 10 Gb

Dostupná síťová karta obsahovala dva 10 Gb porty, z nichž jeden byl použit jako vstupní port aplikace a druhý jako výstupní port aplikace. Celkové schéma zapojení serveru *young-lady* a generátoru síťového provozu Spirent nalezneme v příloze C. Procesor na serveru *young-lady* obsahoval 12 logických jader, naši aplikaci jsme tedy mohli otestovat v až dvanáctiprocesovém režimu. Pro běh aplikace bylo alokováno 8 GB paměti RAM ve formě velkých paměťových stránek.

Testování probíhalo ve 3 různých módech aplikace – jednoprocesovém, šestiprocesovém a dvanáctiprocesovém. Pro porovnání jsme ve stejných podmínkách otestovali i linuxový nástroj *tc*. Aby bylo srovnání co nejrelevantnější, nastavili jsme nástroj *tc* co nejpodobněji struktuře našeho plánovače. Plánování paketů tedy probíhalo ve 4 vrstvách. První 2 vrstvy simulující DPDK vrstvy *subport* a *pipe* byly vytvořeny pomocí plánovací disciplíny HTB. Třetí vrstva simulující DPDK vrstvu *traffic class* byla vytvořena pomocí plánovací disciplíny PRIO. A nakonec spodní vrstva *queue* byla simuována pomocí plánovací disciplíny SFQ. Všechny tyto plánovací disciplíny, stejně jako nástroj *tc*, byly představeny v kapitole 3.

Klíčovým údajem pro měření výkonnosti aplikace je počet paketů, které aplikace zvládne za jednotku času zpracovat. Pro každý paket totiž musí aplikace aktivovat zpracovávací funkci. Všechny pakety v síťovém provozu ale nemají stejnou velikost. Proto jsme zvolili jako

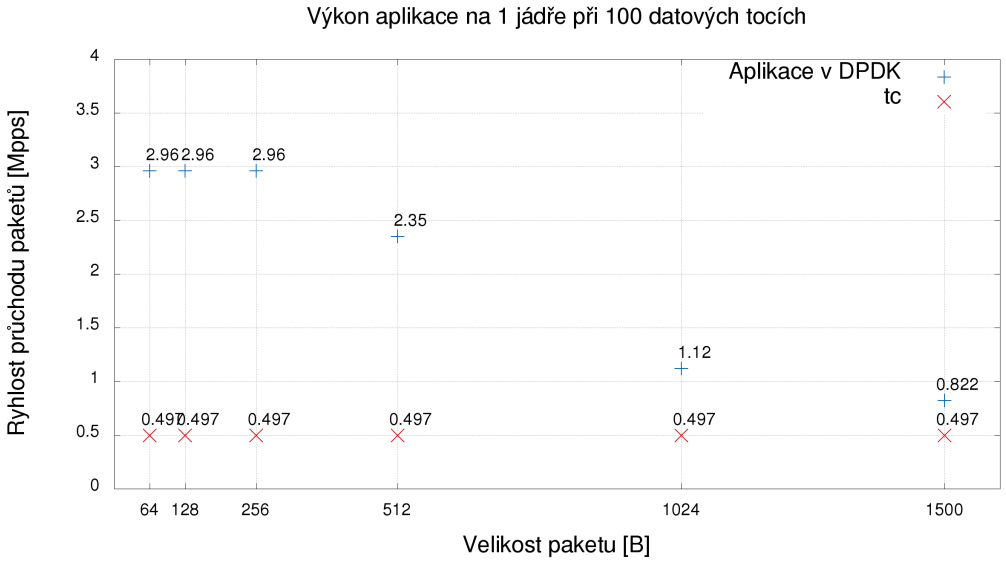
jednotku pro měření výkonnosti aplikace *Milion paketů za sekundu [Mpps]*. Tato jednotka vyjadřuje počet paketů, které prošly bodem v síti za jednotku času. Počet průchozích paketů různých velikostí za jednotku času v síti o šířce pásma 10 Gbps ukazuje tabulka 7.1.

