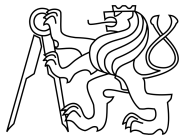
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Analýza mechanizmů kontroly rychlosti datového toku na transportní vrstvě
Student:	Jan Rubín
Vedoucí:	Ing. Viktor Černý
Studijní program:	Informatika
Studijní obor:	Informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Protokol TCP je protokolem čtvrté (neboli transportní) vrstvy ISO-OSI modelu. Jednou z jeho funkcí je kontrola rychlosti datového toku. V současné době však existuje mnoho implementací, které se liší schopnostmi reagovat na výpadek nebo zpomalení přenosové linky. Předpokládaným rozdílem implementací je jejich efektivita na silně nebo slabě chybujících linkách. Dalším faktorem ovlivňujícím efektivitu je zpoždění linky.

Zmapujte schopnosti jednotlivých implementací a najděte jejich silné a slabé stránky.
V součinnosti s vedoucím práce vyberte nejnadějnější varianty a na jejich základě vytvořte hybridní implementaci.
Vytvořte jednoduché testovací prostředí a vhodnou metodiku, které umožní porovnání různých implementací.
Výkonnost vaší varianty porovnejte s referenční implementací standardu protokolu TCP.
Pro implementaci zvolte libovolný programovací nebo skriptovací jazyk.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 17. ledna 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Bakalářská práce

Analýza mechanizmů kontroly rychlosti datového toku na transportní vrstvě

Jan Rubín

Vedoucí práce: Ing. Viktor Černý

8. května 2016

Poděkování

V první řadě bych rád poděkoval Ing. Viktoru Černému za vedení této práce, poskytování cenných informací a za nekončící trpělivosti při mnoha konzultacích. Dále bych rád poděkoval své rodině za podporu v průběhu celého studia a všem dalším, kteří jakkoliv přispěli ke vzniku této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 8. května 2016 .....

Odkaz na tuto práci

Rubín, Jan. *Analýza mechanizmů kontroly rychlosti datového toku na transportní vrstvě*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Obsahem této bakalářské práce je analýza mechanizmů kontroly rychlosti datového toku na transportní vrstvě. Řada mechanizmů, které řeší problém zahlcení sítě, je již v protokolu TCP implementována. Některá řešení ale nejsou efektivní a stále na nich probíhá výzkum. Tato práce jednotlivé mechanizmy studuje, popisuje jejich funkcionalitu a principy, které napomáhají předcházet či řešit zahlcení sítě. Rozebírá také silné a slabé stránky konkrétních implementací, které jsou ukázány ve vyvinutém testovacím prostředí. Součástí práce je i navržení dvou hybridních implementací, které kombinují vlastnosti již existujících řešení a přinášejí i nové funkce, jak kontrolovat datový tok.

Klíčová slova zahlcení sítě, analýza rychlosti datového toku, simulace sítě, algoritmy transportní vrstvy, testovací prostředí

Abstract

In this Bachelor thesis, the flow control mechanisms on the transport layer are analyzed. There are already many algorithms implemented in the TCP that treat network congestion. However, some of them are not efficient enough and they are a topic of ongoing research. These mechanisms, their functionality, as well as the principles helping to prevent or solve network congestion are studied in this thesis. The advantages and the disadvantages of the particular implementations are inspected. These implementations are tested in the developed environment. Furthermore, two hybrid implementations which combine the features of the existing solutions and include functionalities for network congestion are designed.

Keywords network congestion, flow control analysis, network simulation, transport layer algorithms, testing environment

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Internet je dnes součástí každodenního života a většina lidí si jeho absenci nedokáže ani představit. Aby mohl být přenos dat po internetu vůbec realizovatelný a aby fungoval efektivně, je nutné podrobně zkoumat aspekty datové komunikace a směřovat její vývoj správným směrem.

TCP je protokolem čtvrté (transportní) vrstvy ISO-OSI modelu, který musí zajistit spolehlivou komunikaci mezi odesílatelem a příjemcem [1]. Jeho součástí je i kontrola rychlosti datového toku včetně způsobů navázání stabilního spojení bez větších výkyvů. Jedná se o velmi důležité vlastnosti síťových komunikací, protože bez nich by taková komunikace nebyla ani možná. Nedodržování mechanizmů kontroly rychlosti datových toků by mělo velké následky pro dnešní podobu internetu a skončilo by úplným kolapsem sítě.

Řešením problémů, které mohou při komunikaci nastat, se již zabývalo mnoho výzkumů a na transportní vrstvě již existuje mnoho implementací, které je řeší více či méně efektivně. Standardních řešení však už tolik není a často se jedná pouze o naivní algoritmy, které je potřeba blíže zkoumat a neustále rozvíjet. Samotné požadavky na síťovou komunikaci jsou velmi dynamické a tyto algoritmy se jim musejí neustále přizpůsobovat na základě nových výzkumů.

Tato práce podrobně zkoumá problematiku zahlcení a způsoby, kterými lze regulovat rychlost datových toků z důvodu předcházení nebo řešení této problematiky. Uvádí standardní implementace, které jsou již v TCP obsaženy a srovnává jejich výhody a nevýhody s nestandardními algoritmy. Práce navíc přináší dvě hybridní implementace, které zavedené principy mění a rozšiřují. Odlišnosti všech algoritmů jsou názorně ukázány pomocí vyvinutého testovacího prostředí, které simuluje síťovou komunikaci. Součástí práce je i diskuze probíhajících výzkumů a budoucností, kterou se bude problematika zahlcení pravděpodobně ubírat.

Struktura práce

V první kapitole je zaveden obecný problém zahlcení a jaké následky může mít na komunikaci v síti. Následuje seznámení se způsoby regulace datového toku a je zavedeno obecné členění novodobých mechanizmů řešících zahlcení.

Druhá kapitola popisuje existující mechanizmy, které jsou v TCP implementovány. Vysvětluje jejich rozlišnosti ve způsobu detekce a řešení zahlcení. Na konkrétních příkladech ukazuje obecné principy, jak tyto mechanizmy fungují a utváří základ pro bližší porozumění funkčnosti algoritmů. Dále je zde odůvodněn výběr konkrétních implementací, které jsou dopodrobna rozebrány ve třetí kapitole. Nakonec přináší úvod do probíhajících výzkumů, které jsou stále nedílnou součástí zlepšování těchto mechanizmů.

Třetí kapitola se zabývá implementací testovacího prostředí, které simuluje síťovou komunikaci. V tomto prostředí jsou podrobně rozebírány konkrétní algoritmy, řešící problém zahlcení. Jsou zavedeny i obecné výpočty, které odesílatel musí provádět pro správnou funkčnost všech algoritmů. Mezi tyto výpočty patří například přepočítávání časovače pro událost timeout ke konkrétnímu spojení [2].

Čtvrtá kapitola provádí vzorové testy algoritmů popsaných v předchozí kapitole. Jednotlivé testy dělí po sekcích kvůli konkrétním problematikám, které se v síti vyskytují. V uvedených grafech jsou ukázány silné a slabé stránky jednotlivých implementací, patřičně obohacené slovním vysvětlením. Nakonec je uveden souhrn pozorovaných vlastností.

Kapitola 1

Analýza problematiky

1.1 Problematika zahlcení sítě

Mějme modelovou situaci, kdy server naváže spojení s klientem a chce mu začít posílat data. Problém ale může nastat, když příjemce bude velmi pomalé zařízení v pomalé síti. Server dopředu neví, kolik dat může posílat a tyto dvě strany se musejí dohodnout, aby nedošlo k takzvanému zahlcení. Ten samý problém však může nastat, když je přetížen nějaký síťový uzel na cestě k příjemci, přestože on je schopný přijímat stejný (nebo i vyšší) objem dat, jako odesílatel posílá. Tento jev se nazývá Congestion Collapse [3] a podrobněji ho vysvětlím v následující podsekci.

Obecně existují tři způsoby, jak se zahlcení zjišťuje. Nejrozšířenější metodou je detekce pomocí ztrátovosti paketů. Problém tohoto přístupu však je, že zahlcení se zjistí až poté, co k němu již došlo. Tato metoda je označována jako "Black box", protože nevyžaduje žádnou hlubší znalost sítě. Druhým způsobem je pozorování odezvy, jak rychle dokáže příjemce reagovat na posílaná data a podle toho upravovat objem posílaných dat. Tato metoda je přísně závislá na schopnosti počítání přesných RTT a je označována jako "Grey box", protože vyžaduje určité měření navíc. Třetím způsobem je posílání signálů o maximálním množství dat, které je schopný každý uzel v síti přijmout. Označuje se jako "Green box"[4].

1.1.1 Congestion Collapse

Při velké síťové zátěži, kdy spojovací uzly zpracovávající tok dat podléhají většímu náporu, zvětšuje se doba přenosu dat od odesílatele k příjemci a celkový objem dat, který je na cestě [3]. Tento průběh komunikace je při zátěži normální a síťové prvky s ním počítají díky přepočítávání odezvy mezi uzly a koncovými body. Nutnou podmínkou pro tento bezproblémový provoz v zátěži však je, aby každý datagram byl v síti právě jednou. Tomu říkáme, že je zahlcení pod kontrolou. Pokud ale odesílatel začne přeposílat dříve odeslaná

data (například dlouhou dobu nedošlo potvrzení), v síti nastává velké riziko zahlcení.

Znovu-posílání paketů po určitých narůstajících intervalech je běžný způsob, jak se síť vypořádává s běžným zahlcením a často tento způsob stačí pro jeho eliminaci. Narůstání intervalů má obyčejně horní mez. V úvahu se musí ale připustit situace, kdy s cílovým zařízením (nebo síťovým uzlem) komunikuje víc než jedno spojení. Poté je velká pravděpodobnost, že zpoždění na daném prvku naroste o tak velkou hodnotu, že výsledný čas pro doručení naroste rychleji, než odesílatel stačí aktualizovat svůj interval pro přijímání potvrzení. Pokud již odesílatel dospěl k nějakému maximálnímu bodu, po který je ochotný čekat na znovu-poslání paketu, ale běh sítě je zpomalen ještě víc, může se pak stát, že bude odesílatel nechtěně posílat víc a víc stejných paketů do sítě, přestože to není potřeba. Nakonec se všechny buffery v síťových uzlech zcela zaplní a nutně dojde k situaci, kdy se pakety začnou zahazovat. [3]

1.1.2 Congestion Window

Součástí standardu TCP protokolu je takzvané Congestion Window. Slouží k omezení maximálního datového toku, který může odesílatel poslat do sítě pro určité spojení bez nutnosti čekat na potvrzení. Tato hodnota tedy přímo odpovídá počtu nepotvrzených dat v kanálu a nesmí být nikdy po dobu běhu překročena [5].

Aby odesílatel věděl, že byla data v pořádku doručena, příjemce musí posílat nazpět potvrzení. To je označované jako ACK, tedy z angličtiny "acknowledgment". Je to jediný nástroj, který se nějakým způsobem snaží řešit problém dvou armád, přestože problém jako takový nemá řešení [6]. Jeden ACK může potvrzovat jednak jednotlivé segmenty, tak bloky segmentů nebo celé pakety a bloky paketů.

V průběhu komunikace se Congestion Window zvětšuje a zmenšuje v závislosti na pozorovaných vlastnostech sítě. Zpravidla platí, že pokud příjemce bez problémů potvrzuje data, která mu došla, okno roste. Pokud jsou v síti pozorované náznaky zahlcení, okno se zmenšuje. Tímto se zajistí, aby nebyl příjemce zahlcen příliš mnoha daty.

Způsobů, jak hýbat s velikostí okna, je mnoho. Tuto funkcionalitu převážně zajišťují algoritmy Slow-start a Congestion Avoidance popsané níže. Podrobné zkoumání těchto algoritmů podléhá neustálému výzkumu a optimální řešení se stále hledá.

Congestion Window nesmí být zaměňováno s velikostí okna, které má příjemce. Tato okna jsou odlišná a do určité míry nezávislá. Je jisté, že Congestion Window nikdy nebude konvergovat k velikosti, která je větší, než velikost okna příjemce. To by jistě znamenalo vyčerpání místa u příjemce a nutně by následovala ztrátovost paketů, tedy zahlcení.

1.2 Členění obecných principů

TCP protokol se z pohledu zahlcení rozděluje na čtyři na sobě závislé algoritmy. Jejich provázanost je dobře popsaná v [7] nebo v [5], přičemž tři z nich se napříč implementacemi takřka nemění, zatímco Congestion avoidance algoritmy (někdy také přezdívané "TCP flavors") s sebou nesou výpočetní a detekční rozdílnosti zahlcení. Odlišný je i způsob, jakým se při zahlcení zotaví.

1.2.1 Slow-start

Před nasazením Slow-start algoritmu do TCP protokolu odesílatel zpravidla poslal několik paketů najednou do sítě. Tento počet se většinou rovnal velikosti okna příjemce, na kterém se s odesílatelem dopředu domluvili. Tento způsob je v pořádku, pokud je odesílatel i příjemce ve stejné síti (LAN) [7]. Pokud jsou mezi cílovými body nějaké pomalejší síťové uzly, může se lehce stát, že daný pomalejší prvek nestíhá tak rychle přeposílat data a zaplní se. Poté nutně musí následovat ztráta paketů.

Slow-start algoritmus tomuto snaží předcházet způsobem, kdy nejprve pošle jeden první paket do sítě a s dalšími počká, dokud nedostane potvrzení. Když se odeslaný paket v pořádku potvrdí, odesílatel pošle nový paket za již potvrzený a navíc zvětší okno o jeden další. Tím se efektivně zdvojnásobí počet paketů každý RTT. Algoritmus končí, pokud dospěje k předem dané maximální hodnotě *ssthresh*, do které je ochotný okno navyšovat, nebo pokud detekuje příznaky zahlcení (například ztrátu paketu).

Název Slow-start může vzbudit domnění, že se jedná o pomalou metodu posílání nových paketů. Toto je však zavádějící díky exponenciálnímu růstu okna. Tato standardní metoda je naopak považována za velmi agresivní [7].

Existuje více nestandardních implementací, které experimentují s počáteční velikostí okna. Některé studie uvažují větší velikost než jedna pro urychlení exponenciálního růstu. Přestože je tento způsob za určitých podmínek více efektivní [8], nejedná se o doporučený postup zavedení Slow-start algoritmu a nebudu se jím v této práci zabývat.

