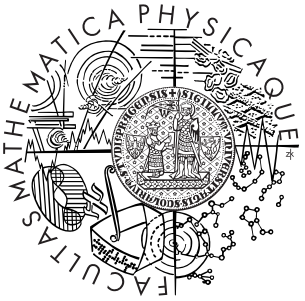
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# DIPLOMOVÁ PRÁCE




Lukáš Turek

## Řízení toku v přístupových bodech bezdrátové sítě IEEE 802.11

Katedra softwarového inženýrství

Vedoucí diplomové práce: Doc. Ing. Jan Janeček, CSc.
Studijní program: Informatika - Softwarové inženýrství

2009

Děkuji především docentu Janu Janečkovi za ochotu vést mé téma a poskytnutí mnoha cenných rad. Děkuji též RNDr. Liboru Forstovi, že zastoupil vedoucího, když onemocněl, a přeji docentu Janečkovi brzké uzdravení. Dále bych chtěl poděkovat občanskému sdružení Praha12.Net za zapůjčení hardwaru nutného pro vývoj a testování.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 4.8.2009 Lukáš Turek

# Obsah

# Seznam obrázků

# Seznam tabulek



# Seznam výpisů

Název práce: Řízení toku v přístupových bodech bezdrátové sítě IEEE 802.11
Autor: Lukáš Turek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Doc. Ing. Jan Janeček, CSc.
e-mail vedoucího: janecek@fel.cvut.cz

Abstrakt: Bezdrátové sítě založené na standardu IEEE 802.11 (Wi-Fi) se dnes často používají jako první míle pro připojení k Internetu. Standard však pro toto použití nebyl navržen a proto se tento způsob nasazení potýká s problémy. Jedním z nich je vzrůstající latence při velkém zatížení sítě, způsobená absencí řízení toku. Předložená práce ukazuje, proč algoritmy řízení toku dostupné v operačních systémech nemohou ve Wi-Fi sítích efektivně pracovat a proč ani doplněk 802.11e není řešení. Následně navrhuje algoritmus určený speciálně pro Wi-Fi sítě, který dělí přenosové pásmo rovnoměrně mezi klienty, minimalizuje latenci a přitom nesnižuje propustnost sítě. Funkce algoritmu je ověřena simulací v prostředí OMNeT++ a následně je algoritmus implementován v operačních systémech Linux a FreeBSD.

Klíčová slova: řízení toku, latence, bezdrátové sítě, Wi-Fi, 802.11

Title: Flow Control for the IEEE 802.11 Access Points
Author: Lukáš Turek
Department: Department of Software Engineering
Supervisor: Doc. Ing. Jan Janeček, CSc.
Supervisor's e-mail address: janecek@fel.cvut.cz

Abstract: Wireless networks based on IEEE 802.11 standard (Wi-Fi) are nowadays often used as a last mile of Internet connection. However the standard was not designed with this use in mind, so this kind of deployment is rife with problems. One of these problems is high latency of network under high load caused by the absence of flow control. The present work explains why flow control algorithms implemented in operating systems cannot work in Wi-Fi networks effectively and why even the 802.11e amendment yields no solution. Subsequently an algorithm designed specially for Wi-Fi networks, which divides bandwidth equally among clients and minimizes latency while not reducing network throughput, is proposed. The function of the algorithm is confirmed by simulation in OMNeT++ environment and the algorithm is then implemented in Linux and FreeBSD operating systems.

Keywords: flow control, latency, wireless, Wi-Fi, 802.11

# Kapitola 1

# Stanovení řešeného problému

## 1.1 Motivace

Český trh s vysokorychlostním připojením k Internetu je velmi specifický, nemá nejspíše obdobu nikde na světě: podle [5] má bezdrátové připojení podíl celých 30%. Další specifikum je, že na trhu působí velký počet malých ISP[1], pokrývajících třeba jen několik vesnic nebo část města [6]. Tito ISP používají takřka výlučně technologie standardu 802.11 [1], obvykle nazývané *Wi-Fi*.

Jiné technologie jako WiMax jsou pro ně nedosažitelné: přestože byl standard WiMax (IEEE 802.16) navržen i pro bezlicenční pásma, v ČR se zatím používá pouze v licencovaném pásmu 3.5 GHz, které je navíc v krajských městech již rozebrané [7]. Cena základnové stanice se navíc pohybuje v řádu stovek tisíc Kč, zatímco Wi-Fi přístupový bod stojí pouze několik tisíc Kč. Na technologii Wi-Fi je dnes možné poskytovat vysokorychlostní připojení k Internetu za cenu od 150 Kč měsíčně [8].

Wi-Fi připojení v pásmu 5 GHz (802.11a) navíc dokáže rychlostí konkurovat i ADSL: na vzdálenost 1km je možné dosáhnout reálné přenosové rychlosti okolo 24 Mbit/s[2], což odpovídá přenosové rychlosti ADSL2+ na stejnou vzdálenost [9]. Spolehlivost bude typicky nižší kvůli riziku rušení, ale to lze vyvážit nižší cenou. Vybudovat Wi-Fi přístupový bod je navíc mnohem levnější než vybudovat předsunutou telefonní ústřednu, nemluvě o ceně za položení telefonního vedení.

Technologie Wi-Fi ale nikdy nebyla určena pro poskytování připojení k Internetu, tzv. „poslední míli“. Byla navržena pro lokální sítě v rámci jedné organizace. Neposkytuje žádné záruky latence a přenosové rychlosti, ani nijak neřeší rovnoměrné rozdělení pásma mezi uživatele. Různí uživatelé mají navíc odlišné a často protichůdné požadavky na síť. Zkušenosti s provozem sítě Praha12.Net ukázaly, že tyto problémy nejsou jen teoretické, uživatelé je reálně pociťují a je proto nutné hledat řešení.

[1] *Internet service provider*, poskytovatel připojení k Internetu

[2] V celé práci budou předpony SI používány v jejich původním významu, tedy 1 Mbit = 1 000 000 bitů. Pro nominální rychlosti sítí se používají skoro výlučně násobky 1000, protože jsou odvozeny od modulačních rychlostí, daných počtem symbolů za vteřinu.

## 1.2 Terminologie

S problematikou dělení přenosového pásma mezi datové toky souvisí několik termínů, které se někdy zaměňují. V této práci budou termínu použity ve významu, který je specifikován v anglické Wikipedii:

**Flow Control**
: řízení přenosové rychlosti mezi dvěma body tak, aby odesílatel nezahltil příjemce

**Quality of Service (QoS)**
: prioritizace určitých datových toků v sítí pro zlepšení kvality služeb

**Traffic Policing**
: snižování datového toku zahazováním paketů, překračuje-li datový tok daný limit

**Traffic Shaping**
: snižování datového toku pozdržením paketů, překračuje-li datový tok daný limit

**Fair Queueing**
: algoritmus pro spravedlivé dělení přenosového pásma mezi datové toky

**Packet Scheduler**
: algoritmus rozhodující o čase a pořadí odesílání paketů

U většiny těchto anglických termínů není rozšířen český překlad. Výjimkou je *Flow Control* překládané jako „řízení toku“, které se ale používá v širším významu, zahrnujícím i *Traffic Shaping*. Proto budu dále používat především tento termín, i když v některých situacích nemusí být zcela přesný.

## 1.3 Požadavky na řízení toku

Tak jako u většiny služeb je rozhodující spokojenost zákazníka, základním kritériem kvality sítě je uspokojení potřeb uživatelů. Tyto potřeby se ale u různých uživatelů mohou lišit, a mohou být dokonce v rozporu. Podle pozorování v síti Praha12.Net jsou nejčastější tyto tři skupiny uživatelů:

1. „Brouzdal“ – uživatel, který si prohlíží web. Chce, aby se mu stránky načítaly co nejrychleji. Potřebuje tedy co největší přenosovou rychlost (velikosti stránek dnes dosahují stovek kB) a zároveň relativně nízkou latenci (načtení jedné stránky je typicky několik požadavků na server, takže se latence[3] sčítá). Jeho datový přenos je velmi nepravidelný: shluky paketů s odstupy v řádu vteřin až desítek vteřin.

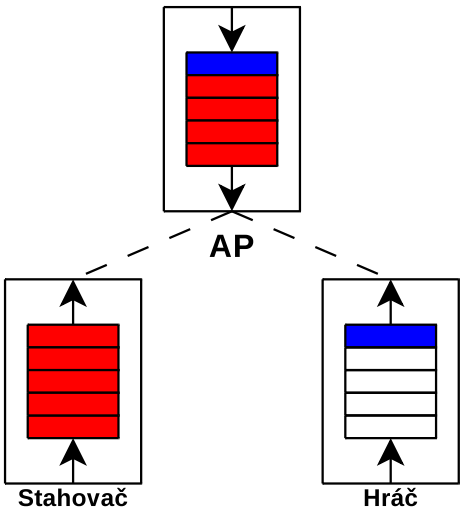
[3]Latencí je myšlena doba odezvy (*roundtrip*) mezi klientem a serverem, jak ji lze změřit příkazem `ping`. Latence způsobená přenosem po Internetu bývá nízká (do 10ms) a není možné ji ovlivnit, proto ji zanedbáme, a budeme uvažovat jen dobu přenosu od klienta k přístupovému bodu a zpět.

2. „Stahovač“ – uživatel, který souvisle stahuje velké objemy dat. Požaduje co největší přenosovou rychlost, ale latence ho příliš nezajímá. Pro síť představuje největší zátěž, při absenci řízení toku omezuje ostatní uživatele. Při nasazení dobrého algoritmu řízení toku by ale mohl využít volnou kapacitu sítě, která by jinak zůstala ladem.

3. „Hráč“ – uživatel, který hraje online hry nebo telefonuje přes Internet (VoIP). Přenáší relativně málo dat, ale vyžaduje co nejmenší latenci a jitter (rozptyl latence). Jeho nároky splňuje Wi-Fi síť bez řízení toku zdaleka nejhůře, při velkém zatížení může latence růst až do řádu vteřin (konkrétní měření budou představena v kapitole 4.2). Uspokojení tohoto typu uživatelů je při návrhu algoritmu řízení toku klíčové.

Požadavky posledních dvou uživatelů jsou naprosto odlišné, každý vyžaduje optimalizaci jiného parametru (propustnosti u Stahovače a latence u Hráče). Požadavky prvního jsou určitou kombinací obou, algoritmus tedy nemůže řešit jen oba krajní případy. Bez řízení toku jsou navíc požadavky Stahovače a Hráče v přímém rozporu: velké zatížení sítě znamená zvýšení latence. Tento rozpor vzniká v důsledku toho, jak fungují aktivní prvky Wi-Fi sítě.

Předpokládejme, že páteřní spoj mezi přístupovým bodem (AP, *Access Point*) má větší přenosovou rychlost než Wi-Fi síť[4]. AP tedy přijímá pakety z Internetu rychleji, než je dokáže odesílat klientům, a musí je ukládat do fronty. Klienti také ukládají pakety do fronty, pokud je generují rychleji, než je stíhají odesílat. Celou situaci znázorňuje obrázek 1.1:




Obrázek 1.1: Fronty paketů

Latence ve směru od klientů do Internetu (*upload*) je způsobena přístupovou metodou CSMA/CA. Klient, který chce odesílat data, detekuje, že pásmo je obsazené. Po skončení přenosu počká náhodnou dobu, než začne vysílat – za tu dobu ho

[4]To v praxi obvykle platí, páteřní spoje bývají optická vlákna nebo směrové (point-to-point) bezdrátové spoje, např. v pásmu 10GHz. Navíc pokud by byl páteřní spoj také na technologii Wi-Fi se stejnou nebo menší kapacitou, nastane na něm stejný problém jako na AP.

může „předběhnout“ někdo jiný svým přenosem. Klient po něm opět čeká, ale pouze po zbytek času, který předtím náhodně určil. Jeho přenos by se tedy měl uskutečnit v konečném čase, který ale nelze deterministicky určit. Délka fronty u Stahovače nicméně nijak neovlivňuje, kdy bude odeslán paket Hráče, všichni mají stejnou pravděpodobnost, že se jim podaří odeslat paket v daném čase. Latence v odchozím směru tedy závisí především na počtu klientů.

Větší latence vzniká ve směru z Internetu ke klientům (*download*). Jednak AP také musí soutěžit o médium v přístupové metodě CSMA/CA, a navíc je jeho fronta společná pro všechny klienty. Bez aplikace řízení toku pracuje tato fronta v režimu FIFO – pakety jsou odesílány v pořadí, v jakém přišly. Jediný paket Hráče tak čeká, než budou odeslány všechny pakety Stahovače. Latence proto závisí především na délce fronty. Bude-li ale fronta krátká a Stahovač ji zaplní, bude Hráčův paket zahozen.

Největší zátěž pro síť představují uživatelé P2P sítí: stahují z více zdrojů najednou a navíc uploadují. Někteří ISP tedy problém řeší omezováním nebo úplným blokováním P2P sítí. To má ale několik problémů:

- Omezování nebo dokonce blokování určitého druhu provozu odporuje principu síťové neutrality. Americký telekomunikační regulátor FCC zakázal společnosti Comcast blokovat Bittorent na své kabelové síti [11]. V Evropě zatím žádný podobný precedens není, ale v Evropském parlamentu probíhají jednání o zavedení principu síťové neutrality do práva Evropské unie.
- Odlišení P2P sítí včetně Bittorentu od ostatního provozu není snadné. P2P sítě dokáží používat libovolné porty, včetně portů používaných běžnými protokoly jako HTTP nebo SMTP. Je tedy potřeba výpočetně náročný filtr na aplikační vrstvě, který je však stále možné obejít šifrováním provozu. Provoz P2P sítí pak může být neodlišitelný od elektronického bankovnictví zabezpečeného SSL.
- Uživatel Bittorentu je také zákazník, který může přejít ke konkurenci, pokud nebude spokojen. Síť s nízkou latencí, ale bez uživatelů, je k ničemu.

Optimální algoritmus řízení toku by neměl rozhodovat, které použití sítě je „dobré“ a které „špatné“, měl by se snažit všem uživatelům vyhovět nejlépe, jak je to možné. Řešení se přitom zdá být snadné: posílat pakety pro Hráče přednostně. Jak je možné toho dosáhnout?

## 1.4 Doplněk 802.11e

Nedostatky Wi-Fi v oblasti *Quality of Service* si uvědomila i pracovní skupina IEEE 802.11, a vypracovala doplněk 802.11e [4]. Ve snaze prosadit jej do praxe zavedla Wi-Fi Alliance certifikaci *Wireless Multimedia Extensions* (WME) [10], pro níž je ale vyžadována podpora pouze části specifikace 802.11e. Dnes dostupný hardware WME podporuje, ale nic dalšího z 802.11e obvykle už ne. Pro řešení problémů s řízením toku je tedy možné použít nanejvýš vlastnosti z WME.

Základem WME je vylepšená přístupová metoda EDCA, popsaná v [12]. Každá stanice je jakoby rozdělena na několik virtuálních stanic pro jednotlivé priority dat. Každá z virtuálních stanic má vlastní frontu pro pakety s danou prioritou a používá jiné parametry přístupové metody CSMA/CA: čím vyšší priorita, tím nižší je maximum rozsahu hodnot, z nichž se náhodně vybírá doba čekání před započetím vysílání. Výsledek je, že pakety s vyšší prioritou budou statisticky čekat na odeslání kratší dobu, než pakety s nižší prioritou.

802.11e vyžaduje podporu alespoň čtyř front pro tyto druhy provozu (řazeno od nejnižší po nejvyšší prioritu):

1. *Background* – přenosy na pozadí
2. *Best Effort* – běžné datové přenosy
3. *Video* – streamované video, videohovor
4. *Voice* – streamovaný zvuk, VoIP

Pakety mohou být rozdělovány do front například podle pole ToS (*Type of Service*) v IP hlavičce paketu, jak je definováno v RFC 1349[5]. To ale vyžaduje podporu aplikací, které musí odesílané pakety označit správnou prioritou. Pozdější detekce může být složitá a při použití šifrování i nemožná, jak bylo popsáno v předchozí kapitole 1.3.

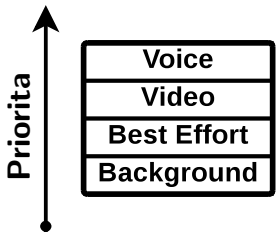
Doplněk 802.11e byl stejně jako původní standard 802.11 navržen pro lokální sítě v kontrolovaném prostředí domácností a organizací. Předpokládá tedy jistou míru spolupráce mezi klienty. Vůbec nepomůže v situaci, kdy aplikace Hráče neumí nastavovat prioritu paketů. Hráč navíc vůbec nemůže ovlivnit označení priority u příchozích paketů z herního serveru nebo od partnera ve VoIP hovoru. Naopak by se dokonce mohlo stát, že P2P aplikace Stahovače zneužije systém priorit a bude posílat pakety s nejvyšší prioritou. Pak by pakety Hráče čekaly dokonce ještě déle než bez QoS!

802.11e také neřeší spravedlivé rozdělení pásma mezi klienty. Pokud na AP přijde 100 paketů pro jednoho klienta s určitou prioritou a pro druhého jediný paket se stejnou prioritou, bude tento paket čekat ve frontě, než se odešlou všechny předchozí pakety. Nebo dokonce může být zahozen, pokud se fronta zaplní.

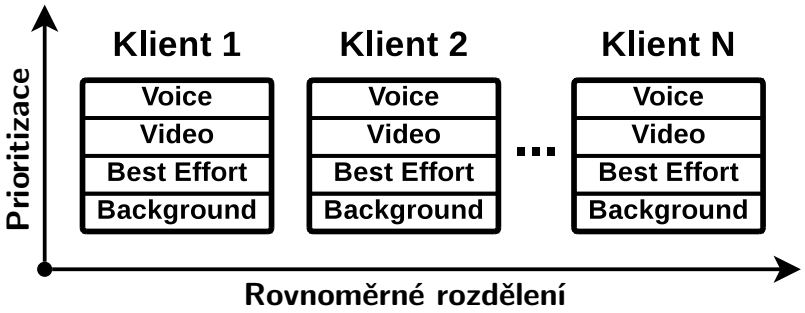
Rozdělování paketů do front definovaných v 802.11e vytváří jednorozměrný model priorit, znázorněný na obrázku 1.2. Situace ve Wi-Fi síti ISP je ale složitější, má dva rozměry (klienty a priority), jak ukazuje obrázek 1.3. Rovnoměrné dělení pásma mezi klienty je úplně jiný problém, ortogonální k systému priorit 802.11e.

Doplněk 802.11e tedy nemůže vyřešit problém s latencí způsobenou frontou paketů v AP, ale zároveň s řízením toku nekoliduje a může jej doplnit. 802.11e je jediný způsob, jak bez úprav klientských stanic snížit latenci způsobenou přístupovou metodou CSMA/CA: AP nemůže rozhodovat, kdy bude který klient vysílat, to dovolují až přístupové metody PCF a HCCA, které běžně dostupný hardware nepodporuje.

[5]Pole *ToS* bylo v RFC 2474 prohlášeno za zastaralé a nahrazeno *DSCP*. Samotné RFC 2474 ale nedefinuje význam jednotlivých hodnot *DSCP*, ponechává jej na síti samotné. V praxi aplikace stále nastavují hodnoty *ToS* podle RFC 1349, například OpenSSH.




Obrázek 1.2: Model priorit 802.11e




Obrázek 1.3: Rozdělení provozu ve skutečné Wi-Fi síti

## 1.5 Specifika řízení toku ve Wi-Fi síti

Problematika řízení toku je dobře zpracována nejen teoreticky, ale existuje i mnoho implementací v operačních systémech (Linux, FreeBSD) a směrovačích. Algoritmy pro řízení datového toku však byly navrženy pro použití na Ethernetu, případně datových okruzích s pevnou přenosovou rychlostí. Od nich se Wi-Fi výrazně odlišuje:

1. Wi-Fi síť je half-duplex, o jedno přenosové pásmo se dělí přenosy ke klientům a od klientů.
2. Nominální přenosová rychlost se může měnit ve velkém rozsahu (od 1 do 54 Mbit/s u 802.11g).
3. Přenos určitého množství dat v malých paketech trvá déle, než přenos stejného množství dat ve velkých paketech.
4. Volitelný mechanismus RTS/CTS způsobuje další snížení efektivní přenosové rychlosti.
5. Neproběhne-li přenos úspěšně kvůli rušení nebo kolizi s jiným přenosem, může být několikrát opakován.

Tyto odlišnosti nejen že nejsou zanedbatelné, ale v praxi znemožňují použití jakýchkoli algoritmů pro řízení toku, založených na znalosti přenosové rychlosti linky. Přenosovou rychlost nelze ani odhadnout, rozdíly mohou být i dva řády, jak lze ukázat následujícím výpočtem:

Uvažujme přístupový bod 802.11g (bez zpětné kompatibility s 802.11b) se dvěma klienty. První klient přenáší rychlostí 54 Mbit/s pakety o maximální velikosti 1500 bajtů[6], druhý klient přenáší rychlostí 1 Mbit/s malé pakety o velikosti 60 bajtů.

Protokol 802.11 má poměrně velkou režii, jak je uvedeno v [13]. K velikosti paketu je nutno přičíst hlavičku linkové vrstvy o velikosti 36 bajtů. Před započetím vysílání je nutno čekat interval DIFS, který je u 802.11g 20 µs. Odeslané rámce jsou potvrzovány na linkové vrstvě. Potvrzení (ACK) má délku 14 bajtů a je posíláno stejnou rychlostí jako data, nejvýše však 24 Mbit/s. Před odesláním potvrzení musí stanice čekat po dobu SIFS, což je 10 µs.

Další režie souvisí s modulací a u obou klientů se liší. První klient používá modulaci OFDM, která vyžaduje 20 µs synchronizační hlavičku před přenosem a dalších 6 µs „signal extension“ po něm. Přenos probíhá po symbolech s délkou 4 µs, které kódují více bitů. Vždy se ale přenáší celý symbol, proto se doba přenosu musí zaokrouhlit nahoru na násobky čtyř. Druhý klient používá modulaci DBPSK, která vyžaduje před přenosem „preamble“ trvající 192 µs. Vše shrnuje tabulka 1.1.

| | | **1. klient** | **2. klient** |
|---|---|---|---|
| Data | DIFS | 28 µs | 28 µs |
| | synchronizační hlavička | 20 µs | 192 µs |
| | obsah | 228 µs | 768 µs |
| | signal extension | 6 µs | – |
| ACK | SIFS | 10 µs | 10 µs |
| | synchronizační hlavička | 20 µs | 192 µs |
| | obsah | 8 µs | 112 µs |
| | signal extension | 6 µs | – |
| **Celkem** | | 326 µs | 1302 µs |

Tabulka 1.1: Výpočet doby přenosu rámce

První klient tedy přijme 1500 bajtů za 326 µs. Pokud by i druhý přijal stejné množství dat, zabere to $25 \times 1302 = 32550$ µs. To je skoro stokrát více, efektivní rychlost Wi-Fi se tedy může lišit o dva řády. A to nebyl brán do úvahy mechanismus RTS/CTS, který je navíc u 802.11g využíván pro zajištění zpětné kompatibility s 802.11b. Algoritmus pro řízení toku na Wi-Fi tedy nesmí vyžadovat stanovení přenosové rychlosti linky.

## 1.6 Dostupné algoritmy pro řízení toku

V předchozích kapitolách 1.3 a 1.5 byly nastíněny požadavky na algoritmus pro řízení toku. Nyní je možné prostudovat dostupné implementace v operačních systémech a vyhodnotit, zda tyto požadavky splňují.

[6] Wi-Fi dokáže přenášet bez fragmentace pakety do velikosti 2304 bajtů [1], ale většina spojení vede alespoň přes jeden Ethernetový spoj s limitem 1500 bajtů. Proto se skoro vždy používá MTU 1500 i na Wi-Fi rozhraních, aby se předešlo fragmentaci.

Budu se zabývat jen open-source systémy – k jiným nemám přístup a navíc bych se nemohl podívat do kódu, jak přesně algoritmus funguje a případně jej přizpůsobit konkrétním potřebám. Nejpoužívanější open-source systémy na routerech jsou dle mých zkušeností ze sítě CZFree Linux a FreeBSD. Jaké metody řízení toku nabízejí?

### 1.6.1 FreeBSD

V aktuální stabilní verzi 7.2 systému FreeBSD jsou k dispozici dvě nezávislé implementace firewallu a řízení toku: PF + ALTQ a IPFW + Dummynet.

Firewall PF [14] a na něj vázaný framework pro řízení toku ALTQ [15] je novější, byl přenesen ze systému OpenBSD. Nabízí několik algoritmů pro řízení toku, ale všechny mají podstatné omezení: dokáží pracovat jen s odchozím provozem na rozhraní. Wi-Fi je však half-duplex, příchozí a odchozí provoz se dělí o společné médium a navzájem se tedy ovlivňují.