Velikost paketu [B]	Počet paketů za sekundu [Mpps]
64	14,88
128	8,45
256	4,53
512	2,35
1024	1,12
1500	0,822

Tabulka 7.1: Počet průchozích paketů za sekundu při rychlosti linky 10 Gbps

Pro všech 6 zobrazených velikostí paketů jsme provedli testování naší aplikace a nástroje *tc* v jednoprocesovém, šestiprocesovém a dvanáctiprocesovém režimu. Všechny testy jsme nejprve provedli pro provoz se 100 různými datovými toky současně. Poté jsme celou sadu testů provedli nad provozem s 1000 různými datovými toky současně. Rozdíly mezi těmito sadami testů byly minimální, proto si v této kapitole rozebereme pouze výsledky pro testy se 100 různými datovými toky. Výsledky testů s 1000 různými datovými toky nalezneme v příloze D.

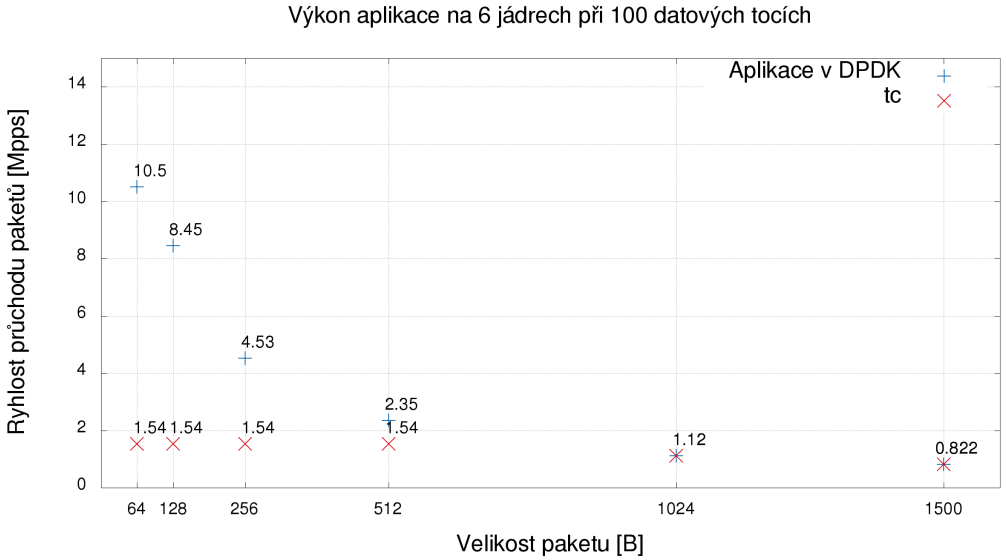
Výsledky testů v jednoprocesovém režimu jsou zobrazeny v grafu 7.1. Z grafu můžeme vyčíst, že naše aplikace překonává linuxový nástroj *tc* v testech všech velikostí paketů. Nástroj *tc* nedokáže zpracovat všechny pakety na lince ani u největších paketů o velikosti 1500 B. Naproti tomu naše aplikace zvládá zpracovat všechny pakety linky od velikosti 512 B. Maximální výkon jednoprocesového režimu nehledě na velikost paketů je 2,96 Mpps.



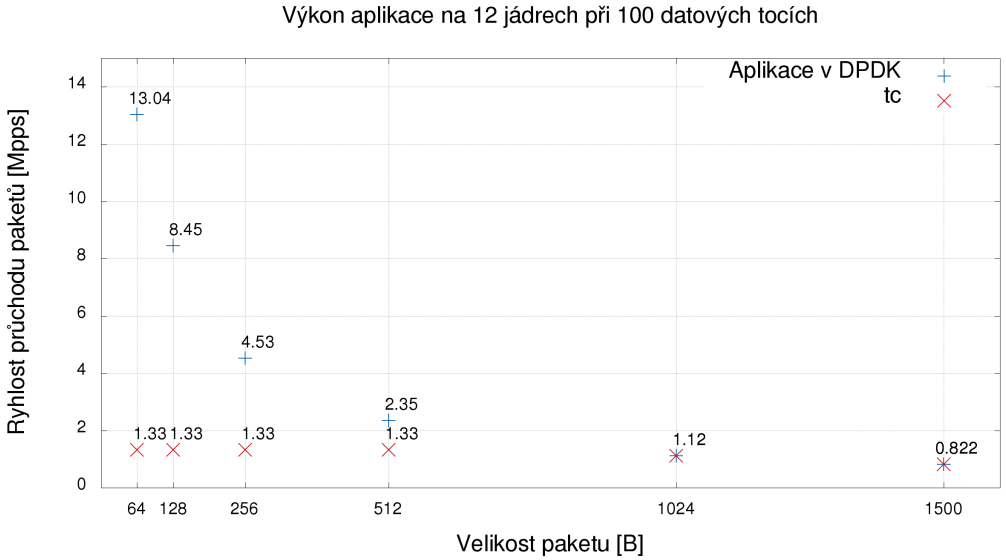
Obrázek 7.1: Výsledky testů pro 100 datových toků na 1 jádře