1.2.2 Congestion avoidance

Jak již bylo řečeno, obecně se zavádějí tři způsoby, jak detekovat zahlcení. Jedním je ztráta paketu, druhým je vypršení doby, kdy odesílatel čeká na odpověď (dojde k timeoutu) a posledním je nějaká vedlejší informace v podobě signálu. Další pohled na detekci zahlcení je takový, že součet všech dat na vstupu je větší, než dokáže nejpomalejší síťový prvek zpracovat na výstupu. Congestion avoidance algoritmus je metodou, jak se vypořádat se zahlcením sítě [7].

Způsob, kterým se zahlcení řeší, záleží na konkrétní implementaci daného Congestion avoidance algoritmu. Někdy se těmto mechanizmům přezdívá "TCP flavors", tedy TCP příchutě. Jejich charakteristika se liší ve způsobu, které události v síti pozorují jako zahlcení a také způsobem, jakým samotné zahlcení řeší. Vždy ale obsluhují změny velikost Congestion Window a tím regulují objem posílaných dat.

V této fázi se nenavyšuje okno s každým příchozím ACK, ale jednou za RTT (Round-trip time). Toto zajišťuje lineární nárůst velikosti okna oproti exponenciálnímu při Slow-startu.

Navrhované Congestion avoidance algoritmy počítají s tím, že výpadek paketu z důvodu poškození je velmi malý (méně než 1%) [7]. Přesto je dobré o slabinách algoritmů vědět pokud možno vše a proto i tuto vlastnost testuji v kapitole 4.

1.2.3 Fast retransmit

Přestože je tento algoritmus často slučován s Congestion avoidance, měl by být Fast retransmit uváděn zvlášť jako nezávislé řešení znovu-poslání paketu při výpadku. Zatímco Congestion avoidance se zaměřuje na obsluhu změny velikosti okna, Fast retransmit opravuje škody, které zahlcení způsobilo.

Když dojde příjemci paket s jiným sekvenčním číslem, než očekával, začne ke každému dalšímu ACK přidávat toto sekvenční číslo. Tím odesílateli signalizuje, že mu došel jiný paket a něco není v pořádku. Je potřeba si uvědomit, že se paket nutně nemusel ztratit, protože v síti může docházet k mírnému přeuspořádání paketů. Fast retransmit z tohoto důvodu počítá duplicitní ACK (tedy stejné ACK přicházející za sebou). Pokud jejich počet přesáhne standardní hodnotu tři, je velmi pravděpodobné, že paket vypadl, a je nutné ho poslat znovu [7] [5].

Některé implementace Congestion avoidance algoritmů zavádějí důmyslnější rozpoznání, kdy je již možné paket poslat. Vždy se tedy nemusí čekat na tři duplicitní ACK, ale nejedná se o standardní řešení. Tyto principy však nesmí být slučovány s Congestion avoidance jako takovém, jelikož se dají implementovat k jakémukoliv Congestion avoidance algoritmu [9].

1.2.4 Fast recovery

Poté, co Fast retransmit pošle paket kvůli výpadku, přejde se na Fast recovery. Efektivně se začne používat Congestion avoidance, rozdíl je pouze v situaci, ve které se komunikace nachází.

Princip Congestion avoidance ve fázi Fast recovery spočívá v tom, že příchozí ACK říkají víc než jen to, že příjemce čeká na určitý paket. ACK s sebou nese i sekvenční číslo paketu, který jistě musel příjemci dojít. Nutně to tedy znamená, že nedošlo ke Congestion Collapse a příjemce stále reaguje. Paket s daným sekvenčním číslem již tedy není v oběhu a nespotřebovává tedy pro-

středky sítě. Pro každý takový příchozí ACK může odesílatel poslat jeden nový paket dat navíc bez obavy, že je síť zahlcena nebo ji zahltí. Tímto způsobem komunikace nadále pokračuje. Do té doby, kdy příjemce odesílá alespoň nějaké ACK, informuje tím odesílatele o funkčnosti sítě. Z těchto důvodů odesílatel nechce zareagovat přílišnou redukcí Congestion Window (tedy množství odesílaných dat) a proto se nezačne provádět Slow-start, ale pokračuje Congestion avoidance. Fast recovery trvá tak dlouho, dokud nepřijde neduplicitní ACK [5].

Kapitola 2

Možnosti řešení

Od doby, kdy začaly první výzkumy ohledně spolehlivého a efektivního přenosu dat po síti, uběhlo mnoho let. Proto již existuje spousta řešení a implementací, které se snaží zabránit zahlcení sítě a pokud k němu dojde, tak se ho snaží co nejrychleji opravit. V této kapitole popíšu některé nejpoužívanější Congestion avoidance algoritmy, jaké mají výhody a nevýhody, které problémy řeší a proč je dobré je používat.

2.1 Existující algoritmy

2.1.1 TCP Tahoe a TCP Reno

V této podkapitole bych rád hned na úvod zmínil dva algoritmy, které se řadí mezi standardy a na nichž samotný výzkum začínal. Můžeme si všimnout, že například v publikaci [7] v sekci Congestion avoidance je popisován přímo algoritmus TCP Reno, přestože tam není takto pojmenován. Tahoe i Reno fungují na obdobném principu, přičemž Tahoe je starší variantou Congestion avoidance a přistupuje agresivněji ke zmenšování okna. Oba algoritmy detekují zahlcení pouze pomocí ztrátovosti paketů a to dvěma způsoby:

- Událost timeout
- Tři duplicitní ACK

Pokud dojde k timeoutu (odesílatel nemá žádný příchozí ACK od příjemce a vyprší RTO – tzv. "retransmission timeout"[2], viz 3.2.2.5), oba algoritmy zmenší okno na velikost jedna a zahájí Slow-start s polovičním *ssthresh*. Pokud se ztráta paketu detekovala pomocí třech duplicitních ACK, Reno sníží velikost okna na polovinu a *ssthresh* nastaví stejně. Nezačne tedy dělat Slow-start, ale zahájí (nebo pokračuje) Congestion avoidance. Tahoe takto nejedná. Ten bezpodmínečně sníží velikost okna na jedna – tedy stejně, jako kdyby došlo k timeoutu a zahájí Slow-start.

2.1.2 Vysokorychlostní algoritmy

S rozšiřováním sítí na celosvětové měřítko se postupem času musel vývoj přemístit i na vysokorychlostní algoritmy. Ty jsou nasazeny na linky s velkou propustností dat a s velkými odezvami (zpožděním). Tyto sítě jsou někdy označované jako "long fat networks"nebo "long, fat pipe"[10].

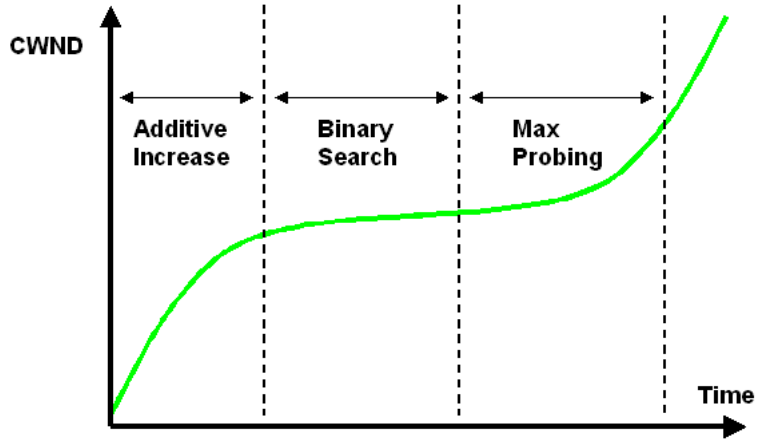
Mezi tyto algoritmy patří například High Speed TCP (HSTCP) [11] nebo BIC TCP ("Binary Increase Congestion control") [12].

HSTCP se chová stejně jako Reno 2.1.1, akorát od určité velikosti Congestion Window 1.1.2 využívá jiného poměru zvětšování a zmenšování okna. Obecně platí, že čím víc se posílá dat, tím víc okno narůstá. Naopak, pokud má dojít ke zmenšení okna (ztráta paketu zjištěná třemi duplicitními ACK), toto zmenšení je agresivnější. Tím je zajištěné dobré chování i v klasických linkách, ale také v linkách, kde je nutné přizpůsobit se velkým zpožděním s mnoha daty v oběhu.

BIC [12] je další způsob, jak se vypořádat s rychlými přenosy velkých objemů dat. To zajišťuje jeho unikátní funkce růstu Congestion Window, která se dělí na tři fáze. Když BIC zaznamená ztrátu paketu, zmenší okno multiplikativní proměnnou β. Těsně před zmenšením se zaznamená velikost okna jako hodnota W_{max} a těsně po redukci jako hodnota W_{min}. Poté BIC spustí binární vyhledávání mezi těmito dvěma hodnotami, které ohraničují interval. Toto vyhledávání odpovídá pouhému rozpůlení intervalu – tedy vybere se prostřední hodnota. Někde zde totiž musí být nutně optimální velikost okna, protože ztráta paketu nastala v rozmezí tohoto intervalu. Vzhledem k tomu, že by skok doprostřed intervalu mohl být moc velký nárůst okna, používá se konstanta zvaná S_{max}, která v tomto případě slouží jako maximum, o které se okno může najednou zvětšit. Pokud nastane ztráta paketu, aktuální velikost okna je nové maximum, pokud nenastane, stanoví se jako nové minimum. Tento proces pokračuje, dokud okno už neroste méně, než určuje hodnota S_{min}. Pokud okno naroste víc než stanovené maximum, pak nastal špatný odhad konvergence a musí se najít nové maximum. Tato fáze se jmenuje "max probing"a je inverzní funkcí k binárnímu vyhledávání. Roste tedy exponenciálně. Pokud brzy nenajde nové maximum, tak od určitého bodu změní své chování na lineární tím, že přičte k velikosti okna předem danou konstantu [12]. Tyto tři fáze ilustruje graf 2.1.

Je nutné zmínit, že algoritmy tohoto typu předpokládají velmi malou chybovost linky. Například u HSTCP předem odhadovaná chybovost souvisí s určením parametrů pro změny poměru růstu nebo zmenšování okna [11]. Tyto výpočty podrobněji ukážu v implementační části 3.3.3.

Vzhledem k tomu, že se jedná o algoritmy, které pracují s obrovskými okny a s malou chybovostí linky, vzniká riziko přílišného nárůstu okna ve fázi Slow-start 1.2.1. Pokud není Slow-start explicitně limitován nějakou menší hodnotou a nárůstu okna nebrání chybovost linky, může se stát, že při každém dalším RTT se okno zvětší o tisíce nových segmentů. To může lehce způsobit



Obrázek 2.1: Průběh funkce v BIC [12]

tisíce ztracených paketů v případě zahlcení sítě. Studie [11], která se zabývá HSTCP, zmiňuje shovívavější metodu Slow-startu, nazvanou "Limited Slow-start". Ta omezuje maximální zvětšení Congestion Window během jednoho RTT. Tato modifikace může být však použita i s jinými Congestion avoidance algoritmy.

2.1.3 Algoritmy s měřením zpoždění

Všechny předchozí zmíněné algoritmy detekují zahlcení pomocí ztrátovosti paketu. To přináší problém, že zahlcení zpravidla poznají až tehdy, kdy už nastalo, a již jsou napáchané škody. Modernější přístup algoritmů této kategorie spočívá v tom, že jako příznak zahlcení berou v potaz odezvy, resp. změny mezi naměřenými odezvami v síti. Tím efektivně odhadují zatíženost síťových prvků a mohou zahlcení předvídat. Tyto algoritmy ale těžce spoléhají na přesné výpočty RTT. Pokud bude výpočet nepřesný, bude odesílatel při menším odhadu posílat méně dat, než by mohl. Naopak při větším odhadu způsobí zahlcení linky.

Jedna z vlajkových lodí, která tento princip využívá, je algoritmus TCP Vegas. Byl vyvinut na University of Arizona, přičemž tvůrci ho pojmenovali podle největšího města Nevady.

V TCP Vegas probíhají dvě fáze výpočtu. První se soustředí na odhad očekávané propustnosti. Druhou je aktuální propustnost, která skutečně nastala. Myšlenka, proč dělat očekávané odhady, je předpokládaná souvislost objemu dat v síti s tím, kdy data dorazí. Tím TCP Vegas měří a upravuje objem dat, která jsou v oběhu navíc. To jsou data, která by odesílatel do sítě neposlal, kdyby věděl přesné kapacity sítě [9].

Aby mohly být tyto výpočty provedeny, TCP Vegas musí průběžně vypočítávat hodnotu $BaseRTT$. Ta odpovídá nejmenšímu RTT, který v síti nastal.

Typicky je tato hodnota rovná RTT pro první segment, který se v rámci Slow-startu pošle do sítě, protože v tuto dobu není síť zpravidla nijak zatížená a pakety na routerech nejsou bufferovány. Potom odhadovaná propustnost odpovídá rovnici

$$Expected = \frac{WindowSize}{BaseRTT}$$

kde $WindowSize$ je velikost Congestion Window 1.1.2, která odpovídá objemu dat v síti. Dále TCP Vegas potřebuje již zmíněnou aktuální propustnost, pojmenovanou *Actual*. Ta se spočítá tak, že se pro každý paket zaznamená čas, kdy byl odeslán do sítě a kdy dorazil jeho ACK. Toto umožní stanovit RTT pro daný paket. Dále se spočítá objem dat, který byl v tomto intervalu odeslán do sítě. Tato hodnota se vydělí získaným RTT. Celý výpočet se provádí jednou za round-trip time (zde ve smyslu jednou za celé odeslané okno).

V poslední řadě udělá TCP Vegas rozdíl těchto naměřených hodnoty, tedy

$$Diff = Expected - Actual$$

a dle této hodnoty upravuje velikost okna. Hodnota $Diff$ není nikdy záporná, protože tato skutečnost by nutně vyžadovala aktualizaci hodnoty $BaseRTT$ (našel se nový minimální RTT). Tedy nutně platí

$$Actual \leq Expected$$

a podmínka je vždy splněna. Hodnotu $Diff$ TCP Vegas porovnává se dvěma hranicemi α a β. Jejich hodnoty jsou zaváděné jako experimentální [9] a v průběhu výzkumů se mohou měnit. Vždy ale platí, že $\alpha \leq \beta$. Pokud je $Diff \leq \alpha$, TCP Vegas zvětší velikost okna, protože usuzuje, že v síti je málo dat. Naopak, když $Diff \geq \beta$, okno se zmenší, protože je v síti příliš mnoho dat. Optimální velikost okna tedy nutně spadá do intervalu $\alpha < Diff < \beta$. Pokud tomu tak je, velikost okna se nemění.