**Class Based Queuing**
: **CBQ** umožňuje dělit datové toky do tříd. Každá třída má přířazenu přenosovou rychlost a prioritu. Struktura tříd je hierarchická s kořenovou třídou, která musí mít nastavenu přenosovou rychlost celého spoje v Mbit/s.

**Hierarchical Fair Service Curve Packet Scheduler**
: Algoritmus **HFSC** je podobný CBQ, ale je o něco komplexnější. Umožňuje kontrolu nejen nad přenosovou rychlostí, ale i nad latencí jednotlivých tříd. Také však vyžaduje určení přenosové rychlosti spoje.

**Priority Queuing**
: **PRIQ** pouze dělí datový tok do front s různou prioritou, podobně jako 802.11e. Opět vyžaduje nastavení přenosové rychlosti spoje.

**Random Early Detection**
: **RED** řeší jiný problém (zahazování paketů při zaplnění fronty v routeru), nedá se použít pro dělení pásma mezi klienty.

Původní FreeBSD firewall IPFW obsahuje nástroj pro řízení toku *Dummynet*. *Dummynet* byl původně vyvinut pro testování protokolů (reakci na zahlcení a ztráty paketů), ne pro řízení toku. Proto pracuje s trochu jiným konceptem: algoritmus řízení toku není vázán na jedno síťové rozhraní, ale na virtuální spoj nazývaný *pipe*. Pakety procházející firewallem pak mohou být poslány přes tyto virtuální spoje – i několik za sebou. Na rozdíl od ALTQ tedy umožňuje pracovat najednou s oběma směry provozu, od klientů i ke klientům.

*Pipe* může (ale nemusí) mít nastaveno omezení přenosové rychlosti a případně i umělé zpoždění paketů. *Pipe* může být rozdělena na několik front (*queue*), které se pak dělí o přenosovou rychlost. Dělení nemusí být rovnoměrné, fronta může mít nastavenu váhu. Přestože je možné rozdělit na fronty i *pipe* bez omezení datového toku, výsledek nefunguje, k žádnému snížení latence nedojde. Důvody tohoto chování budou vysvětleny později v kapitole 5.1.

Všechny algoritmy pro řízení toku ve FreeBSD tedy vyžadují stanovení přenosové rychlosti linky. To u Wi-Fi není možné, jak bylo ukázáno v kapitole 1.5. Žádný z algoritmů dostupných ve FreeBSD proto není použitelný pro řízení toku na Wi-Fi.

### 1.6.2 Linux

Nabídka algoritmů řízení toku v aktuálním Linuxovém jádře 2.6.30 je ještě bohatší než ve FreeBSD. Dokumentace k algoritmům je k dispozici ve formě manuálových stránek [16].

**Class Based Queueing**
: **CBQ** je obdoba stejnojmenného algoritmu z FreeBSD. Dělí datové toky do hierarchické struktury tříd. Opět ale vyžaduje nastavení přenosové rychlosti celého spoje v Mbit/s.

**Hierarchical Token Bucket**
: Algoritmus **HTB** funguje podobně jako CBQ, liší se pouze jednodušší konfigurací. Bez nastavení přenosové rychlosti spoje se také neobejde.

**Hierarchical Fair Service Curve**
: **HFSC** je opět obdoba stejnojmenného algoritmu z FreeBSD a sdílí s ním i nutnost stanovit přenosovou rychlost spoje.

**Multi Band Priority Queueing**
: **PRIO** umožňuje dělení datového toku do front s různou prioritou. Počet front je nutno určit předem, a pakety se do nich rozdělují podle položky *ToS* v hlavičce IP paketu. PRIO tedy nabízí skoro totéž co 802.11e a problém spravedlivého dělení přenosového pásma mezi klienty neřeší.

**Hardware Multiqueue-aware Multi Band Queuing**
: Plánovač **MULTIQ** je určen pro použití se síťovým hardwarem s více než jednou frontou odchozích paketů. Tím ale není myšlen Wi-Fi hardware podporující 802.11e, ale některé high-end ethernetové síťové karty.

**Stochastic Fairness Queueing**
: **SFQ** automaticky dělí pakety do velkého počtu front podle haše parametrů spojení (IP adresy, čísla portů). Dělení nelze nijak konfigurovat, takže není možné použít SFQ pro dělení pásma mezi klienty. Stahovač, který používá Bittorent a stahuje přes několik TCP spojení, bude dokonce zvýhodněn, protože zabere mnoho front, zatímco pakety Hráče budou jen v jedné frontě.

**Token Bucket Filter**
: **TBF** je jednoduchý algoritmus, který dokáže pouze omezit datový tok na danou hodnotu (tak jako jedna třída CBQ nebo HTB).

**Random Early Detection**
: **RED** není algoritmus pro dělení pásma, řeší problém zahazování paketů při zaplnění fronty v routeru.

**Generic Random Early Detection**

**GRED** je komplexnější verze algoritmu RED, stále ale ovlivňuje pouze chování při hrozícím zaplnění fronty.

**Deficit Round Robin**

**DRR** je novinka v jádře 2.6.29. Nemá manuálovou stránku a nepovedlo se mi bohužel najít ani žádnou dokumentaci nebo příklad použití.

Ani Linux tedy nenabízí žádný algoritmus řízení toku, který by splňoval požadavky uvedené v kapitolách 1.3 a 1.5. Nezbývá, než takovýto algoritmus navrhnout a implementovat.

# Kapitola 2

# Přehled algoritmů řízení toku

V předchozí kapitole 1.6 bylo ukázáno, že žádná implementace řízení toku dostupná v rozšířených open-source operačních systémech není aplikovatelná na specifické podmínky Wi-Fi sítí. To však neznamená, že by byly nepoužitelné samotné algoritmy, na nichž jsou tyto implementace založeny. Má proto smysl je prozkoumat společně s dalšími, které dostupné implementace nemají. Pokud nebudou použitelné přímo, mohou alespoň posloužit jako inspirace.

Algoritmy pro řízení toku je možné rozdělit do tří skupin podle problému, který řeší:

**Algoritmy pro *Traffic Shaping*** zajišťují, že výsledný datový tok odpovídá danému parametru. Tímto parametrem je typicky přenosová rychlost a nastavené hodnoty se dosahuje tak, že jsou některé pakety pozdrženy[1]. Jednotlivé algoritmy se liší v tom, jak přenosovou rychlost měří. Datový tok je totiž tvořen pakety různé délky, nikdy tedy nemůže být zcela rovnoměrný. Některé algoritmy dovolují vyslat za sebou několik paketů maximální rychlostí, tzv. *burst*.

**Algoritmy pro *Packet Scheduling*** naopak nezdržují pakety, ale mění jejich pořadí. Pakety jsou rozděleny do tříd a poté mohou být ty z prioritní třídy odesílány přednostně, nebo naopak mohou být odesílány rovnoměrně ze všech tříd. Základním předpokladem je ale existence fronty, v níž pakety čekají na odeslání. Tu dokáží vytvořit algoritmy pro *Traffic Shaping*.

Poslední skupinou jsou **algoritmy pro optimalizaci využití přenosového pásma**. V bezdrátové síti nemají nikdy všechny stanice stejné podmínky pro příjem signálu. Proto fyzická vrstva nabízí několik modulačních rychlostí a stanice s horším signálem může použít nižší rychlost. Přenosové médium je ale sdílené, takže stanice s nižší rychlostí snižuje celkovou efektivitu sítě.

[1] Zatímco *Traffic Policing* pakety, které se nevejdou do zadaných parametrů, zahazuje. Základy algoritmů pro *Traffic Shaping* i *Traffic Policing* jsou ale stejné, a proto se pro ně používají i stejné názvy.

## 2.1 Algoritmy pro Traffic Shaping

### 2.1.1 Leaky Bucket

Nejjednodušším algoritmem pro *Traffic Shaping* je *Leaky Bucket* [17]. Název je odvozen od představy kbelíku s otvorem, kterým vytéká voda – neustále stejnou rychlostí. *Leaky Bucket* obdobně omezuje datový tok na pevnou přenosovou rychlost. „Kbelík“ je reprezentován frontou s omezenou délkou, pokud se zaplní, jsou další příchozí pakety zahozeny.

V sítích pracujících s pakety proměnlivé délky (kam patří i Wi-Fi, na rozdíl od ATM sítí) je možné *Leaky Bucket* implementovat tak, že má-li být cílová přenosová rychlost $r$ bit/s a právě začalo odesílání paketu délky $l$ bajtů, další paket se začne odesílat za $8l/r$ sekund. V praxi se ale u těchto sítí používají jiné algoritmy než *Leaky Bucket*.

### 2.1.2 Token Bucket

Název *Token Bucket* [17] by se dal přeložit jako „kbelík žetonů“. Žetony mohou v síti s proměnlivou délkou paketů reprezentovat bajty. Paket délky $l$ je pak možné odeslat jen tehdy, je-li v kbelíku alespoň $l$ žetonů, a stejný počet žetonů je poté odebrán. Žetony jsou průběžně doplňovány danou rychlostí, ve výsledku je tedy průměrná přenosová rychlost omezena rychlostí doplňování žetonů.

Rozdíl oproti algoritmu *Leaky Bucket* je v tom, že *Token Bucket* umožňuje odeslat shluk paketů (*burst*) maximální rychlostí – tolik, kolik je maximální počet žetonů v kbelíku. To lépe odpovídá reálným datovým tokům, v nichž se shluky často vyskytují (například načítání WWW stránky). Na *Token Bucket* je mimo jiné založena implementace řízení toku HTB v Linuxu, zmíněná v kapitole 1.6.2.

### 2.1.3 Jumping Window

Algoritmus *Jumping Window* [17] počítá přenesená data za pevně stanovené časové okno. Překročí-li množství přenesených dat od začátku časového okna definovaný limit, budou další pakety pozdrženy až do začátku nového časového okna (kdy se čítač vynuluje). Je-li limit nastaven na $n$ bajtů a délka časového okna $t$ sekund, bude přenosová rychlost omezena na $8n/t$ bit/s. Je však možné odeslat maximální rychlostí až $2n$ bajtů, pokud změna časového okna nastane přesně uprostřed přenosu.

### 2.1.4 Triggered Jumping Window

Algoritmus *Triggered Jumping Window* [17] se od *Jumping Window* odlišuje tím, že časová okna nezačínají pravidelně v pevných intervalech. Nové okno začne teprve v okamžiku příchodu prvního paketu. Mezi okny tedy mohou být mezery, pokud

nebyla přenášena žádná data. Tato změna nemá vliv na maximální dosažitelnou přenosovou rychlost, ale omezuje největší možný *burst* na $n$ bajtů.

### 2.1.5 Moving Window

Nevýhodou *Jumping Window* je skokové nulování čítače při začátku okna. Pakety pak čekají do začátku nového okna, jsou rychle odeslány a další zase čekají. I původně rovnoměrný datový tok se tak stává nepravidelným. Řešením je okno posouvat plynule, což činí algoritmus *Moving Window* [17].

Velikost každého odeslaného paketu je přičtena k čítači, ale zároveň je tato velikost zaznamenána spolu s časem odeslání. Později, po uběhnutí $t$ sekund, je velikost paketu od čítače zase odečtena (a záznam může být vymazán). Hodnota čítače v každý okamžik tedy vyjadřuje počet bajtů přenesených za posledních $t$ sekund.

Pokud by po přičtení velikosti paketu hodnota čítače překročila definovaný limit $n$, je paket pozdržen, dokud hodnota čítače dostatečně neklesne. Výsledná maximální přenosová rychlost je stále $8n/t$ bit/s a největší *burst* $n$ bajtů jako u algoritmu *Jumping Window*. Pokud však pakety neustále přicházejí rychleji, než mohou být odesílány, budou odcházet pravidelně jako u algoritmu *Leaky Bucket* místo po shlucích na začátku okna.

Cenou za pravidelnější datový tok je vyšší paměťová náročnost. Zatímco předchozí algoritmy si vystačily s jedním čítačem a mají tedy konstantní paměťovou složitost, *Moving Window* vyžaduje udržovat v paměti informace o všech paketech poslaných během časového okna. Potřebná paměť tedy závisí na délce okna, nastaveném limitu a průměrné velikosti paketů, kterou lze předem jen těžko odhadnout.

### 2.1.6 Exponentially Weighted Moving Average Window

Algoritmus *Jumping Window* zapomíná staré pakety skokově, což způsobuje nepravidelnost datového toku. *Moving Window* zapomíná plynule, ale za cenu složitější implementace s vyššími a těžko odhadnutelnými nároky na paměť. Určitého kompromisu je možné dosáhnout použitím exponenciálně váženého průměru [18].

Obecný vážený průměr je dán vztahem $\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$, kde $x_i$ je $i$-tá naměřená hodnota a $w_i$ je váha této hodnoty. U exponenciálně váženého průměru je dán parametr $\alpha$, $0 < \alpha < 1$ a pro váhy platí $w_i = (1-\alpha)^{n-i}$. Váhy tedy exponenciálně rostou směrem k novějším hodnotám (klesají směrem ke starším hodnotám). Normalizace vah je rovna

$$\sum_{i=1}^{n} w_i = \sum_{i=1}^{n} (1-\alpha)^{n-i} = \sum_{i=0}^{n-1} (1-\alpha)^{i} = \frac{1-(1-\alpha)^n}{1-(1-\alpha)},$$

což pro velká $n$ konverguje k $\frac{1}{\alpha}$.

Exponenciálně vážený průměr $n$ hodnot lze spočítat z průměru $(n-1)$ hodnot:

$$\begin{aligned}\bar{x}_n &= \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i}x_i = \alpha x_n + \sum_{i=1}^{n-1} \alpha(1-\alpha)^{n-i}x_i = \\ &= \alpha x_n + (1-\alpha)\sum_{i=1}^{n-1} \alpha(1-\alpha)^{n-1-i}x_i = \\ &= \alpha x_n + (1-\alpha)\bar{x}_{n-1}\end{aligned} \tag{2.1}$$

Klouzavý průměr je definován jako průměr posledních $N$ hodnot. Váha starých hodnot však klesá exponenciálně, takže je můžeme zanedbat. Není tedy nutné ukládat všechny naměřené hodnoty jako u *Moving Window*, postačí jeden čítač s aktuální hodnotou exponenciálně váženého průměru.

Nevýhodou *EWMA Window* je poměrně komplexní a těžko předvídatelné chování. Nelze jednoduše určit, kdy budou staré údaje zapomenuty. Teoreticky nikdy, prakticky je možné považovat za zapomenuté ty hodnoty, jejichž součet (normalizovaných) vah je menší než nějaká mez, například 1%. Jaká má být hodnota parametru $\alpha$, aby součet vah $k$ nejnovějších hodnot byl roven $p$?

Součet vah prvních $(n-k)$ hodnot je pro dané $k \ll n$ roven

$$\begin{aligned}\sum_{i=1}^{n-k} \alpha(1-\alpha)^{n-i} &= \alpha(1-\alpha)^k \cdot \sum_{i=1}^{n-k}(1-\alpha)^{n-k-i} = \alpha(1-\alpha)^k \cdot \sum_{i=0}^{n-k-1}(1-\alpha)^i = \\ &= \alpha(1-\alpha)^k \cdot \frac{1-(1-\alpha)^{n-k}}{1-(1-\alpha)} = (1-\alpha)^k \cdot (1-(1-\alpha)^{n-k}),\end{aligned}$$

což pro velké $n$ konverguje k $(1-\alpha)^k$. Tento výsledek položíme rovný $1-p$:

$$\begin{aligned}(1-\alpha)^k &= 1-p \\ 1-\alpha &= \sqrt[k]{1-p} \\ \alpha &= 1-\sqrt[k]{1-p}\end{aligned} \tag{2.2}$$

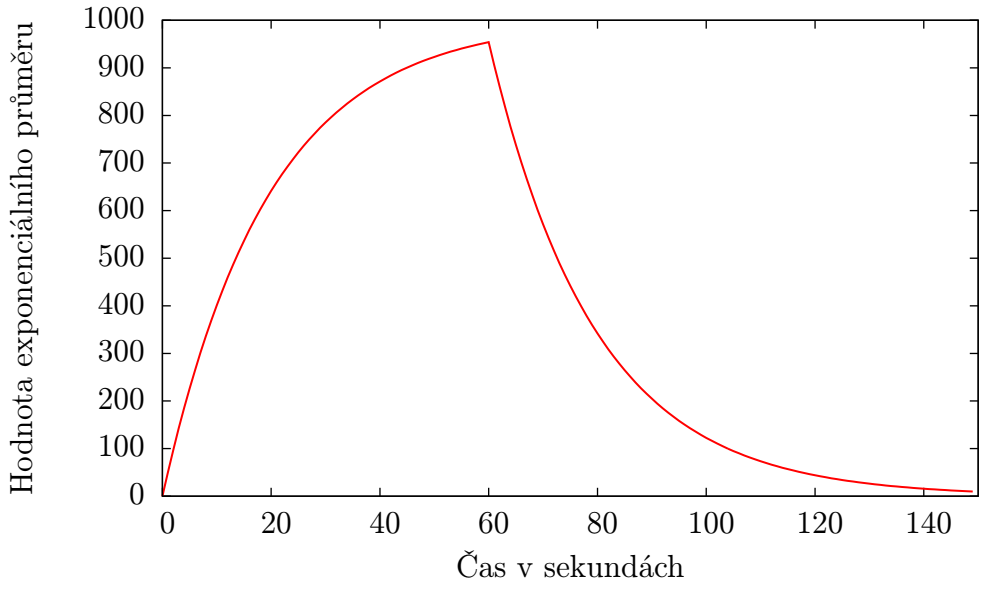
Pro $p = 99\%$ a $k = 100$ vychází $\alpha \doteq 0.045$.

Standardní klouzavý průměr předpokládá, že hodnoty jsou měřeny pravidelně. To v počítačové síti neplatí, měřené hodnoty jsou velikosti odesílaných paketů, a pakety nejsou odesílány pravidelně. Vzorec pro výpočet klouzavého průměru je proto nutno upravit na

$$\bar{x}_n = (1-\alpha)\bar{x}_{n-1} + \frac{\alpha l_i}{t_i - t_{i-1}}, \tag{2.3}$$

kde $l_i$ je délka $i$-tého paketu a $t_i$ je čas odeslání $i$-tého paketu.

Představu o chování EWMA lze získat z grafu 2.1, který zobrazuje hodnoty průměru po vteřinách. Prvních 60 sekund byl odesílán jeden paket o velikosti 1000 bajtů každou vteřinu, pak již byl datový tok nulový a průměr tedy postupně klesá k nule. Parametr $\alpha$ má hodnotu 0.05.




Obrázek 2.1: Hodnota exponenciálně váženého průměru v čase

### 2.1.7 Posouzení algoritmů

Algoritmy pro *Traffic Shaping* standardně pracují s přenosovou rychlostí v Mbit/s. To nelze aplikovat na Wi-Fi sítě z důvodů uvedených v kapitole 1.5. Místo přenosové rychlosti je ale možné použít jinou metriku, například čas: na jakou dobu bylo zabráno přenosové médium. Vyžadovalo by to však znát dopředu, kolik času zabere přenos paketu, což není možné: paket může být vysílán opakovaně, pokud se přenos nezdařil kvůli kolizi nebo rušení. Navíc se o jedno přenosové pásmo dělí oba směry komunikace, od klientů i ke klientům.

Na všechny algoritmy pro *Traffic Shaping* je ale možné nahlížet jiným způsobem. Z abstraktního pohledu všechny nějak odhadují průměrnou přenosovou rychlost datového toku, porovnávají výsledek s definovanou hodnotou a podle toho rozhodují, zda je možné paket odeslat. A tento první krok, výpočet průměru, lze využít i u Wi-Fi pro hodnocení klientů a a následně pro výběr, čí paket odeslat. Takovýto přístup je ale spíše doménou algoritmů pro *Packet Scheduling*.

## 2.2 Algoritmy pro Packet Scheduling

### 2.2.1 Fair Queueing

Ideální spravedlivé dělení přenosového pásma mezi datové toky by vypadalo tak, že data by byla odesílána z jednotlivých toků cyklicky po bitech. To v paketových sítích není možné, paket je nutné odeslat v celku, ideální stav lze pouze aproximovat. Jednu z možností aproximace představuje algoritmus *Fair Queueing* navržený v [19].

*Fair Queueing* simuluje odesílání po bitech tak, že pro každý paket počítá čas $F$, kdy by při odesílání po bitech skončil jeho přenos. Příchozí pakety jsou vkládány do fronty uspořádané podle času $F$, a k odeslání je vždy vybrán paket s nejmenším $F$ ve frontě.

Čas dokončení přenosu však závisí na přenosové rychlosti linky, a navíc příchod paketu z nového datového toku znamená nutnost přepočítat $F$ všech paketů ve frontě, což by vedlo na časovou složitost $O(n)$. Pro plánování ale není potřeba znát přesný čas, pouze uspořádání paketů podle tohoto času, a to se příchodem nového paketu nemění. Proto se zavádí tzv. virtuální čas $V(t)$. $V(t)$ roste nepřímo úměrně počtu datových toků, které přenáší data: za 1 jednotku virtuálního času odešlou všechny aktivní datové toky 1 byte.

Přenos paketu $k$ z datového toku $i$ o délce $L_i^k$ skončí $L_i^k$ jednotek virtuálního času po začátku jeho přenosu. A odesílání paketu, který dorazil v čase $a_i^k$ začne buď po odeslání předchozího paketu ve frontě datového toku $i$, nebo ihned, pokud je tato fronta prázdná. Z toho už lze sestavit výsledný vzorec pro $F_i^k$:

$$F_i^k = \max(V(a_i^k), F_i^{k-1}) + L_i^k \tag{2.4}$$

Existuje i zobecnění algoritmu nazývané *Weighted Fair Queueing*. Každý datový tok má přiřazenu váhu $W_i$. Datové toky se pak nedělí o přenosové pásmo rovnoměrně, ale v poměru vah. Vzorec pro výpočet virtuálního času konce přenosu se pak mění na:

$$F_i^k = \max(V(a_i^k), F_i^{k-1}) + \frac{L_i^k}{W_i} \tag{2.5}$$

Časová složitost algoritmu závisí na složitosti vkládání do uspořádané fronty, která nemůže být lepší než $O(\log n)$ [2]. Přesto je možné algoritmus zrychlit: v uspořádané frontě nemusí být všechny pakety. Pakety jednoho datového toku jsou odesílány v pořadí, v jakém přišly, mohou být tedy v obyčejné frontě (FIFO). Do uspořádané fronty pak stačí vkládat vždy pouze první paket z každého datového toku. Potom složitost vložení do uspořádané fronty nezávisí na počtu paketů ve frontě $n$, ale na počtu datových toků $m$ a je rovna $O(\log m)$.

Složitost závisející na počtu datových toků může činit problém u routerů na páteřních linkách, kde počet datových toků (daných obvykle zdrojovou a cílovou IP adresou) dosahuje řádu milionů. U Wi-Fi je situace jiná, počet klientů na jednom přístupovém bodu dosahuje nejvýše desítek (maximum v síti Praha12.Net je okolo dvaceti).

### 2.2.2 Stochastic Fair Queueing

Algoritmus *Stochastic Fair Queueing* [20] snižuje časovou složitost *Fair Queueing* na $O(1)$ použitím hašování. Počet front je pevně daný, nezávisí na počtu datových toků. Pakety jsou pak do front rozdělovány podle haše zdrojové a cílové adresy. Výpočet

[2] Pomocí uspořádané fronty je možné třídit, a žádný třídící algoritmus založený na porovnávání nemůže mít lepší časovou složitost než $O(n \log n)$.

haše probíhá v konstantním čase a délka prioritní fronty je také konstantní, proto je časová složitost přidání paketu do fronty $O(1)$.

*Stochastic Fair Queueing* předpokládá, že datových toků je velké množství, a statisticky tak budou rozděleny spravedlivě. U Wi-Fi, kde jeden datový tok představují data jednoho klienta a klientů je málo (nejvýše několik desítek), ale tento předpoklad neplatí. A zároveň také není optimalizace časové složitosti tolik potřebná, naopak by přidané hašování mohlo znamenat zpomalení.

### 2.2.3 Deficit Round Robin

Algoritmus *Deficit Round Robin* [21] používá jiný přístup než *Fair Queueing*: neprázdné fronty jsou obsluhovány cyklicky (*round-robin*). Triviální řešení by bylo při průchodu z každé neprázdné fronty odeslat jeden paket. Pracovalo by s časovou složitostí $O(1)$, ale nebylo by spravedlivé: pakety mají proměnlivou délku, datový tok posílající velké pakety by tedy byl zvýhodněn.

*Deficit Round Robin* bere do úvahy velikost paketů. U každé fronty si udržuje čítač nazvaný *deficit*. Při každém cyklickém průchodu front je k čítači přičtena konstanta pojmenovaná *kvantum*. Datový tok pak může odeslat tolik bajtů, kolik činí jeho *deficit*, a *deficit* je o stejnou hodnotu snížen. Zbylá hodnota je zachována do dalšího kola. Fronta si tedy může nevyužité bajty „schovat“ do příštího průchodu a výsledné chování se trochu podobá algoritmu *Token Bucket* z kapitoly 2.1.2. Kvanta však není možné „hromadit“: pokud ve frontě nečekají žádné pakety na odeslání, je čítač vynulován.