Výsledky testů v šestiprocesovém a dvanáctiprocesovém režimu najdeme v grafech 7.2 a 7.3. Výsledky jsou si velice podobné a narážíme zde už na maximální limit výkonu aplikace. Linuxový nástroj *tc* opět zaostává za naší aplikací a ani při vícejádrovém zpracování

nedokáže obsloužit všechny pakety na lince při velikosti paketů menší než 1024 B. Naproti tomu naše aplikace si neporadí se všemi pakety na lince jen v případě paketů o velikosti 64 B. Tyto pakety tvoří nejmenší možné Ethernetové rámce a v reálných situacích se s provozem paketů o velikosti pouze 64 B nesetkáme [17].



Obrázek 7.2: Výsledky testů pro 100 datových toků na 6 jádrech



Obrázek 7.3: Výsledky testů pro 100 datových toků na 12 jádrech

Kapitola 8

Možná rozšíření práce

V předchozí kapitole jsme se zaměřili především na test maximálního výkonu našeho plánovače. Úkolem plánovače síťového provozu je ale také správné rozložení provozu podle požadavků uživatele. V tomto ohledu jsme provedli několik testů, které odhalily malý nedostatek našeho plánovače. Pro každý proces aplikace je vytvořen samostatný plánovač, který je nastaven podle uživatelem dodaného konfiguračního souboru. Omezení provozu pro konkrétního uživatele probíhá správně, protože všechen tento provoz bývá přidělen pomocí technologie RSS jednomu procesu aplikace.

Problém nastává při omezení celkového provozu proudícího do aplikace. Maximální požadovaná rychlost celkového provozu je totiž nastavena plánovačům v každém procesu aplikace. Skutečná maximální propustnost plánovače se tedy znásobí podle počtu procesů aplikace. Možným řešením pro budoucí implementaci by bylo přidat jeden centrální plánovač, který by běžel v samostaném procesu. Tento plánovač by sbíral pakety ze všech ostatních plánovačů a určoval by, které pakety půjdou na výstupní rozhraní. Otázkou je, nakolik by tato úprava ovlivnila výkon aplikace.

Dalším možným rozšířením by mohla být implementace dynamické rekonfigurace klasifikátoru aplikace a parametrů plánovače. Způsob klasifikace paketů je v tuto chvíli napevno implementován v kódu aplikace. Parametry plánovače se nastavují zadáním konfiguračního souboru při startu aplikace. Možnost tato nastavení upravovat za běhu aplikace dle svých potřeb by zlepšila uživatelskou přívětivost aplikace.

Kapitola 9

Závěr

Cílem této práce bylo vytvořit plánovač síťového provozu pro vysokorychlostní sítě o šířce pásma až 10 Gbps. V úvodních kapitolách byly představeny obecné mechanismy plánování síťového provozu a implementace těchto mechanismů v operačním systému Linux pomocí nástroje *tc*. Tento nástroj byl později použit pro porovnání výkonu námi implementovaného plánovače.

Plánovač představený v této práci je implementován v programovacím jazyku C za použití frameworku DPDK. Tento framework poskytuje knihovny a ovladače pro rychlé zpracování paketů v uživatelském prostoru systému. V návrhu aplikace byl představen nejen plánovač paketů, ale také mechanismy pro rychlé předzpracování a klasifikaci paketů. Implementace plánovače byla realizována s důrazem na maximální výkon a jednoduchost. Pro dosažení maximálního výkonu je třeba aplikaci spustit na co nejvíce jádrech procesoru.

Výsledky testování ukázaly, že plánovač je schopen dosáhnout výkonu požadovaného v zadání práce. Aplikace výrazně překonala výkon linuxového nástroje *tc*. Omezením pro aplikaci může být nedostatečný výkon hardware, na kterém je spuštěna.