Publikace [9] také zavádí upravené algoritmy Slow-startu 2.1.3.1 a Fast retransmitu 2.1.3.2. Tyto upravené verze algoritmů lze ale použít i k ostatním Congestion avoidance algoritmům 1.2.2 a neměly by být tedy uvažovány jako přímá součást řešení TCP Vegas.

2.1.3.1 TCP Vegas Slow-start

Vzhledem k tomu, že TCP Reno efektivně zdvojnásobuje velikost okna každý RTT (bez omezení bufferů), může se nakonec předpokládat zahození celé poloviny okna, když odesílatel přesáhne kapacitu linky. Z tohoto důvodu TCP Vegas zavádí Slow-start s omezením, že dovoluje exponenciální růst okna každý *druhý* RTT. Mezitím se okno nemění. [9]

TCP Vegas také zavádí mezní hodnotu γ. Pokud je ve fázi Slow-startu aktuální propustnost *Actual* menší, než γ, tak se Slow-start ukončí a začne Congestion avoidance. Stejně jako hodnoty α a β (zmíněné výše) je i γ určena experimentálně.

2.1.3.2 TCP Vegas Fast retransmit

TCP Reno detekuje výpadek paketu, když odesílateli dojde n duplicitních ACK. Tato hodnota je standardně $n = 3$, jak je popsáno v [5]. TCP Vegas tuto detekci rozšiřuje o další dva principy:

- Když přijde odesílateli duplicitní ACK, porovná aktuální čas s časovou značkou, kterou paket získal při jeho vytvoření (resp. odeslání do sítě). Pokud je tento interval větší, než timeout pro dané spojení, TCP Vegas nečeká na n duplicitních ACK a hned pošle paket znovu.
- Pokud odesílateli dojde neduplicitní ACK – a to konkrétně první nebo druhý po znovu-poslání paketu – TCP Vegas se opět podívá na časové značky těchto paketů a zopakuje postup v bodě výše.

TCP Vegas stále obsahuje standardní detekci pro určení, kdy se má paket poslat znovu. Toto jsou pouze návrhy pro nestandardní rozšíření. [9]

2.2 Výběr algoritmů

Spolu s vedoucím práce jsem vybral několik vzorových algoritmů, pomocí nichž budu v kapitole Testování 4 demonstrovat problémy, které mohou v síti nastat. Pro výběr algoritmů jsme stanovili hned několik požadavků:

- Zahrnutí standardní implementace
- Zaměření na algoritmy, které obsluhuje odesílatel
- Zaměření na algoritmy, které detekují zahlcení sítě pomocí ztrátovosti paketů
- Vybrat kandidáty s různým chováním v pomalých a rychlých linkách

Tímto zúžením jsem vybral tři implementace, konkrétně:

- TCP Tahoe
- TCP Reno
- High Speed TCP

Dále jsem se po prozkoumání možností rozhodl implementovat dva hybridní algoritmy. První vychází z TCP Vegas a TCP Reno. Důvod pro inspiraci algoritmem TCP Vegas je zcela odlišný přístup k detekci zahlcení. Ta se nedělá pomocí ztrátovosti paketů, ale pomocí zpoždění, se kterým odesílateli přicházejí nazpět potvrzení. Blíže tento hybridní algoritmus popíši v patřičné kapitole 3.3.5. Druhým hybridním algoritmem je implementace využívající selektivního potvrzování paketů (SACK). Blíže je popsán v 3.3.4

2.3 Probíhající výzkumy

Na akademické půdě probíhá mnoho výzkumů, které přicházejí s novými nápady na vylepšení řešení problematiky zahlcení. Přestože řada výzkumů není ani zdaleka hotová, rád bych zde zmínil některé zajímavosti [13], které sami o sobě nutí k zamyšlení:

- **Nečinná spojení.** Pokud odesílatel dlouho negeneruje žádná nová data, která by příjemce potřeboval, spojení se stane nečinné. Je nutné si uvědomit, že v síti většinou nekomunikuje pouze jedna strana s příjemcem, nebo přes síťový uzel probíhá více komunikací. Pokud je ale náš odesílatel nečinný, ostatní komunikace ho ani nevidí a kvůli standardní implementaci Slow-startu 1.2.1 nepředpokládají, že by nějaké nové spojení nárazově zatížilo síť. Odesílatelovo spojení však není nové a může si pamatovat svoji předchozí velikost okna. Z toho by mohly nastat velké problémy pro chod sítě. Publikace [5] uvádí restartování okna. Výzkum se ale ubírá směrem postupného snižování okna každý RTO, když není odesílatel činný.

- **Počet duplicitních ACK.** Jak uvažují některé publikace (např. [9]), někdy je vhodné změnit svoji činnost při prvním a druhém duplicitním ACK. Jedna metoda s názvem "Limited transmit"navrhuje posílat nová data za každý z těchto prvních dvou ACK. Některé metody zase navrhují zmenšit počet potřebných duplicitních ACK pro spuštění Fast retransmit algoritmu 1.2.3, ovšem jen za určitých podmínek.

- **Rozšíření SACK.** Pokud je v síti implementovaný SACK, zvažují se nové mechanismy pro jeho hlubší rozšíření. Jedná se především o vylepšení přenosu, kdy síť masivně přeuspořádá pořadí paketů.

Kapitola 3

Realizace

V této kapitole rozeberu jednotlivé algoritmy, které jsem se rozhodl implementovat. Pochopení těchto algoritmů je důležité pro správné zpracování a interpretování výsledků v kapitole Testovaní 4. I z tohoto důvodu budu u algoritmů uvádět i jednotlivé vzorce, kterými se přepočítává velikost Congestion Window 1.1.2.

Aby bylo možné algoritmy testovat a porovnávat, musel jsem vyvinout testovací prostředí, které v této kapitole také dopodrobna popíši.

3.1 Výběr programovacího jazyka

Pro celkovou implementaci bylo nutné zvolit nejvhodnější programovací jazyk, který umožní vývoj testovacího prostředí jako vícevláknovou aplikaci s objektovým řešením problematiky. Kvůli těmto požadavkům a kvůli více zkušenostem s jazykem C++ jsem zvolil tuto možnost, přestože v úvahu připadaly i jiné jazyky, například Java nebo Python.

3.2 Simulace síťového prostředí

Pro celou úlohu bylo zapotřebí vyvinout testovací prostředí, které bude názorně simulovat komunikaci v síti. Aby bylo takové prostředí dobře použitelné, nutným požadavkem byla parametrizace jednotlivých problematik, které s sebou komunikace po síti nese. Mezi tyto parametry mimo jiné patří:

- Taktování linky
- Chybovost linky
- Zpoždění linky
- Velikost bufferů

- Počáteční RTO a timeout
- Velikost posílaných paketů
- Který Congestion avoidance algoritmus se má použít
- Velikost meze *ssthresh*

Tyto parametry hrají klíčovou roli při testování a srovnávání algoritmů. Díky nim je možné prostředí pozměnit a tím pozorovat, jak si určitý algoritmus s problémem poradí.

Mnou navržené testovací prostředí nezahrnuje přeuspořádání paketů v síti. Tato problematika není pro danou úlohu podstatná. I přes to ale musejí algoritmy počítat s možností, že přeuspořádání nastane, kvůli zpoždění reakce algoritmu Fast retransmit. Tato vlastnost musí být zachována a testovací prostředí s tím počítá.

3.2.1 Návrh tříd

Pro usnadnění tvorby kódu a pro lepší orientaci bylo nutné řešit problematiku objektově. K tomuto účelu jsem vytvořil strukturu *Packet*, která obsahuje jednoznačný identifikátor (ID), ACK, datovou velikost a čas vytvoření. Paket svojí strukturou tedy částečně odpovídá reálnému paketu spolu s jeho hlavičkou.

Pro lepší obsluhu Congestion Window u odesílatele a bufferů u obslužných vláken či u příjemce bylo nutné zavést dvě třídy:

- Window
- Buffer

Obě třídy jsou si velmi podobné a to jak svými položkami, tak svými metodami. Každá z těchto tříd v sobě obsahuje zapouzdřenou frontu, která simuluje dané okno nebo buffer. Do této fronty se vkládají jednotlivé pakety – instance třídy *Packet*. Třída *Window* však obsahuje více způsobů, jak se svojí frontou může pracovat.

3.2.2 Odesílatel

Odesílatel je implementován jako samostatné vlákno, které vytváří nové pakety a vkládá je do svého okna. Toto okno odpovídá Congestion Window 1.1.2. Přestože má odesílatel ve standardu TCP pouze jedno okno, pro snazší implementaci a obsluhu jednotlivých částí jsem zavedl oken více. Toto usnadnění nemá žádný vliv na výslednou funkčnost oproti standardní implementaci. Jednotlivá okna, která odesílatel obsahuje, jsou:

- Congestion Window – odesílací okno

- Okno pro přijímání potvrzení
- Okno s dosud nepotvrzenými pakety

Congestion Window slouží jako odesílací okno, kam odesílatel vkládá nově vytvořené pakety. To dělá pouze tehdy, když je pro nový paket místo. Tato funkčnost je zajištěna udržovanými proměnnými ve třídě *Window*. Místo pro nový paket vznikne zpravidla tehdy, když odesílateli dojde potvrzení ACK od příjemce, nebo když se navýšilo okno. Tyto příchozí potvrzení se vkládají do okna pro přijímání potvrzení. Odesílatel pravidelně kontroluje, zda v tomto oknu nepřibyl nový paket a při této události ho vyjme a zpracuje. Posledním oknem je fronta se záznamy všech odeslaných, ale dosud nepotvrzených paketů.

3.2.2.1 Generování nových paketů

Odesílatel vytvoří nový paket vždy, když se uvolní místo v Congestion Window. Buď byl paket potvrzen, nebo bylo provedeno zvětšení okna. Odesílatel si pamatuje poslední vytvořené sekvenční číslo paketu, které s každým novým paketem navyšuje. Tento proces docílí jednoznačnost identifikátorů v hlavičkách paketů. K novému paketu se také přidá časové razítko, které odpovídá času vytvoření. Velikost paketu se v testovacím prostředí zadává intervalem, z kterého se vybere náhodné číslo. Pokud je dolní i horní mez intervalu stejná, velikost paketů je konstantní.

3.2.2.2 Stavy odesílatele

Z praktického hlediska existují dva stavy, ve kterém se odesílatel nachází:

- Slow-start
- Congestion

Ve stavu Slow-start odesílatel předpokládá, že síť není zahlcená a provádí Slow-start algoritmus 1.2.1. Opuštění tohoto stavu zapříčiní ztráta paketu, nebo překročení meze *ssthresh*, definované v [5]. Tato mez slouží jako maximum, pro které je možné provádět exponenciální růst okna při Slow-startu (tedy bez náznaku zahlcení). Obyčejně je tato mez velké číslo, protože není žádoucí limitovat propustnost vnějším vlivem. Přestože je počáteční hodnota *ssthresh* dána parametrem, nejedná se o konstantu. Naopak, pro správnou manipulaci s velikostí okna je velmi důležité tuto hodnotu při zahlcení měnit. Způsob těchto změn závisí na konkrétním Congestion avoidance algoritmu.

Odesílatel obsahuje všechny čtyři algoritmy popsané v 1.2, nicméně pro Fast retransmit a Fast recovery není potřeba rozpoznávat stav odesílatele. To je dáno tím, že oba tyto algoritmy pracují samostatně. Druhý zmiňovaný navíc funguje pouze tehdy, kdy už došlo k zahlcení a odesílatel je ve fázi Congestion avoidance.

3.2.2.3 Členění příjmu potvrzení

Odesílatel svoji činnost volí nejen podle toho, v jakém je stavu, ale také podle hlavičky paketu, který mu přišel jako potvrzení. Toto členění by se dalo rozdělit do těchto částí:

- ID doraženého potvrzení je shodné s jeho číslem ACK
  - Nejedná se tedy o duplicitní ACK a potvrzuje se ten samý paket
- ID doraženého potvrzení není shodné s jeho číslem ACK
  - Jedná se o první ACK, který se neshoduje s ID, nebo se jedná o duplicitní ACK

První bod znamená, že je komunikace v pořádku a nedochází ke ztrátovosti paketů. Jedná se o ideální stav. Druhý bod znamená, že se sítí něco není v pořádku. Jak jsem již popsal v 1.2.3, nemusí se nutně jednat o ztrátovost paketu, ale mohlo dojít k přeuspořádání paketů v síti. Přestože přeuspořádání paketů v testovacím prostředí nemám implementované, prodleva zjištění ztráty paketu musí být zachována. Pokud se jedná o první paket, který takto není v pořádku, odesílatel začne každý další ACK počítat jako duplicitní, dokud opět nenastane situace popsaná v první bodě. Tento mechanismus zaručí správnou aktivaci algoritmu Fast retransmit.

3.2.2.4 Fast retransmit a Fast recovery

Jak již bylo zmíněno, oba algoritmy spadají do fáze, kdy došlo ke ztrátě paketu a je potřeba ji řešit, přičemž algoritmem Fast retransmit tato fáze začíná. Ten počítá duplicitní ACK a když je jejich počet $n = 3$ (dle standardu [5]), tak je paket považován za ztracený a pošle se znovu. Pokud byl odesílatel ve stavu Slow-start, tak tento stav opustí a musí zahájit Congestion avoidance.

Příchozí (duplicitní) potvrzení s sebou nesou více informací, než že určitý paket vypadl. Tou je především ID paketu, ke kterému je konkrétní číslo ACK přiřazeno. Díky této informaci odesílatel ví, že daný paket opustil síť a nespotřebovává již tedy její prostředky. Navíc má jistotu, že příjemce daný paket už přijal a není potřeba se jím už nadále zabývat. Lze tedy poslat nový paket do sítě bez obavy, že by došlo k zahlcení, nebo že by se zahlcení prohloubilo. Z tohoto důvodu Fast recovery při znovu-poslání paketu nezahajuje Slow-start, ale pokračuje v Congestion avoidance.