Je-li kvantum větší nebo rovno MTU, bude při průchodu z každé neprázdné fronty odeslán alespoň jeden paket. Budeme-li neprázdné fronty udržovat v odděleném seznamu, bude časová složitost výběru paketu k odeslání rovna $O(1)$.

Algoritmus *Deficit Round Robin* je oproti *Fair Queueing* rychlejší a zároveň jednodušší na implementaci. Má ale i nevýhodu, latenci: přijde-li nový paket do prázdné fronty, je tato fronta podle definice v [21] zařazena na konec seznamu. Paket tedy musí čekat, než všechny fronty vyčerpají své kvantum.

### 2.2.4 Posouzení algoritmů

Společnou vlastností všech algoritmů pro *Packet Scheduling* je, že musí pracovat nad frontou, kde pakety čekají na odeslání. Tuto frontu dokáží vytvořit algoritmy pro *Traffic Shaping*, ale ty u Wi-Fi nelze použít, jak bylo vysvětleno v kapitole 2.1.7. U Wi-Fi proto musí algoritmus pro *Packet Scheduling* pracovat nad tou frontou, která přirozeně vzniká a odkud si hardware bere pakety k odeslání. Hledáním, kde se tato fronta nachází, se budu zabývat později v kapitole 5.1.

Algoritmy pro *Packet Scheduling* zajišťují spravedlivé dělení přenosového pásma, ale pro svoji specifickou definici spravedlnosti: pásmo se v každý okamžik dělí rovnoměrně mezi aktivní datové toky. Klíčové je slovo „aktivní“: zatímco Stahovač přenáší data neustále, Brouzdal jen občas. Brouzdal ale nebude nijak zvýhodněn, v době,

kdy budou přenášet data oba, dostane každý přesně polovinu přenosového pásma. Vše je otázka definice „spravedlivosti“, kterou se budu podrobněji zabývat v kapitole 3.2.2.

## 2.3 Algoritmy pro optimalizaci využití přenosového pásma

### 2.3.1 Maximum Throughput Scheduling

Cílem algoritmu *Maximum Throughput Scheduling* [22] je maximalizovat celkovou přenosovou rychlost sítě, bez ohledu na klienty. U Wi-Fi by to znamenalo, že přístupový bod pošle přednostně pakety těm klientům, kteří mají nejlepší signál a dokáží tedy přijmout data vyslaná nejvyšší modulační rychlostí.

Algoritmus *Maximum Throughput Scheduling* je navržen hlavně pro mobilní sítě, kde se stanice pohybují a síla jejich signálu kolísá. U Wi-Fi použité na poslední míli pro připojení k Internetu se ale klienti nepohybují, připojují se typicky přes anténu na domě. Nasazení *Maximum Throughput Scheduling* v této síti by znamenalo, že dokud má AP nějaká data pro stanice schopné komunikovat maximální rychlostí, na ostatní se vůbec nedostane.

Přes zřejmou nespravedlnost se však někdy ve Wi-Fi sítích určitá obdoba *Maximum Throughput Scheduling* používá. Automatické řízení modulační rychlosti se jednoduše vypne a AP posílá všem klientům data maximální rychlostí. Všichni klienti musí mít anténu s dostatečným ziskem, která jim dovolí data přijímat – pokud to z nějakého důvodu u klienta není možné, nemůže se připojit.

### 2.3.2 Proportionally Fair Scheduling

Algoritmus *Proportionally Fair Scheduling* [23] počítá pro každou stanici $i$ hodnotu prioritní funkce

$$P_i = \frac{T_i^\alpha}{R_i^\beta},$$

kde $T_i$ je přenosová rychlost dosažitelná stanicí $i$ v daný okamžik, $R_i$ je dlouhodobý průměr dosažitelné rychlosti a $\alpha$, $\beta$ jsou konstanty.

Hodnoty konstant $\alpha = 1$ a $\beta = 0$ vedou na *Maximum Throughput Scheduling*. V mobilních sítích se používají hodnoty $\alpha = 1$ a $\beta = 1$: stanice přenáší data, má-li momentálně lepší signál než obvykle. Ve Wi-Fi síti pro připojení k Internetu se signál stanic příliš nemění, a proto se nemění ani dosažitelná přenosová rychlost, tedy $R_i = T_i$. Prioritní funkce pak zdegeneruje na $P_i = T_i^{\alpha-\beta}$, což je pro $\alpha > \beta$ opět *Maximum Throughput Scheduling*.

### 2.3.3 Posouzení algoritmů

Algoritmy pro optimalizaci využití přenosového pásma byly navrženy především pro mobilní sítě. Uvedl jsem jen ty nejzákladnější, složitější algoritmy se zaměřují na řešení problémů s interferencemi mezi buňkami mobilních sítí a předávání klientů mezi buňkami, což u Wi-Fi sítí se stacionárními stanicemi není relevantní. Základní princip plánování podle přenosové rychlosti ale relevantní je, modulační rychlost se u 802.11g pohybuje ve velkém rozsahu 1 až 54 Mbit/s.

Algoritmy tak, jak byly popsány, používají dosažitelnou přenosovou rychlost jako kritérium pro výběr, kterého klienta obsloužit. Existuje ale i alternativní přístup: brát hodnotu prioritní funkce jako váhu, například pro algoritmus *Weighted Fair Queueing* popsaný v kapitole 2.2.1. V nejjednodušším případě je jako váha použita přímo přenosová rychlost.

U algoritmu *Weighted Fair Queueing* se stanice dělí o datový tok v poměru jejich vah: za daný časový úsek může stanice $i$ odeslat $W_i$ bitů. Odeslání $W_i$ bitů rychlostí $R_i$ trvá $W_i/R_i$ sekund. Pokud platí $W_i = R_i$, zabere každá stanice přenosové médium na stejný čas, budou se o ně tedy dělit zcela spravedlivě.

# Kapitola 3

# Návrh algoritmu řízení toku

## 3.1 Specifikace požadavků

Funkční požadavky na algoritmus řízení toku byly naznačeny v kapitole 1.3 a specifika Wi-Fi, s nimiž se musí vypořádat, v kapitole 1.5. Základem je zajistit, aby se uživatelé co nejméně negativně ovlivňovali, s ohledem na odlišné potřeby tří typů uživatelů (Brouzdala, Hráče a Stahovače).

Vedle funkčních požadavků je ale nutno uvést i požadavky nefunkční, například na výkon. Algoritmus musí fungovat na hardwaru používaném v přístupových bodech Wi-Fi sítí, který má menší výkon a méně paměti než běžná PC a místo harddisku paměť Flash.

### P1 Rozlišování klientů AP

Algoritmus musí rozlišovat jednotlivé klienty přístupového bodu, nemůže se rozhodovat jen na základě IP adres. Jeden klient může mít přidělen celý blok adres, nebo používat několik adres, má-li Wi-Fi zařízení nastaveno do režimu bridge. Sice by bylo možné tyto adresy zadat do konfigurace řízení toku, ale to by komplikovalo implementaci i následnou správu. Pro zjednodušení nebudu uvažovat, že by dva zákazníci mohli být připojeni přes společnou anténu a Wi-Fi zařízení. Tato situace sice v síti Praha12.Net nastává, ale je naprostou výjimkou.

### P2 Spravedlivé dělení pásma

Omezeným zdrojem ve Wi-Fi síti je přenosové médium: v jeden okamžik může buď vysílat jeden klient, nebo může AP posílat data jednomu z klientů. Každý klient má stejné právo přenášet data, o přenosové pásmo by se proto měli dělit rovnoměrně. Dva klienti, kteří přenáší data neustále, by měli přenést za stejný čas stejné množství dat. Klient, který nevytěžuje přenosové pásmo na maximum, by za to měl být odměněn nižší latencí. Pro zjednodušení nebudu uvažovat různé priority klientů (například podle toho, kolik platí).

### P3 Dlouhodobá paměť

Algoritmus musí umět rozlišit Stahovače, který přenáší data souvisle celý den nebo i déle, a Brouzdala, který může občas stahovat i několik minut, například aktualizaci operačního systému. Musí si tedy pamatovat historii přenesených dat, ne však po neomezenou dobu, protože chování uživatelů se může v čase měnit. Algoritmus by si ale měl vystačit pouze s operační pamětí, neměl by ukládat data na disk. Některé embedded platformy používané na přístupových bodech totiž klasický disk nemají, operační systém běží z ramdisku a na Flash pro uložení firmwaru a konfigurace se přistupuje zvláštním způsobem.

### P4 Maximální propustnost

Algoritmus by neměl snižovat propustnost sítě umělým zdržováním paketů. Kdykoli přístupový bod může vysílat a ve frontě je alespoň jeden paket, bude odeslán. Je-li na AP připojen pouze jeden klient, má pro sebe veškerou kapacitu sítě. Případné omezování klientů podle zaplaceného tarifu lze řešit dostupnými algoritmy pro *Traffic Shaping* (viz kapitola 1.6) a nemusí být tedy součástí algoritmu.

### P5 Minimalizace latence

Přijde-li paket pro Hráče, musí být odeslán co nejdříve. Není možné použít cyklickou obsluhu (*round-robin*), čekání, než budou odeslány pakety pro všechny klienty by znamenalo příliš velkou latenci: poslání paketů velikosti 1500 bajtů 10 klientům rychlostí 1 Mbit/s trvá 120 ms, a to i když neuvažujeme režii přístupové metody a pakety ve směru od klientů, které zvyšují zpoždění zcela nedeterministicky. Algoritmus tedy musí vybírat další paket k odeslání co nejpozději, ideálně až ve chvíli, kdy byl odeslán předchozí.

### P6 Síťová neutralita

Algoritmus by měl odpovídat zásadám síťové neutrality z důvodů uvedených v kapitole 1.3. Měl by rozlišovat klienty podle charakteru komunikace (např. množství přenesených dat), ne podle používaných protokolů. Detekce protokolů je náročná na výkon a při použití šifrování může být i zcela nemožná.

### P7 Rozlišení efektivity přenosů

Algoritmus nemůže pracovat jen s přenesenými bajty. Doba přenosu stejného množství dat se může lišit až stonásobně, jak bylo ukázáno v kapitole 1.5. Sdílený prostředek je vysílací čas, ne proud bitů. Algoritmus by měl proto odhadovat, jak dlouho přenos trval, a tento čas používat pro posuzování klientů namísto přenesených bajtů.

### P8 Download i upload zároveň

Wi-Fi síť je half-duplex: download i upload klientů sdílí stejné přenosové médium. Klient, který intenzivně uploaduje, zpomaluje ostatním download, což algoritmus musí brát v potaz. Přímo ovlivnit, kdy bude který klient vysílat, je ale těžké, rozhoduje o tom náhoda v přístupové metodě CSMA/CA. Upload je možné částečně ovlivnit využitím zpětné vazby protokolu TCP, který reaguje na zpoždění nebo ztrátu paketů zpomalením. Využít doplněk 802.11e bohužel nelze, jak bylo uvedeno v kapitole 1.4: prioritu odesílaných paketů si určují klienti sami.

### P9 Bez úprav klientů

Algoritmus nesmí vyžadovat změny na straně klientů. V síti Praha12.Net jsou klientská zařízení majetkem klientů a jsou velmi různorodá, v jiných nekomerčních bezdrátových sítích je situace obdobná. Výměna všech klientských zařízení za jednotný typ není reálná – nejen z finančních důvodů. A těžko bude možné vytvořit upravený firmware nebo ovladače pro všechna zařízení.

### P10 Softwarová implementace

Implementace algoritmu by měla vyžadovat pouze změny v softwaru přístupového bodu. Vývoj vlastního hardwaru je mimo rozsah práce, a také mimo finanční možnosti většiny bezdrátových sítí. Je nicméně možné, že bude nutné najít vhodný kompatibilní hardware, který dovolí nutné změny (například díky open-source ovladači nebo firmwaru). Musí však být cenově dostupný (cca do 3000 Kč).

### P11 Efektivita

Algoritmus (nebo spíše jeho implementace) nesmí být příliš náročná na velikost paměti a výkon procesoru, aby nevyžadovala výkonné PC, ale vystačila si s embedded platformou (ty nejlevnější mají MIPS nebo ARM procesor okolo 100MHz a 8MB paměti). Na druhou stranu nemusí příliš dobře škálovat při rostoucím počtu klientů, na jednom sektoru AP (jednom Wi-Fi rozhraní) budou nanejvýš desítky klientů.

### P12 Přenositelnost

Algoritmus by neměl být svázán s konkrétním operačním systémem nebo hardwarovou platformou. Minimálně by měl být implementovatelný v operačních systémech Linux a FreeBSD a přenositelný na platformy Intel x86 (běžné PC) a MIPS (platformy Mikrotik RouterBoard a Ubiquity Nanostation často používané ve Wi-Fi routerech).

## 3.2 Hledání vhodného algoritmu

V předchozích kapitolách byly uvedeny algoritmy řízení toku popsané v literatuře a požadavky na algoritmus řízení toku na Wi-Fi. Nyní nadešel čas zjistit, zda některý z těchto algoritmů, případně jejich kombinace, může uvedené požadavky splnit.

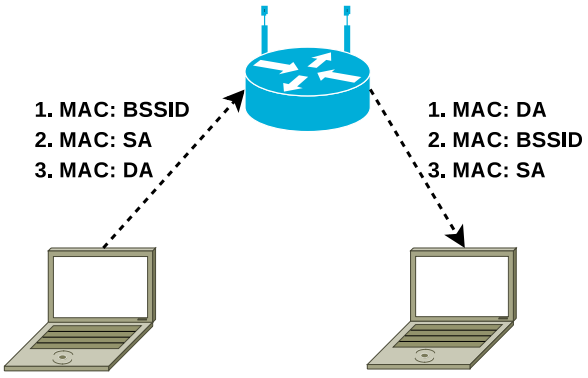
Základní požadavky na funkce jsou body P1, P2 a P3, ostatní jsou spíše omezující podmínky. Cílem proto bude splnit tyto základní body a neporušit přitom žádné omezení.

### 3.2.1 Rozlišování klientů

Požadavek P1 znamená, že algoritmus musí klienty rozlišovat podle MAC adres, tedy pracovat na 2. vrstvě ISO/OSI modelu. Klient je definován bezdrátovým zařízením, což je typicky „krabička“ typu Ovislink 5460 nebo WRT-314, ale může to být i Wi-Fi karta v notebooku nebo desktopu. Určuje ale MAC adresa v datovém rámci jednoznačně bezdrátové zařízení?

Adresace ve Wi-Fi síti je složitější než na Ethernetu. Tak jako u Ethernetu může být rámec přenášen několika kabely, u Wi-Fi může být přenášen několikrát vzduchem. Na rozdíl od Ethernetu je ale u Wi-Fi nutné u každého přenosu adresovat příjemce. V rámcích protokolu 802.11 proto mohou být až čtyři MAC adresy, aby bylo možno rozlišit skutečný zdroj a cíl rámce a odesílatele a příjemce u jednoho přenosu.

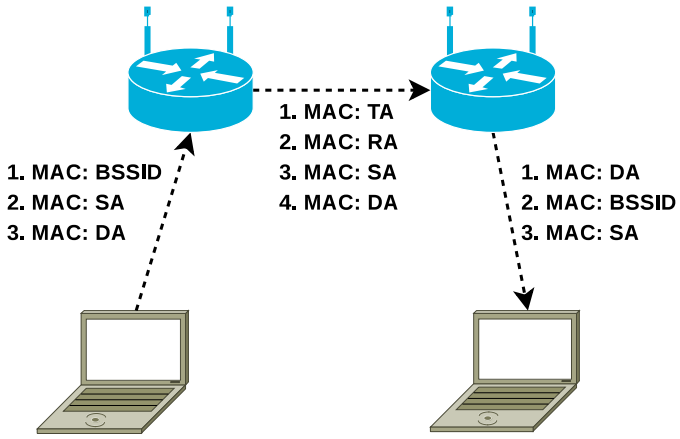
Typický příklad je rámec od jednoho klienta AP ke druhému. Ten je přenášen dvakrát: od 1. klienta na AP a pak od AP k 2. klientovi. U obou přenosů je skutečný zdroj a cíl stejný (adresy klientů), ale při prvním přenosu bude příjemcem AP a u druhého zase odesílatelem. Situace je znázorněna na obrázku 3.1. SA (*Sender address*) je skutečný zdroj, DA (*Destination address*) skutečný cíl a BSSID (*Basic service set ID*) adresa AP.




Obrázek 3.1: Adresace při přenosu rámce mezi dvěma klienty AP

Čtyři adresy se použijí, pokud jsou dvě sítě LAN propojené bezdrátovým mostem a nebo dva přístupové body propojené Wi-Fi spojem, jako na obrázku 3.2. V této

situaci se používá zvláštní režim nazývaný *Wireless Distribution System* (WDS), kdy jsou přístupové body nastaveny pro vzájemnou spolupráci. Rámce přenášené přes WDS spoj mají vedle adres skutečného zdroje a cíle i adresy odesílajícího AP (TA, *Transmitter Address*) a přijímajícího AP (RA, *Receiver Address*). Režim WDS však není certifikován organizací Wi-Fi Alliance, takže není zajištěna kompatibilita hardwaru různých výrobců.




Obrázek 3.2: Adresace při přenosu rámce ve WDS

Uvažujme nyní zákazníka, který chce do bezdrátové sítě připojit svoji LAN s několika počítači. Jeho Wi-Fi zařízení musí fungovat v režimu klient, WDS nelze použít kvůli problémům s kompatibilitou a omezení jen na několik WDS partnerů. Běžné zařízení v režimu klient bude od AP dostávat jen rámce na svoji vlastní MAC adresu, nemůže tedy fungovat jako klasický most. Jak tedy může zákazník připojit svoji LAN? Řešení je několik:

1. Použít zařízení, které bude fungovat jako směrovač. To je ideální, vyžaduje však buď nasadit překlad adres (NAT), nebo je nutná spolupráce správce sítě, který klientovi přidělí podsíť. Je též potřeba odpovídající hardware, který umí fungovat v režimu směrovače. To byl problém především v minulosti, kdy tato zařízení byla výrazně dražší, dnes je již cenový rozdíl minimální.

2. Zařízení může maskovat MAC adresy počítačů v LAN za svoji MAC adresu metodou zvanou *ARP Proxy*. Ve všech odchozích paketech mění MAC adresu odesílatele za svoji, a u příchozích paketů mění MAC adresu cíle podle interní tabulky. Tento režim je v konfiguraci skoro vždy nesprávně nazýván „bridge“.

3. Zařízení se může za každý počítač v LAN jakoby znovu asociovat na AP. Všechny počítače v LAN pak vypadají jako klienti AP a dostávají data na svoji skutečnou MAC adresu. S tímto řešením jsem se setkal jen u jednoho typu hardwaru z roku 2003. Žádný dnes dostupný hardware nezávisle na výrobci (Ovislink, Zcomax, Straightcore, …) tento postup již nepoužívá. Z dnešního pohledu je tedy toto originální řešení pouze historickou kuriozitou.

U 1. a 2. řešení opravdu platí, že klient je jednoznačně definován MAC adresou, Wi-Fi zařízení přijímá i odesílá rámce pouze s jednou MAC adresou. Problém by byl pouze u hardwaru používajícího třetí řešení. Ten se ale dnes již na trhu nevyskytuje a případných několik zákazníků, kteří by jej mohli stále používat, můžeme zanedbat. Rozlišování klientů podle MAC adresy je tedy dostačující.

### 3.2.2 Spravedlivé dělení pásma

Ke splnění požadavku P2 se zdají být nejvhodnější algoritmy pro *Packet Scheduling* popsané v kapitole 2.2.

Chování algoritmu *Deficit Round Robin* odporuje požadavku P5: přijde-li paket pro klienta, jehož fronta je prázdná, bude čekat, až všichni ostatní klienti odešlou alespoň jeden paket. Algoritmus *Stochastic Fair Queueing* hašuje datové toky podle zdrojové a cílové adresy, nerozlišuje jednotlivé klienty, nesplňuje tedy požadavek P1. Zbývá pouze *Fair Queueing*, který oběma podmínkám vyhovuje. Jeho časová složitost $O(\log n)$ není problém, $n$ je zde počet klientů a ten dosahuje nanejvýš desítek.

Náznak, jak splnit požadavek P7 (rozlišení efektivity přenosů), byl ukázán v kapitole 2.3.3: přenosová rychlost se použije jako váha pro algoritmus *Weighted Fair Queueing*. Tento postup by ale nebyl zcela přesný, efektivita přenosu nezávisí jen na přenosové rychlosti, ale i na délce paketů. Navíc pokud přenos selže, může být několikrát opakován.

Podle původní definice *Fair Queueing* z kapitoly 2.2.1 má být $F_i^k$ *čas*, kdy bude odesílání paketu dokončeno. Nejpřesnější proto bude pracovat se skutečným časem přenosu místo délky paketu. Vzorec (2.4) se pak změní na

$$F_i^k = \max(V(a_i^k), F_i^{k-1}) + D_i^k,$$

kde $D_i^k$ je čas přenosu (*duration*) paketu $k$ od klienta $i$.

Takovýto algoritmus ale nelze implementovat: doba přenosu paketu závisí i na počtu retransmisí, je tedy známa až po dokončení přenosu. V okamžiku, kdy je vybírán další paket k odeslání, je znám pouze čas začátku přenosu $S_i^k$. Použijeme-li jej místo času dokončení přenosu, dostaneme vzorec

$$S_i^k = \max(V(a_i^k), F_i^{k-1})$$

Narozdíl od první úpravy tato již mění sémantiku *Fair Queueing*. Základní princip ale zachovává, včetně nevýhody popsané v kapitole 2.2.4: o přenosové pásmo se v daný čas dělí rovnoměrně aktivní datové toky (ty, které mají nějaké pakety ve frontě). Brouzdal, který přenáší data nepravidelně v dávkách, bude znevýhodněn proti Stahovači, který přenáší data souvisle. Brouzdal využije nanejvýš polovinu pásma, zatímco Stahovač je bude mít po většinu času pro sebe celé a jen občas se o ně podělí s Brouzdalem. Každý klient by ale měl mít právo přenést stejné množství dat, takže toto chování je v rozporu s požadavkem P2.

Popsané chování je však základem algoritmu *Fair Queueing*. Pokud bychom jej chtěli změnit, aby algoritmus bral do úvahy čas, kdy klient žádná data nepřenášel, museli bychom ze vzorce odstranit výběr maxima:

$$S_i^k = F_i^{k-1}$$

Tím by ale algoritmus zdegeneroval na obyčejný čítač, který by už s původním *Fair Queueing* neměl nic společného. A nebyl by ani použitelný: klient, který by se nově připojil, by měl veškeré pásmo pro sebe, dokud by nepřenesl tolik dat, jako předtím ostatní – latence by byla teoreticky nekonečná. Je tedy zřejmé, že algoritmus *Fair Queueing* není použitelný pro spravedlivé dělení pásma na Wi-Fi a ani jeho modifikace nevedou k použitelnému řešení.

Výsledkem úprav však byl jednoduchý čítač a existují algoritmy řízení toku, které s právě takovým čítačem pracují – byly popsány v kapitole 2.1. Může být některý z nich vhodný pro spravedlivé dělení pásma na Wi-Fi?

Algoritmus *Jumping Window* by trpěl podobným problémem jako *Deficit Round Robin*: na začátku okna se čítače vynulují, mohlo by se tedy stát, že paket pro Hráče bude čekat, než se odešlou pakety pro všechny Stahovače, což je v rozporu s požadavkem P5.

Algoritmus *Moving Window* tímto problémem netrpí, ale za cenu velké režie. Místo jednoho čítače potřebuje seznam paketů, paměťová složitost je tedy $O(n)$, kde $n$ je počet paketů odeslaných během časového okna. Maximální počet paketů je těžko předvídatelný (závisí na velikosti paketů a přenosové rychlosti), nelze tedy paměť alokovat staticky, musel by se nejspíše použít spojový seznam. Časová složitost zjištění aktuální hodnoty čítače by navíc byla také $O(n)$ – může být nutné odečíst staré pakety až za celou délku okna.

Režie *Moving Window* je ale zbytečná, existuje kompromis. Na začátku okna může být čítač nastaven na spotřebovaný čas za předchozí okno. Tím se vyřeší problém s latencí na začátku okna a přitom časová i paměťová složitost zůstane $O(1)$.

Zbývá ještě vyřešit otázku velikosti okna. Okno určuje maximální latenci: pokud 1. klient přenášel data souvisle po dobu celého okna a 2. klient vůbec a na začátku dalšího okna začne přenášet druhý, první během okna nepřenese nic. Aby tento výpadek přenosu nedělal problém TCP spojení, měla by být velikost okna nejvýše několik stovek milisekund, například 200 ms. Pro VoIP by byla i latence 200 ms příliš velká, ale VoIP zatíží přenosové pásmo nanejvýš na několik procent[1]. Zpoždění je nejvýše rovné času, na který zabral pásmo přenos VoIP paketů, a jednotky procent z 200 ms jsou zanedbatelné desítky milisekund.