Přínosem této práce je vytvoření funkčního plánovače síťového provozu pro vysokorychlostní sítě ve frameworku DPDK. Tento plánovač může být dále použit jako součást komplexních nástrojů pro správu vysokorychlostní sítě. Aplikaci je možné dále rozšiřovat, aby bylo dosaženo větší uživatelské přívětivosti. Možným rozšířením v tomto směru by byla implementace dynamické rekonfigurace klasifikátoru paketů a parametrů plánovače pro rozložení provozu.

Literatura

[1] Almquist, P.: *Type of Service in the Internet Protocol Suite*. RFC 1349, Červenec 1992, [Online; navštíveno 13.05.2017].
URL https://www.rfc-editor.org/rfc/rfc1349.txt

[2] Benvenuti, C.: *Understanding Linux network internals*. Sebastapol, CA: O'Reilly, c2006, ISBN 978-0-12-088588-6.

[3] Brown, M. A.; Bolelli, F.; Patriciello, N.: *Traffic Control HOWTO*. 2006, [Online; navštíveno 13.05.2017].
URL http://tldp.org/en/Traffic-Control-HOWTO/

[4] Deering, S.; Hinden, R.; Cisco; aj.: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460, Prosinec 1998, [Online; navštíveno 15.05.2017].
URL https://www.rfc-editor.org/rfc/rfc2460.txt

[5] DPDK: *23. Quality of Service (QoS) Framework - Data Plane Development Kit 17.05.0 documentation*. 2017, [Online; navštíveno 14.05.2017].
URL http://dpdk.org/doc/guides/prog_guide/qos_framework.html

[6] DPDK: *26. QoS Scheduler Sample Application - Data Plane Development Kit 17.05.0 documentation*. 2017, [Online; navštíveno 14.05.2017].
URL http://dpdk.org/doc/guides/sample_app_ug/qos_scheduler.html

[7] DPDK: *2. Overview - Data Plane Development Kit 17.05.0 documentation*. 2017, [Online; navštíveno 13.05.2017].
URL http://dpdk.org/doc/guides/prog_guide/overview.html

[8] DPDK: *3. Environment Abstraction Layer - Data Plane Development Kit 17.05.0 documentation*. 2017, [Online; navštíveno 13.05.2017].
URL http://dpdk.org/doc/guides/prog_guide/env_abstraction_layer.html

[9] DPDK: *4. Compiling and Running Sample Applications - Data Plane Development Kit 17.05.0 documentation*. 2017, [Online; navštíveno 13.05.2017].
URL http://dpdk.org/doc/guides/linux_gsg/build_sample_apps.html

[10] DPDK: *7. Poll Mode Driver - Data Plane Development Kit 17.05.0 documentation*. 2017, [Online; navštíveno 15.05.2017].
URL http://dpdk.org/doc/guides/prog_guide/poll_mode_drv.html

[11] DPDK: *Supported NICs*. 2017, [Online; navštíveno 15.05.2017].
URL http://dpdk.org/doc/nics

[12] Foundation, T. L.: *iproute2*. Červenec 2016, [Online; navštíveno 13.05.2017].
URL https://wiki.linuxfoundation.org/networking/iproute2

[13] Heinanen, J.; Guerin, R.; Finland, T.: *A Single Rate Three Color Marker*. RFC 2697, Září 1999, [Online; navštíveno 14.05.2017].
URL https://www.rfc-editor.org/rfc/rfc2697.txt

[14] Herbert, T.; de Bruijn, W.: *Scaling in the Linux Networking Stack*. 2017, [Online; navštíveno 13.05.2017].
URL https://www.kernel.org/doc/Documentation/networking/scaling.txt

[15] Hubert, B.: *tc - show / manipulate traffic control settings*. Prosinec 2001, [Online; navštíveno 13.05.2017].
URL http://lartc.org/manpages/tc.txt

[16] Hubert, B.; Graf, T.; Maxwell, G.; aj.: *Linux Advanced Routing & Traffic Control HOWTO*. 2012, [Online; navštíveno 13.05.2017].
URL http://lartc.org/howto/

[17] Kurose, J. F.; Ross, K. W.: *Computer networking: a top-down approach*. Boston: Pearson, 6 vydání, 2013, ISBN 978-0-13-285620-1.

[18] Matoušek, P.: *Síťové služby a jejich architektura*. Publishing house of Brno University of Technology VUTIUM, 2014, ISBN 978-80-214-3766-1.