Vzhledem k tomu, že Fast retransmit pošle znovu paket na základě třech duplicitních ACK, může nastat ještě jedna nepříjemná situace, kterou musí odesílatel efektivně řešit. Může totiž nastat problém, kdy vypadne paket, který již dříve byl poslán znovu (tedy ten samý paket vypadne dvakrát po sobě). V tomto případě by Fast recovery trval věčně, protože ten skončí jen za situace, kdy odesílateli dojde potvrzení znovu-poslaného paketu. Z tohoto

důvodu je pro tento paket potřeba nastavit časovač [3], který bude pravidelně kontrolovat, zda paketu netrvá cesta moc dlouho. V dokumentaci se nepíší přesné hodnoty, kterými se tento časovač má řídit. Je pouze uvedeno, že tato hodnota se s každým znovu-posláním zvětšuje. V mém testovacím prostředí jsem stanovil experimentální hodnoty časovače z počátku na $RTO+3$ vteřiny. S každým dalším znovu-posláním počká odesílatel dalších 5 vteřin navíc.

3.2.2.5 Výpočet RTO a detekce timeoutu

Pro správnou funkčnost odesílatele je nutné vypočítávat Retransmission timeout (RTO) [14], který se používá jako časovač pro určité spojení. Tento časový interval reaguje na dobu, kdy přijde odesílateli nazpět jakákoliv odpověď [2].

Než proběhne první měření Round-trip time (RTT), měl by být RTO nastaven na jednu vteřinu [2]. Některé starší publikace (např. [15]) navrhují počáteční RTO s velikostí tří vteřin. Důvod, proč je dnes standardně RTO menší, je celkové zrychlení sítí, se kterým dřívější RFC nepočítaly.

Aby mohl být RTO vypočítán, musejí se zavést dvě pomocné proměnné [2]:

- SRTT ("Smoothed Round-trip time")
- RTTVAR ("Round-trip time variation")

Při prvním výpočtu RTO (tedy po prvním zjištění RTT) probíhá výpočet následovně:

$$SRTT = RTT$$

$$RTTVAR = RTT/2$$

$$RTO = SRTT + max(G, K * RTTVAR)$$

kde $G \leq 100$ je doporučená granularita, $K = 4$ je doporučený koeficient a funkce $max()$ vybere větší číslo ze svých parametrů. V mém testovacím prostředí 3.2 používám $G = 1000$ pro lepší výsledky měření. Publikace [2] tuto možnost povoluje.

Při každém dalším naměřeném RTT probíhá výpočet následovně:

$$RTTVAR = (1 - \beta) * RTTVAR + \beta * |SRTT - RTT|$$

$$SRTT = (1 - \alpha) * SRTT + \alpha * RTT$$

kde standardně $\alpha = 1/8$ a $\beta = 1/4$. Vzhledem k tomu, že druhá rovnice závisí na první, musí být dodrženo pořadí výpočtu. Poté, co jsou hodnoty $RTTVAR$ a $SRTT$ vypočítány, opět se vypočte:

$$RTO = SRTT + max(G, K * RTTVAR)$$

a pokud platí $RTO \geq 1\ sec$, tak je nový výpočet RTO hotov. Jinak se nastaví $RTO = 1\ sec$. Některé implementace zavádějí i horní mez $RTO \leq 60\ sec$ [2], ale standardně není vyžadována.

Z výpočtů se dá jednoduše vypozorovat, že přepočítávání RTO má paměť a někdy může být potřeba hodnoty $SRTT$ a $RTTVAR$ uvést do původního stavu. Toto je obzvlášť žádoucí, když nastane několik timeoutů za sebou [2]. Tato funkčnost není součástí standardu a nemám ji v testovacím prostředí implementovanou.

V [2] je také uvedeno, že TCP musí používat Karnův algoritmus [16]. V mém testovacím prostředí tomu tak není, protože používám časová razítka pro každý paket. Při správných aktualizacích časových razítek během znovu-posílání paketu není nutné používat Karnův algoritmus.

Tradičně se RTO vypočítává jednou za RTT. Pokud ale TCP používá časová razítka, potom každý ACK může být použit jako nové měření [2]. Publikace [17] navrhuje, že při velkých Congestion Window je vhodné měřit RTO několikrát během jednoho RTT. Měření RTO s každým příchozím ACK nevede k lepším výsledkům, nicméně výsledky nezhoršuje. Může se ovšem stát, že díky velké četnosti měření RTO se bude držet neadekvátní historie měření a může být vhodné ji vymazat [2]. V mém testovacím prostředí se RTO počítá s každým příchozím ACK.

Pokud příjemce z nějakého důvodu nereaguje a odesílateli vyprší časovač odpovídající hodnotě RTO, je nutné ohlásit timeout a poslat paket znovu. Následuje takzvaná "back-off"strategie, která multiplikativně zvětší RTO a tím dá větší prostor pro paket, aby stihl dorazit. Standardně se RTO zdvojnásobí, tedy:

$$RTO = RTO * 2.$$

Odesílatel musí přijímat potvrzení od příjemce a zároveň musí kontrolovat, zda nedošlo k timeoutu. V mém testovacím prostředí pravidelně kontroluji, zda rozdíl aktuálního času a časovače, který se při každém příchozím ACK obnoví, není větší než vypočtený RTO. Pokud se podmínka splní, nastal timeout a je potřeba poslat první doposud nepotvrzený paket. Podobný princip se používá, pokud vypadl již znovu-poslaný paket. V tomto případě se ale dělá rozdíl aktuálního času s časovým razítkem daného paketu.

3.2.3 Příjemce

Podobně jako odesílatel, i příjemce je implementován jako samostatné vlákno. Žádné pakety ale nevytváří, pouze je přijímá a patřičně upravuje. Příjemce tedy nepřidává žádná data do sítě. Změna paketu spočívá ve vyplnění potvrzení ACK v jeho hlavičce.

V TCP je příjemce standardně omezený velikostí svého okna. Je to omezující paměť, se kterou může pracovat. Tomuto oknu se však neříká Congestion Window, ale *Receiver Window*, tedy okno příjemce [5]. V mém prostředí toto

omezení nemám implementované, protože stejného výsledku při testování se dá docílit změnou velikosti bufferů na lince.

3.2.3.1 Princip přijímání paketu

Příjemce kontroluje stav, kdy mu do jeho okna přijde nový paket. Na tuto událost zareaguje porovnáním ID paketu se sekvenčním číslem, které očekává. Toto číslo vždy doplní do položky ACK v hlavičce paketu. Příjemce si vede záznamy o paketech, které mu ještě nedorazily. Tato funkce je lehce upravená činnost Receiver Window standardního TCP, která nemá na výsledek žádný vliv. Doraziví pakety se dají kategorizovat takto:

- ID příchozího paketu se shoduje se sekvenčním číslem. Je tedy v pořádku a jeho ACK se vyplní číslem ID
- Příchozí paket má větší ID, než je sekvenční číslo. Vznikne duplicitní ACK, který se vyplní sekvenčním číslem, které příjemce očekával
- Příchozí paket má menší ID, než je sekvenční číslo. Příjemce tedy již paket dříve obdržel a u odesílatele došlo pravděpodobně k předčasnému timeoutu. Do ACK paketu se vyplní sekvenční číslo, které příjemce očekával

Pokud příjemce již na nějaký ztracený paket čekal, postup popsaný výše je stejný. Akorát se opakovaně vyplňuje ID chybějícího paketu jako duplicitní ACK, namísto sekvenčního čísla. Chybějící paket je totiž stále očekávaný. To se děje do té doby, než příjemce obdrží požadovaný paket [5].

Příjemce by měl okamžitě poslat duplicitní ACK, když mu přijde jiný paket, než očekává [5]. Odesílatel by se měl o možném problému dozvědět co nejdříve kvůli správné činnosti algoritmu Fast retransmit 1.2.3. V mém testovacím prostředí příjemce vždy neprodleně pošle duplicitní ACK.

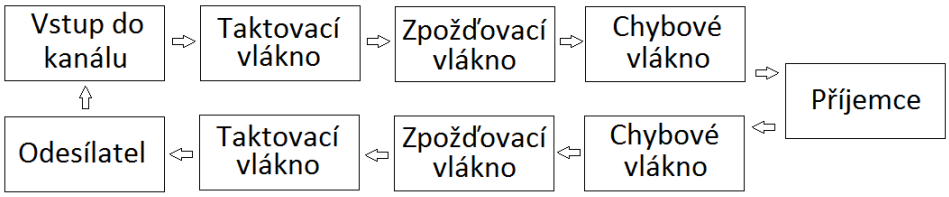
Publikace [5] také zmiňuje, že příjemce může potvrzovat jeden paket (či segment) větším počtem potvrzení [1], nebo naopak více paketů jedním potvrzením. V testovacím prostředí neprovádím segmentaci datového toku, tedy nikdy nevznikne situace, kdy by se pomocí více ACK potvrdil jeden paket. Opačná situace ale může snadno nastat a to například při ztrátovosti potvrzení na cestě zpět k odesílateli.

3.2.4 Obslužná vlákna sítě

Komunikace mezi odesílatelem a příjemcem je zprostředkovaná pomocí obslužných vláken, která simulují síť. Jedná se o sadu vláken, přes jejichž buffery se předávají pakety. Každé vlákno nějakým způsobem modifikuje datový tok a využívá parametrů nastavených uživatelem při startu testovacího prostředí.

[1] Tento princip se někdy nazývá "ACK Division"a je blíže popsán v [18].

Každé z těchto vláken je implementováno dvakrát, kromě vlákna vstupního 3.2.4.1. Důvod pro dvojí implementaci je komunikace oběma směry – jak od odesílatele k příjemci (nová data), tak od příjemce k odesílateli (potvrzení). Testovací prostředí tedy umí modifikovat oba směry komunikace nezávisle.



Obrázek 3.1: Propojení vláken mezi odesílatelem a příjemcem

3.2.4.1 Vstup dat do kanálu

Poté, co odesílatel naplní své okno novými daty, převezme toto vlákno kontrolu nad přenosem. Všechna tato data najednou pošle ke zpracování k dalšímu vláknu. Funguje tedy jako takové "úzké hrdlo", které omezí nárazovou vlnu paketů, kterou je schopný pomyslný síťový uzel přijmout. Toto omezení je dáno velikostí bufferu a pakety, které se do něj nevejdou, se zahodí. Tato ztrátovost je příznak zahlcení linky. Velikost bufferů (i vstupního) je dána parametrem a je pro všechna obslužná vlákna stejná.

3.2.4.2 Taktovací vlákno

Síťové uzly zpracovávají příchozí pakety v nějakém taktu. Tento takt odpovídá rychlosti daného síťového prvku a je dán parametrem. Dalším faktorem, který ovlivňuje takt přeposílání dat, je velikost konkrétního paketu. Ze základu jsem tuto hodnotu nastavil jako desetinu milisekundy za každý poslaný bajt. Tato hodnota se v testovacím prostředí může změnit.

3.2.4.3 Zpožďovací vlákno

Každá reálná síť má určité zpoždění a jeho simulaci je zde také velmi nutné implementovat. Zpožďovací vlákno znázorňuje zatížení uzlu, které se v průběhu komunikace může měnit. To odpovídá reálné situaci, kdy s uzlem začne komunikovat další spojení.

Aby bylo možné zpoždění implementovat, musí se ke každému paketu při vstupu do vlákna přidat časové razítko. Ve stejném taktu, jako pracuje Taktovací vlákno 3.2.4.2, se vždy zkontroluje, jestli je již možné paket poslat dál. To se docílí porovnáním časového razítka a aktuálního času.

O kolik se má paket zpozdit, je též dáno parametrem. Stejně tak je implementovaná funkce, která v průběhu komunikace zpoždění může měnit. Ta

funguje na principu, kdy n paketů má konstantní zpoždění určené parametrem, a dalších n paketů má zpoždění několikanásobné a postup se opakuje. Tento násobek je také dán předem při spuštění prostředí pomocí příslušného parametru.

3.2.4.4 Chybové vlákno

Komunikace po síti může probíhat přes různá spojení a ta nemusí být stejně spolehlivá. Přenos vzduchem (např. WiFi) bude mít skoro jistě větší chybovost, než optické kabely. Tento princip zajišťuje Chybové vlákno. Využívá funkci, která zavádí dva způsoby chybovosti:

- Modulární
- Blokovou

Modulární chybovost spočívá v zahození každého n-tého paketu, kdežto bloková chybovost je velmi podobná funkci zpoždění popsané v 3.2.4.3 – prvních n paketů se přepošle bez výpadku, dalších n se zahodí a postup se opakuje. Vzhledem k tomu, že je někdy pro testování vhodné chybovost zahájit až později (kvůli dostatečnému rozběhnutí komunikace), dají se oba typy chybovosti zpozdit až do nějakého množství odeslaných paketů.

Stejně jako předchozí vlákna, i zde jsou všechny vstupní hodnoty parametrizovány při spuštění testovacího prostředí.

3.2.5 Výstup testovacího prostředí

Testovací prostředí kromě interaktivních výpisů průběhu komunikace také ukládá výsledky do souborů. Z těchto souborů lze poté libovolným programem vygenerovat grafy. Vhodným nástrojem je například *Gnuplot* [19], který jsem použil na vygenerování grafů v kapitole Testování 4.

Zápis do souborů probíhá na straně příjemce, tedy vždy, když mu přijde jakýkoliv paket. Každý tento záznam je na samostatném řádku a jeho položky jsou oddělené mezerami. Položky sloupců odpovídají těmto záznamům:

1. Čas záznamu
2. ID paketu
3. ACK paketu
4. Čas vytvoření paketu
5. Aktuální velikost Congestion Window
6. Aktuální *ssthresh*
7. Aktuální RTO

Tyto položky postačují pro zjištění všech potřebných informací ze síťové komunikace. V jednotlivých testech v 4 budu srovnávat první a pátý sloupec, protože poskytují srovnání časové osy s aktuální velikostí okna. Toto srovnání je pro tuto práci nejdůležitější kvůli souvislosti se zahlcením sítě, popsané hned na úvod v kapitole 1.

3.3 Implementace algoritmů

V této sekci popíši jednotlivé algoritmy, které jsem se rozhodl vypracovat a otestovat. Jak jsem již zmínil v 2.2, výběr těchto algoritmů jsem učinil spolu s vedoucím práce na základě předem stanovených požadavků.

3.3.1 TCP Tahoe

TCP Tahoe patří mezi nejstarší Congestion avoidance algoritmy. Když byl tento algoritmus vyvinut, ještě nebyl vynalezen princip Fast recovery a z tohoto důvodu Tahoe redukuje velikost okna daleko přísněji, než TCP Reno 3.3.2. Při detekci ztráty paketu totiž bezpodmínečně nastaví velikost okna na jeden paket a hodnotu *ssthresh* redukuje na polovinu. Ztracený paket se pošle znovu a zahájí se Slow-start nehledě na to, co bylo příčinou výpadku paketu. Tahoe nerozlišuje mezi událostí timeoutu nebo detekcí ztráty paketu pomocí třech duplicitních ACK [20].