Poslední detail je splnění požadavku P8, řízení uploadu. Není možné ovlivnit, kdy bude který klient vysílat, to by vyžadovalo úpravu na straně klientů, což zakazuje požadavek P9. Jediná možnost je využít vlastnosti protokolu TCP. Čas přenosu

[1]Uvažujme nejběžnější kodek G.711. Ten přenáší oběma směry 35 UDP paketů velikosti 172B za vteřinu, to je s hlavičkami IP a linkové vrstvy 250B. Přenos 70 rámců velikosti 250B rychlostí 11Mbit/s podle výpočtu z kapitoly 1.5 zabere přibližně 50ms, tedy 5% vysílacího času.

paketů od klienta bude také přičítán k čítači. Odchozí pakety od klienta pak budou zdržovat příchozí pakety včetně potvrzení (ACK), na což by měl protokol TCP reagovat zpomalením.

### 3.2.3 Dlouhodobý průměr

To, co bylo navrženo v předchozí kapitole 3.2.2, může být pouze základ algoritmu. Pracuje jen s údaji za dvě časová okna dlouhá dohromady nanejvýš vteřinu, což rozhodně nelze považovat za dlouhodobou paměť. Pro splnění požadavku P3 je nutné přidat další úroveň, výpočet dlouhodobého průměru zatížení přenosového pásma. Tento průměr se pak použije jako váha, podobně jako u algoritmu *Weighted Fair Queueing*.

Součástí požadavku P3 je zapomínání starých dat, protože chování uživatelů se může v čase měnit. To lze opět řešit časovým oknem a některým z algoritmů popsaných v kapitole 2.1. Nejjednodušší algoritmus *Jumping Window* by zde mohl fungovat, reset čítačů někdy uprostřed noci by pravděpodobně mnoho klientů nezaznamenalo. Přesto není ideální – už proto, že vyžaduje od uživatele zadání času resetu a také přístup k správně nastaveným hodinám.

Použití *Moving Window* by znamenalo příliš velkou paměťovou náročnost. Uvažujme velikost okna 100 ms a paměť na celý den. Při velikosti záznamu 4 bajty vychází potřebná paměť skoro 3.5 MB na klienta. A záleží na počtu klientů, kteří se připojili během dne, ne na počtu aktivních klientů. 20 klientů je poměrně konzervativní odhad, a výsledných 69 MB je už mimo možnosti embedded platforem. Nemluvě o tom, že by bylo snadné udělat na AP denial-of-service útok posláním mnoha paketů z různých falešných MAC adres, což by způsobilo vyčerpání paměti.

Řešením paměťové náročnosti je algoritmus *EWMA Window*, popsaný v kapitole 2.1.6. Ten si vystačí s jedinou proměnnou, která bude aktualizována vždy na konci časového okna podle vzorce (2.1). Naměřenými hodnotami, z nichž je počítán průměr, jsou zde podíly času, na který klient zabral přenosové pásmo – tedy podíl součtu časů přenosu paketů k délce okna $\tau$:

$$\bar{x}_n = \frac{\alpha}{\tau} \sum D_i + (1 - \alpha)\bar{x}_{n-1} \tag{3.1}$$

Před použitím je nezbytné zvolit vhodnou hodnotu parametru $\alpha$. Vypočítat ji lze podle vzorce (2.2). Mají-li mít hodnoty za posledních 24 hodin váhu 99%, vychází pro $\tau = 200$ ms hodnota $\alpha$ přibližně $10^{-5}$:

$$\alpha = 1 - \sqrt[k]{1-p} = 1 - \sqrt[(86400/0.2)]{1-0.99} \doteq 1.066 \cdot 10^{-5}$$

Nyní je možné propojit exponenciální průměr s aktualizací čítače času přenosů v rámci okna. V předchozí kapitole 3.2.2 byl definován jednoduchý mechanismus aktualizace čítače $C_i$ klienta $i$ po odeslání paketu $k$, které zabralo čas $D_k$:

$$C_i := C_i + D_k$$

Triviální řešení by bylo jednoduše čas $D_i$ násobit hodnotou exponenciálního průměru $A_i$:

$$C_i := C_i + A_i D_k$$

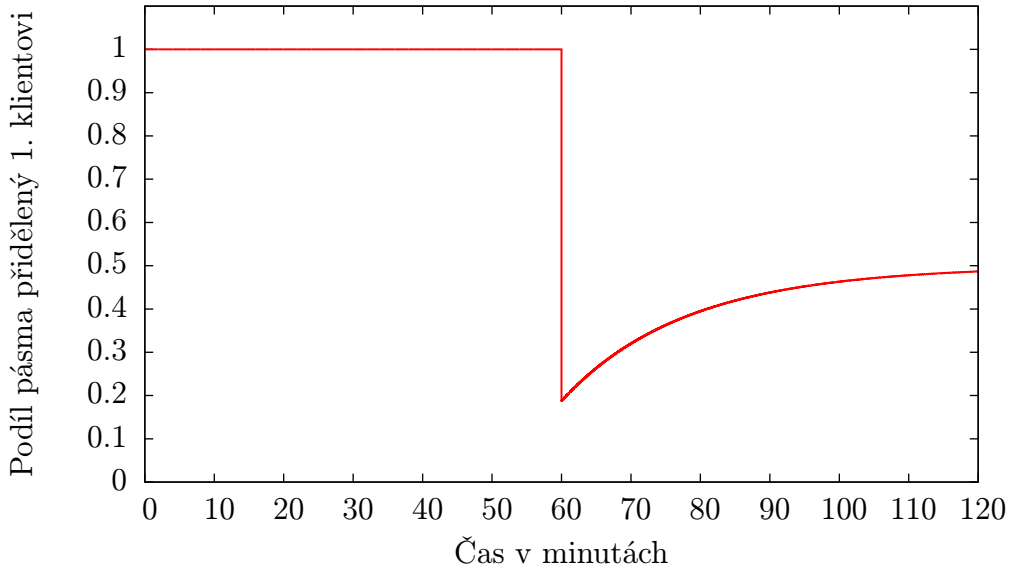
Hodnota exponenciálního průměru ale může být nulová. Po spuštění AP budou všechny průměry nulové, k čítači přenesených dat v rámci okna by se tedy přičítaly samé nuly a základní algoritmus popsaný v předchozí kapitole 3.2.2 by vůbec nefungoval. Váha proto bude z průměru odvozována složitěji tak, aby byla vždy alespoň 1. Navíc přibude parametr $\beta$, který určuje, jak velký význam má mít exponenciální průměr:

$$C_i := C_i + D_k(1 + \beta A_i) \tag{3.2}$$

Přenosové pásmo se pak bude mezi dva klienty dělit nanejvýš v poměru $1 : (1 + \beta)$.

Nevýhoda daná použitím *EWMA Window* je, že výsledné chování algoritmu je poměrně těžko odhadnutelné. Základní představu o chování *EWMA Window* lze získat z grafu 2.1. Ten ale zobrazuje jen vývoj průměru v čase u osamoceného klienta. Pokud se o přenosové pásmo dělí dva klienti, dojde ke zpětné vazbě: průměr ovlivňuje poměr, v němž se klienti dělí o vysílací čas, a z využitého času se zase počítá průměr.

Graf 3.3 ukazuje vývoj poměru, v němž se klienti dělí o pásmo, přenáší-li první data souvisle hodinu plnou rychlostí a pak začne přenášet druhý. Koeficient exponenciálního průměru $\alpha$ má hodnotu $10^{-4}$, koeficient váhy $\beta$ hodnotu 4 a délka časového okna $\tau$ je 200 ms. Graf zobrazuje podíl pásma přidělený prvnímu klientovi, druhý dostane doplněk do jedné.




Obrázek 3.3: Vývoj poměru, v němž se dělí dva klienti o přenosové pásmo

## 3.3 Pseudokód algoritmu

Nyní zbývá jen shrnout části výsledného návrhu algoritmu z předchozí kapitoly 3.2 ve formě pseudokódu. Výsledkem nebude jeden souvislý kód, ale jednotlivé procedury, reagující na různé události (např. přijetí a odeslání paketu).

Algoritmus je parametrizován třemi konstantami:

- $\boldsymbol{\alpha}$ – parametr EWMA, $0 < \alpha < 1$, vyšší hodnota znamená kratší „paměť“
- $\boldsymbol{\beta}$ – určuje, jak EWMA ovlivňuje dělení pásma mezi klienty během časového okna, $\beta \geq 0$
- $\boldsymbol{\tau}$ – délka časového okna v sekundách, $\tau > 0$

Příchod nového rámce k odeslání

**Vstup** : Seznam klientů clientList, rámec k odeslání frame

client ← `GetClientByMAC`(clientsList, frame.destination);
`Enqueue`(client.queue, frame);

Výběr dalšího rámce k odeslání

**Vstup** : Seznam klientů clientList
**Výstup**: Rámec k odeslání frame

client ← klient s neprázdnou frontou a nejnižší hodnotou client.weightedCounter;
**return** `Dequeue`(client.queue);

Odeslání rámce dokončeno

**Vstup** : Seznam klientů clientList, odeslaný rámec frame

client ← `GetClientByMAC`(clientsList, frame.destination);
`CountFrameIn`(client, frame);

Přijetí rámce dokončeno

**Vstup** : Seznam klientů clientList, přijatý rámec frame

client ← `GetClientByMAC`(clientsList, frame.source);
`CountFrameIn`(client, frame);

**Funkce** `CountFrameIn`(client, frame)

**Vstup** : Klient client, rámec k odeslání frame
duration ← `TransmitDuration`(frame);
client.durCounter ← client.durCounter + duration;
client.weightedCounter ← client.weightedCounter + duration · (1 + $\beta$ · client.average);

Konec časového okna (pravidelně po $\tau$ sekundách)

**Vstup** : Seznam klientů clientList

**foreach** client *in* clientsList **do**
    client.average $\leftarrow \alpha \cdot$ client.durCounter $\div \tau + (1 - \alpha) \cdot$ client.average;
    client.weightedCounter $\leftarrow$ client.durCounter $\cdot (1 + \beta \cdot$ client.average);
    client.durCounter $\leftarrow 0$;
**end**

V pseudokódu chybí definice některých částí, které jsou již věcí implementace:

**Výběr klienta s nejnižší hodnotou čítače**
Je možné použít sekvenční průchod přes všechny klienty s časovou složitostí $O(n)$, nebo nějakou stromovou datovou strukturu, například haldu, která pracuje s asymptotickou časovou složitostí $O(\log n)$. Není však jisté, že bude v praxi halda skutečně rychlejší, bude-li klientů jen několik (do deseti).

**Funkce** `GetClientByMAC`
Vyhledat klienta podle MAC adresy je možné opět sekvenčním průchodem, a nebo použitím hašovací tabulky.

**Funkce** `TransmitDuration`
Výpočet, jak dlouho přenos trval, závisí na tom, kolik údajů poskytne hardware. Nutností je přenosová rychlost, ale výpočet může dále zpřesnit informace o počtu retransmisí, nebo zda byl využit mechanismus RTS/CTS.

Popsaný algoritmus teoreticky splňuje skoro všechny požadavky z kapitoly 3.1. Jedinou nedořešenou otázkou je splnění požadavku P10: nebude implementace algoritmu vyžadovat zásah do hardwaru? Tím se budu zabývat později v kapitole 5.1.

Nyní je na čase ověřit, zda se algoritmus chová podle teoretických předpokladů i ve skutečné Wi-Fi síti. Chování Wi-Fi sítě je nedeterministické, závisí na vnějších vlivech (rušení) a navíc je náhoda přímo součástí přístupové metody CSMA/CA. Má-li být výsledek testů opakovatelný, musí být provedeny v simulačním prostředí.

# Kapitola 4

# Simulace chování algoritmu

Pro ověření chování algoritmu je nutné najít simulační prostředí, které dostatečně přesně modeluje linkovou vrstvu IEEE 802.11. Musí též být open-source, aby bylo možné provést úpravy nezbytné pro implementaci řízení toku. Oba tyto požadavky splňuje software *OMNeT++*.

## 4.1 Úvod do OMNeT++

OMNeT++ [24] je open-source simulační prostředí dostupné pro akademické použití zdarma, licence dovoluje distribuovat modifikovaný kód. OMNeT++ je ale pouze obecná knihovna pro diskrétní simulace, konkrétní simulaci provádí oddělený software, simulační model. Pro simulaci počítačových sítí existuje model INET [25], šířený pod licencí GPL.

Vývoj OMNeT++ stále pokračuje, aktuální verze má číslo 4.0. Ta ale ještě nebyla k dispozici v době, kdy jsem s vývojem začínal. Pro OMNeT++ 4.0 navíc ani nebyla vydána stabilní verze modelu INET, existuje jen vývojová verze v GIT repozitáři. Simulace proto byly prováděny na OMNeT++ ve verzi 3.3p1 a modelu INET ve verzi 20061020.

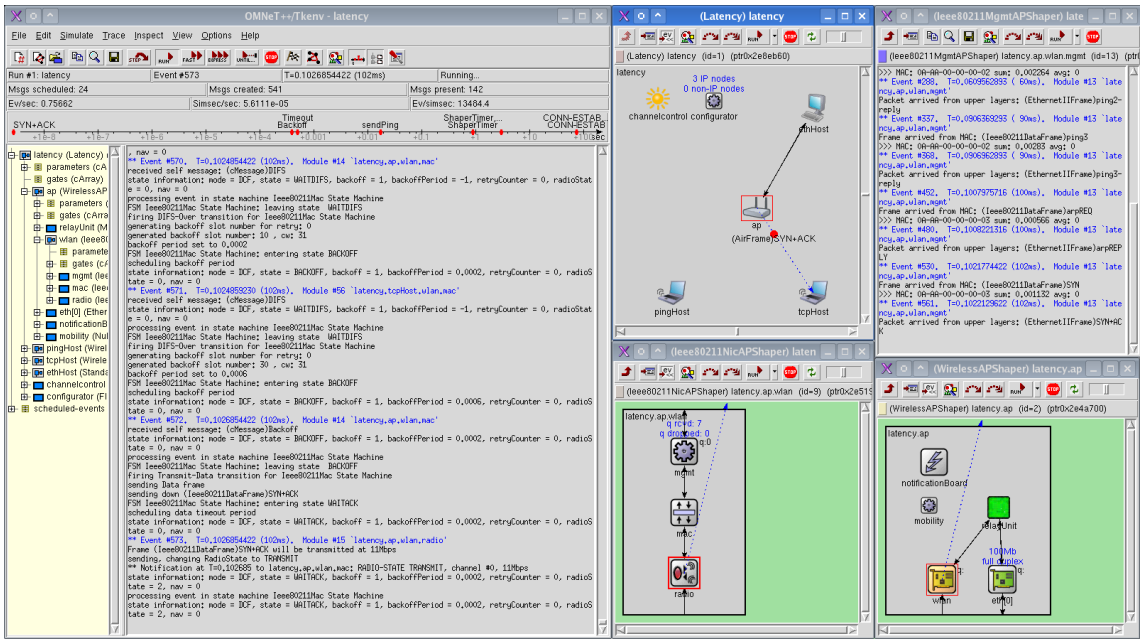
Základem diskrétní simulace jsou objekty a zprávy. Objekty simulují entity reálného světa, které spolu komunikují posíláním zpráv. U simulace sítě jsou objekty jednotlivé počítače a zprávy přenášené rámce. Objekty mohou být složeny z jiných objektů, např. přístupový bod z bezdrátové a ethernetové části. Stejně tak zprávy mohou v sobě nést další zprávy, což je ideální pro simulaci TCP/IP, kde je např. TCP fragment zabalen do IP paketu a ten do ethernetového rámce.

Každá zpráva má definován čas doručení, pomocí něhož se simuluje doba přenosu: přijetí zprávy bude naplánováno na čas, kdy bude přenos dokončen. Mechanismus zpráv se používá i pro časovače (např. ACK timeout u Wi-Fi): objekt pošle sám sobě zprávu s časem doručení nastaveným na dobu, kdy timeout vyprší.

V OMNeT++ se při odesílání zpráv neadresuje přímo objekt příjemce, ale spoj, kterým má být zpráva odeslána. Spojení objektů se definuje až v konfiguraci, což

umožňuje velkou flexibilitu, pro vytvoření nové simulace není nutný zásah do kódu objektů. Drobnou nevýhodou je, že linka vždy spojuje právě dva objekty, všechny posílané zprávy jsou tedy *unicast*. U Wi-Fi každý přenos zachytí všichni klienti, zpráva tedy musí být rozeslána všem pomocí zvláštního objektu, simulujícího přenosové pásmo.

OMNeT++ má i grafické prostředí, jehož ukázka je na obrázku 4.1. V hlavním okně (vlevo) je logaritmická časová osa se zprávami, které čekají na doručení (nahoře), strom se strukturou objektů v simulaci (vlevo) a výpis událostí (vpravo). Vedle toho je graficky znázorněna struktura celé simulace (uprostřed) a je možné zobrazit strukturu jednotlivých objektů (vpravo dole) a seznam událostí jednoho objektu (vpravo nahoře). Vedle toho lze prohlížet strukturu zpráv a některé datové struktury objektů.



Obrázek 4.1: Grafické prostředí OMNeT++

Přenos zpráv může být plynule animován, to ale nelze zaměňovat se spojitou simulací: animace dvou přenosů, které probíhají v jednu chvíli (např. jedno vysílání AP, které přijmou oba klienti), budou probíhat jedna po druhé, ne současně.

OMNeT++ je naprogramován v jazyce C++, grafické prostředí v jazyce Tcl/Tk. Objekty simulačních modelů jsou také psány v jazyce C++, anebo skládány z jiných objektů pomocí speciálního jazyka NED (*NEtwork Description*).

Simulace je tvořena objektem, který reprezentuje síť (popsaným v jazyce NED), a konfigurací v podobě INI souboru, kde jsou definovány parametry všech objektů. Konfigurace bohužel bývá dosti dlouhá, protože je nutné zadat všechny parametry, vynechané se výchozími hodnotami nedoplní. V textu práce proto budou uvedeny pouze výtahy z konfigurace, celá je k dispozici na přiloženém CD v adresáři `simul`.

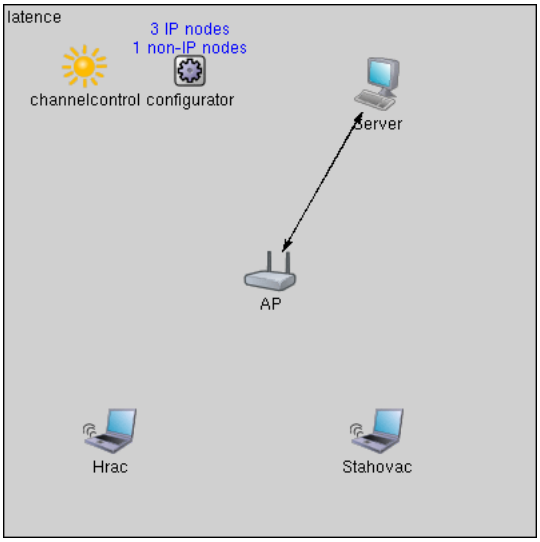
Návod, jak použít OMNeT++ k vytvoření simulace, je v obsáhlém dokumentu [26]. Byl sice napsán pro verzi 3.0 a naposledy aktualizován v roce 2004, ale většina

obsažených informací je ve verzi 3.3 stále platná. Dokument pokrývá OMNeT++ včetně syntaxe jazyka NED, ne však simulační model INET. Dokumentace k modelu INET vygenerovaná ze zdrojových kódů je v [27].

## 4.2 Model Wi-Fi sítě

Zatímco poměr, v němž se budou dělit klienti o pásmo, je možné vypočítat docela přesně (viz graf 3.3), odhad latence je složitější. Latence závisí na chování přístupové metody CSMA/CA, vyžaduje tedy přesnou simulaci linkové vrstvy.

Podle kapitoly 1.3 jsou největším zdrojem latence Stahovači a nejcitlivější na vysokou latenci Hráči. Testovací síť se tedy bude skládat z přístupového bodu, Hráče a Stahovače. Vedle nich musí být v síti nějaký server, který bude fungovat jako protistrana komunikace obou klientů. Server bude připojen k přístupovému bodu Ethernetem, přístupový bod tedy bude fungovat jako most mezi Wi-Fi a Ethernetem. Schéma sítě tak, jak je zobrazuje OMNeT++, je na obrázku 4.2.




Obrázek 4.2: Modelová Wi-Fi síť

INET nabízí dva modely Wi-Fi sítě: jednoduchý, kde jsou klienti automaticky připojeni na AP, a složitější, který simuluje roaming – klienti vyhledávají AP a asociují se na něj. Algoritmus řízení toku je navržen pro síť se stacionárními stanicemi, které jsou připojeny neustále, jednoduchý model je proto zcela dostačující.

Definice sítě v jazyce NED je ve výpisu 4.1. Nepodstatné detaily jako souřadnice objektů v grafické reprezentaci byly vynechány. Kód si zaslouží několik vysvětlivek:

- `WirelessAPWithEthSimplified` je přístupový bod s bezdrátovým a ethernetovým rozhraním a přepínačem mezi nimi.
- `WirelessHostSimplified` je klient AP s podporou TCP/IP (protokoly TCP, UDP a ICMP (ping)).

- `StandardHost` je počítač s ethernetovým rozhraním a TCP/IP (protokoly TCP, UDP a ICMP (ping)).
- `ChannelControl` simuluje přenosové pásmo – rozesílá odeslané rámce všem bezdrátovým rozhraním.
- `FlatNetworkConfigurator` automaticky přiřazuje počítačům IP adresy ze zadaného rozsahu. Parametr `moduleTypes` určuje typy objektů, kterým mají být adresy přiřazeny, parametr `nonIPModuleTypes` ty ostatní.
- Klíčovým slovem `connections` se definují spojení objektů. Bezdrátová rozhraní se nespojují, to zařídí automaticky `ChannelControl`, definován je zde pouze ethernetový spoj mezi AP a serverem (v obou směrech).

```
module Latence
    submodules:
        AP: WirelessAPWithEthSimplified;
        Hrac: WirelessHostSimplified;
        Stahovac: WirelessHostSimplified;
        Server: StandardHost;
        channelcontrol: ChannelControl;
        configurator: FlatNetworkConfigurator;
            parameters:
                moduleTypes = "WirelessHostSimplified StandardHost",
                nonIPModuleTypes = "WirelessAPWithEthSimplified",
                networkAddress = "192.168.0.0",
                netmask = "255.255.0.0";
    connections nocheck:
        AP.ethOut++ --> Server.ethIn++;
        Server.ethOut++ --> AP.ethIn++;
endmodule
```

Výpis 4.1: Definice sítě v jazyce NED

Největší vliv na latenci má délka fronty v přístupovém bodu, jak bylo ukázáno v kapitole 1.3. Aby simulace odpovídala realitě, musí být tato délka stejná, jako u skutečných přístupových bodů. Zvolil jsem hodnotu 199, kterou používá Linux[1].

Model bude simulovat nejhorší případ, kdy má Stahovač velmi slabý signál a AP s ním komunikuje nejnižší modulační rychlost 1 Mbit/s. Právě v této situaci je řízení toku nejpotřebnější. Model Wi-Fi v OMNeT++ bohužel neumožňuje nastavit pro různé klienty AP různou přenosovou rychlost, nastavení je pro Wi-Fi rozhraní přístupového bodu pevné. To by ale výsledek nemělo příliš ovlivnit, pakety pro Hráče jsou malé, doba jejich přenosu proto nebude významná složka latence.

[1] Zjištěno na Wi-Fi kartě s chipsetem Atheros a ovladačem Madwifi příkazem `ifconfig wifi0`. Délka fronty je číslo u `txquelen` ve výstupu příkazu.