[19] Medhi, D.; Ramasamy, K.: *Network routing: algorithms, protocols, and architectures*. Boston: Elsevier/Morgan Kaufmann Publishers, c2007, ISBN 978-0-12-088588-6.

[20] Nichols, K.; Blake, S.; Baker, F.; aj.: *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474, Prosinec 1998, [Online; navštíveno 13.05.2017].
URL https://www.rfc-editor.org/rfc/rfc2474.txt

[21] Piat, F.; Capper, S.; aj.: *Hugepages - Debian Wiki*. Srpen 2015, [Online; navštíveno 13.05.2017].
URL https://wiki.debian.org/Hugepages

[22] Postel, J.: *Internet Protocol*. RFC 791, Září 1981, [Online; navštíveno 13.05.2017].
URL https://www.rfc-editor.org/rfc/rfc791.txt

[23] Spirent Communications, I.: *Spirent TestCenter SPT-2000A, SPT-5000A AND SPT-9000A CHASSIS*. 2007, [Online; navštíveno 16.05.2017].
URL http://www.livingston-products.com/products/pdf/142602_1_en.pdf

[24] Woo, S.; Park, K.: *Scalable TCP Session Monitoring with Symmetric Receive-side Scaling*. KAIST Technical Report, 2012, [Online; navštíveno 14.05.2017].
URL http://www.ndsl.kaist.edu/~kyoungsoo/papers/TR-symRSS.pdf

Příloha A

Obsah CD

- **src** – adresář zdrojových souborů aplikace, souboru MAKEFILE a adresáře *build* se spustitelnou aplikací
- **xdolez64** – adresář obsahující technickou zprávu ve formátu pdf
- **latex** – adresář obsahující technickou zprávu ve formátu tex
- **tc** – adresář s profilem nástroje *tc* použitým pro výkonnostní testování a skriptem pro jeho vygenerování
- **README** – soubor s pokyny k instalaci a spuštění aplikace

Příloha B

Parametry pro spuštění

--pfc "RX port, TX port" Dvojice vstupní a výstupní port. Je možné nakonfigurovat několik dvojic pro jedno spuštění programu.

--cfg FILE Konfigurační soubor s parametry rozložení provozu pro plánovač

--rsz "A, B" Velikost bufferů.

- **A** Určuje velikost bufferu pro příchozí pakety pro každé jádro. Velikost se určuje počtem položek bufferu.
- **B** Určuje velikost bufferu pro odchozí pakety pro každé jádro. Velikost se určuje počtem položek bufferu.

--bsz "A, B, C, D" Velikost shluku paketů.

- **A** Určuje velikost dávky paketů, kterou každé jádro čte ze vstupního rozhraní.
- **B** Určuje velikost dávky paketů, kterou jádro zařadí do plánovače.
- **C** Určuje velikost dávky paketů, kterou si jádro vyžádá od plánovače.
- **D** Určuje velikost dávky paketů, kterou každé jádro zapíše na výstupní rozhraní.

--msz M Určuje velikost alokované paměti ve velkých paměťových stránkách pro každou dvojici vstupního a výstupního portu. Velikost se určuje v počtu struktur `rte_mbuf`. Tato struktura obsahuje všechny dostupné informace o jednom paketu.

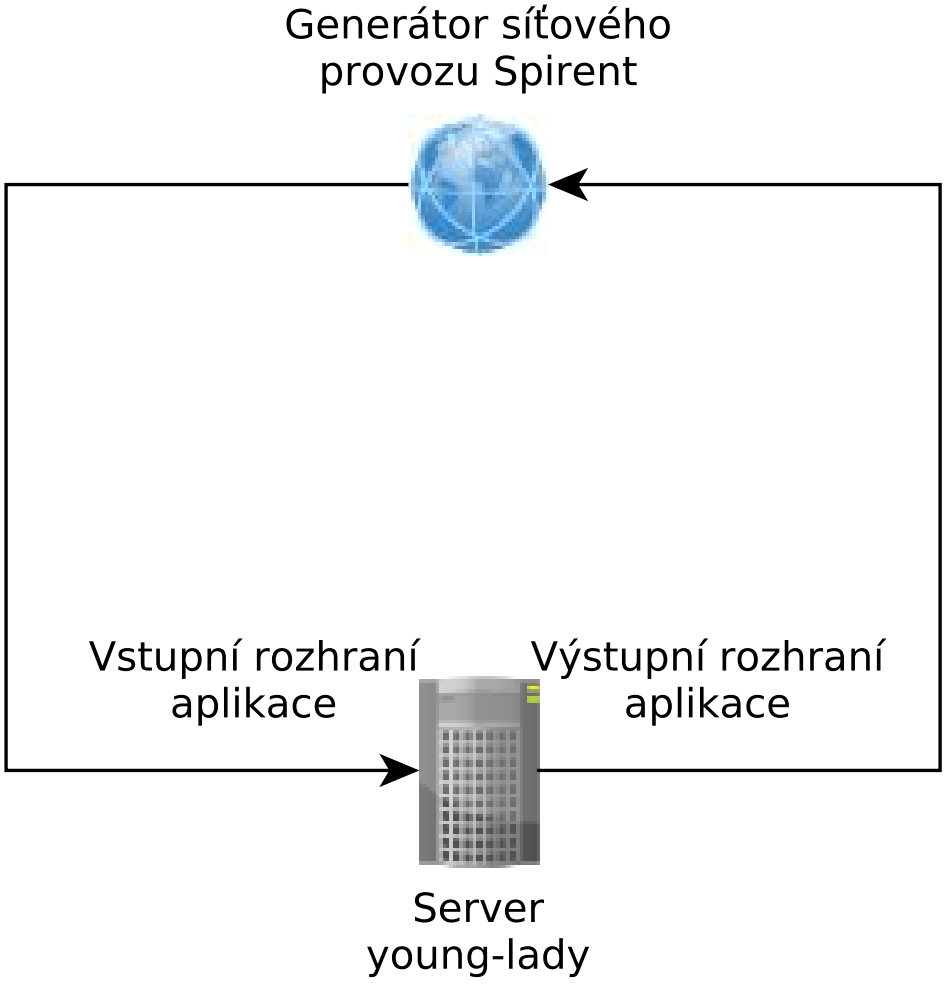
--rth "A, B, C" Nastavuje parametry vstupních front pro předzpracování paketů. Význam hodnot se lze dočíst v dokumentaci [5].

--tth "A, B, C" Nastavuje parametry výstupních front pro předzpracování paketů. Význam hodnot se lze dočíst v dokumentaci [5].

Aplikace musí být spuštěna s alespoň jedním parametrem `--pfc`. Ostatní parametry mají v aplikaci nastaveny výchozí hodnoty.