3.3.2 TCP Reno

Když publikace hovoří o standardní implementaci (např. [7], [5] nebo [21]), často mají na mysli právě algoritmus TCP Reno.

Narozdíl od Tahoe, když algoritmus zaznamená výpadek paketu pomocí třech duplicitních ACK, daný paket se pošle znovu a velikost Congestion Window se zmenší na polovinu. Hodnota *ssthresh* se nastaví stejně, jako velikost okna. Poté se zahájí Fast recovery, který odpovídá Congestion avoidance, protože příchozí potvrzení s sebou nesou více informací, než že paket vypadl (viz 3.2.2.4). Tato fáze trvá do té doby, než dorazí neduplicitní ACK [5].

Při timeoutu se TCP Reno chová stejně, jako TCP Tahoe – tedy nastaví velikost Congestion Window na jeden paket, hodnotu *ssthresh* redukuje na polovinu a zahájí Slow-start. Tento timeout je reakcí na neodpovídání sítě (jakéhokoliv ACK). Nejedná se o časovač ke konkrétnímu paketu, ale k jakékoliv odpovědi [2].

3.3.3 High Speed TCP

Standardní TCP algoritmy ztrácejí na efektivnosti, když jsou nasazeny na sítě s velkou šířkou pásma a s velkým RTT. Tyto algoritmy potom nemohou využít

celou kapacitu sítě. High Speed TCP (HSTCP) navrhuje změnu v přepočtech velikosti okna ve spojení s velkými Congestion Window [11].

HSTCP abstrahuje činnost TCP Reno v Congestion avoidance fázi následujícím způsobem:

- Pokud má dojít ke zvětšení okna, toto navýšení odpovídá hodnotě:

$$\frac{\alpha(w)}{w}$$

- Pokud má dojít ke zmenšení okna, toto zmenšení odpovídá hodnotě:

$$(1 - \beta(w)) * w$$

kde w je aktuální velikost Congestion Window, $\alpha(w)$ a $\beta(w)$ jsou míry pro zvětšení a zmenšení okna. Je nutné si uvědomit, že pokud se položí $\alpha(w) = 1$ a $\beta(w) = \frac{1}{2}$, předpisy odpovídají přesně chování TCP Reno. Díky tomu lze algoritmus použít jak na klasických linkách, tak na linkách s obrovskými objemy dat.

HSTCP používá princip, který mění velikost hodnot $\alpha(w)$ a $\beta(w)$ v závislosti na velikosti aktuálního Congestion Window. Předpokládá totiž, že pokud okno přesáhlo určité mezní hodnoty velikosti okna, jedná se o vysokokapacitní síť a může si dovolit větší přírůstky okna pomocí zvětšení $\alpha(w)$. Naopak, pokud dojde k zahlcení, HSTCP umí rychleji zareagovat větším snížením okna díky menší hodnotě $\beta(w)$.

3.3.3.1 Algoritmus pro výpočet mezních hodnot

Publikace [11] zavádí rozsáhlá pozorování a odůvodnění, jak určovat hranice pro velikost okna w a hodnoty $\alpha(w)$ a $\beta(w)$. Z počátku je nutné připomenout, že vysokorychlostní algoritmy obecně spoléhají na malou ztrátovost linky (viz 2.1.2). Pokud by byla ztrátovost velká, Slow-start ani Congestion avoidance nikdy nedosáhnou oken velkých rozměrů. Vtah mezi ztrátovostí, průměrnou velikostí okna a RTT mezi ztrátami v HSTCP jsem uvedl v tabulce 3.1.

Podobně jako [11], v testovacím prostředí jsem se také zaměřil na maximální ztrátovost 10^{-3}, tedy druhý řádek v tabulce 3.1. Pro správnou ilustraci vzorců do algoritmu zjišťování hranic pro změny hodnot $\alpha(w)$ a $\beta(w)$ je nutné zavést několik pomocných údajů z téže publikace:

- $Low\_Window = 38$ a odpovídá nejnižší (první) hranici, kdy se zvětší $\alpha(w)$ a zmenší $\beta(w)$. Tato hodnota koresponduje se ztrátovostí 10^{-3} z tabulky 3.1

- $High\_Window = 83000$ a odpovídá horní hranici, kterou chceme dosáhnout při nastaveném $High\_Decrease$

- $High\_Decrease = 10^{-1}$ a odpovídá zmenšovacímu parametru pro $\beta(w)$ kde $w = High\_Window$. Tedy v tomto případě $\beta(83000) = 10^{-1}$

Výpočet patřičných mezních hodnot pro velikost okna w vypadá následovně:

$$\beta(w) = \frac{(High\_Decrease - 0,5) * (\log_{10}(w) - \log_{10}(Low\_Window))}{\log_{10}(High\_Window) - \log_{10}(Low\_Window)} + 0,5$$

a

$$\alpha(w) = \frac{2 * w^2 * \beta(w)}{(2 - \beta(w)) * w^{1,2} * 12,8}$$

Pomocí těchto rovnic z [11] lze jednoduše vypočítat hodnoty $\alpha(w)$ a $\beta(w)$. Několik jich uvádím v tabulce 3.2.

Tabulka 3.1: Vztah ztrátovosti, průměrné velikosti okna a RTT u HSTCP [11]

Ztrátovost paketů	Průměrná velikost okna	RTT mezi ztrátami
10^{-2}	12	8
10^{-3}	38	25
10^{-4}	263	38
10^{-5}	1795	57
10^{-6}	12279	83
10^{-7}	83981	123
10^{-8}	574356	180
10^{-9}	3928088	264
10^{-10}	26864653	388

Tabulka 3.2: Ukázka hodnot pro $\alpha(w)$ a $\beta(w)$ u HSTCP [11]

w	$\alpha(w)$	$\beta(w)$
38	1	0,50
118	2	0,44
221	3	0,41
347	4	0,38
495	5	0,37
663	6	0,35
851	7	0,34
1058	8	0,33
1284	9	0,32
...	...	...

Z tabulky 3.2 je vidět, že s přibývající velikostí okna w se hodnota $\alpha(w)$ zvětšuje a hodnota $\beta(w)$ zmenšuje. To je žádoucí kvůli požadavkům zvětšovacího výpočtu $\frac{\alpha(w)}{w}$ a zmenšovacího výpočtu $(1 - \beta(w)) * w$.

Algoritmus tyto hodnoty počítá opakovaným aplikováním vzorců popsaných výše. Pokud známe hodnoty jako v tabulce 3.2, je již jednoduché patřičně měnit velikost Congestion Window a HSTCP je kompletní.

Hodnoty, které uvádí publikace [11] a které zde také zmiňuji, jsou pouze experimentální a pravděpodobně se nejedná o nejlepší řešení, jak dělat HSTCP algoritmus. V tomto odvětví je zapotřebí hlubší výzkum problematiky.

Algoritmus Slow-start a událost timeoutu není nijak změněná oproti TCP Reno, přestože publikace [11] modifikaci Slow-startu jako nestandardní implementaci nabízí. Spočívá v omezení maximálního zvětšení okna, o které se může Congestion Window navýšit během jednoho RTT. Bližší popis limitovaného Slow-startu je popsán v publikaci [22], ale v mém testovacím prostředí ho nemám naimplementovaný a blíže se jím nezabývám.

3.3.4 Hybridní SACK

Rád bych představil algoritmus, který využívá zcela jiný princip detekce ztráty paketů. Nazval jsem ho Hybridní SACK ("Selective Acknowledgement"), protože využívá selektivního potvrzování paketů, popsaného v [23].

Ostatní algoritmy nejsou příliš efektivní, pokud je v síti velká ztrátovost paketů a komunikace s sebou nese nedostatek informací kvůli malému počtu potvrzení. Odesílatel se totiž o výpadku paketu dozví s prodlevou jednoho RTT. SACK tento problém překonává, protože každé potvrzení selektivně potvrzuje pouze jeden konkrétní paket, ke kterému patří.

Mnou navržený algoritmus je hybridní, protože přímo nerespektuje návrh uvedený v [23], ale sdílí stejnou myšlenku.

3.3.4.1 Detekce zahlcení

Jak již bylo popsáno, každý paket obsahuje časové razítko, kdy byl paket vytvořen. Hybridní SACK nastaví časovač pro každý tento paket a pokud čas pro doručení toho paketu překročí daný interval (odpovídá vypočtenému RTO 3.2.2.5), dojde k timeoutu a paket se neprodleně pošle znovu. Je to jediný způsob, kterým Hybridní SACK detekuje výpadek paketu. To s sebou může nést řadu výhod a nevýhod, včetně potenciální pozdní reakce na výpadek. Výhody i nevýhody demonstruji v kapitole Testování 4.

3.3.4.2 Princip funkčnosti

Stejně jako ostatní algoritmy, i Hybridní SACK používá Slow-start se stejnými pravidly. Pokud dojde ke ztrátě paketu (dojde k timeoutu pro konkrétní paket), algoritmus nastaví velikost okna na jeden paket a hodnotu *ssthresh* redukuje na polovinu. Ztracený paket pošle znovu a zahájí Slow-start. Pokud velikost Congestion Window překročí hranici *ssthresh*, zahájí se Congestion avoidance s lineárním zvětšováním okna.

Algoritmus tedy nerespektuje členění popsané v 3.2.2.3 a nevyužívá algoritmů Fast retransmit a Fast recovery.

Když odesílateli dojde nové potvrzení, Hybridní SACK projde všechny doposud nepotvrzené pakety. Pokud se ID nějakého paketu shoduje s ID příchozího ACK, paket se potvrdí a patřičně se uvolní místo v okně pro poslání nového paketu.

3.3.5 Hybridní Vegas

Poslední algoritmus, který jsem implementoval do testovacího prostředí, je lehce upravený TCP Vegas spolu s kombinací TCP Reno. Klade si za úkol detekovat zahlcení jinak, než pomocí ztrátovosti paketů, ale přitom chce zachovat stejnou jednoduchost, kterou nabízí TCP Reno.

3.3.5.1 Detekce zahlcení

Na rozdíl od ostatních implementací, Hybridní Vegas detekuje zahlcení pomocí měnících se zpoždění v síti. Množství posílaných dat do sítě totiž přímo souvisí s dobou, za kterou se tato data stihnou poslat. Pokud se však tato očekávaná doba změní o příliš velkou hodnotu, je to příznak toho, že je síť zahlcená, nebo se v ní naopak uvolnily prostředky. V obou případech je potřeba měnit velikost Congestion Window.

3.3.5.2 Princip funkčnosti

TCP Vegas detekuje zahlcení stejným způsobem popsaným výše. Rozdíl oproti Hybridnímu Vegas je v jiných výpočtech propustnosti sítě. Výpočty algoritmu TCP Vegas jsem již uvedl v 2.1.3.

Nehledě na to, jestli nějaký paket vypadl nebo ne, Hybridní Vegas sám rozhodne, zda se bude okno zvětšovat nebo zmenšovat. Přestože algoritmus respektuje členění odesílatele popsané v 3.2.2.3, sám vyhodnocuje závažnost situace a podle jednotlivých fází nehodnotí zahlcení.

Aby mohl algoritmus správně fungovat, potřebuje zjistit nejnižší RTT, který v síti nastal, nazvaný $baseRTT$. Pojmenovaný je tedy stejně, jako v TCP Vegas [9]. Tuto hodnotu aktualizuje s každým příchozím potvrzením, pokud se naskytl menší RTT. Výpočet odpovídá rozdílu časového razítka paketu s aktuálním časem. Hodnota $baseRTT$ se tedy typicky rovná prvnímu RTT, kdy síť ještě není zahlcená a nejsou vložené žádné pakety v bufferech pro toto spojení.

Když Hybridní Vegas opustí Slow-start (což je ze stejných důvodů, jako tomu je u TCP Reno), přejde do režimu, kdy se každý RTT provede kontrola odezvy. Aby byla tato kontrola možná, algoritmus uloží čas, kdy začalo nové měření RTT s názvem $StartRTT$. Dále uloží aktuální velikost okna, která od-

povídá počtu paketů, které se odešlou v nadcházejícím měření [2]. Tuto hodnotu nazvěme *Actual*. Doba RTT se jednoduše vypočítá odečtením času kontroly s počátečním časem $StartRTT$.

Dále tento algoritmus potřebuje spočítat očekávanou propustnost, tedy jak dlouho trval RTT v poměru s nejlepším časem:

$$Expected = RTT/baseRTT$$

kde *Expected* je požadovaný poměr. Tato hodnota odpovídá počtu paketů, které by byly odeslány, kdyby byla síť v nejlepší kondici. Teď už je jen nutné spočítat rozdíl těchto hodnot

$$Diff = Actual - Expected$$

a vypozorovat, co toto číslo znamená.

Je potřeba si uvědomit, že $Diff$ oproti implementaci TCP Vegas může nabývat kladných i záporných hodnot. Snahou Hybridního Vegas je držet $Diff = 0$, pokud je to možné. To totiž znamená, že síť je v očekávané kondici a přestože odesílatel posílá více dat do sítě, nemá to na zpětnou odezvu žádný vliv. Pokud je $Diff < 0$, síť si vede hůř, než se očekávalo. Je možné, že do sítě začalo zasahovat nějaké jiné spojení a zvětšilo se zpoždění na síťových uzlech. Naopak, pokud $Diff > 0$, v síti se zpoždění pravděpodobně zmenšilo a síť si vede lépe. Hodnota RTT se totiž nezvětšila o tolik, jako se zvětšil počet odeslaných dat, tedy hodnota *Actual*.

Podobně jako TCP Vegas, i zde algoritmus porovnává hodnotu $Diff$ s hranicemi α a β. Pokud je $Diff > \beta$, jedná se o výtečný stav sítě a okno se může navýšit o jeden paket. Naopak, pokud $Diff < \alpha$, okno se musí zmenšit o jeden paket. Cílem je držet $Diff$ mezi hodnotami α a β, přičemž v mé implementaci jsou tyto hodnoty též nastaveny experimentálně, a to $\alpha = -5$ a $\beta = 5$.

Algoritmus používá všechna standardní členění, tedy není žádným způsobem upravený Slow-start, Fast retransmit, Fast recovery nebo detekce timeoutu oproti TCP Reno. Přestože publikace [9] některé změny doporučuje, nechtěl jsem v tomto testovacím prostředí zavádět jiné alternativní postupy, abych nezkreslil funkčnost Congestion avoidance algoritmu jako takového.