Podstatné části konfigurace Wi-Fi jsou ve výpisu 4.2. Přenosová rychlost se nastavuje dvakrát: zvlášť pro fyzickou a linkovou vrstvu. Mechanismus RTS/CTS je vypnutý, model simuluje síť bez skrytých stanic.

```
**.ap.wlan.mac.address = "10:00:00:00:00:00"
**.mgmt.accessPointAddress = "10:00:00:00:00:00"
**.mgmt.frameCapacity = 199
**.radio.bitrate = 1e6
**.wlan.mac.bitrate = 1e6
**.wlan.mac.rtsThresholdBytes = 3000
```

Výpis 4.2: Konfigurace Wi-Fi

Aktivita hráče bude simulována pingem, který zároveň měří dobu odezvy. Ping bude napodobovat VoIP hovor používající kodek G.711: 35 UDP paketů o velikosti 172B za vteřinu oběma směry. Obsah ICMP paketu tedy bude mít velikost 172 bajtů a budou posílány v intervalu 0.03 sekundy. Základ konfigurace je ve výpisu 4.3.

```
*.Hrac.pingApp.destAddr="Server"
**.pingApp.packetSize=172
**.pingApp.interval=0.03
**.pingApp.startTime=0
**.pingApp.printPing=true
```

Výpis 4.3: Konfigurace pingu u Hráče

Aktivita Stahovače bude simulovat Bittorent, kde se soubory stahují z mnoha zdrojů zároveň. Stahovač tedy naváže deset TCP spojení na Server a po všech mu bude server posílat data neomezeně dlouho (v konfiguraci je nastaven objem dat 1 GB, ale tolik se určitě za dobu běhu simulace nepřenese). Základ konfigurace TCP spojení Serveru je ve výpisu 4.4, základ konfigurace Stahovače ve výpisu 4.5.

```
*.Server.numTcpApps=10
*.Server.tcpApp[0].port=1000
                ...
*.Server.tcpApp[9].port=1009

*.Server.tcpApp[*].active=false
*.Server.tcpApp[*].connectAddress=""
*.Server.tcpApp[*].connectPort=-1
*.Server.tcpApp[*].tOpen=0
*.Server.tcpApp[*].tSend=0.2
*.Server.tcpApp[*].sendBytes=1e9  # 1GB
```

Výpis 4.4: Konfigurace TCP spojení na Serveru

Výsledek simulace po jedné minutě běhu je ve výpisu 4.6: průměrná latence se blíží ke dvěma vteřinám! To činí Wi-Fi připojení naprosto nepoužitelné nejen

```
*.Stahovac.numTcpApps=10
*.Stahovac.tcpApp[0].connectPort=1000
                  ...
*.Stahovac.tcpApp[9].connectPort=1009

*.Stahovac.tcpApp[*].active=true
*.Stahovac.tcpApp[*].port=-1
*.Stahovac.tcpApp[*].connectAddress="Server"
*.Stahovac.tcpApp[*].tOpen=0.1
*.Stahovac.tcpApp[*].tSend=0.2
*.Stahovac.tcpApp[*].sendBytes=0
```

Výpis 4.5: Konfigurace TCP spojení Stahovače

pro VoIP, ale i pro jakékoli interaktivní aplikace včetně vzdáleného přístupu (SSH, VNC). Dvouvteřinové zpoždění způsobí i znatelné zpomalení načítání WWW stránek. Výsledek simulace ale odpovídá realitě, podobná latence byla skutečně pozorována v síti Praha12.Net.

Ztráta každého osmdesátého paketu navíc dokazuje, že pakety Stahovače zaplnily celou frontu přístupového bodu. Nyní nadešel čas odpovědět na základní otázku: je algoritmus řízení toku navržený v kapitole 3.3 řešením předvedené situace?

```
sent: 2000   drop rate (%): 1.25
round-trip min/avg/max (ms): 4.41913/1813.74/2676.66
stddev (ms): 490.911   variance:0.240994
```

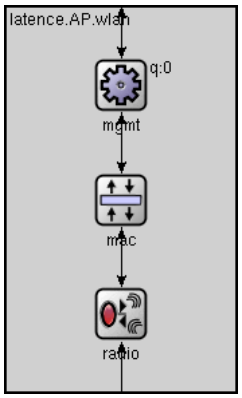
Výpis 4.6: Výsledek simulace (latence Hráče)

## 4.3 Implementace řízení toku v OMNeT++

Implementace řízení toku si rozhodně nevystačí se skládáním objektů pomocí jazyka NED, model INET ani neobsahuje žádnou podporu pro *QoS*. Bude nezbytný zásah do C++ kódu objektů, jimiž je tvořen přístupový bod. Objekt `Ieee80211NicAP` reprezentující Wi-Fi rozhraní přístupového bodu se skládá ze tří objektů:

- `Ieee80211MgmtAP` modeluje řízení sítě (přihlašování a odhlašování klientů), ale ve zjednodušené verzi `Ieee80211MgmtAPSimplified` nedělá skoro nic.
- `Ieee80211Mac` modeluje linkovou vrstvu a přístupovou metodu CSMA/CA.
- `Ieee80211Radio` modeluje fyzickou vrstvu (příjem a vysílání rámců a případné kolize).

Vztah mezi těmito objekty je graficky znázorněn na obrázku 4.3.

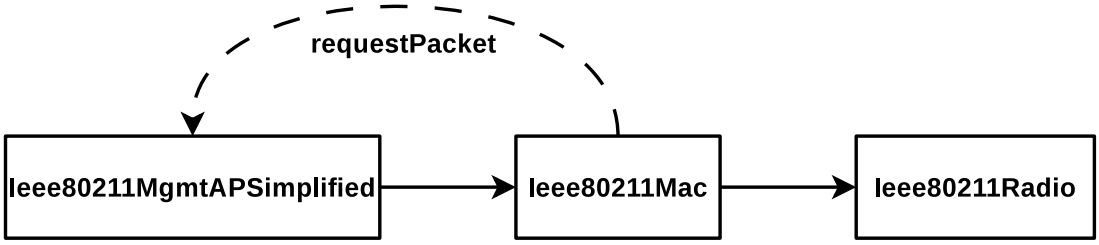



Obrázek 4.3: Součásti modelu Wi-Fi rozhraní

Algoritmus řízení toku musí pracovat nad frontou, kde čekají pakety na odeslání, z důvodů uvedených v kapitole 2.2.4. Každý z objektů má ale automaticky alespoň jednu frontu příchozích zpráv – která z nich je ekvivalentem fronty paketů?

Odpověď na tuto otázku se skrývá v objektu `Ieee80211Mac`. Jedním z jeho parametrů je `queueModule`, určující právě hledaný objekt s frontou paketů. V definici `Ieee80211NicAPSimplified` je tímto objektem `Ieee80211MgmtAPSimplified`. `Ieee80211MgmtAPSimplified` implementuje rozhraní `IPassiveQueue`: frontu, která posílá pakety jen na vyžádání.

Celá interakce funguje následovně: `Ieee80211Mac` požádá zavoláním metody `requestPacket` objekt `Ieee80211MgmtAPSimplified` o poslání dalšího paketu. Objekt `Ieee80211MgmtAPSimplified` buď pošle paket (po spoji mezi objekty ve formě běžné zprávy), nebo v případně prázdné fronty zvýší čítač a paket pošle až ve chvíli, kdy se ve frontě nějaký objeví. Interakce je znázorněna na obrázku 4.4.




Obrázek 4.4: Interakce objektů v modelu Wi-Fi rozhraní

Předek `Ieee80211MgmtAPSimplified` je třída `PassiveQueueBase`, která má dvě základní metody s výstižnými jmény `enqueue` a `dequeue`. Tyto metody jsou naštěstí virtuální, takže není nutné přímo zasahovat do kódu `Ieee80211MgmtAPSimplified`, stačí vytvořit zděděnou třídu `Ieee80211MgmtAPSched`, která bude místo obyčejného FIFO implementovat algoritmus řízení toku.

Většina kódu `Ieee80211MgmtAPSched` je přímočará implementace pseudokódu z kapitoly 3.3. Příchod rámce k odeslání implementuje metoda `enqueue`, výběr rámce k odeslání metoda `dequeue`. Přijetí rámce na Wi-Fi rozhraní řeší metoda `handleDataFrame`. Časovač pro konce časových oken je implementován zprávou,

kterou objekt posílá sám sobě.

Zbývají činnosti nedefinované v pseudokódu. K vyhledání klienta podle MAC adresy je použit STL kontejner `map`. Výběr klienta s nejnižší hodnotou čítače je řešen obyčejným průchodem všech klientů – u diskrétní simulace doba zpracování zprávy nemá vliv na výsledek simulace, takže na rychlosti příliš nezáleží, simulace je i na běžném PC dostatečně rychlá.

Poslední krok chybějící v pseudokódu je výpočet času přenosu. V modelu Wi-Fi rozhraní bohužel objekt MAC vrstvy neoznamuje řídící vrstvě dokončení přenosu s časem jeho trvání. Nicméně vzhledem k tomu, že rádio používá pevnou modulační rychlost, je možné tento čas dopočítat, tak jako v kapitole 1.5. Jen je potřeba přidat přenosovou rychlost jako parametr `Ieee80211MgmtAPSched` – tak, jako už je parametrem linkové a fyzické vrstvy. Chybějící informace o počtu opakování přenosu nicméně způsobí malou nepřesnost simulace.

Objekt `Ieee80211MgmtAPSched` bude tedy mít navíc čtyři parametry oproti `Ieee80211MgmtAPSimplified`:

- `expfactor` – parametr $\alpha$ exponenciálního průměru
- `avgweight` – parametr $\beta$ (váha exponenciálního průměru)
- `wintime` – parametr $\tau$ (délka časového okna v sekundách)
- `bitrate` – přenosová rychlost v bit/s (musí se shodovat se stejnojmennými parametry objektů `Ieee80211Mac` a `Ieee80211Radio`)
- `ackrate` – přenosová rychlost potvrzení (ACK) v bit/s, `Ieee80211Mac` používá napevno 2 Mbit/s

Kompletní dokumentace metod objektu `Ieee80211MgmtAPSched` je v příloze A. Byla vygenerována z komentářů v kódu programem Doxygen tak jako dokumentace celého modelu INET [27]. Komentáře v objektech modelu INET jsou v angličtině, pro konzistenci jsou tedy anglicky i komentáře v kódu `Ieee80211MgmtAPSched` a z nich vygenerovaná dokumentace.

Nyní zbývá jen vytvořit složené objekty Wi-Fi rozhraní a celého přístupového bodu, které budou obsahovat řízení toku. Hierarchie těchto objektů musí kopírovat hierarchii původních objektů v modelu INET. Pro přehlednost jsou názvy původních objektů a ekvivalentních objektů s řízením toku v tabulce 4.1.

| | **Původní objekty** | **S řízením toku** |
|---|---|---|
| **Řídící vrstva** | `Ieee80211MgmtAPSimplified` | `Ieee80211MgmtAPSched` |
| **Wi-Fi rozhraní** | `Ieee80211NicAPSimplified` | `Ieee80211NicAPSched` |
| **Přístupový bod** | `WirelessAPWithEthSimplified` | `WirelessAPSched` |

Tabulka 4.1: Objekty OMNeT++ s řízením toku a bez něj

## 4.4 Výsledky simulace

Nyní je možné zopakovat simulaci z kapitoly 4.2 s přístupovým bodem implementujícím řízení toku a porovnat výsledky. Dojde k nějakému zlepšení, a bude toto zlepšení dostatečně významné?

Použití řízení toku si vyžádá několik úprav v simulaci. V definici sítě v jazyce NED musí být místo `WirelessAPWithEthSimplified` použit přístupový bod s řízením toku `WirelessAPSched`. Pak také musí být zadány parametry simulace.

OMNeT++ dovoluje definovat v INI souboru různé průběhy simulace, odlišující se jen některými parametry. Typ objektu je však obvykle zadán napevno v souboru NED. Naštěstí je ale možné učinit třídu objektu modelujícího přístupový bod parametrem simulace. Nutné změny v definici sítě v jazyce NED jsou ve výpisu 4.7.

```
module Latence
    parameters:
        ...
        APClass: string;
    submodules:
        AP: APClass like WirelessAPWithEthSimplified;
        ...
endmodule
```

Výpis 4.7: Definice sítě parametrizovaná třídou objektu

Zbývá ještě zadat parametry algoritmu řízení toku. Pro simulaci jsem použil hodnoty odvozené v kapitole 3.2: $\alpha = 10^{-5}$, $\beta = 4$, $\tau = 0.2$ s. Nutné změny v INI souboru s parametry simulace jsou ve výpisu 4.8.

```
*.APClass = "WirelessAPSched"
**.mgmt.ackrate = 2e6
**.mgmt.wintime = 0.2
**.mgmt.expfactor = 1e-5
**.mgmt.avgweight = 4.0
```

Výpis 4.8: Přidané parametry v INI souboru vyžadované řízením toku

Výsledek simulace je ve výpisu 4.9. Porovnání s výsledkem simulace bez řízení toku je v tabulce 4.2. Zlepšení je opravdu významné: maximální latence klesla na desetinu, průměrná latence se snížila dokonce třicetkrát. Zcela zmizely i ztráty paketů: Hráč má vlastní frontu, přeplnění fronty Stahovače na něj nemá vliv.

```
sent: 2000   drop rate (%): 0
round-trip min/avg/max (ms): 4.41913/56.8607/263.872
stddev (ms): 29.3966   variance:0.000864162
```

Výpis 4.9: Výsledek simulace s řízením toku (latence Hráče)

| | Bez řízení toku | S řízením toku | Podíl |
|---|---|---|---|
| Minimální latence (ms) | 4.419 | 4.419 | 1.0 |
| Průměrná latence (ms) | 1813.740 | 56.861 | 31.90 |
| Maximální latence (ms) | 2676.660 | 263.872 | 10.14 |
| Směrodatná odchylka (ms) | 490.911 | 29.397 | 16.70 |
| Ztrátovost (%) | 1.25 | 0.0 | — |

Tabulka 4.2: Naměřené hodnoty v simulaci

Výsledek je tedy poměrně dobrý – ale je nejlepší možný? Zkusme odhadnout minimální průměrnou latenci. Přenos samotného pingu po Wi-Fi trvá přibližně 2.5 ms, přenáší se tam a zpět, tedy celkem 5 ms. Přenos po Ethernetu k Serveru můžeme zanedbat. Podstatná část latence je způsobena čekáním:

- Ve chvíli, kdy chce Hráč odeslat ping, bude pravděpodobně AP přenášet paket pro Stahovače. Hráč tedy v průměru čeká polovinu přenosu paketu pro Stahovače.
- Poté, co skončí přenos pro hráče, bude Hráč soupeřit s AP o pásmo v přístupové metodě CSMA/CA. V polovině případů prohraje a bude čekat po dobu dalšího přenosu paketu pro Stahovače.
- Objekt `Ieee80211Mac` si udržuje ve své frontě stále dva rámce. Jeden bude průměrně v polovině přenosu, druhý čeká, a paket Hráče se může zařadit až za něj. Toto chování odpovídá ovladači skutečného hardwaru, což bude ukázáno později v kapitole 5.1.

Celkem tedy paket pro Hráče čeká na přenos 2.5 paketu pro Stahovače, což potrvá přibližně $2.5 \cdot (478 + 8 \cdot 1536)$ µs $\doteq$ 32 ms, po přičtení 5 ms vyjde přibližně 37 ms. To je poměrně blízko 56 ms dosaženým v simulaci, určitě není možné další násobné zlepšení. Algoritmus se tedy chová podle předpokladů: průměrná latence se blíží teoretické minimální hodnotě. Rozdíl způsobují občasné kolize a také TCP ACK pakety od Stahovače, které nebyly započítány.

Na hodnotě maximální latence se názorně projevuje to, že přístupová metoda 802.11 je nedeterministická: v závislosti na počáteční hodnotě náhodného generátoru se pohybuje mezi 170 a 280 ms. Takto velká latence vzniká, sejde-li se hned několik nešťastných náhod (opakované kolize kombinované s několikanásobnou „prohrou“ v přístupové metodě CSMA/CA). Nic z toho algoritmus řízení toku ovlivnit nemůže.

### 4.4.1 Simulace s uploadem Stahovače

První simulace vůbec neuvažovala upload Stahovače (kromě paketů TCP ACK). Používá-li ale Stahovač P2P sítě (například BitTorent), bude zároveň odesílat data, a to přibližně stejnou rychlostí, jako přijímat. V druhé simulaci proto bude po stejných TCP spojeních, jako posílá Server data Stahovači, posílat i Stahovač data Serveru.

K tomu stačí jednoduché nastavení `*.Stahovac.tcpApp[*].sendBytes=1e9` v INI souboru.

Tuto simulaci už ale nelze provádět na modulační rychlosti 1 Mbit/s: mezi odesláním paketů od Hráče průměrně dojde k jednomu přenosu paketu od Stahovače a jednoho pro Stahovače. Ty dohromady trvají asi 25.4 ms, po přičtení 5 ms na přenos paketů Hráče vychází 30.4 ms, což je víc než interval posílání pingů (30 ms). A to neuvažujeme kolize a další režii. Hráč by tedy své pakety nestíhal odesílat a řízení toku by na tom nemohlo nic změnit. Druhou simulaci je proto nutné provést s vyšší přenosovou rychlostí, alespoň 2 Mbit/s.

Výsledky simulace bez řízení toku a s řízením toku jsou v tabulce 4.3. Rozdíl je menší, ale stále významný: průměrná latence je stále nižší skoro třicetkrát, maximální již pouze sedmkrát. Menší rozdíl maximální latence plyne z delšího čekání při několikanásobné „prohře“ v přístupové metodě. Algoritmus ale stále funguje poměrně dobře – snižuje latenci tak, jak je to jen možné.

| | **Bez řízení toku** | **S řízením toku** | **Podíl** |
|---|---|---|---|
| Minimální latence (ms) | 2.691 | 2.691 | 1.0 |
| Průměrná latence (ms) | 1529.880 | 56.715 | 26.97 |
| Maximální latence (ms) | 2094.990 | 283.902 | 7.38 |
| Směrodatná odchylka (ms) | 234.082 | 47.331 | 4.95 |
| Ztrátovost (%) | 14.3 | 0.0 | — |

Tabulka 4.3: Naměřené hodnoty v simulaci s uploadem Stahovače

Simulací bylo ověřeno, že se algoritmus chová podle předpokladů a dokáže významně snížit latenci sítě. Zbývá poslední, ale důležitá otázka: je možné jej implementovat ve skutečných operačních systémech a na skutečném hardwaru?

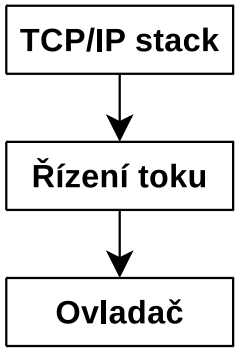
# Kapitola 5

# Implementace algoritmu

Skutečné operační systémy jsou mnohem složitější než simulace v OMNeT++. Prvním krokem k implementaci algoritmu je proto pochopení jejich struktury a nalezení, jak do ní algoritmus začlenit.

## 5.1 Struktura TCP/IP v operačních systémech

Jediným aktuálním zdrojem informací o implementaci TCP/IP v operačních systémech Linux a FreeBSD jsou jejich zdrojové kódy[1]. Základní struktura je v obou systémech stejná a je znázorněna na obrázku 5.1. TCP/IP stack provádí směrování a rozhoduje, kterým síťovým rozhraním má paket odejít. Řízení toku je již specifické pro síťové rozhraní. Může zpožďovat pakety předávané ovladači hardwaru, případně měnit jejich pořadí.




Obrázek 5.1: Struktura TCP/IP v operačním systému

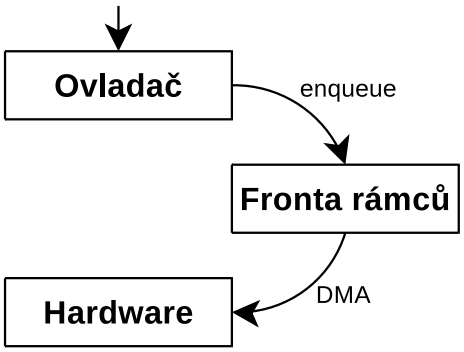
Řízení toku je v Linuxu nazýváno *Queueing discipline*, zkráceně *Qdisc*. Jeho obdobou ve FreeBSD je framework ALTQ přenesený z OpenBSD. Vedle ALTQ nabízí FreeBSD nástroj *Dummynet*, který však není vázán na síťové rozhraní. Nemůže tedy

[1]Linux nemá stabilní interní API a mění se rychlostí 1000 změn denně [28]. Publikované články proto velmi rychle zastarávají, o knihách ani nemluvě. Pro FreeBSD platí v menší míře totéž.

žádným způsobem reagovat na proměnlivou přenosovou rychlost Wi-Fi, může jen omezovat datový tok na pevně danou hodnotu.

V hypotetickém ideálním systému by fronta paketů mohla být pouze uvnitř řízení toku, které by ovladači poslalo paket, počkalo až bude odeslán a pak poslalo další. Na Wi-Fi však může začít další přenos během několika desítek mikrosekund – takto rychle by operační systém nemusel stihnout zareagovat. Rozhodnutí v *Queueing discipline* může být poměrně komplikované, operační systém navíc může čekat na uvolnění zámků a také obsluhovat hned několik Wi-Fi zařízení najednou (v síti Praha12.Net máme v provozu i router s osmi Wi-Fi kartami).

Hardware tedy má svoji vlastní frontu rámců, znázorněnou na obrázku 5.2. Ta je sice v paměti RAM, ale hardware si z ní bere rámce asynchronně přes DMA (*Direct Memory Access*). Sám prochází spojový seznam vytvořený ovladačem a pouze oznamuje dokončení přenosu rámce vyvoláním přerušení. Pořadí rámců, které byly do této fronty přidány, už nelze měnit – hardware může některý z nich právě odesílat. Délka fronty je navíc poměrně dlouhá, řádově stovky rámců[2]. Latenci v řádu vteřin ukázanou v kapitole 4.2 tedy může způsobit pouze tato fronta.




Obrázek 5.2: Fronta rámců v ovladači

Fronta rámců je interní pro ovladač, řízení toku nemá informace o její délce. Může pouze naslepo posílat ovladači pakety pevnou přenosovou rychlostí – o málo nižší, než jakou jsou pakety odesílány, aby se délka fronty držela na malých hodnotách. Na Wi-Fi ale nelze tuto rychlost stanovit, jak bylo ukázáno v kapitole 1.5. Proto žádný algoritmus řízení toku popsaný v kapitole 1.6 na Wi-Fi nemůže fungovat a řešení není ani vytvoření nového algoritmu řízení toku v rámci *Queueing discipline*.

Implementace algoritmu řízení toku se tedy nevyhne úpravám ovladače hardwaru. Především je nutné zkrátit frontu rámců připravených k odeslání. Nelze ji zkrátit na jediný rámec – hardware potřebuje ihned po dokončení přenosu vědět, zda se má znovu ucházet o přenosové pásmo v přístupové metodě CSMA/CA. Zkrácení na dva rámce by už fungovat mohlo, teprve testy ukáží, je-li tomu tak.

Pokud se fronta rámců zaplní, ovladač oznamuje operačnímu systému, že mu nemá předávat další pakety, a obdobně oznamuje, že se již místo ve frontě uvolnilo. Neoznamuje však každý jednotlivý odeslaný paket, jen tyto mezní stavy. A také neoznamuje, jakou přenosovou rychlostí byl paket odeslán a kolikrát byl přenos

[2]V ovladači Madwifi je délka fronty omezena konstantou `ATH_TXBUF` na 200 rámců.

opakován – tato informace v ovladači zaniká, využívá se nanejvýš k automatickému výběru přenosové rychlosti.

Pokud by byl algoritmus řízení toku implementován v rámci *Queueing discipline*, neměl by k dispozici informaci o efektivitě přenosu a také by nemohl započítávat příchozí pakety (upload) – *Queueing disciplines* pro příchozí (*ingress*) a odchozí (*egress*) pakety jsou nezávislé. Rozšíření komunikačního protokolu mezi ovladačem a *Queueing discipline* by bylo možné, ale poměrně komplikované, a navíc by nebylo přenositelné mezi Linuxem a FreeBSD. Problém by byl i s aplikací změn na nové verze obou systémů.

Mnohem jednodušší bude implementovat řízení toku čistě v rámci ovladače hardwaru. Implementace pseudokódu z kapitoly 3.3 nebude složitá, půjde nanejvýš o stovky řádků oproti tisícům řádků ovladače. Algoritmus řízení toku je dokonce jednodušší než algoritmus pro automatický výběr přenosové rychlosti (*rate control*), který je také implementován uvnitř ovladače. Řízení toku navíc může být samostatný modul, který bude moci využít více ovladačů.

## 5.2 Ovladače Wi-Fi v operačních systémech

Nutnost implementace v ovladači hardwaru znamená nutnost zvolit konkrétní ovladač. A výběr ovladače je zároveň výběrem hardwaru. Tento hardware musí být dostupný v ČR, podporovat režim AP a také být ověřený v praxi. Tak jako Wi-Fi standardy je i hardware vyvíjen pro lokální sítě uvnitř budov a s nasazením v zarušeném venkovním prostředí může mít problémy.

Ve **FreeBSD** by podle [29] měly podporovat režim AP všechny nativní ovladače (tj. všechny kromě NDIS wrapperu ovladačů z Windows). V praxi vyzkoušený je však režim AP pouze na hardwaru s čipy od Atherosu, které podporuje ovladač `ath`.