Příloha C

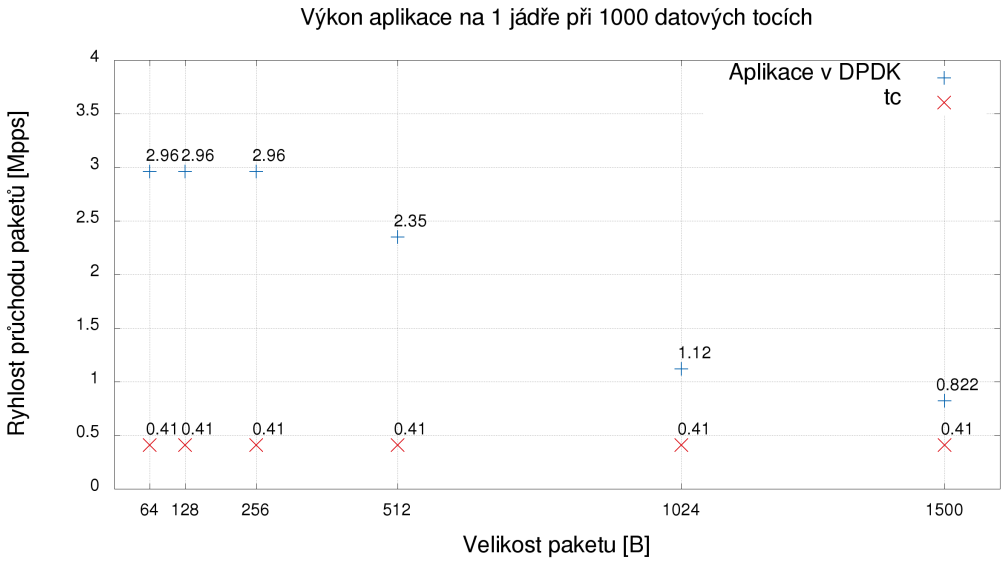
Schéma testování



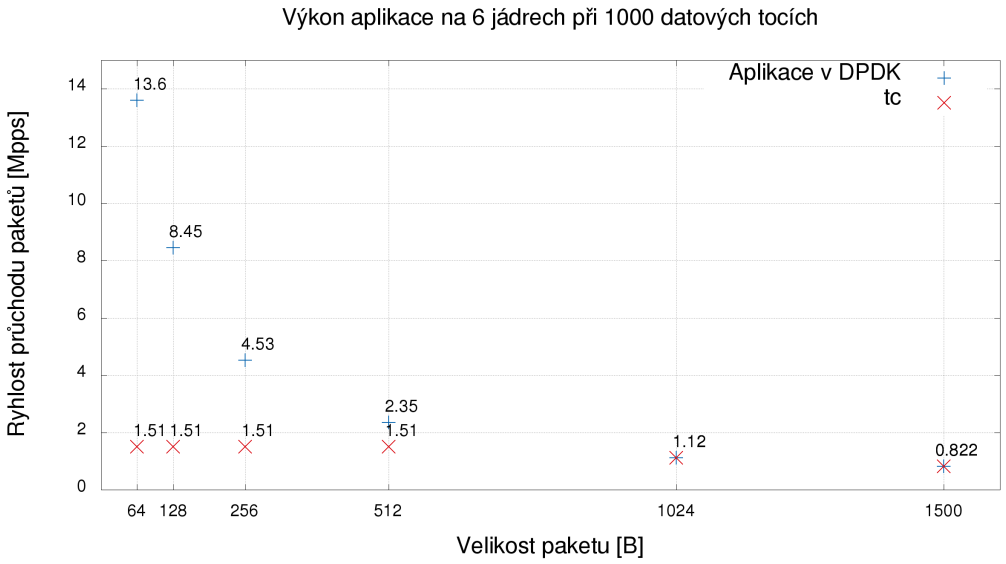
Obrázek C.1: Schéma testování aplikace

Příloha D

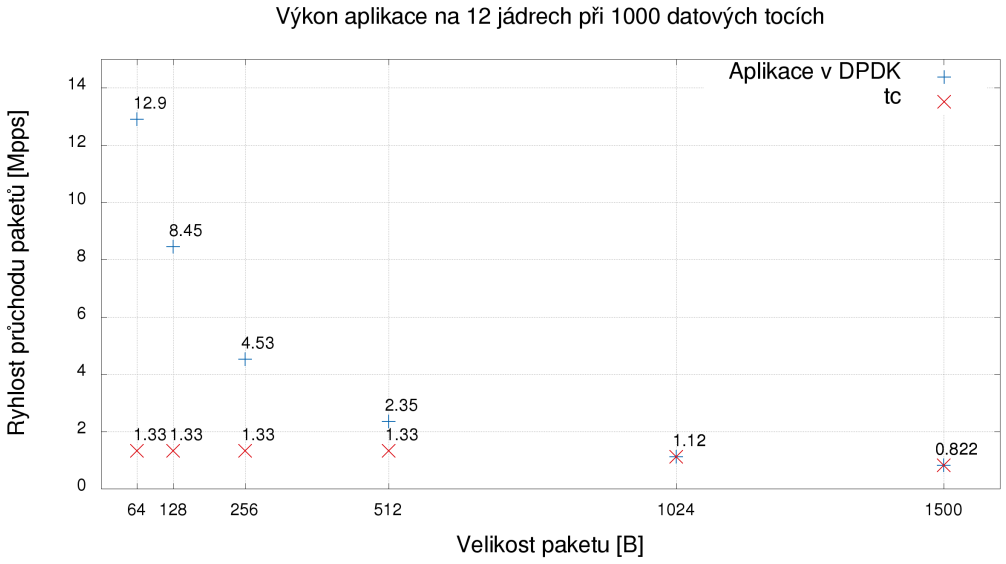
Testování aplikace nad provozem s 1000 různými datovými toky



Obrázek D.1: Výsledky testů pro 1000 datových toků na 1 jádře



Obrázek D.2: Výsledky testů pro 1000 datových toků na 6 jádrech



Obrázek D.3: Výsledky testů pro 1000 datových toků na 12 jádrech