[2] Velikost okna je zde uvedena v počtu paketů, ne v počtu bajtů.

Kapitola 4

Testování

V této kapitole provedu sérii jednoduchých testů, na kterých demonstruji výhody a nevýhody jednotlivých algoritmů popsaných v 3.3. Testování umožňuje implementované testovací prostředí 3.2, které díky své parametrizaci může upravovat podmínky komunikace a tím měnit vlastnosti sítě.

Aby byly následující testy čitelné a grafy se neslévaly, nastavil jsem velikost posílaných paketů na 1 KB. Tato velikost navýší takt síťových uzlů na dostatečnou hodnotu, aby mohly být změny v síti dobře zaznamenávány. Takt linky je v základu nastaven na 10 milisekund.

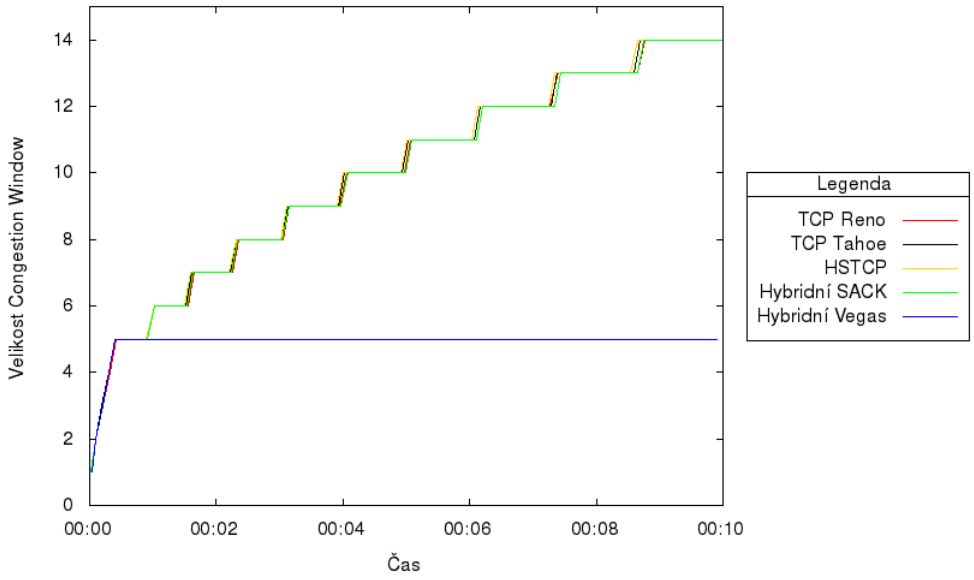
4.1 Přenos dat

4.1.1 Slow-start a Congestion Avoidance

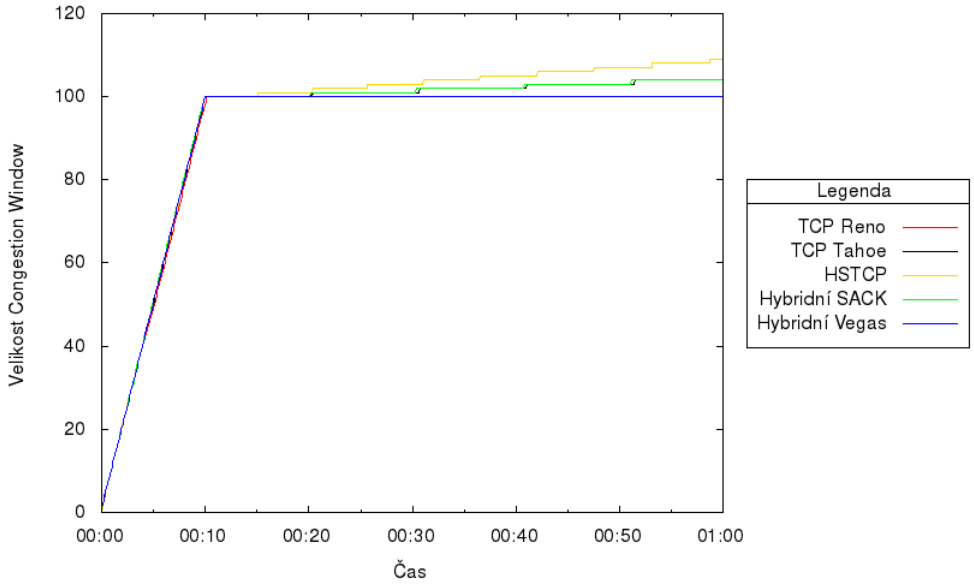
Na úvod bych rád ilustroval průběh komunikace, kdy algoritmus dosáhne mezní hodnoty $ssthresh = 5$ pro Slow-start a následně se přepne na Congestion avoidance. Poté je vidět lineární navyšování okna, které v grafu 4.1 vytváří schody začínající v hodnotě 5 – tedy při přepnutí na Congestion avoidance.

Z grafu 4.1 je také patrné, že průběh Congestion avoidance algoritmů je téměř totožný (odchylky vytváří pouze nepravidelné střídání vláken v prostředí). Jenom Hybridní Vegas drží Congestion Window na velikosti 5. Tato vlastnost je žádoucí, protože podmínky v síti se nemění. Neaplikuje se totiž žádné zpoždění ani chybovost linky.

V reálné komunikace je *ssthresh* velké číslo, protože komunikaci není dobré zvnějšku takto omezovat. Následující graf 4.2 již ilustruje více posílaných dat. Průběh TCP Reno, Tahoe, Hybridní SACK i HSTCP je opět takřka totožný. Je ale patrné rychlejší zvětšování okna u HSTCP, protože okno již přesáhlo velikost 38 a aplikuje se jiný poměr přírůstku okna každý RTT, viz 3.2. Tento rozdíl by byl v delší komunikaci čím dál víc znát, jak ukazuje graf 4.3.



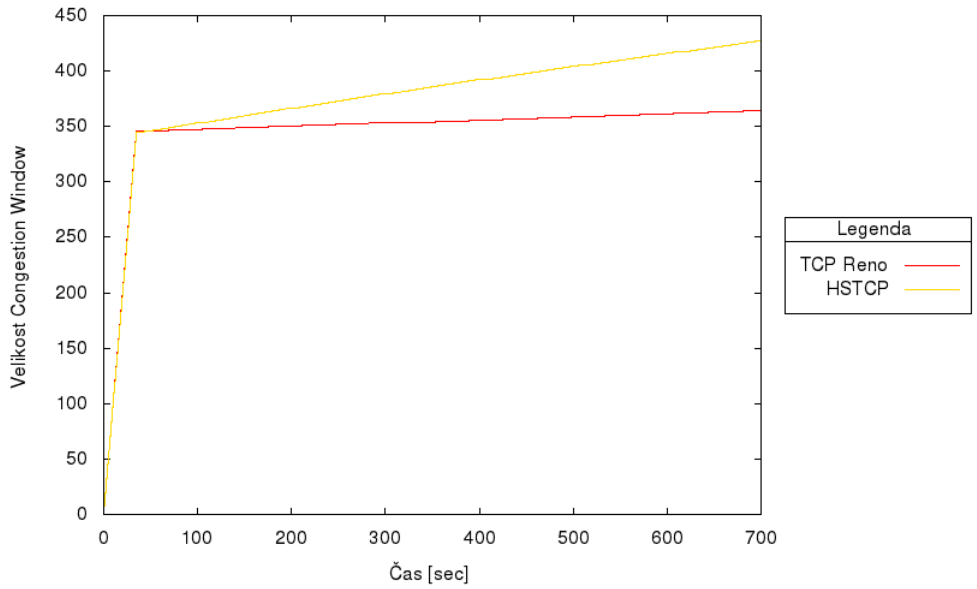
Obrázek 4.1: Výsledek testu malého přenosu dat, $ssthresh = 5$



Obrázek 4.2: Výsledek testu většího přenosu dat, $ssthresh = 100$

4.1.2 Vysokokapacitní síť

Tento test jsem zaměřil na situaci, kdy se posílá množství dat v řádu tisíců paketů a jedná se o vysokokapacitní sít. V takové komunikaci nedochází k zahlcení a všechny síťové uzly mají dostatek paměti ve svých bufferech. V této situaci lze jednoduše demonstrovat lepší využití linky algoritmem HSTCP. V následujícím grafu 4.3 porovnávám pouze HSTCP s TCP Reno, protože TCP Tahoe a Hybridní SACK mají obdobné chování jako Reno. Hybridní Vegas v tomto testu pozbývá na smyslu.



Obrázek 4.3: Výsledek testu s vysokokapacitním přenosem

Za tento test se přeneslo skoro sedm tisíc paketů a komunikace trvala 700 vteřin. Za tu dobu HSTCP stihl navýšit okno přesně o 63 paketů oproti algoritmu TCP Reno. Tento průběh v grafu je znázorněn rozevírajícími se nůžkami.

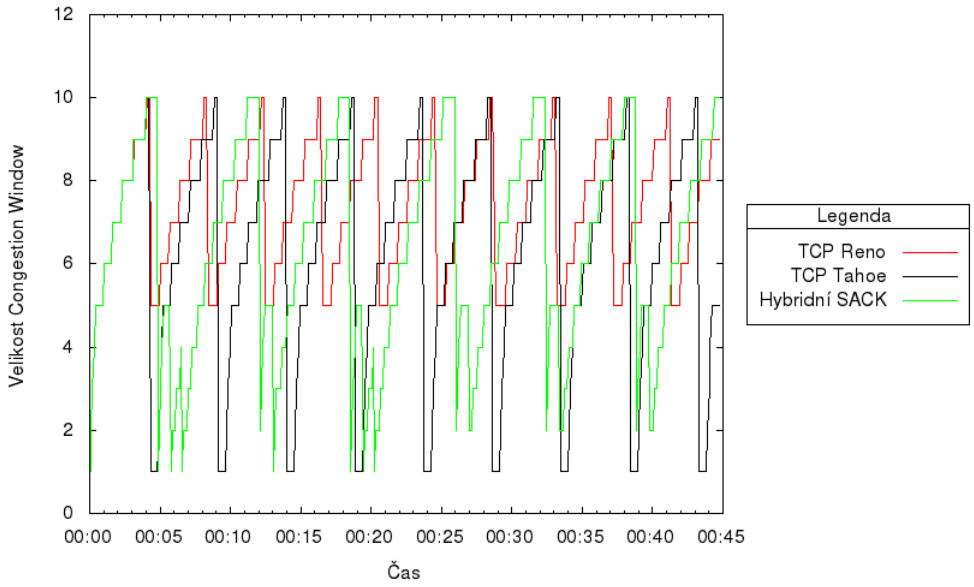
4.1.3 Omezení bufferů

Obecně mohou nastat dva scénáře, kdy je síť zahlcena omezením bufferů:

- Algoritmus dojde k maximu sítě ve fázi Congestion avoidance, viz graf 4.4
- Algoritmus dojde k maximu sítě ve fázi Slow-start, viz graf 4.5

První scénář je lepší, protože odesílatel posílá nové pakety pomaleji a nedojde k tak velkému nárazovém zahlcení, jako ve druhém bodě. Slow-start je velmi agresivní a projeví se výraznější ztrátovostí, se kterou se síť musí vypořádat.

Pro správnou simulaci obou testů je měněn pouze parametr *ssthresh*, kdy v prvním je nastaven na hodnotu 5 a v druhém již není nijak omezen (odpovídá tedy reálné síti). Velikost bufferů je omezena na 8 paketů.

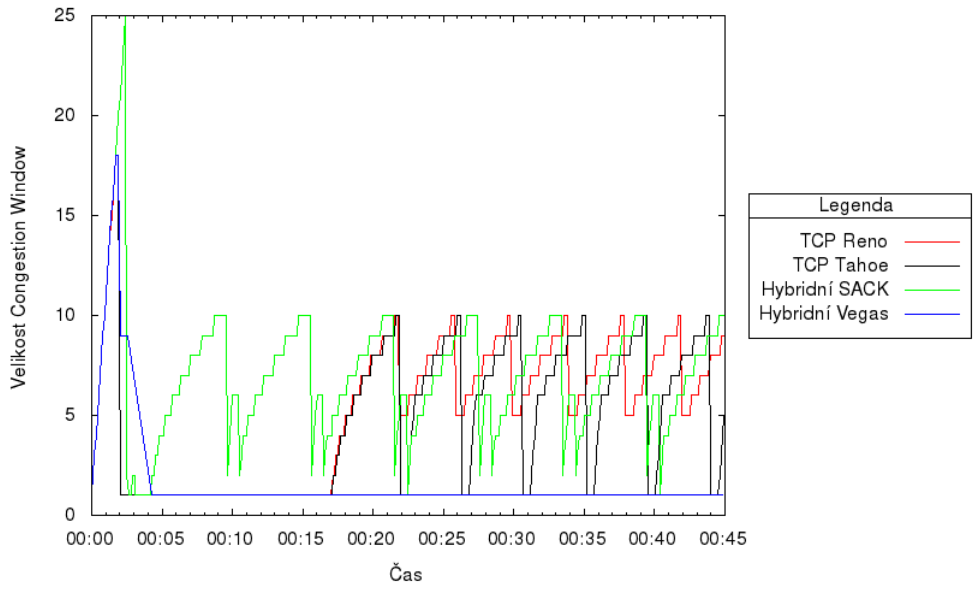


Obrázek 4.4: Výsledek testu s omezenou velikostí bufferů, omezený *ssthresh*

Vzhledem k tomu, že síť není ovlivňována vnějšími jevy (například měnícím se zpožděním) a *ssthresh* je menší než velikost bufferů, nemá v 4.4 testování Hybridního Vegas smysl. Ten by totiž zůstal na hodnotě $ssthresh = 5$ a dále by se neměnil. Stejně tak vynechám z dalšího testování HSTCP, protože okna v testech nenabývají tak velkých hodnot, aby se algoritmus nějak lišil od TCP Reno. Pokud bude test, kde bude mít Congestion Window prostor pro nárůst větších hodnot, HSTCP uvedu.

Pozorovat ostatní algoritmy však smysl má. V grafu 4.4 pozorujeme hlavní rozdíly mezi TCP Reno, Tahoe a Hybridním SACK. Z černé a zelené křivky je vidět, že Tahoe a Hybridní SACK díky detekci výpadku paketu pomocí timeoutu (pro Tahoe je tato událost ekvivalentní se třemi duplicitními ACK, přestože zde u něj k timeoutu nedošlo) zareagují výrazně větším zmenšením okna, než TCP Reno. Díky Slow-startu se opět rychle vyšplhají na větší hodnoty okna, ale už budou vždy posunuty oproti Renu. V tomto testu jsou tedy oba méně efektivní.