V **Linuxu** je situace složitější, režim AP je podporován jen u některého hardwaru:

- Po dlouhou dobu fungoval režim AP pouze s ovladačem HostAP pro chipset Prism 2.5 (PCI karty Z-COM XI-626). Tento chipset ale podporoval pouze 802.11b a už se několik let nevyrábí.
- 802.11 a/b/g hardware od Atherosu má podporu režimu AP díky ovladači Madwifi, který byl původně přenesen z FreeBSD. Ovladač funguje relativně dobře, ale není kompletně open-source a proto nemohl být nikdy začleněn do jádra Linuxu. Jeho vývoj byl již ukončen, vývojáři přešli k novému ovladači `ath5k`, který ale stále nedosahuje stability Madwifi.
- Režim AP v 802.11 stacku `mac80211` je novinkou v kernelu 2.6.29. Zapnut je jen v ovladačích pro čipy Broadcom (`b43` a `b43legacy`), Ralink (`rt2x00`), Prism54 (`p54`) a 802.11n čipy Atherosu (`ath9k`). U tohoto hardwaru zatím není v praxi ověřeno, jak se chová ve venkovním prostředí ani zda jsou ovladače

dostatečně stabilní pro produkční nasazení. U ovladače `ath5k` podporujícího stejný hardware jako Madwifi ještě není podpora režimu AP stabilní a není proto zapnuta, je nutné aplikovat patch.

Jediný hardware s podporou režimu AP v Linuxu i FreeBSD ověřený v praxi jsou karty s 802.11 a/b/g chipsetem od Atherosu. Vzhledem k tomu, že situace je jednodušší na FreeBSD (jen jeden ovladač, dobře otestovaný v praxi, menší množství kódu), bude implementace vyvinuta a odladěna v tomto systému a následně přenesena na Linux. FreeBSD implementace bude popsána podrobně, zatímco v popise Linuxové implementace budou uvedeny pouze rozdíly oproti FreeBSD verzi.

## 5.3 Implementace ve FreeBSD

Řízení toku pro FreeBSD bylo implementováno ve FreeBSD verze 7.1. Má formu modulu `ath_sched` poskytujícího funkce, které jsou volány z ovladače `ath`. Rozhraní modulu `ath_sched` popsané v příloze B je podobné implementaci v OMNeT++ (příloha A).

Základní datovou strukturou je opět spojový seznam klientů. Datová struktura klienta `ath_sched_client` obsahující frontu paketů a čítače je alokována dynamicky při příchodu prvního paketu. Naplánování konce okna zajišťují standardní FreeBSD časovače [30]. Zde však podobnost s implementací v OMNeT++ končí. Implementace pro FreeBSD se musí vypořádat s několika dalšími problémy:

- Kód ovladače `ath` je mnohem složitější než kód objektů v OMNeT++. Celý ovladač má přes 250 kB a z toho je skoro 190 kB v jediném souboru. Funkce dlouhé několik set řádek nejsou výjimkou, velké části kódu se navíc kompilují podmíněně podle direktiv preprocesoru. Není snadné najít, jak do ovladače řízení toku zaintegrovat.
- Na rozdíl od simulace u skutečné implementace záleží na výkonu: mnohý hardware je dimenzován jen na práci jako přístupový bod a router. Pokud by algoritmus řízení toku přidával další zátěž, snížila by se propustnost sítě. Algoritmus proto musí především dostatečně rychle vyhledávat klienta podle MAC adresy a efektivně vybírat další paket k odeslání.
- Zatímco události v OMNeT++ se zpracovávají sekvenčně, ve skutečném operačním systému běžícím na počítači s více procesory může být několik událostí obsluhováno paralelně. Datové struktury je proto nutné chránit zámky.
- V jádře Linuxu ani FreeBSD nelze používat výpočty s pohyblivou desetinnou čárkou (*floating-point*). Při přechodu do režimu kernelu se neukládá obsah *floating-point* registrů, výpočet by tedy poškodil data přerušeného procesu. Algoritmus si musí vystačit pouze s operacemi s celými čísly.
- Na rozdíl od simulace musí být možné měnit parametry algoritmu za běhu (včetně jeho vypnutí a zapnutí) – nelze pro změnu parametrů vyžadovat restart počítače.

Zvoleným řešením uvedených problémů se budu nyní věnovat podrobněji.

### 5.3.1 Integrace do ovladače

Pochopení, jak ovladač `ath` funguje, se nakonec ukázalo jako nejtěžší část celé práce. K ovladači neexistuje žádná dokumentace, jedinou možností bylo prostudovat více než šest a půl tisíce řádek nepříliš přehledného kódu. Vývoj též neulehčilo to, že při pokusu o unload ovladače (i bez mých změn) počítač zatuhl, takže každý test nové verze znamenal restart počítače.

Centrálním bodem ovladače je funkce `ath_start`. Má 250 řádek, ale základní princip je poměrně jednoduchý: funkce bere pakety z fronty TCP/IP stacku a vkládá je do fronty hardwaru, dokud jsou ve frontě TCP/IP stacku nějaké pakety a ovladač má volné buffery. Je volána jednak z TCP/IP stacku, když vložil do fronty další paket, a také když bylo dokončeno vysílání (a nějaký buffer se uvolnil).

Úprava funkce `ath_start` vedoucí ke zkrácení fronty hardwaru na minimum je relativně snadná: všechny pakety z fronty TCP/IP stacku budou předány řízení toku voláním funkce `ath_sched_enqueue`. Poté bude fronta hardwaru doplněna na délku 2 rámci, které vybralo řízení toku (funkce `ath_sched_dequeue`).

Na rozdíl od simulace v OMNeT++ musí řízení toku reagovat na změnu stavu rozhraní: je-li rozhraní vypnuto (příkazem `ifconfig ath0 down`), měl by se zastavit časovač, aby zbytečně nespotřebovával výkon procesoru, a také by měly být uvolněny pakety ve frontách klientů, které už nebude možné odeslat. Změnu stavu provádí funkce `ath_newstate`. Odtud bude volána funkce `ath_sched_set_running` přepínající stav řízení toku – řízení toku bude zapnuté, je-li rozhraní ve stavu `IEEE80211_S_RUN`.

Nejsložitější je získat z ovladače informace o délce přenosu. Oznámení hardwaru o dokončení odesílání a příjmu zpracovávají funkce `ath_tx_processq` resp. `ath_rx_proc`. Hardware předává informace v deskriptoru, jehož součástí je kromě síly signálu i použitá přenosová rychlost a v případě odchozího rámce i údaj o počtu opakování přenosu. Bohužel deskriptor neobsahuje informaci, zda byl použit mechanismus RTS/CTS ani zda měl rámec krátkou nebo dlouhou preambuli. Informaci o počtu opakování přenosu příchozího rámce nelze zjistit vůbec, v rámcích se žádné počítadlo pokusů nepřenáší. Výpočet tedy nebude zcela přesný, ale tato chyba chování algoritmu příliš neovlivní.

Naštěstí vše, co je potřeba k výpočtu doby přenosu, je už v ovladači připraveno, neboť výpočet doby přenosu je nutný pro stanovení hodnoty NAV (*Network allocation vector*) v rámci RTS. Výpočet NAV se nachází ve funkci `ath_tx_start` a používá funkci `ath_hal_computetxtime`, která bere do úvahy i velikost hlavičky a preambuli rámce v závislosti na nastaveném režimu (802.11 a/b/g). K výsledku je nutno přičíst trvání SIFS a ACK (čas je předpočítán v tabulce `HAL_RATE_TABLE`) a nakonec délku DIFS podle aktuálního režimu. Výsledek je pak spolu s MAC adresou klienta předán funkci `ath_sched_count_in`.

Poslední úprava je již triviální: přidat inicializaci řízení toku `ath_sched_attach`

do funkce `ath_attach` a deinicializaci `ath_sched_detach` do funkce `ath_detach`.

### 5.3.2 Vyhledání klienta podle MAC adresy

Nejefektivnější datovou strukturou pro vyhledání klienta dle MAC adresy je hašovací tabulka. FreeBSD nabízí funkce pro vytvoření hašovací tabulky [31], ale ty nemají nic společného například s `hash_map` v knihovně STL: vytvořená tabulka je jen pole spojových seznamů, vyhledávání a vkládání je ponecháno na programátorovi. Použití těchto funkcí by neušetřilo ani řádek kódu, a naopak by zkomplikovalo následný přenos na Linux.

Díky znalosti struktury MAC adres je navíc možné vytvořit efektivnější hašovací funkci. První byte je u běžných adres vždy nulový (jinou hodnotu má jen u automaticky generovaných dočasných adres). Druhý a třetí byte má přidělen výrobce hardwaru. Poslední tři bajty pak výrobce sekvenčně přiděluje svým produktům – MAC adresy několika kusů hardwaru od jednoho výrobce, které byly koupeny najednou, se budou lišit jen posledními bity.

Není tedy nutné počítat haš celé MAC adresy, postačí přímo použít posledních $N$ bitů MAC adresy jako index do tabulky velikosti $2^N$. Adresy hardwaru od jednoho výrobce v této jednoduché hašovací funkci kolidovat nebudou, a pravděpodobnost kolize adres hardwaru od různých výrobců je stejná jako u haše celé MAC adresy.

V implementaci je použito obyčejné hašování se separovanými řetězci: řetězce jsou tvořeny položkou `hash_entry` v datové struktuře klienta `ath_sched_client`, nevyžadují další paměť. V implementaci je použita hašovací tabulka velikosti 64 – velikost je zanedbatelných 256 bajtů na 32-bitových systémech a očekávaná délka řetězce při 32 klientech je $\frac{1}{2}$.

### 5.3.3 Výběr rámce k odeslání

Podle pseudokódu z kapitoly 3.3 má funkce `ath_sched_dequeue` vybrat první rámec z fronty klienta, jenž má nejmenší hodnotu čítače `weighted_counter` mezi klienty s neprázdnou frontou.

V seznamu klientů jsou všechny počítače, které byly na AP někdy připojeny, včetně těch, které se již odpojily (údaj o množství přenesených dat by odpojením neměl být zapomenut, toho by klienti mohli zneužívat). Obvykle ale jen malý počet klientů má ve frontě nějaké pakety připravené k odeslání. Proto bude vhodné pro zvýšení efektivity umístit tyto tzv. „aktivní“ klienty do zvláštní datové struktury.

Existují dvě datové struktury, které lze pro seznam aktivních klientů použít: obyčejný spojový seznam a halda, uspořádaná podle hodnoty čítače `weighted_counter`. Halda umožňuje vybrat klienta s nejmenší hodnotou čítače v konstantním čase, ale za cenu složitější změny hodnoty čítače – klient musí být odebrán z haldy a znovu vložen, obojí v čase $O(\log n)$. Časovou složitost operací u obou datových struktur shrnuje tabulka 5.1.

| | **Spojový seznam** | **Halda** |
|---|---|---|
| **Výběr rámce k odeslání** | $O(n)$ | $O(1)$ |
| **Započtení příchozího rámce** | $O(1)$ | $2 \cdot O(\log n)$ |
| **Započtení odchozího rámce** | $O(1)$ | $2 \cdot O(\log n)$ |

Tabulka 5.1: Porovnání složitosti operací v haldě a spojovém seznamu

Všechny operace v tabulce 5.1 probíhají přibližně stejně často: každý odeslaný paket je započítán a příchozích paketů je přibližně stejně jako odchozích (u TCP spojení jsou fragmenty potvrzovány, u VoIP hovoru prochází stejný počet paketů oběma směry).

Zkusme nyní odhadnout, pro jaký počet aktivních klientů bude již halda efektivnější než spojový seznam. Pro jednoduchost uvažujme, že jeden krok průchodu spojovým seznamem trvá stejně dlouho, jako dílčí operace v haldě. Ve skutečnosti bude krok ve spojovém seznamu vyžadovat méně instrukcí a spojový seznam bude tedy mírně znevýhodněn. Zanedbáme-li operace v konstantním čase, vyjde rovnice

$$4 \cdot \log(n) = n$$

Tato rovnice je transcendentní, ale výsledek lze „uhodnout“, je jím číslo 16. Haldu je tedy výhodné použít jen pro 17 a více aktivních klientů. Při tomto počtu klientů však již narůstá počet kolizí do takové míry, která činí síť skoro nepoužitelnou[3]. Algoritmus by tedy měl být optimalizován spíše na méně než 16 klientů, a proto je vhodnější použít spojový seznam místo „chytřejší“ datové struktury.

### 5.3.4 Synchronizace

Spojový seznam ve FreeBSD je implementován pouze sadou maker, která neobsahují žádnou synchronizaci, ochrana proti souběhu je ponechána na programátorovi. Spojové seznamy musí být chráněny nejen proti souběžné modifikaci, ale i proti modifikaci během procházení seznamem.

Současná infrastruktura pro synchronizaci ve FreeBSD je popsána v [32]. K dispozici jsou mutexy [33] (zkratka *Mutual Exclusion*, vzájemné vyloučení) pro implementaci kritické sekce a od FreeBSD 7.0 i read/write zámky [34]. Právě RW zámky jsou nejvhodnější pro ochranu seznamů proti souběhu: několik vláken může seznam paralelně procházet, ale jen jedno vlákno jej může modifikovat, a to jen tehdy, kdy jej žádné jiné neprochází.

Další datovou strukturou, kterou je nutno chránit proti paralelnímu přístupu, je struktura klienta `ath_sched_client`. Výpočty si nevystačí s atomickými proměnnými, hodnoty `counter` a `weighted_counter` se musí měnit najednou. Nicméně zde by by případný souběh způsobil pouze nepřesnost hodnot, ne pád systému. Horší

[3] V přístupové metodě CSMA/CA u 802.11g [3] stanice čeká před vysíláním náhodný počet slotů z intervalu $\langle 1, 15 \rangle$. Při počtu 16 klientů tedy podle Dirichletova principu dojde ke kolizi vždy: alespoň dvě stanice musí zvolit stejnou hodnotu a začít vysílat současně.

situace nastane, bude-li struktura `ath_sched_client` dealokována ve chvíli, kdy s ní pracuje jiné vlákno: na jejím místě v paměti už může být jiná datová struktura, a zápis by způsobil chybu v naprosto nesouvisející části systému.

Před dealokací struktury klienta proto musí být zamknut mutex klienta a klient odebrán ze všech seznamů. Funkce, která vybírá klienta ze seznamu, jej musí zamknout, ještě než odemkne seznam – jinak by mezitím klient mohl být dealokován. To znamená, že funkce vyhledávající klienta podle MAC adresy musí vracet strukturu klienta zamčenou a teprve volající funkce ji odemkne. To bohužel trochu kazí návrh (zamčení a odemčení není symetricky v jedné funkci), ale jiné řešení neexistuje.

Použití několika zámků přináší riziko uváznutí (deadlocku). K deadlocku dojde tehdy, pokud jsou splněny všechny Coffmanovy podmínky popsané v [35]:

1. Prostředek může v daný čas využívat jen jeden proces
2. Proces může žádat o další prostředky, když už má jiné přiděleny
3. Přidělený prostředek nelze procesu odejmout
4. V orientovaném grafu čekání a držení prostředků je cyklus

Pro prevenci deadlocku stačí znemožnit splnění jediné podmínky. První a třetí podmínka je dána přímo důvodem, proč jsou zámky použity: datové struktury může modifikovat jen jedno vlákno, před dokončením modifikace je datová struktura v nekonzistentním stavu. Obojí však platí jen pro exkluzivní zámky (mutexy a zámky pro zápis).

Splnění druhé podmínky by bylo možné vyloučit jednoduše tak, že by byl použit jen jediný zámek, zamykaný vždy ve funkcích na nejvyšší úrovni (tj. ve funkcích volaných přímo z ovladače). Tím by se ale znemožnil jakýkoli paralelismus.

Nejvhodnější je proto zabránit splnění čtvrté podmínky tak, že bude definováno pořadí, v jakém mohou vlákna žádat zámky. Tím se zabrání uzavření cyklu v grafu čekání. V implementaci je zvoleno následující pořadí:

1. Seznam aktivních klientů (zápis)
2. Seznam všech klientů (zápis)
3. Hašovací tabulka (zápis)
4. Datová struktura klienta

Toto pořadí odpovídá přirozenému pořadí v původním kódu, s jedinou výjimkou: funkce, která zařazuje paket do fronty klienta, musí na začátku zamknout seznam aktivních klientů exkluzivně pro případ, že je tento paket zařazen do prázdné fronty a klient se přidává do seznamu aktivních. Ve většině případů je zámek zbytečný, ale nelze jej zamykat až následně: při uvolňování klientů se napřed exkluzivně zamykají všechny seznamy a pak teprve jednotliví klienti. Pokud by dvě funkce zamykaly seznam aktivních klientů a strukturu klienta v různém pořadí, nastal by dříve či později deadlock.

### 5.3.5 Převod výpočtů do celých čísel

Zatímco simulace v OMNeT++ používá výpočty s pohyblivou desetinnou čárkou, implementace pro FreeBSD si musí vystačit s výpočty s 32-bitovými celými čísly. Přestože skoro všechna dnes prodávaná PC mají 64-bitové procesory, v routerech se používají spíše starší PC nebo embedded platformy jako PC Engines Alix (32-bit x86 procesor AMD Geode) nebo Mikrotik Routerboard a Ubiquity Nanostation (32-bit MIPS).

I procesory v embedded platformách obsahují jednotku pro práci s čísly s pohyblivou desetinnou čárkou (FPU), ale tu nelze používat v jádře OS. Operační systém při přechodu z uživatelského režimu do režimu jádra ukládá na zásobník jen celočíselné registry. V registrech FPU je tedy to, co zde proces zanechal, a co také předpokládá, že zde po návratu z kernelu nalezne. Ovladač by sice mohl registry ukládat sám, ale to by kromě ztráty výkonu znamenalo i ztrátu přenositelnosti: k ukládání registrů je potřeba kód v assembleru.

Všechny výpočty v pohyblivé desetinné čárce je naštěstí možné převést na pevnou desetinnou čárku poměrně snadno: všechna čísla se pohybují v relativně malém rozsahu, který je shora omezen. Exponenciální průměr nemůže přesáhnout 1.0, čítač časů přenosů (`counter`) dosahuje nejvýše délky okna (rozumné maximum je vteřina). Vážený čítač (`weighted_counter`) bude také omezen, pokud omezíme maximální hodnotu parametru $\beta$.

Aby výpočty využily co nejvíce z 32-bitové přesnosti, musí se tato maxima blížit co nejvíce k $2^{32}$, tj. přibližně $4 \cdot 10^9$. Hranici 32-bitového rozsahu však nesmí nikdy překročit nejen výsledné hodnoty, ale ani žádné mezivýsledky výpočtu.

Exponenciální průměr bude z intervalu $\langle 0, 1\rangle$ převeden do $\langle 0, 10^9\rangle$. Parametr $\tau$ bude zadáván v milisekundách, padne tedy do intervalu $\langle 0, 10^3\rangle$. Místo $\alpha$ bude výpočet pracovat s $\alpha^{-1}$ a vzorec (3.1) se změní na

$$\begin{aligned} \texttt{window\_avg} &= 10^3 \cdot \texttt{counter}/\tau \cdot 10^3 \\ \texttt{average} &= \texttt{average} - \texttt{average}/\alpha^{-1} + \texttt{window\_avg}/\alpha^{-1} \end{aligned}$$

Časy přenosů předávané z ovladače jsou v mikrosekundách, hodnota čítače je tedy při maximální délce okna 1s v intervalu $\langle 0, 10^6\rangle$. Vážený čítač pokrývá dvě okna, váha proto musí být v intervalu $\langle 0, 2100\rangle$. Vzorec (3.2) se tedy změní na

$$\texttt{weighted\_counter} = \texttt{weighted\_counter} + \texttt{duration} \cdot (100 + \beta \cdot \texttt{average}/10^7)$$

Maximální hodnota parametru $\beta$, kdy nedojde k přetečení, pak vychází 20.

### 5.3.6 Nastavování parametrů za běhu

Na rozdíl od simulace v OMNeT++ musí být možné řízení toku vypínat a zapínat za běhu. Výchozí stav bude vypnuté řízení toku, protože ve většině případů nemá smysl (je-li rozhraní v režimu klient, a nebo i v režimu AP, ale na dvoubodovém

spoji). Volání všech funkcí modulu `ath_sched` z ovladače proto bude podmíněno testem `ath_sched_enabled()`.

Standardní způsob nastavování parametrů jádra ve FreeBSD je příkaz `sysctl`. Sysctl parametry mají hierarchickou strukturu, modul řízení toku proto může přidat uzel `sched` k parametrům síťového rozhraní. Parametry řízení toku pak budou pod tímto uzlem, například `dev.ath.0.sched.enabled`.

Většina parametrů se jednoduše změní bez vedlejších efektů a změna se projeví při výpočtu v následujícím časovém okně. Jediná výjimka je parametr `enabled`, zapínání řízení toku. Při zapnutí je spuštěn časovač, vypnutím se časovač ukončí a jsou uvolněny datové struktury klientů včetně paketů ve frontách.

Všechny parametry i s jejich výchozí, minimální a maximální hodnotou podle předchozí kapitoly 5.3.5 shrnuje tabulka 5.2.

| **parametr** | **sysctl** | **výchozí** | **minimum** | **maximum** |
|---|---|---|---|---|
| stav | `enabled` | 0 | 0 | 1 |
| $\alpha^{-1}$ | `expfactor` | 1000 | 1 | $10^9$ |
| $\beta$ | `avgweight` | 4 | 0 | 20 |
| $\tau$ | `wintime` | 200 | 10 | 1000 |

Tabulka 5.2: Sysctl parametry řízení toku

## 5.4 Implementace v Linuxu

Jak již bylo uvedeno v kapitole 5.2, situace s ovladači hardwaru s čipy Atheros je v Linuxu složitější než ve FreeBSD. Ovladač `ath5k`, který je součástí Linuxu, jen pomalu dosahuje stability nutné pro rutinní používání v režimu klient. Režim AP je v ovladači vypnut (k zapnutí je potřeba patch), protože stále nefunguje dostatečně spolehlivě. V produkčním prostředí tedy zatím není ovladač `ath5k` použitelný.

V praxi se proto stále používá starý ovladač Madwifi, jehož pohnutá historie je shrnuta v [36]. Madwifi pochází ze stejného kódu jako ovladač `ath` ve FreeBSD. Oba závisí na komponentě HAL (*Hardware abstraction layer*), která obsluhuje hardware na nejnižší úrovni. HAL nebyl donedávna[4] open-source, byl dodáván jen v binární podobě. Zatímco autor HAL Sam Leffler vyvíjel i ovladač `ath`, vývojáři Madwifi neměli ke zdrojovým kódům HAL přístup, což se projevilo na kvalitě ovladače. Uzavřený HAL navíc znemožnil začlenění Madwifi do jádra Linuxu.

Situaci navíc dále komplikuje, že vývoj Madwifi probíhá v několika větvích najednou. Některé větve se nedostaly příliš daleko (např. pokusy o port na 802.11 stack `mac80211` nebo open-source HAL z OpenBSD), ale tři jsou ve více či méně použitelném stavu:

[4] Zdrojové kódy verze HAL použité ve FreeBSD byly zveřejněny v listopadu 2008, ve FreeBSD verze 7.2 je již ovladač `ath` kompletně open-source.

- `madwifi-old` je nejstarší vývojová větev. Vznikla přímo přenosem ovladače `ath` z FreeBSD na Linux.
- Větev `madwifi-ng` vznikla, když vývojáři Atherosu vzali tehdejší Madwifi a provedli v něm rozsáhlé změny, které již nebylo možné sloučit s `madwifi-old`. Z této větve pochází poslední stabilní vydání `madwifi-0.9.4`.
- `madwifi-svn` je současná vývojová verze, která zatím neměla žádné stabilní vydání, je dostupná jen v SVN. Nyní je pravděpodobné, že ani žádné stabilní vydání nikdy nebude, vývoj se v podstatě zastavil.

Rozhodnutí, do které větve implementovat řízení toku, je poměrně snadné: upravovat zastaralé `madwifi-old` nemá příliš smysl, a nestabilní `madwifi-svn` by nebylo použitelné v produkčním prostředí. Zbývá tedy jen `madwifi-0.9.4` z větve `madwifi-ng`.

Řízení toku jsem testoval na jádře 2.6.27, zatímco `madwifi-0.9.4` bylo vydáno v době, kdy byla aktuální verze Linuxu 2.6.24. Mezi těmito verzemi došlo v jádře k mnoha změnám, z nichž některé nebyly s Madwifi kompatibilní. Bylo proto nutné aplikovat několik patchů z SVN repository Madwifi, aby se ovladač vůbec podařilo zkompilovat.

Další komplikací vývoje na Linuxu je dokumentace. Zatímco ve FreeBSD má skoro každá funkce jádra manuálovou stránku, Linux interní API neudržuje stabilní. To znamená, že jakákoli dokumentace funkcí jádra velmi rychle zastarává. Vývojáři zastarávání předchází jednoduše: žádnou dokumentaci nepíší. Jen málo funkcí jádra tedy má manuálové stránky, a pokud existují, jsou velice stručné a mnohdy neaktuální.