Graf 4.5 znázorňuje horší scénář zahlcení, než byl ukázán v 4.4. To je z důvodu, že ve fázi Slow-start odesílatel posílá daleko více paketů do sítě. Zvětšuje totiž Congestion Window exponenciálně, na rozdíl od lineárního Congestion avoidance.



Obrázek 4.5: Výsledek testu s omezenou velikostí bufferů, neomezený *ssthresh*

Zde Hybridní SACK zareaguje na ztrátu paketů daleko později, než ostatní algoritmy, kvůli rozdílné detekci ztráty paketu. Všechny algoritmy – kromě Hybridního SACK – mohou relativně dobře zareagovat třemi duplicitními ACK, ale síť je natolik zahlcená, že to zvládnou pouze jednou. To se projeví jedním schodem v prvním poklesu velikosti Congestion Window. Síť je totiž tak zahlcená, že odesílatel už nemá prostředky posílat další pakety ve Fast recovery a následuje série timeoutů. Trvá přibližně 10 vteřin, než odesílatel postupně po jednom odešle všechny nepotvrzené pakety opakovaným timeoutem. Je důležité zdůraznit, že vzhledem k velkému počtu opakujících se timeoutů hodnota *ssthresh* klesne na jeden paket, takže algoritmy nezahájí Slow-start, ale Congestion avoidance. Toto je nesmírně důležitá vlastnost, protože bez ní by se předchozí scénář periodicky opakoval. S touto reakcí již algoritmy šetrněji přistupují k síti a průběh komunikace je již velmi podobný tomu v 4.4.

Z grafu 4.5 je vidět ještě jedna důležitá vlastnost, která demonstruje sílu Hybridního SACK. Díky nastavování časovače pro každý jednotlivý paket, tento algoritmus provede timeout na všechny pakety najednou (pouze s odstupem času vytváření paketů). Tím se naprosto eliminuje doba zotavení ze sítě popsaná v odstavci výše. V tomto testu Hybridní SACK dokonce dvakrát úplně navýší okno, zatímco ostatní algoritmy posílají pakety po jednom a zotavují se ze zahlcení.

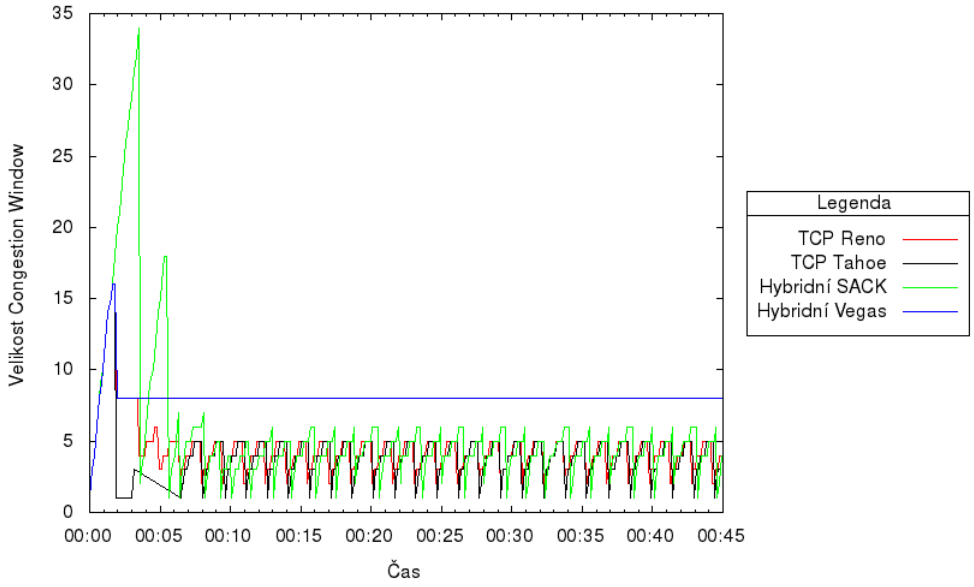
4.2 Chyby v síti

Chybovost v síti je pro mnoho algoritmů příznak zahlcení. Odpovídá nekvalitnímu připojení (např. WiFi) a projeví se ztrátou paketů. V testovacím prostředí jsou implementovány dva druhy chybovosti, které se testují zvlášť.

4.2.1 Modulární chybovost

Modulární chybovost odpovídá ztrátě jednoho paketu za každých n paketů. Tato chybovost například simuluje poškození jednotlivých paketů při přenosu přes nekvalitní síťové uzly.

Z grafu 4.6 je nejpatrněji vidět přestřelení okna algoritmem Hybridní SACK. To je dáno pozdní reakcí na zahlcení z důvodu detekce timeoutem. Algoritmus velmi poklesne, ještě jednou se vyšplhá na poloviční velikost a dál už reaguje podobně, jako ostatní.



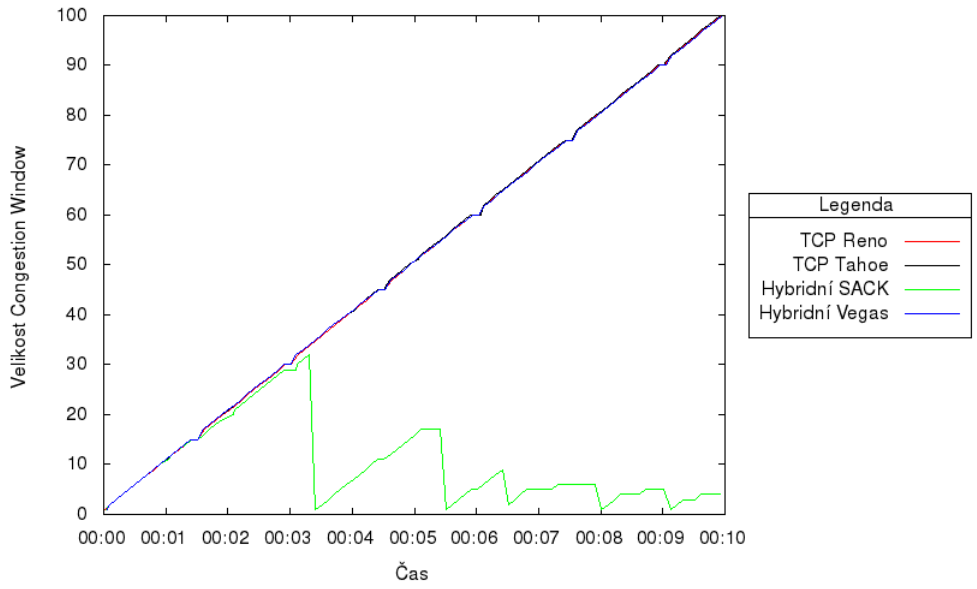
Obrázek 4.6: Výsledek testu s modulární chybovostí sítě od odesílatele

Hlavní rozdíl, na který bych chtěl poukázat, je mezi TCP Reno a TCP Tahoe. Je vidět, že Tahoe příliš zmenší velikost okna při třech duplicitních ACK. Tím se sám připraví o cenné prostředky a mezi třetí a šestou vteřinou dokonce skončí timeoutem, protože nemá dostatečnou kapacitu pro další přenos pomocí Slow-startu (je až příliš mnoho nepotvrzených paketů). TCP Reno se lépe dobere k optimální velikosti okna postupným snižováním, jak je vidět z červené křivky.

Nutno dodat, že Hybridní Vegas v tomto testu funguje nejefektivněji. Sice kvůli reakci na výpadek ve Slow-startu sníží okno na velikost 8, dál ale již veli-

kost nemění, protože chybovost není ovlivněná zatížeností sítě. Tato funkčnost je výjimečná právě rozdílnou detekcí zahlcení, než používají ostatní algoritmy.

Zajímavé pozorování také vznikne, když se chybovost vyskytne pouze po cestě zpět – tedy od příjemce k odesílateli – jako tomu je v grafu 4.7. Ztrácejí se tedy jen potvrzení. Vzhledem k tomu, že algoritmy standardně čekají na tři duplicitní ACK, ztracené potvrzení se nahradí nadcházejícím, které potvrdí dva pakety najednou. Jediný algoritmus, který má s tímto zásadní problém, je Hybridní SACK. Ten totiž potřebuje jedno konkrétní potvrzení ke konkrétnímu paketu. Proto má průběh totožný, jako v grafu 4.6.



Obrázek 4.7: Výsledek testu s modulární chybovostí sítě od příjemce

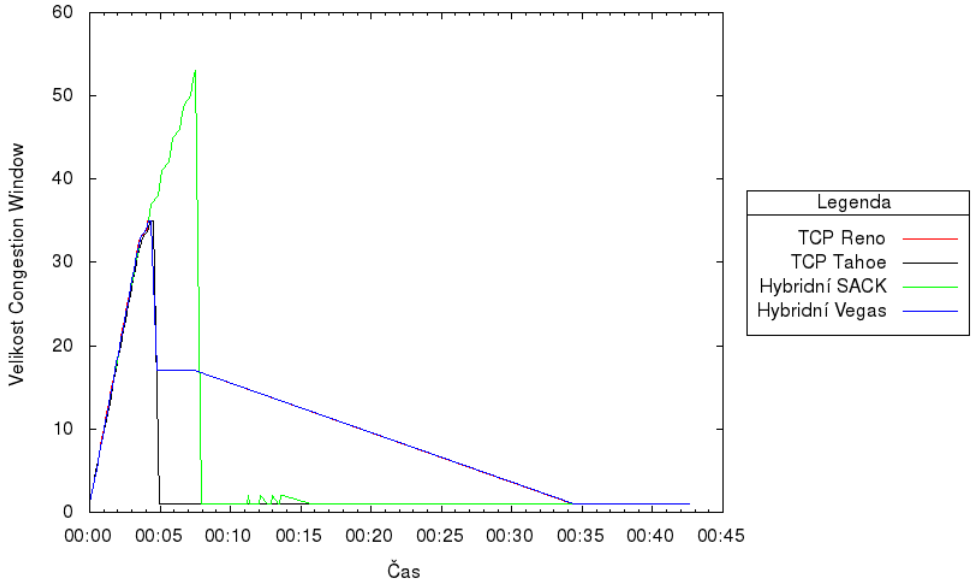
4.2.2 Bloková chybovost

V komunikaci se může stát, že se spojení na nějaký časový okamžik přeruší a ztratí se tím určitý blok dat. Tyto pakety je potřeba poslat znovu a algoritmy se s touto situací musejí vypořádat. Bloková chybovost je v testovacím prostředí implementována jako střídavé posílání a zahazování paketů. Nejprve se n paketů odešle bez problémů, poté se n paketů zahodí a proces se opakuje. Pro tento test jsem nastavil tyto parametry chybovosti:

- Bloková chybovost o modulu 3
- Zpoždění startu chybovosti 30 paketů

Z grafu 4.8 je vidět, že dokud chybovost nezačne po třiceti odeslaných paketech, algoritmy běží bez problémů. Jakmile ale tato 50% chybovost nastane,

všechny algoritmy skončí timeoutem a úplně zkolabují. Hybridní SACK se sice snaží o mírný nárůst okna mezi časy 00:10 a 00:15, ale k lepší výsledkům to nevede. Tento nárůst je daný dřívějšími timeouty všech paketů, obdobně jako v grafu 4.5.



Obrázek 4.8: Výsledek testu s blokovou chybovostí sítě od odesílatele

TCP Reno, Tahoe a Hybridní Vegas mají totožný průběh. V sedmé vteřině jim dojdou prostředky pro komunikaci a čekají celý RTO na událost timeoutu. Ta nastane těsně před 35. vteřinou běhu.

Tento test mimo jiné demonstruje tvrzení v 1.2.2, že Congestion avoidance algoritmy nepracují dobře s většími výpadky než 1%. Zde je to ještě prohloubeno kvůli blokovému zabírání místa v oknu odesílatele v podobě nepotvrzených paketů.

Scénář blokové chybovosti cestou od příjemce je obdobný s modulární chybovostí v grafu 4.7 a proto jej zde neuvádím.

4.3 Proměnlivé zpoždění sítě

Když v komunikaci naroste zpoždění, může to znamenat, že s nějakým síťovým uzlem začala komunikovat ještě nějaká další strana. Naopak, pokud se zpoždění zmenší, jiná komunikace nejspíše ukončila spojení.

Pro první test s proměnlivým zpožděním a s grafem 4.9 jsem nastavil takovéto parametry:

- $ssthresh = 12$

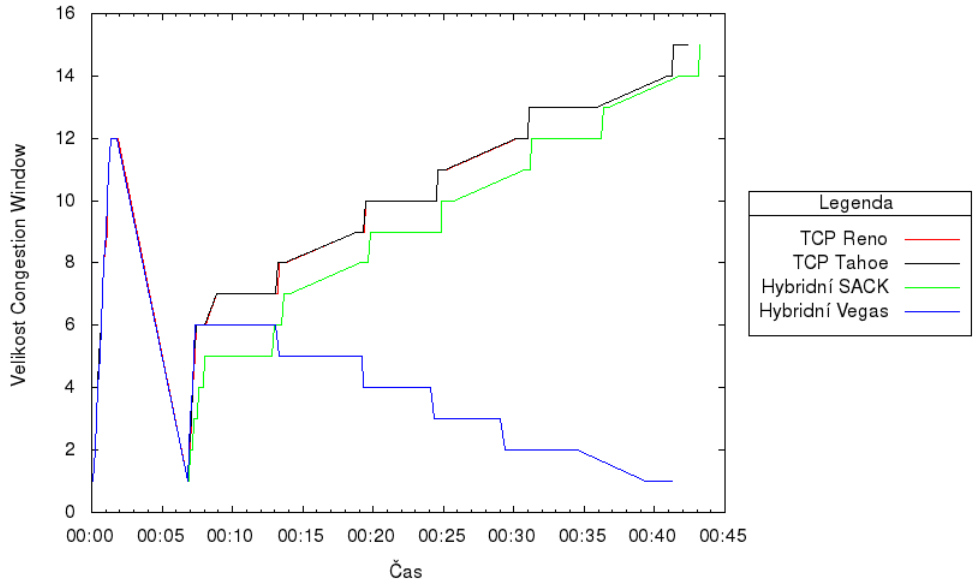
- Střídání zpoždění $n = 15$
- Klidové zpoždění 100 ms
- Koeficient pro zvětšení roven 50, tedy zátěžové zpoždění 5 vteřin

Z grafu 4.9 je vidět, že žádný algoritmus tuto nárazovou zátěž nejdříve nezvládne a skončí timeoutem. Během tohoto času se však dokáže patřičně přepočítat RTO podle vzorců uvedených v 3.2.2.5 a ani opakující se výkyvy ve zpoždění už nezapříčiní další úplný kolaps.