Lepším zdrojem informací je adresář `Documentation` ve zdrojových kódech jádra, ale zde jsou popsané jen některé aspekty jádra, a to především novinky, ne základní struktura. *The Linux Kernel Module Programming Guide* [37] popisuje pouze obecné principy vývoje (např. jak vytvořit jaderný modul), není to žádný referenční manuál. Často tedy nezbývá než studovat přímo zdrojové kódy.

Naštěstí byla implementace odladěna na FreeBSD, takže zbývalo jen najít Linuxové ekvivalenty funkcí jádra FreeBSD a přizpůsobit integraci do ovladače rozdílům mezi `madwifi-old` a `madwifi-ng`. Základní struktura implementace algoritmu je v Linuxu a FreeBSD stejná, API popsané v příloze C je skoro shodné s API FreeBSD verze (příloha B).

Podstatný rozdíl je pouze řešení synchronizace: Linux má pokročilejší mechanismy synchronizace než FreeBSD, které by byla škoda nevyužít.

### 5.4.1 Synchronizace

V Linuxu jsou k dispozici všechna klasická synchronizační primitiva: spinlocky, mutexy i read/write zámky ve formě semaforů. Vedle nich ale Linux nabízí novinku: bezzámkové algoritmy založené na RCU (*read-copy-update*). Místo modifikace datové

struktury je vytvořena její kopie, která je pak atomicky zaměněna s původní (změna ukazatele je na moderních procesorech atomická operace). Stará datová struktura je uvolněna poté, co jsou ukončeny všechny přístupy k ní.

Typické použití RCU je jako náhrada read/write zámků u spojových seznamů. Příklad aplikace je uveden v [38]: místo zámků pro čtení se použijí RCU „zámky“ (`rcu_read_lock()` a `rcu_read_unlock()`). Klasický zámek chrání seznam jen proti paralelním změnám, procházení seznamu je na něm nezávislé.

Kód pro přidávání položek seznamu se nijak dále nemění, složitější je pouze uvolňování položek: položku není možné uvolnit hned, je nutné voláním `call_rcu` zaregistrovat funkci, která bude zavolána poté, co všechna vlákna opustí kritickou sekci. RCU také klade omezení na kód v kritické sekci: nemůže blokovat. Datová struktura klienta zamykaná v RCU sekci tedy musí být chráněna spinlockem místo mutexu.

Použitím RCU se výrazně zjednodušily závislosti zámků. Hašovací tabulka a seznam klientů se mění vždy najednou, mohou tedy mít jeden společný zámek. Dealokace klienta se provádí mimo funkci `ath_sched_stop`, která odebírá klienty ze seznamů – mizí tedy závislost zámku klienta a zámku seznamu aktivních klientů. Pořadí zamykání pro prevenci uváznutí je možné změnit následovně:

1. Seznam klientů a hašovací tabulka (zápis)
2. Datová struktura klienta
3. Seznam aktivních klientů (zápis)

V novém pořadí je seznam aktivních klientů zamykán až po klientovi. Může tedy být zamknut jen pokud se opravdu mění, ne vždy na začátku `ath_sched_enqueue` a `ath_sched_dequeue`, jako bylo nutné ve FreeBSD.

Pokud procházení seznamem klientů nevyžaduje zámek, trvají skoro všechny kritické sekce konstatní čas, nezávislý na délce seznamů. Není proto problém použít pro zamykání spinlocky, systém nebude blokován příliš dlouho. Jedinou výjimkou je funkce `ath_sched_stop`, která provádí dealokaci všech klientů. Ta ale není volána za normálního běhu, pouze při vypínání rozhraní.

Linux nicméně nabízí několik typů spinlocků pro použití v různých situacích podle toho, z jakého kontextu může být k datové struktuře přistupováno. Spinlock proto může plnit až čtyři úkoly:

1. Vypnutí hardwarových přerušení
2. Vypnutí softwarových přerušení (taskletů)
3. Zákaz preempce (předání procesoru jinému vláknu)
4. Blokování jiných procesorů před vstupem do kritické sekce

Funkce řízení toku mohou být volány jen z kontextu uživatelského procesu nebo taskletu, nikdy ne z přerušení. Není proto nutné vypínat hardwarová přerušení (`spin_lock_irqsave`), ale pouze softwarová přerušení (tasklety), což zajistí funkce

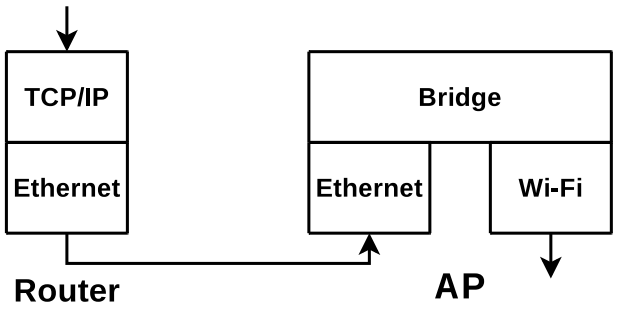
`spin_lock_bh`[5]. Zákaz taskletů i preempce obsahuje počítadlo zanoření, je tedy možné držet několik zámků zároveň.

## 5.5 Implementace v hardwarovém AP

Návrh i implementace algoritmu byly zatím postaveny na předpokladu, že přístupový bod funguje jako router s jedním nebo více páteřními spoji (uplinky) a jedním nebo více Wi-Fi sektory pro připojení klientů. Tedy že má podobu počítače s několika PCI nebo miniPCI Wi-Fi kartami.

Počítač nemusí být jen klasické PC, ale i x86 embedded platformy jako PC Engines Alix. Podobná situace je u platformy Mikrotik RouterBoard, která má sice procesor s instrukční sadou MIPS, zbytek je však stejný: PCI sběrnice a běžné miniPCI Wi-Fi karty.

Existuje ale i alternativa: oddělit Wi-Fi zařízení od routeru. Router pak má místo Wi-Fi karty obyčejné ethernetové rozhraní a do něj je připojeno hardwarové AP, které funguje jako most mezi Wi-Fi a Ethernetem. Struktura takto rozděleného přístupového bodu je znázorněna na obrázku 5.3.




Obrázek 5.3: Wi-Fi zařízení oddělené od routeru

Algoritmus řízení toku je vázán na Wi-Fi hardware, musí být tedy implementován v AP. Algoritmus je na toto rozdělení připraven, pracuje na 2. vrstvě ISO/OSI modelu, není závislý na TCP/IP. Problém je čistě praktický: je nutné upravit firmware v hardwarovém AP.

Hardwarová AP používají pro úsporu nákladů speciální chipsety, které integrují MIPS procesor i Wi-Fi rádio. Dnes se nejčastěji používají zařízení postavená na dvou chipsetech:

**Realtek 8186**
: Chipset RTL8186 používají nejlevnější 802.11 b/g zařízení na trhu jako OvisLink WL-5460AP, StraightCore WRT-314 a Zcomax WA-2204. Jejich firmware je založen na Linuxu a zdrojové kódy jsou dostupné. Bohužel ale nejsou kompletní, ovladač Wi-Fi je dodáván jen v binární podobě (což je pravděpodobně

[5] „BH“ je zkratka *bottom half*, dřívějšího názvu pro tasklet.

porušení licence GPL). Implementace řízení toku vyžaduje úpravu ovladače, pro tento hardware tedy není možná.

**Atheros AR2312**
802.11 a/b/g chipset AR2312 používají zařízení jako OvisLink WLA-5000AP a Ubiquity NanoStation/Bullet. Všechna mají firmware založený na Linuxu a na rozdíl od RTL8186 je kompletně open-source. Navíc již byl vytvořen alternativní firmware založený na distribuci OpenWRT [39]. Chipset AR2312 navíc také podporuje ovladač Madwifi, v němž se již podařilo řízení toku implementovat.

Řízení toku je naprogramováno v čistém C a je tedy přenositelné i na architekturu MIPS. Zdálo by se, že stačí zkompilovat ovladač Madwifi pro MIPS, ale situace je složitější. HAL verze 0.9.18.0 použitý v `madwifi-0.9.4` ještě chipset AR5312 nepodporuje, podpora je až v novější verzi 0.10.5.6, která ale nemá kompatibilní API. Bylo proto nutné přenést změny pro kompatibilitu s novým API z vývojové verze `madwifi-svn` do `madwifi-0.9.4`. Výsledný ovladač byl otestován v zařízení OvisLink WLA-5000AP.

## 5.6 Testy implementací

Nyní zbývá poslední, avšak rozhodující krok: otestovat, zda implementace algoritmu fungují správně. V ideálním případě by se měly chovat stejně jako simulace v OMNeT++, jejíž výsledky jsou v kapitole 4.4. Aby byly výsledky srovnatelné, musí být testovací síť co nejvíc podobná simulačnímu modelu z kapitoly 4.2:

- **Server** nabízí ke stažení protokolem HTTP dynamicky generovaný soubor `infinity.dat`, který obsahuje nekonečně mnoho nulových bajtů získaných z `/dev/zero`[6].
- **Stahovač** stahuje soubor `infinity.dat` desetkrát najednou programem `wget` (data nikam neukládá).
- **Hráč** simuluje VoIP hovor pingem se stejnými parametry jako v OMNeT++ modelu (tj. `ping -s 172 -i 0.03 -c 2000`).

Aktivita Stahovače a Hráče začíná současně, test je ukončen po přijetí posledního pingu, tj. po jedné minutě stejně jako u simulace. Test je opakován se zapnutým a vypnutým řízením toku a také pro dva různé přístupové body (jeden s Linuxem a druhý s FreeBSD). Přenosová rychlost na AP byla vždy nastavena na 1 Mbit/s. Parametry řízení toku byly ponechány na výchozích hodnotách z kapitoly 5.3.6.

[6] Wi-Fi nepoužívá žádnou kompresi, takže není rozdíl mezi stahováním samých nul a náhodných dat. Generování náhodných dat je výpočetně náročné, což by mohlo testy ovlivnit.

### 5.6.1 Test implementace ve FreeBSD

AP se systémem FreeBSD 7.1 běželo na hardware PC Engines Alix 2C2 s procesorem AMD Geode na 500 MHz a 256 MB RAM. Jako Wi-Fi rozhraní byla použita miniPCI karta Wistron CM9 s chipsetem Atheros AR5213.

Výsledky testů s vypnutým a zapnutým řízením toku jsou v tabulce 5.3. Naměřené hodnoty jsou skutečně blízké výsledkům simulace, dokonce ještě lepší: průměrná latence se snížila více než padesátinásobně. Bohužel se nepodařilo zcela zabránit ztrátě paketů, tu však mohou způsobovat i vnější vlivy jako rušení.

| | **Bez řízení toku** | **S řízením toku** | **Podíl** |
|---|---|---|---|
| Minimální latence (ms) | 2.688 | 2.736 | 0.98 |
| Průměrná latence (ms) | 2270.061 | 42.586 | 53.31 |
| Maximální latence (ms) | 2703.469 | 235.011 | 11.50 |
| Směrodatná odchylka (ms) | 366.845 | 23.441 | 15.65 |
| Ztrátovost (%) | 1.2 | 0.4 | 3.0 |

Tabulka 5.3: Naměřené hodnoty v testu FreeBSD implementace

### 5.6.2 Test implementace v Linuxu

Pro test v Linuxu jsem zvolil hardware Ovislink WLA-5000AP s 32 MB RAM a procesorem MIPS s taktem 180 MHz. Wi-Fi rozhraní je součástí chipsetu AR2312.

Výsledky testů s vypnutým a zapnutým řízením toku jsou v tabulce 5.4. Naměřené hodnoty jsou opět velmi blízké výsledkům simulace. Velký rozdíl oproti výsledkům FreeBSD implementace je ztrátovost paketů. V Linuxové implementaci běžící na klasickém PC ale k těmto ztrátám nedochází, takže budou pravděpodobně důsledkem vlastností hardwaru, ne chyb v implementaci.

| | **Bez řízení toku** | **S řízením toku** | **Podíl** |
|---|---|---|---|
| Minimální latence (ms) | 2.856 | 2.893 | 0.99 |
| Průměrná latence (ms) | 2036.366 | 54.552 | 37.33 |
| Maximální latence (ms) | 3009.811 | 209.818 | 14.34 |
| Směrodatná odchylka (ms) | 510.748 | 29.265 | 17.45 |
| Ztrátovost (%) | 3.5 | 1.0 | 3.5 |

Tabulka 5.4: Naměřené hodnoty v testu Linuxové implementace

### 5.6.3 Souhrn výsledků

Naměřené hodnoty průměrné, minimální a maximální latence bez řízení toku jsou v tabulce 5.5, hodnoty s řízením toku v tabulce 5.6.

| | **OMNeT++** | **FreeBSD** | **Linux** |
|---|---|---|---|
| Minimální latence (ms) | 4.419 | 2.688 | 2.856 |
| Průměrná latence (ms) | 1813.740 | 2270.061 | 2036.366 |
| Maximální latence (ms) | 2676.660 | 2703.469 | 3009.811 |
| Směrodatná odchylka (ms) | 490.911 | 366.845 | 510.748 |
| Ztrátovost (%) | 1.25 | 1.2 | 3.5 |

Tabulka 5.5: Souhrn naměřených hodnot bez řízení toku

| | **OMNeT++** | **FreeBSD** | **Linux** |
|---|---|---|---|
| Minimální latence (ms) | 4.419 | 2.736 | 2.893 |
| Průměrná latence (ms) | 56.861 | 42.586 | 54.552 |
| Maximální latence (ms) | 263.872 | 235.011 | 209.818 |
| Směrodatná odchylka (ms) | 29.397 | 23.441 | 29.265 |
| Ztrátovost (%) | 0.0 | 0.4 | 1.0 |

Tabulka 5.6: Souhrn naměřených hodnot s řízením toku (v milisekundách)

# Kapitola 6

# Závěr

Cíl práce se podařilo splnit: byl navržen algoritmus řízení toku, který se dokáže vypořádat se specifickými podmínkami Wi-Fi sítí. Následně byl algoritmus implementován pro operační systémy Linux a FreeBSD, které se v přístupových bodech Wi-Fi sítí reálně používají. Implementace byla dovedena do stavu, kdy může být ihned nasazena v praxi.

Simulace v prostředí OMNeT++ ověřila, že navržený algoritmus dokáže významně snížit latenci sítě. Testy prováděné ve skutečné síti naznačují, že se stejně chovají i obě implementace algoritmu. Pokles latence je bohužel limitován vlastnostmi linkové vrstvy IEEE 802.11: v přístupové metodě CSMA/CA je AP zcela rovnocenné s klienty, nemůže ovlivnit, kdy bude který klient vysílat. Průměrná latence tedy roste alespoň lineárně s počtem klientů, kteří odesílají data, kvůli kolizím spíše rychleji.

Přestože byl hlavní cíl splněn, výsledek rozhodně není konečný, nabízí se několik směrů dalšího vývoje:

**Konfigurovatelnost**
: Z požadavků v kapitole 1.3 bylo explicitně vyjmuto nastavování priorit jednotlivých klientů a uživatelsky definované rozlišování klientů. Pro širší použití by bylo vhodné tato omezení odstranit. Konfigurační rozhraní pro zadání priorit klientů by si však již nevystačilo se sysctl parametry. Rozlišování klientů jinak než podle MAC adres by bylo ještě složitější. U odchozích paketů je možné využít značky firewallu, ale u příchozích toto fungovat nemůže. Nanejvýš lze dovolit sloučení několika různých MAC adres pod jednoho klienta.

**Integrace s řízením toku v OS**
: Řízení toku bylo implementováno pouze v rámci ovladače hardwaru, zcela nezávisle na frameworku řízení toku operačního systému. Návrh tak nebyl svázán omezeními řízení toku v OS, ale zároveň ztratil flexibilitu, klienty rozlišuje jen pevným způsobem podle MAC adres. Bylo by vhodné upravit řízení toku v OS tak, aby mohlo zahrnout specifika Wi-Fi sítí. Změny by však byly dosti rozsáhlé, neboť je nutné změnit i základní princip, jak řízení toku v OS pracuje: musí posílat vybraný paket na vyžádání z ovladače, ne v čase dle vlastního

uvážení. Je též nezbytné přidat zpětný kanál, kudy by ovladač informoval o efektivitě přenosu odesílaných paketů.

**Úpravy linkové vrstvy**

Pro další snížení latence je potřeba, aby AP mohlo ovlivňovat, kdy bude který klient vysílat. Řešení kompatibilní se stávajícími klienty by teoreticky mohlo být postaveno na mechanismu RTS/CTS: AP může některému klientovi na rámec RTS neodpovědět a pozdržet tak jeho vysílání. Bohužel alespoň na čipech Atheros je mechanismus RTS/CTS implementován hardwarově, z ovladače může být jen globálně vypnut. Musel by být nalezen jiný hardware, který implementuje mechanismus RTS/CTS v modifikovatelném firmwaru.

# Literatura


[1] *IEEE Std 802.11-2007 – Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE ComputerSociety, 2007.
http://standards.ieee.org/getieee802/download/802.11-2007.pdf

[2] *IEEE Std 802.11b-1999 – Higher-Speed Physical Layer Extension in the 2.4 GHz Band*, IEEE ComputerSociety, 2003.
http://standards.ieee.org/getieee802/download/802.11b-1999.pdf

[3] *IEEE Std 802.11g-2003 – Further Higher Data Rate Extension in the 2.4 GHz Band*, IEEE ComputerSociety, 2003.
http://standards.ieee.org/getieee802/download/802.11g-2003.pdf

[4] *IEEE Std 802.11e-2005 – Medium Access Control (MAC) Quality of Service Enhancements*, IEEE ComputerSociety, 2003.
http://standards.ieee.org/getieee802/download/802.11e-2005.pdf

[5] Peterka, J.: *Internet v ČR: 2 miliony vysokorychlostních přípojek a 5 milionů uživatelů*, Lupa.cz, 2009.
http://www.lupa.cz/clanky/internet-vnbspcr-2-miliony-pripojek/

[6] Internet pro všechny: *Bezdrátové sítě v České republice*, [citováno 28.4.2009].
http://www.internetprovsechny.cz/wifi.php

[7] wimax.cz: *Časté dotazy*, [citováno 28.4.2009].
http://www.wimax.cz/index.php?sectionid=3&id=7&Itemid=25

[8] slfree.net: *Jak platit členské příspěvky?*, [citováno 28.4.2009].
http://slfree.net/informace-pro-zajemce/jak-zaplatit-clenske-prispevky

[9] Wolf, K.: *ADSL na konci světa: dosáhnete na 16 Mbit/s?*, Lupa.cz, 2008.
http://www.lupa.cz/clanky/adsl-na-konci-sveta-aneb-jakou-roli-hraje/

[10] Wi-Fi Alliance Knowledge Center: *Wi-Fi Multimedia™ (WMM®)*, [citováno 29.4.2009].
http://www.wi-fi.org/knowledge_center/wmm

[11] von Lohmann, F.: *FCC Rules Against Comcast for BitTorrent Blocking*, eff.org, 2008.
http://www.eff.org/deeplinks/2008/08/fcc-rules-against-comcast-bit-torrent-blocking

[12] Mangold, S. et al.: *IEEE 802.11e Wireless LAN for Quality of Service*, European Wireless, 2002.
http://www.ing.unipi.it/ew2002/proceedings/H2006.pdf

[13] Gast, M: *When Is 54 Not Equal to 54? A Look at 802.11a, b, and g Throughput*, oreillynet.com, 2003.
http://www.oreillynet.com/pub/a/wireless/2003/08/08/wireless_throughput.html

[14] *The OpenBSD Packet Filter (PF) and ALTQ*, FreeBSD Handbook, [citováno 8.5.2009].
http://www.freebsd.org/doc/en/books/handbook/firewalls-pf.html

[15] *PF: Packet Queueing and Prioritization*, OpenBSD documentation, [citováno 8.5.2009].
http://www.openbsd.org/faq/pf/queueing.html

[16] *iproute2 manual pages*, [citováno 9.5.2009].
http://linux.die.net/man/

[17] Ravishankar, K., Kalyanaraman, S., Karthekeyan, C.: *A Report on Traffic Shaping and Congestion Control*, 2005.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.689

[18] *Exponential Moving Average*, Wikipedia, [citováno 19.5.2009].
http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average

[19] Demers, A., Keshav, S., Shenker, S.: *Analysis and Simulation of a Fair Queueing Algorithm*, SIGCOMM, 1989.
http://www.sigcomm.org/ccr/archive/1995/jan95/ccr-9501-shenker.pdf

[20] McKenney, P.: *Stochastic Fairness Queuing*, 1991.
http://www.rdrop.com/users/paulmck/sfq.ps.Z

[21] Shreedhar, M., Varghese, G.: *Efficient Fair Queuing Using Deficit Round Robin*, IEEE/ACM Transactions on Networking, 1995.
http://www-cse.ucsd.edu/users/varghese/PAPERS/fq.ps

[22] *Maximum throughput scheduling*, Wikipedia, [citováno 22.5.2009].
http://en.wikipedia.org/wiki/Maximum_throughput_scheduling

[23] *Proportionally fair scheduling*, Wikipedia, [citováno 22.5.2009].
http://en.wikipedia.org/wiki/Proportional_fairness

[24] *Omnet++: Discrete Event Simulation System*, [citováno 7.6.2009].
http://www.omnetpp.org/

[25] *INET Framework for OMNeT++*, [citováno 7.6.2009].
http://inet.omnetpp.org/

[26] Varga, A.: *OMNeT++ Discrete Event Simulation System Version 3.0 User Manual*, 2004.
http://www.omnetpp.org/doc/omnetpp33/manual/usman.html

[27] *INET Framework for OMNeT++/OMNEST*, 2006.
http://www.omnetpp.org/doc/INET/neddoc/index.html

[28] Linux Foundation: *How to Participate in the Linux Community*, [citováno 21.6.2009].
http://ldn.linuxfoundation.org/how-participate-linux-community

[29] *FreeBSD Host Access Points*, FreeBSD Handbook, oddíl 31.3.5, [citováno 18.6.2009].
http://www.freebsd.org/doc/en/books/handbook/network-wireless.html#NETWORK-WIRELESS-AP

[30] *TIMEOUT(9)*, FreeBSD Kernel Developer's Manual, [citováno 11.7.2009].
http://www.freebsd.org/cgi/man.cgi?query=timeout&sektion=9

[31] *HASHINIT(9)*, FreeBSD Kernel Developer's Manual, [citováno 7.7.2009].
http://www.freebsd.org/cgi/man.cgi?query=hashinit

[32] Rao, A.: *The locking infrastructure in the FreeBSD kernel*, AsiaBSDCon, 2009.
http://2009.asiabsdcon.org/papers/abc2009-P6A-paper.pdf

[33] *MUTEX(9)*, FreeBSD Kernel Developer's Manual, [citováno 9.7.2009].
http://www.freebsd.org/cgi/man.cgi?query=mutex

[34] *RWLOCK(9)*, FreeBSD Kernel Developer's Manual, [citováno 12.7.2009].
http://www.freebsd.org/cgi/man.cgi?query=rwlock

[35] Coffman, E. G., Elphick, M., Shoshani, A.: *System Deadlocks*, ACM Computing Surveys, Volume 3, Issue 2 (1971), 67-78.
http://portal.acm.org/citation.cfm?id=356586.356588

[36] Slabý, J.: *ath5k: Past, present and future*, NFX Data communications and networking workshop, 2009.
http://www.nfx.cz/wiki/images/7/71/Ath5k-nfx.pdf

[37] Salzman, P. J., Burian, M., Pomerantz, O.: *The Linux Kernel Module Programming Guide*, 2007.
http://tldp.org/LDP/lkmpg/2.6/html/

[38] McKenney, P. E.: *What is RCU, Really?*, [citováno 18.7.2009].
http://rdrop.com/~paulmck/RCU/whatisRCU.html

[39] *OpenWrt pro AR531X*, [citováno 19.7.2009].
http://atheros.openwrt.net/

# Příloha A

# OMNeT++ Implementation API

*Dokumentace byla automaticky vygenerována z komentářů ve zdrojových kódech a je proto v angličtině.*

## A.1 Class Documentation

### A.1.1 Ieee80211MgmtAPSched Class Reference

#### Classes

- class Client

#### Public Member Functions

- Ieee80211MgmtAPSched ()
- virtual ~Ieee80211MgmtAPSched ()

#### Protected Types

- typedef std::map< MACAddress, Client, Ieee80211MgmtAP::MAC_compare > ClientMap

#### Protected Member Functions

- virtual void initialize (int stage)
- virtual void handleTimer (cMessage *msg)
- virtual void handleDataFrame (Ieee80211DataFrame *frame)
- virtual bool enqueue (cMessage *msg)
- virtual cMessage * dequeue ()
- void windowEnd ()
- void scheduleWindowTimer ()
- double weightedDuration (double duration, double average)
- double transmitDuration (Ieee80211DataFrame *frame)

- double frameDuration (int bits, double bitrate)
- void countFrameIn (Ieee80211DataFrame *frame)
- Client & getClientByMAC (MACAddress mac)

**Protected Attributes**

- ClientMap clientMap
- cMessage * windowTimer
- double bitrate
- double ackrate
- double wintime
- double expfactor
- double avgweight

#### A.1.1.1 Detailed Description

Ieee80211MgmtAPSched extends Ieee80211MgmtAPSimplified by a packet scheduler. Single transmit queue is replaced by separate queues for every client (identified by MAC address).

Bandwidth allocation is based on both short-term counter (total duration of transmits that happened in a time window) and long-term exponential average, recalculated at the end of the time window, which is used as weight for the short-term counter.

The scheduler operates with transmit durations as opposed to packet sizes and only reorders the packets, never adds artificial delays.

#### A.1.1.2 Member Typedef Documentation

**typedef std::map<MACAddress, Client, Ieee80211MgmtAP::MAC_compare> ClientMap** `[protected]`

Type of the binary tree for searching a client by MAC address.

#### A.1.1.3 Constructor & Destructor Documentation

**Ieee80211MgmtAPSched ()**

Constructor. Real initialization is done by the method initialize().

**virtual ~Ieee80211MgmtAPSched ()** `[virtual]`

Destructor, frees timer message.

#### A.1.1.4 Member Function Documentation

**virtual void initialize (int *stage*)** `[protected, virtual]`

Loads object parameters and starts timer by sending message windowTimer to itself.

**Parameters:**

***stage*** Not used, only passed to parent.

**virtual void handleTimer (cMessage * *msg*)** `[protected, virtual]`

Redefined from parent to add window end timer messsage (windowTimer).

**Parameters:**

***msg*** Received timer message.

**virtual void handleDataFrame (Ieee80211DataFrame * *frame*)** `[protected, virtual]`

Redefined from parent, transmit duration of a frame received from Wi-Fi network is added to sender's counters.

**Parameters:**

***frame*** Frame received from Wi-Fi network.

**virtual bool enqueue (cMessage * *msg*)** `[protected, virtual]`

Redefined from PassiveQueue, frame received from upper layer is enqueued in client's queue instead of a global queue.

**Parameters:**

***msg*** Frame received from upper layer.

**virtual cMessage* dequeue ()** `[protected, virtual]`

Redefined from PassiveQueue, the frame is dequeued from the queue of the client with the lowest value of weightedCounter instead of a global queue.

**Returns:** Dequeued frame or NULL if all queues are empty.

**void windowEnd ()** `[protected]`

Updates clients' exponential averages and resets counters. Called when windowTimer message is received, which happens periodically every wintime seconds.

**void scheduleWindowTimer ()** `[protected]`

Sends self-message windowTimer to be received in wintime seconds.

**double weightedDuration (double *duration*, double *average*)** `[protected]`

Calculates frame duration multiplied by weight.

**Parameters:**

***duration*** Frame duration in seconds.

***average*** Current value of exponential average.

**Returns:** $duration \cdot (1 + avgweight \cdot average)$

**double transmitDuration (Ieee80211DataFrame * *frame*)** `[protected]`

Estimates the duration of frame transmit sequence. The sequence consists of DIFS interval, frame transmit, SIFS interval and ACK transmit (propagation delay is neglected).

802.11b timings, frame transmit rate bitrate and ACK transmit rate ackrate are used for the calculation.

**Parameters:**

***frame*** Data frame transmitted in the sequence.

**Returns:** Transmit duration in seconds.

**double frameDuration (int *bits*, double *bitrate*)** `[protected]`

Calculates the duration of a frame transmit (including preamble and PHY header).

**Parameters:**

***bits*** Frame length in bits.

***bitrate*** Data transmit rate.

**Returns:** Transmit duration in seconds.

**void countFrameIn (Ieee80211DataFrame * *frame*)** `[protected]`

Adds the duration of complete frame transmit (including ACK) to sender's counters in case of RX frame or recipient's counters in case of TX frame.

**Parameters:**

***frame*** Received or transmitted frame.

**Client& getClientByMAC (MACAddress *mac*)** `[protected]`

Finds a client by MAC address or allocates a new one if it does not exist in clientMap yet.

**Parameters:**

***mac*** MAC addres of a client.

**Returns:** Client with specified MAC address.

#### A.1.1.5 Member Data Documentation

**ClientMap clientMap** `[protected]`

Binary tree (STL map) for searching a client by MAC address.

**cMessage* windowTimer** `[protected]`

Self-message used as a timer to schedule periodic window ends.

**double bitrate** `[protected]`

Transmit rate of data frames in bit/s. Must be set in INI configuration to the value of the same parameter of Ieee80211Mac and Ieee80211Radio.

**double ackrate** `[protected]`

Transmit rate of ACK frames in bit/s. Must be set in INI configuration to the ACK rate of Ieee80211Mac, which is currently hard-coded as 2 Mbit/s.

**double wintime** `[protected]`

Length of the time window in seconds. Tunable via INI parameter with the same name.

**double expfactor** `[protected]`

Determines the aging of the exponential average (how fast older values are forgotten). Values measured in $k$ last windows have fraction $p$ of total weight for $expfactor = 1 - \sqrt[k]{1-p}$. Tunable via INI parameter with the same name.

**double avgweight** `[protected]`

Determines how much the exponential average affects bandwidth share. The bandwith is split between two clients at most in ratio $1 : (1 + avgweight)$. Tunable via INI parameter with the same name.

### A.1.2 Ieee80211MgmtAPSched::Client Class Reference

#### Public Attributes

- double average
- double durCounter
- double weightedCounter
- cQueue queue

#### A.1.2.1 Detailed Description

Represents a single client, identified by his MAC address. Allocated dynamically on arrival of a first frame from the client.

#### A.1.2.2 Member Data Documentation

**double average**

Exponentially weighted moving average of client's bandwidth share. Updated at the end of time window (ie. every wintime seconds) using formula $average = average \cdot (1 - expfactor) + share \cdot expfactor$.

**double durCounter**

Total duration of transmits from or to the client in current window.

**double weightedCounter**

Total duration of transmits from or to the client in current and previous

window multiplied by weight factor (derived from average using formula $weight = 1 + avgweight \cdot average$).

**cQueue queue**

Queue of packets to be sent to the client.

# Příloha B

# FreeBSD Implementation API

*Dokumentace byla automaticky vygenerována z komentářů ve zdrojových kódech a je proto v angličtině.*

## B.1 File Documentation

### B.1.1 ath_sched.h File Reference

ath_sched is a packet scheduler for Wi-Fi devices.

#### Data Structures

- struct **ath_sched_list**
- struct **ath_sched_entry**
- struct ath_sched_client
- struct ath_sched_state

#### Defines

- #define ATH_SCHED_HASHSIZE 64
- #define ATH_SCHED_DEF_WINTIME 200
- #define ATH_SCHED_DEF_EXPFACTOR 1000
- #define ATH_SCHED_DEF_AVGWEIGHT 4

#### Functions

- void ath_sched_attach (struct ath_sched_state *state, device_t netdev, struct ifnet *ifp)
- void ath_sched_detach (struct ath_sched_state *state)
- void ath_sched_enqueue (struct ath_sched_state *state, uint8_t *macaddr, struct mbuf *m)
- struct mbuf * ath_sched_dequeue (struct ath_sched_state *state)

- void ath_sched_count_in (struct ath_sched_state *state, uint8_t *macaddr, int duration)
- void ath_sched_set_running (struct ath_sched_state *state, int running)
- static int ath_sched_enabled (struct ath_sched_state *state)

#### B.1.1.1 Detailed Description

ath_sched is a packet scheduler for Wi-Fi devices.

The scheduler is meant to be integrated into the driver of a Wi-Fi interface acting as an access point. When the scheduler is enabled, available bandwidth is divided equally amongst clients.

Bandwidth allocation is based on both short-term counter (total duration of transmits that happened in a time window) and long-term exponential average, recalculated at the end of the time window, which is used as weight for the short-term counter.

The scheduler operates with transmit durations as opposed to packet sizes and only reorders the packets, never adds artificial delays. This is not possible in current traffic control framework, so the scheduler has to interface directly with a device driver.

Modifications of a driver required for the integration of the scheduler consist of limiting the length of hardware frame queue as much as possible (optimally to just 2 frames) and then adding calls to packet scheduler methods as specified in their description.

#### B.1.1.2 Define Documentation

**#define ATH_SCHED_HASHSIZE 64**

Size of a hash table for searching a client by his MAC address. Must be a power of two lower or equal to 256.

**#define ATH_SCHED_DEF_WINTIME 200**

Default value of sysctl parameter wintime (in milliseconds) (see ath_sched_state.wintime).

**#define ATH_SCHED_DEF_EXPFACTOR 1000**

Default value of sysctl parameter expfactor (see ath_sched_state.expfactor).

**#define ATH_SCHED_DEF_AVGWEIGHT 4**

Default value of sysctl parameter avgweight (see ath_sched_state.avgweight).

#### B.1.1.3 Function Documentation

**void ath_sched_attach (struct ath_sched_state * *state*, device_t *netdev*, struct ifnet * *ifp*)**

Initializes the packet scheduler state. Must be called before any other method.

The state should be embedded in driver private data and this function should be called from device attach handler.

**Parameters:**

| | |
|---|---|
| ***state*** | Scheduler state to be initialized. |
| ***netdev*** | Network device the scheduler is attached to. |
| ***ifp*** | Interface the scheduler is attached to. |

**void ath_sched_detach (struct ath_sched_state * *state*)**

Destroys the packet scheduler state. Should be called from device detach handler.

**Parameters:**

| | |
|---|---|
| ***state*** | Packet scheduler state. |

**void ath_sched_enqueue (struct ath_sched_state * *state*, uint8_t * *macaddr*, struct mbuf * *m*)**

Adds a frame to the scheduler's queue. The frame might be dropped (eg. if memory allocation fails). Every data frame received by the driver from TCP/IP stack should be passed to this function.

**Parameters:**

| | |
|---|---|
| ***state*** | Packet scheduler state. |
| ***macaddr*** | MAC address of the recipient of the frame. |
| ***m*** | The frame to be enqueued. |

**struct mbuf* ath_sched_dequeue (struct ath_sched_state * *state*)** `[read]`

Chooses a frame to be transmitted. Should be called when there's free space in hardware frame queue to select the next frame to be transmitted.

The frame is dequeued from the queue of the client with the lowest value of ath_sched_client.weighted_counter.

**Parameters:**

| | |
|---|---|
| ***state*** | Packet scheduler state. |

**Returns:** Frame to be sent or NULL if all queues are empty.

**void ath_sched_count_in (struct ath_sched_state * *state*, uint8_t * *macaddr*, int *duration*)**

Adds the duration of a RX or TX transmit to client's counters. Should be called from the handler of send and receive completion event.

The calculation of transmit duration (including ACK and DIFS) must be implemented in the driver. It should be as exact as possible with available information (transmit rate, retry count).

**Parameters:**

| | |
|---|---|
| ***state*** | Packet scheduler state. |

***macaddr*** MAC address of the recipient in case of TX frame or sender in case of RX frame.

***duration*** Duration of the transmit in milliseconds.

**void ath_sched_set_running (struct ath_sched_state * *state*, int *running*)**

Starts or stops the packet scheduler. Should be called when device state changes.

**Parameters:**

***state*** Packet scheduler state.

***running*** Boolean: whether the device is now running.

**static int ath_sched_enabled (struct ath_sched_state * *state*)** `[inline, static]`

Returns whether the packet scheduler is enabled. Driver must check that the scheduler is enabled before calling functions ath_sched_enqueue(), ath_sched_dequeue() or ath_sched_count_in().

**Parameters:**

***state*** Packet scheduler state.

**Returns:** Boolean: whether the packet scheduler is enabled.

## B.2 Data Structure Documentation

### B.2.1 ath_sched_client Struct Reference

#### Data Fields

- uint8_t macaddr [ETHER_ADDR_LEN]
- struct ifqueue queue
- struct mtx lock
- unsigned int counter
- unsigned int weighted_counter
- unsigned int weight
- unsigned int average
- ath_sched_entry clients_entry
- ath_sched_entry active_entry
- ath_sched_entry hash_entry
- int in_active_list

#### B.2.1.1 Detailed Description

Represents a single client, identified by his MAC address. Allocated dynamically on arrival of a first frame from the client.

#### B.2.1.2 Field Documentation

**uint8_t macaddr[ETHER_ADDR_LEN]**

MAC address (unique identifier of the client).

**struct ifqueue queue** `[read]`

Queue of packets to be sent to the client.

**struct mtx lock** `[read]`

Mutex protecting against race conditions in counter updates. When anyone holds a pointer to the client, he must hold this mutex too, otherwise the client might be deallocated!

**unsigned int counter**

Total duration of transmits from or to the client in current window.

**unsigned int weighted_counter**

Total duration of transmits from or to the client in current and previous window multiplied by weight.

**unsigned int weight**

Factor for weighted_counter calculation. Available bandwidth is divided amongst clients in inverse proportion to their weight. Derived from average using formula $weight = 1 + avgweight \cdot average$.

**unsigned int average**

Exponentially weighted moving average of client's bandwidth share. Updated at the end of time window (ie. every wintime milliseconds) using formula $average = average - average/expfactor + share/expfactor$.

**ath_sched_entry clients_entry**

Entry in the list of all clients (ath_sched_state.clients).

**ath_sched_entry active_entry**

Entry in the list of active clients (ath_sched_state.active).

**ath_sched_entry hash_entry**

Entry in the row of the hash table (ath_sched_state.hash).

**int in_active_list**

Boolean: whether the client is present in the list of active clients (ath_sched_state.active).

### B.2.2 ath_sched_state Struct Reference

#### Data Fields

- struct ifnet * ifp
- struct ath_sched_list clients
- struct ath_sched_list active
- struct ath_sched_list hash [64]
- struct rwlock clients_lock
- struct rwlock active_lock
- struct rwlock hash_lock
- struct callout timer
- int enabled
- int running
- unsigned int wintime
- unsigned int expfactor
- unsigned int avgweight

#### B.2.2.1 Detailed Description

Main data structure of the packet scheduler instance. Should be embedded into driver private data (softc).

#### B.2.2.2 Field Documentation

**struct ifnet* ifp** `[read]`

Interface the scheduler is attached to. Used only to write device name in messages.

**struct ath_sched_list clients** `[read]`

List of all clients.

**struct ath_sched_list active** `[read]`

List of active clients (those with non-empty queue).

**struct ath_sched_list hash[64]** `[read]`

Hash table for searching a client by MAC address. Last bits of the address are used as a row index.

**struct rwlock clients_lock** `[read]`

Read/write lock protecting the list clients.

**struct rwlock active_lock** `[read]`

Read/write lock protecting the list active.

**struct rwlock hash_lock** `[read]`

Read/write lock protecting all rows of hash table (hash).

**struct callout timer** `[read]`

Timer for scheduling periodic ends of windows.

**int enabled**

Boolean: whether the scheduler is enabled. Tunable via sysctl parameter with the same name.

**int running**

Boolean: whether the interface the scheduler is attached to is running (up and associated).

**unsigned int wintime**

Length of the time window in milliseconds. Minimum value is 10, maximum 1000. Tunable via sysctl parameter with the same name.

**unsigned int expfactor**

Determines the aging of the exponential average (how fast older values are forgotten). Values measured in $k$ last windows have fraction $p$ of total weight for $expfactor = \frac{1}{1-\sqrt[k]{1-p}}$. Minimum value is 1, maximum $10^9$. Tunable via sysctl parameter with the same name.

**unsigned int avgweight**

Determines how much the exponential average affects bandwidth share. The bandwith is split between two clients at most in ratio $1 : (1 + avgweight)$. Minimum value is 0, maximum 20. Tunable via sysctl parameter with the same name.

# Příloha C

# Linux Implementation API

*Dokumentace byla automaticky vygenerována z komentářů ve zdrojových kódech a je proto v angličtině.*

## C.1 File Documentation

### C.1.1 ath_sched.h File Reference

ath_sched is a packet scheduler for Wi-Fi devices.

#### Data Structures

- struct ath_sched_client
- struct ath_sched_state

#### Defines

- #define ATH_SCHED_HASHSIZE 64
- #define ATH_SCHED_DEF_WINTIME 200
- #define ATH_SCHED_DEF_EXPFACTOR 1000
- #define ATH_SCHED_DEF_AVGWEIGHT 4

#### Functions

- void ath_sched_attach (struct ath_sched_state *state, char *devname)
- void ath_sched_detach (struct ath_sched_state *state)
- void ath_sched_enqueue (struct ath_sched_state *state, uint8_t *macaddr, struct sk_buff *skb)
- struct sk_buff * ath_sched_dequeue (struct ath_sched_state *state)
- void ath_sched_count_in (struct ath_sched_state *state, uint8_t *macaddr, int duration)
- void ath_sched_set_running (struct ath_sched_state *state, int running)

- ctl_table ath_sched_make_sysctl (struct ath_sched_state *state)
- static int ath_sched_enabled (struct ath_sched_state *state)

#### C.1.1.1 Detailed Description

ath_sched is a packet scheduler for Wi-Fi devices.

The scheduler is meant to be integrated into the driver of a Wi-Fi interface acting as an access point. When the scheduler is enabled, available bandwidth is divided equally amongst clients.

Bandwidth allocation is based on both short-term counter (total duration of transmits that happened in a time window) and long-term exponential average, recalculated at the end of the time window, which is used as weight for the short-term counter.

The scheduler operates with transmit durations as opposed to packet sizes and only reorders the packets, never adds artificial delays. This is not possible in current traffic control framework, so the scheduler has to interface directly with a device driver.

Modifications of a driver required for the integration of the scheduler consist of limiting the length of hardware frame queue as much as possible (optimally to just 2 frames) and then adding calls to packet scheduler methods as specified in their description.

#### C.1.1.2 Define Documentation

**#define ATH_SCHED_HASHSIZE 64**

Size of a hash table for searching a client by his MAC address. Must be a power of two lower or equal to 256.

**#define ATH_SCHED_DEF_WINTIME 200**

Default value of sysctl parameter wintime (in milliseconds) (see ath_sched_state.wintime).

**#define ATH_SCHED_DEF_EXPFACTOR 1000**

Default value of sysctl parameter expfactor (see ath_sched_state.expfactor).

**#define ATH_SCHED_DEF_AVGWEIGHT 4**

Default value of sysctl parameter avgweight (see ath_sched_state.avgweight).

#### C.1.1.3 Function Documentation

**void ath_sched_attach (struct ath_sched_state * *state*, char * *devname*)**

Initializes the packet scheduler state. Must be called before any other method.

The state should be embedded in driver private data and this function should be called from device attach handler.

**Parameters:**

***state*** Scheduler state to be initialized.

***devname*** Name of the network interface the scheduler is attached to.

**void ath_sched_detach (struct ath_sched_state * *state*)**

Destroys the packet scheduler state. Should be called from device detach handler.

**Parameters:**

***state*** Packet scheduler state.

**void ath_sched_enqueue (struct ath_sched_state * *state*, uint8_t * *macaddr*, struct sk_buff * *skb*)**

Adds a frame to the scheduler's queue. The frame might be dropped (eg. if memory allocation fails). Every data frame received by the driver from TCP/IP stack should be passed to this function.

**Parameters:**

***state*** Packet scheduler state.

***macaddr*** MAC address of the recipient of the frame.

***skb*** The frame to be enqueued.

**struct sk_buff* ath_sched_dequeue (struct ath_sched_state * *state*)** `[read]`

Chooses a frame to be transmitted. Should be called when there's free space in hardware frame queue to select the next frame to be transmitted.

The frame is dequeued from the queue of the client with the lowest value of ath_sched_client.weighted_counter.

**Parameters:**

***state*** Packet scheduler state.

**Returns:** Frame to be sent or NULL if all queues are empty.

**void ath_sched_count_in (struct ath_sched_state * *state*, uint8_t * *macaddr*, int *duration*)**

Adds the duration of a RX or TX transmit to client's counters. Should be called from the handler of send and receive completion event.

The calculation of transmit duration (including ACK and DIFS) must be implemented in the driver. It should be as exact as possible with available information (transmit rate, retry count).

**Parameters:**

***state*** Packet scheduler state.

***macaddr*** MAC address of the recipient in case of TX frame or sender in case of RX frame.

***duration*** Duration of the transmit in milliseconds.

**void ath_sched_set_running (struct ath_sched_state * *state*, int *running*)**

Starts or stops the packet scheduler. Should be called when device state changes.

**Parameters:**

***state*** Packet scheduler state.

***running*** Boolean: whether the device is now running.

**ctl_table ath_sched_make_sysctl (struct ath_sched_state * *state*)**

Creates sysctl parameters for tuning scheduler properties. Returned table row should be added to device parameters.

**Parameters:**

***state*** Packet scheduler state.

**Returns:** A single row of sysctl table.

**static int ath_sched_enabled (struct ath_sched_state * *state*)** `[inline, static]`

Returns whether the packet scheduler is enabled. Driver must check that the scheduler is enabled before calling functions ath_sched_enqueue(), ath_sched_dequeue() or ath_sched_count_in().

**Parameters:**

***state*** Packet scheduler state.

**Returns:** Boolean: whether the packet scheduler is enabled.

## C.2 Data Structure Documentation

### C.2.1 ath_sched_client Struct Reference

#### Data Fields

- uint8_t macaddr [IEEE80211_ADDR_LEN]
- struct sk_buff_head queue
- spinlock_t lock
- unsigned int counter
- unsigned int weighted_counter
- unsigned int weight
- unsigned int average
- struct list_head clients_entry
- struct list_head active_entry
- struct list_head hash_entry
- int in_active_list
- struct rcu_head rcu

#### C.2.1.1 Detailed Description

Represents a single client, identified by his MAC address. Allocated dynamically on arrival of a first frame from the client.

#### C.2.1.2 Field Documentation

**uint8_t macaddr[IEEE80211_ADDR_LEN]**

MAC address (unique identifier of the client).

**struct sk_buff_head queue** `[read]`

Queue of packets to be sent to the client.

**spinlock_t lock**

Spin lock protecting against race conditions in counter updates. When anyone holds a pointer to the client, he must hold this lock too, otherwise the client might be deallocated!

**unsigned int counter**

Total duration of transmits from or to the client in current window.

**unsigned int weighted_counter**

Total duration of transmits from or to the client in current and previous window multiplied by weight.

**unsigned int weight**

Factor for weighted_counter calculation. Available bandwidth is divided amongst clients in inverse proportion to their weight. Derived from average using formula $weight = 1 + avgweight \cdot average$.

**unsigned int average**

Exponentially weighted moving average of client's bandwidth share. Updated at the end of time window (ie. every wintime milliseconds) using formula $average = average - average/expfactor + share/expfactor$.

**struct list_head clients_entry** `[read]`

Entry in the list of all clients (ath_sched_state.clients).

**struct list_head active_entry** `[read]`

Entry in the list of active clients (ath_sched_state.active).

**struct list_head hash_entry** `[read]`

Entry in the row of the hash table (ath_sched_state.hash).

**int in_active_list**

Boolean: whether the client is present in the list of active clients (ath_sched_state.active).

**struct rcu_head rcu** `[read]`

Deallocation of client structure is delayed using RCU until no one can see the old data.

### C.2.2 ath_sched_state Struct Reference

#### Data Fields

- char * devname
- struct list_head clients
- struct list_head active
- struct list_head hash [64]
- spinlock_t clients_lock
- spinlock_t active_lock
- struct timer_list timer
- int enabled
- int running
- unsigned int wintime
- unsigned int expfactor
- unsigned int avgweight

#### C.2.2.1 Detailed Description

Main data structure of the packet scheduler instance. Should be embedded into driver private data (softc).

#### C.2.2.2 Field Documentation

**char* devname**

Name of the network interface the scheduler is attached to (for messages).

**struct list_head clients** `[read]`

List of all clients.

**struct list_head active** `[read]`

List of active clients (those with non-empty queue).

**struct list_head hash[64]** `[read]`

Hash table for searching a client by MAC address. Last bits of the address are used as a row index.

**spinlock_t clients_lock**

Spin lock protecting against concurrent modifications of the list clients and hash table (hash).

**spinlock_t active_lock**

Spin lock protecting against concurrent modifications of the list active.

**struct timer_list timer** `[read]`

Timer for scheduling periodic ends of windows.

**int enabled**

Boolean: whether the scheduler is enabled. Tunable via sysctl parameter with the same name.

**int running**

Boolean: whether the interface the scheduler is attached to is running (up and associated).

**unsigned int wintime**

Length of the time window in milliseconds. Minimum value is 10, maximum 1000. Tunable via sysctl parameter with the same name.

**unsigned int expfactor**

Determines the aging of the exponential average (how fast older values are forgotten). Values measured in $k$ last windows have fraction $p$ of total weight for $expfactor = \frac{1}{1-\sqrt[k]{1-p}}$. Minimum value is 1, maximum $10^9$. Tunable via sysctl parameter with the same name.

**unsigned int avgweight**

Determines how much the exponential average affects bandwidth share. The bandwith is split between two clients at most in ratio $1 : (1 + avgweight)$. Minimum value is 0, maximum 20. Tunable via sysctl parameter with the same name.