Pro algoritmus Hybridní Vegas je tento test nejtěžší, protože se z jeho pohledu jedná o opakované zahlcení sítě. Je patrné, že v těchto chvílích vždy zmenší velikost okna. Ostatní algoritmy si vedou lépe, protože v síti kolísá pouze zpoždění, ale není zde žádná chybovost. Až si přepočítají správnou velikost RTO, nic nebrání ve zvětšování okna. Nárůst je pouze nepravidelný (není čistě schodovitý jako v předešlých testech) kvůli síťovým fluktuacím.

Nepatrný rozdíl je ještě zaznamenán u Hybridního SACK, kdy se vzpamatovává delší dobu, než TCP Reno a Tahoe. To je opět dáno jiným způsobem detekce timeoutu, kdy kontroluje naráz všechny pakety, ne pouze první z nepotvrzených. Tím pádem se ještě nestihne dostatečně přepočítat RTO a algoritmus se tím zbrzdí.

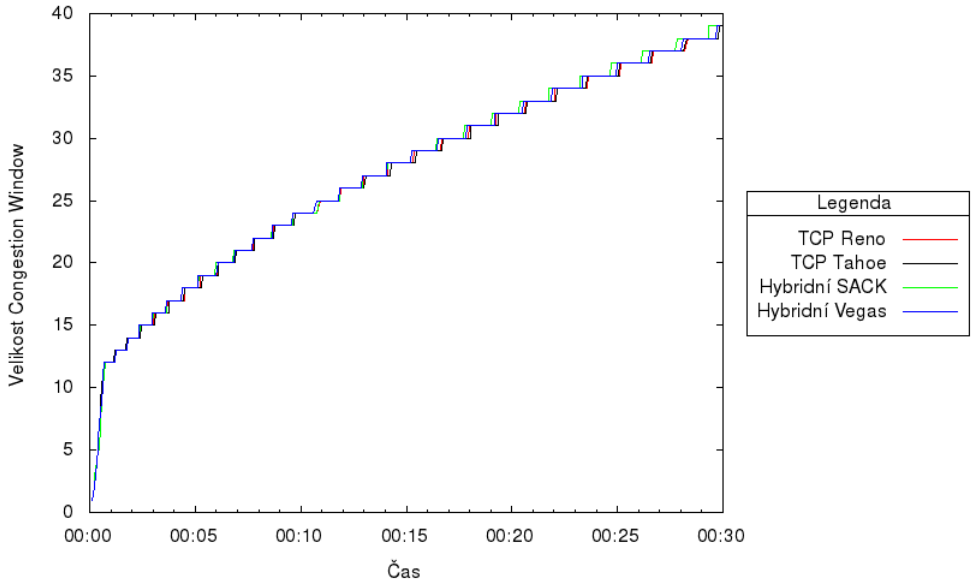
Tento test jsem provedl pouze na cestě od odesílatele k příjemci, protože zde algoritmům záleží na celkovém zpoždění komunikace. Výsledek by byl tedy totožný.



Obrázek 4.9: Výsledek testu s narůstajícím zpožděním každých 15 paketů

Zcela opačný scénář pro Hybridní Vegas nastane, když se ve stejných místech bude zpoždění zmenšovat. Koeficient je v tomto testu nastaven na 0.001, tedy sníží zpoždění tisíckrát. Aby bylo možné dostatečně zmenšit zpoždění, v tomto testu jsem musel zmenšit počet bajtů paketů, protože jinak by komunikaci brzdilo taktování 3.2.4.2 a menší zpoždění by se neprojevilo. V tomto testu nesou pakety pouze třetinu dat, než v předešlých, tedy 300 B.

Jak je z grafu 4.10 vidět, Hybridní Vegas narůstá stejnou mírou, jako ostatní algoritmy. Síť je tedy v tak dobré kondici, že algoritmus navyšuje okno každý RTT a nedochází k zahlcení.



Obrázek 4.10: Výsledek testu se snižujícím zpožděním každých 15 paketů

4.4 Souhrn pozorovaných vlastností

Z jednotlivých testů lze vypozorovat několik vlastností, které charakterizují konkrétní funkčnosti algoritmů. Jak je z řady testů vidět, Slow-start je skutečně agresivní metoda nárůstu okna, pokud se chybovost linky projeví až po nějakém množství odeslaných dat. Podobný problém nastane, když je síť limitována pamětí síťových uzlů.

Ztrátovost paketů je významným příznakem zahlcení a je dobré ji vždy zohlednit. Její největší nevýhodou však je, že zahlcení detekuje až tehdy, když k němu již došlo. V řadě testů tento problém skončil i sérií timeoutů v řadě. V reálné komunikaci by tento problém znamenal několikavteřinovou prodlevu v komunikaci.

Řešením těchto problémů je do určité míry princip selektivního potvrzování, který všechny pakety pokládá jako samostatné instance přenosu dat. Tímto oddělením funkčnosti nicméně dojde k znemožnění pozorování kontextu, ve kterém se spojení nachází. Toto je názorně vidět v grafu 4.7, kde kvůli neznalosti ostatních potvrzení algoritmus snižuje okno, přestože žádný paket ve skutečnosti nevypadl.

Další ze způsobů detekce zahlcení je měření proměnlivého zpoždění v odpovědích od příjemce. Tato funkčnost je v testech znázorněna algoritmem Hybridní Vegas. Přestože je tento princip napřed ve zhodnocení kvality komunikace, nelze ho považovat za nejlepší řešení. Například v grafu 4.5 má úplně nejhorší výsledky, protože v síti se nemění zpoždění a on zůstane na konstantní velikosti okna. V reálné komunikaci by takový postup znamenal nesmyslně pomalý přenos dat.

Každý algoritmus je dobrý v něčem jiném. Optimální řešení se hledá těžko, ale možná se skrývá ve vhodné kombinaci výše uvedených principů. To ale musejí zjistit budoucí výzkumy a do jaké míry se podaří kontrolovat rychlost datových toků ukáže jen čas.

Závěr

V práci byl nejdříve definován problém zahlcení sítě a jaké následky může mít na komunikaci mezi odesílatelem a příjemcem. Rovněž byl uveden způsob členění, kterým se algoritmy řídí za úkolem předcházet či napravovat zahlcení sítě. Toto členění obsahuje čtyři části tvořící společný základ novodobých implementací. K regulaci datového toku slouží zvětšování nebo zmenšování odesílatelova okna.

Součástí práce je několik vzorových řešení, která jsou v protokolu TCP již vyvinuta a dnes se běžně používají. Ta se liší ve způsobu detekce zahlcení a jejich reakcí, pokud k zahlcení došlo. Obsažené jsou v navrženém testovacím prostředí, které simuluje chod sítě díky vícevláknové obsluze komunikačního kanálu. Tato vlákna tvoří odesílatele, příjemce a parametrizované komunikační prostředky, simulující síťové nepřízně. Mezi ně patří rychlost síťových uzlů, zpoždění linky, nebo chybovost sítě. Díky tomu bylo možné jednotlivé implementace otestovat a porovnat jejich chování v určitých podmínkách. Každý z těchto algoritmů je lepší v něčem jiném a to také stěžuje hledání vhodné implementace pro konkrétní spojení na internetu.

Přestože jsou v práci popsány obecné principy fungování algoritmů řešících zahlcení, nejedná se o konečná řešení problematiky zahlcení sítě. I v této době probíhá mnoho výzkumů, které se snaží zlepšit podmínky pro komunikaci a lépe regulovat datový tok po sítích. Kromě implementace standardních algoritmů byly do testovacího prostředí vyvinuty i hybridní implementace, které přinášejí jiný pohled na problematiku zahlcení. Již zmíněné výzkumy jdou stejným směrem – nesnaží se jen čerpat výhody již vynalezených postupů, ale snaží se také rozvinout tyto postupy do lepších implementací. Zde je obrovské místo pro nové vynálezy. Do této práce lze dodatečně implementovat další algoritmy pro potřeby bližšího testování a srovnání se standardy TCP.

Konkrétní zaměření této práce bylo na algoritmy se správou u odesílatele. Nejedná se však o jediný způsob regulace datových toků. Tyto výzkumy jsou ale ještě na začátku a čeká je teprve budoucí rozvoj.

Literatura

[1] Information Sciences Institute, University of Southern California: Transmission Control Protocol. RFC 793, RFC Editor, September 1981. Dostupné z: https://www.ietf.org/rfc/rfc793.txt

[2] Paxson, V.; Allman, M.; Chu, J.; aj.: Computing TCP's Retransmission Timer. RFC 6298, RFC Editor, June 2011. Dostupné z: https://tools.ietf.org/pdf/rfc6298.pdf

[3] Nagle, J.: Congestion Control in IP/TCP Internetworks. RFC 896, RFC Editor, January 1984. Dostupné z: https://tools.ietf.org/pdf/rfc896.pdf

[4] L. Mamtas and V. Tsaoussidis and Chi Zhang: Approaches to Congestion Control in packet networks. Technická zpráva, Demokritos University; Florida International University, October 2004. Dostupné z: http://www.intersys-lab.org/media/papers/2004/tr-duth-ee-2004-10.pdf

[5] Allman, M.; Paxson, V.: TCP Congestion Control. RFC 5681, RFC Editor, September 2009. Dostupné z: https://tools.ietf.org/pdf/rfc5681.pdf

[6] E. A. Akkoyunlu and K. Ekanadham and R. V. Huber: Some constraints and tradeoffs in the design of network communications. Technická zpráva, Department of Computer Science, State University of New York at Stony Brook, November 1975. Dostupné z: http://hydra.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/download/1975_Akkoyunlu,%20Ekanadham,%20Huber_Some%20constraints%20and%20tradeoffs%20in%20the%20design%20of%20network%20communications.pdf

[7] Stevens, W. R.: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, RFC Editor, January 1997. Dostupné z: https://tools.ietf.org/pdf/rfc2001.pdf

[8] Allman, M.; Floyd, S.; Partridge, C.: Increasing TCP's Initial Window. RFC 3390, RFC Editor, October 2002. Dostupné z: https://tools.ietf.org/pdf/rfc3390.pdf

[9] Brakmo, L. S.; O'Malley, S. W.; Peterson, L. L.: TCP Vegas: New Techniques for Congestion Detection and Avoidance. Technická zpráva, Department of Computer Science, The University of Arizona, February 1994. Dostupné z: http://pages.cs.wisc.edu/~akella/CS740/S08/740-Papers/BOP94.pdf

[10] Jacobson, V.; Braden, R.: TCP Extensions for Long-Delay Paths. RFC 1072, RFC Editor, October 1988. Dostupné z: https://tools.ietf.org/pdf/rfc1072.pdf

[11] Floyd, S.: HighSpeed TCP for Large Congestion Windows. RFC 3649, RFC Editor, December 2003. Dostupné z: https://tools.ietf.org/pdf/rfc3649.pdf

[12] Ha, S.; Le, L.; Xu, L.; aj.: BIC and CUBIC [Online]. 2007, [cit. 8.5.2016]. Dostupné z: http://research.csc.ncsu.edu/netsrv/?q=content/bic-and-cubic

[13] Fahmy, S.; Karwa, T. P.: TCP Congestion Control: Overview and Survey Of Ongoing Research. Technická zpráva, Department of Computer Science, Purdue University, 2001. Dostupné z: https://pdfs.semanticscholar.org/317f/549d822cffc04d13f590198def2ba1bf5d7a.pdf

[14] Braden, R.: Requirements for Internet Hosts – Communication Layers. RFC 1122, RFC Editor, October 1989. Dostupné z: https://tools.ietf.org/pdf/rfc1122.pdf

[15] Paxson, V.; Allman, M.: Computing TCP's Retransmission Timer. RFC 2988, RFC Editor, November 2000. Dostupné z: https://tools.ietf.org/pdf/rfc2988.pdf

[16] Karn, P.; Partridge, C.: Improving Round-Trip Time Estimates in Reliable Transport Protocols. Technická zpráva, 1987. Dostupné z: http://ccr.sigcomm.org/archive/1995/jan95/ccr-9501-partridge87.pdf

[17] Jacobson, V.; Braden, B.; Borman, D.: TCP Extensions for High Performance. RFC 1323, RFC Editor, May 1992. Dostupné z: https://tools.ietf.org/pdf/rfc1323.pdf

[18] Savage, S.; Cardwell, N.; Wetherall, D.; aj.: TCP Congestion Control with a Misbehaving Receiver. Technická zpráva, Department of Computer Science and Engineering, University of Washington, Seattle, October 1999. Dostupné z: https://cseweb.ucsd.edu/~savage/papers/CCR99.pdf

[19] Gnuplot [software]. [cit. 8.5.2016]. Dostupné z: http://www.gnuplot.info/

[20] Kurose, J. F.; Ross, K. W.: *Computer Networking: A Top-Down Approach (6th edition)*. Pearson, 2012, ISBN 978-0132856201.

[21] Jacobson, V.: Congestion avoidance and control. Technická zpráva, Stanford, CA, August 1988.

[22] Floyd, S.: Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742, RFC Editor, March 2004. Dostupné z: https://tools.ietf.org/pdf/rfc3742.pdf

[23] Mathis, M.; Mahdavi, J.; Floyd, S.; aj.: TCP Selective Acknowledgment Options. RFC 2018, RFC Editor, October 1996. Dostupné z: https://tools.ietf.org/pdf/rfc2018.pdf

Příloha **A**

Seznam použitých zkratek

TCP Transmission Control Protocol

ISO International Organization for Standardization

OSI Open Systems Interconnection

RTT Round-trip time

ACK Acknowledgement

SACK Selective Acknowledgement

LAN Local Area Network

ssthresh Slow-start threshold

RTO Retransmission Timeout

BIC Binary Increase Congestion control

HSTCP High Speed TCP

cwnd Velikost okna Congestion Window

ID Idetifier

SRTT Smoothed Round-trip time

RTTVAR Round-trip time variation

RFC Request for Comments

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `manual.txt` ..... stručný manuál k obsluze testovacího prostředí
  - `main.cpp` ..... zdrojový kód implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF