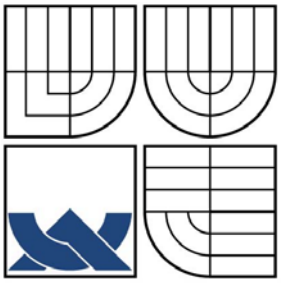
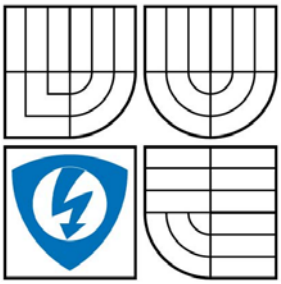


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKACNÍCH TECHNOLOGIÍ ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION EPARTMENT OF TELECOMMUNICATIONS

NÁZEV DIPLOMOVÉ PRÁCE

TITLE

ALGORITMUS VIVALDI PRO NALEZENÍ POZICE STANICE V INTERNETU

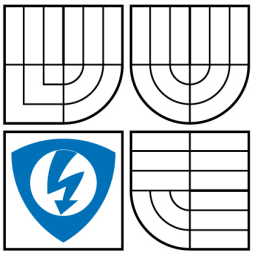
DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — BC.TOMÁŠ HANDL
AUTHOR

VEDOUCÍ PRÁCE — ING. DAN KOMOSNÝ, PH.D..
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Tomáš Handl	***ID:***	85689
Ročník:	2	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

Algoritmus Vivaldi pro nalezení pozice stanice v Internetu

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte algoritmus Vivaldi. Zaměřte se na jeho využití pro 2D souřadnicový systém s výškou. Realizujte algoritmus Vivaldi ve formě knihovny, která bude implementovat dvě varianty algoritmu - s konstantním a proměnným časovým krokem. Vytvořenou knihovnu přehledně zdokumentujte. Ke knihovně zhotovte aplikaci, která bude umožňovat přehlednou konfiguraci parametrů algoritmu a zobrazovat výstupy v textové podobně. Proveďte simulace nalezení pozice stanic pomocí reálných hodnot RTT (Round-Trip Time) v Internetu.

DOPORUČENÁ LITERATURA:

[1] DABEK, F., COX, R., KAASHOEK, F., MORRIS, R. Vivaldi: a decentralized network coordinate system. ACM SIGCOMM Computer Communication Review. Association for Computing Machinery, 2004. ISSN: 0146-4833.
[2] EUGENE, T. S., ZHANG, H. Predicting Internet Network Distance with Coordinates-Based Approaches. Proceedings of 21st Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE, 2002.

Termín zadání:	9.2.2009	***Termín odevzdání:***	26.5.2009

Vedoucí práce: Ing. Dan Komosný, Ph.D.

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Abstrakt v češtině

Práce se zabývá využitím umělých souřadnicových systémů k lokalizaci stanice v síti Internet a predikci zpoždění mezi stanicemi. Jsou zde popsány a srovnány základní vlastnosti centralizovaných a decentralizovaných algoritmů umožňujících lokalizaci stanice v síti Internet a predikci RTT. Podrobněji jsou zde popsáni hlavní představitelé obou typů algoritmů jako GNP, IDMAPS, nebo Lighthouse. Hlavní část práce je zaměřena na seznámení s distribuovaným algoritmem Vivaldi. Je zde popsán základní princip tohoto algoritmu pro variantu s proměnným a konstantním časovým krokem využívající 2-rozměrný souřadnicový systém s třetím parametrem výškou. Dále je zde popsána implementace tohoto algoritmu v podobě knihovny Vivaldi-lib v prostředí programovacího jazyka JAVA. Součástí práce jsou i simulace chování tohoto algoritmu pro obě varianty provedené na umělých sítích a datech získaných z experimentální sítě PlanetLab pomocí vytvořeného simulačního programu VIVALDIMONITOR.

KLÍČOVÁ SLOVA

Klíčová slova v češtině

Algoritmus Vivaldi, umělý souřadnicový systém, predikce RTT, určení polohy stanice v Internetu, decentralizovaný algoritmus.

ABSTRACT

Abstract in English

Diploma thesis deals with usage of artificial coordinate systems used for localization of a station on the internet and prediction of delay between the stations. There are described and compared basic properties of centralized and decentralized algorithms providing station localization on the internet and RTT prediction. More in depth are presented main representatives of both types of algorithms such as GNP, IDMAPS or Lighthouse. Central part of thesis is aimed at getting to know Vivaldi distributed algorithm. Basic principle of the algorithm for constant and variable time step, using two dimensional coordinate system with 3rd parameter height, is here outlined. Further more implementation of this algorithm as a library Vivaldi-lib in the environment of Java is implemented. Part of the thesis are simulations of behaviour of this algorithm for both variations realized on artificial networks and data obtained from PlanetLab experimental network, using simulation created program VIVALDIMONITOR.

KEYWORDS

Keywords in English

Algorithm Vivaldi, synthetic coordinates systém, prediction of RTT, location of host in Internet, decentralized algorithm.

HANDL, T. *Algoritmus Vivaldi pro nalezení pozice stanice v Internetu*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 69 s. Vedoucí diplomové práce Ing. Dan Komosný, Ph.D.

Prohlášení o původnosti práce

Prohlašuji, že svou diplomovou práci na téma „*Algoritmus Vivaldi pro nalezení pozice stanice v Internetu“* jsem vypracoval samostatně pod vedením vedoucího diplomové práce s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně
dne . . . . . .

. . . . . .
(podpis autora)

Poděkování

Děkuji vedoucímu diplomové práce Ing. Danu Komosnému, Ph.D. za velmi užitečnou metodickou pomoc a cenné rady při zpracování této práce a dále děkuji Bc. Jaroslavu Švédovy za výbornou spolupráci při implementaci vyvinuté knihovny do simulační programu a spolupráci při testování chování algoritmu Vivaldi.

V Brně
dne . . . . . .

. . . . . .
(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

Úvod

Tato diplomová práce je zaměřena na seznámení s principy algoritmu souřadnicového systému Vivaldi, který umožňuje nalezení stanic v Internetu na základě predikce hodnoty RTT (Round Trip Time Delay). Parametr RTT je definován jako obousměrné zpoždění, které zahrnuje dobu přenosu paketu od zdroje k příjemci a zpět a také dobu zpracování paketu. RTT k libovolnému uzlu v síti lze jednoduše změřit přímo z libovolné stanice pomocí příkazu ping nebo traceroute, které představují základní nástroje správy sítě. Oba nástroje používají zprávy signalizačního protokolu ICMP [1]. Cílem práce byla implementace dvou základních variant tohoto algoritmu, a to varianty s konstantním a proměnným časovým krokem využívající 2-rozměrný souřadnicový prostor s třetím parametrem výškou v podobě knihovny Vivaldi-lib a ověření základního chování tohoto algoritmu v praxi. K samotnému provádění simulací byla následně použita aplikace VIVALDIMONITOR [2], kterou vyvíjel Bc. Jaroslav Švéda. Základem této aplikace byla právě vyvinutá knihovna Vivaldi-lib implementující obě varianty algoritmu Vivaldi. Samotná aplikace VIVALDIMONITOR představuje simulační program s grafickým ovládacím rozhraním. Toto grafické rozhraní poskytuje širokou nabídkou funkcí pro snadnější ovládání a zpracování výsledků, jako grafické zobrazení všech důležitých charakteristik, import a export dat, nastavení jednotlivých parametrů atd. Aplikace umožní budoucím uživatelům podrobné simulace a testování algoritmu Vivaldi na datech získaných z umělých a skutečných sítí a studium tohoto algoritmu pro nasazení v reálném provozu v síti Internet.

Pro samotnou implementaci knihovny Vivaldi-lib jsem zvolil vývojové prostředí programovacího jazyka JAVA vzhledem k požadavku na grafickou nástavbu programu, neboť prostředí jazyka JAVA poskytuje širokou podporu grafických nástrojů a výhodou je i přenositelnost programu na různé operační systémy.

První kapitola této práce je zaměřena na historii a současnost použití umělých souřadnicových systémů k predikci hodnoty RTT v síti Internet. Součástí této kapitoly je i shrnutí základních požadavků na souřadnicové systémy.

Druhá kapitola seznamuje se základními algoritmy pro určení polohy stanic v síti Internet a predikci zpoždění. V této části je popsáno základní rozdělení algoritmů na centralizované a decentralizované a seznámení s jejich výhodami a nevýhodami při reálném nasazení.

Třetí kapitola se již zabývá přímo algoritmem Vivaldi. Popisuje základní princip tohoto algoritmu. Zabývá se významem časového kroku na rychlost konvergence a přesnost správně předpovídat hodnotu RTT. V této kapitole jsou popsány různé souřadnicové prostory, které může tento algoritmus implementovat. Jejich výhody, nevýhody a vysvětlení, proč byl pro samotnou implementaci zvolen právě 2-rozměrný souřadnicový prostor s třetím parametrem výškou.

Čtvrtá kapitola popisuje již samotnou vytvořenou knihovnu Vivaldi-lib implementující obě varianty algoritmu Vivaldi v prostředí programovacího jazyka Java. Je zde popsána základní struktura knihovny a její členění. Dále jsou zde popsány základní třídy, rozhraní a jejich nejdůležitější parametry a funkce. Jsou zde shrnuty základní testy pro ověření funkčnosti samotné knihovny a správnosti implementace algoritmu Vivaldi. Součástí této kapitoly je i základní popis simulačního programu včetně implementace samotné knihovny do tohoto programu.

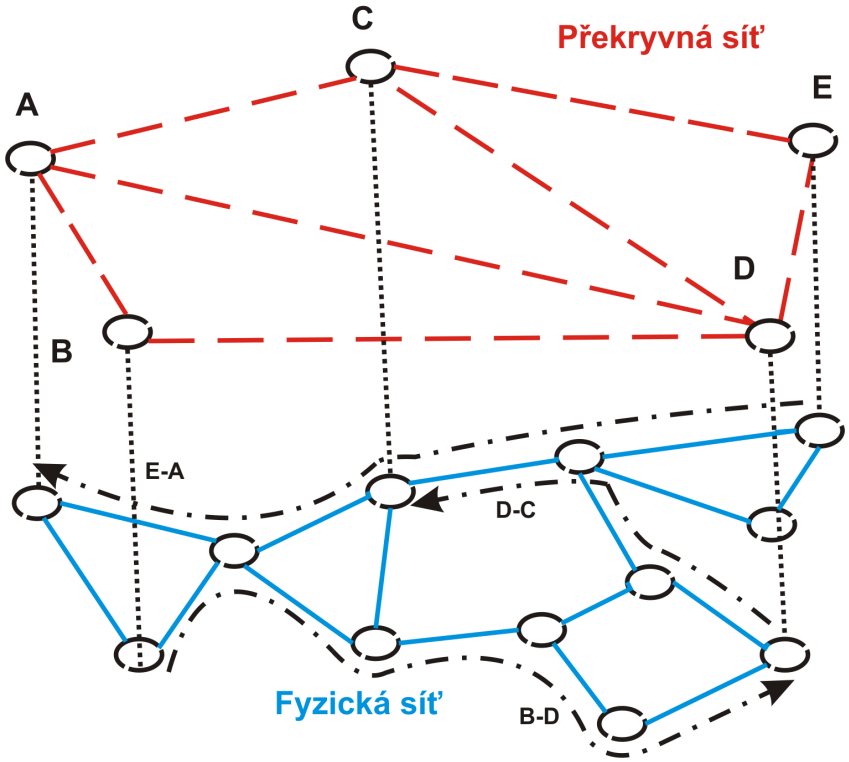
Poslední kapitola je již zaměřena na samotné simulace obou variant algoritmu Vivaldi na umělých sítích a datech získaných z experimentální sítě PlanetLab. Všechny provedené simulace jsou zde podrobněji popsány a doplněny řadou grafů spolu s výsledky měření.

1 Význam souřadnicových systémů při určování polohy a vzájemné vzdálenosti stanic

1.1 Možnosti určení pozice stanic v Internetu

S neustálým rozvojem v oblasti telekomunikací a exponenciálním nárůstem počtu uživatelů připojujících se celosvětově k síti Internet se v posledních letech významně zvýšilo používání síťových služeb a aplikací. Mezi tyto aplikace a služby patři multicastové přenosy, které jsou používány pro streamové multimediální přenosy např. pro vysílání internetové televize označované jako IPTV, peer - to - peer systémy pro distribuci a sdílení souborů (například BitTorrent [3], Emula [4], KaZaA [5]), VOIP (Voice over IP) systémy pro hlasovou a obrazovou komunikaci (například Skype, ICQ) a další internetové aplikace. Většina z těchto aplikací optimalizuje svůj výkon právě na základě znalosti topologie sítí. Například budování stromů multicast vysílání, nebo výběr nejvhodnějšího z více replikujících serverů pro aplikace sdílející soubory může být efektivnější při znalosti pozic jednotlivých stanic v síti. Tyto sítě představují z hlediska zatížení podstatnou část provozu v síti Internet. Všechny stanice v této síti jsou pospojovány na základě logických spojů, které nemusejí odpovídat samotné fyzické struktuře sítě, viz obr. 1.1. U překryvných sítí může docházet k tomu, že topologie sítě neodpovídá komunikaci mezi stanicemi a dochází tak k velmi neefektivní komunikaci, která představuje významnou zátěž pro samotnou síť. Právě tyto systémy mohou těžit z inteligentního výběru cesty nebo stanic. Síťové charakteristiky, jako zpoždění nebo šířka pásma, jsou podstatné pro tyto aplikace a měly by být měřeny. Základní přístup k získávání těchto charakteristik potřebných k určení pozice stanice v síti je získáván sondováním všech stanic v síti. Cena spojená s aktivním monitorováním za účelem odhadnutí potřebných atributů není však v těchto případech zanedbatelná. Cenu představuje hodnota zatížení provozu, zpoždění atd. Především u sítí typu peer - to - peer a jiných sítí by přímé měření výše uvedených parametrů mohlo způsobit až přetížení sítě, neboť počet měření roste kvadraticky s počtem stanic. Pro N stanic je potřebné periodické měření $N\times(N\text{-}1)$ vzdáleností, což při velikosti peer - to - peer sítí není možné. Naopak umělé souřadnicové systémy poskytují přesnou a výkonnou službu, která dovoluje stanicím v síti Internet, aby určily zpoždění k libovolným stanicím bez aktivního monitorování všech stanic v síti. Vzhledem k předchozí uvedené definici pojmu ceny se jedná o nízkocenovou komunikační službu, která umožňuje předpovídat zpoždění mezi libovolnými stanicemi v síti. Stanice počítají souřadnice ve stejném souřadnicovém prostoru tak, že vzdálenost mezi souřadnicemi dvou stanic v prostoru předpovídá hodnotu zpoždění mezi nimi v Internetu. Jestliže stanice A zjistí souřadnice stanice B, tak A nemusí vykonávat explicitní měření k této stanici, aby určila potřebné zpoždění. Místo toho je vzdálenost mezi stanicemi A a B v souřadnicovém prostoru přesným odhadem zpoždění. Většina z těchto umělých systémů byla navržena s předpokladem, že všechny stanice v systému jsou nesobecké. Tento předpoklad však může být porušen oslabenými stanicemi, které jednají ve zlém úmyslu s cílem sesadit přesnost souřadnicového systému. Může se jednat o stanice napadané útočníkem, který chce z nějakého důvodu narušit správné fungování systému. Ohroženější skupinou systémů na útok zevnitř jsou systémy decentralizované, které

nepoužívají fixní infrastrukturu. V těchto systémech každá stanice určuje svoji polohu vůči libovolné skupině jiných stanic.



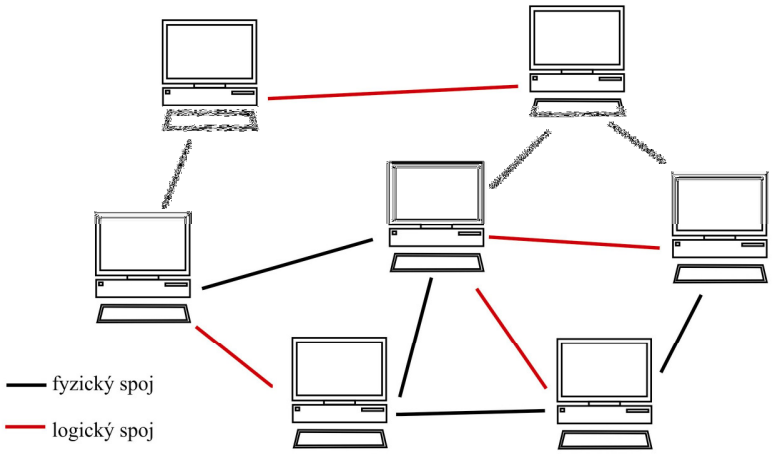
Obr. 1.1: Rozdíl mezi fyzickou a logickou strukturou sítě

1.2 Umělé souřadnicové systémy

Geometrický model síťových vzdáleností představuje mnoho výhod pro síťové aplikace. Jedna z hlavních výhod je schopnost spočítat vzdálenosti mezi body ve zvoleném prostoru. Jakmile jedna stanice spočítá svoji souřadnici, může odhadnout svoji vzdálenost představující hodnotu RTT k jiným stanicím v daném souřadnicovém systému bez explicitního měření k těmto stanicím. Namísto ukládání N^2 vzdáleností stačí sumarizovat topologické vzájemné vztahy mezi stanicemi, kde N představuje počet stanic systému. Umělé souřadnicové systémy byly navrhovány jako nízkocenová komunikační služba, která má přesně předpovídat zpoždění mezi libovolnými stanicemi v síti. Tyto systémy umožní stanici určit svoje vlastní souřadnice, založené na znalosti minimálního množství získaných hodnot zpoždění, které se odhaduje k podmnožině referenčních stanic. Máme několik hlavních architektur pro umělé souřadnicové systémy, které slouží k předpovídání zpoždění mezi jednotlivými stanicemi uvnitř sítě. Mezi prvními myšlenkami spojovanými se síťovými souřadnicovými systémy byl návrh algoritmu GNP (Global Network Positioning) [4]. Ten představuje model Internetu jako 2-rozměrného geometrického prostoru. Stanice počítá svoji polohu v tomto prostoru, která charakterizuje její pozici v Internetu. Podstatou tohoto řešení je, že každá stanice odvozuje a mapuje svoji vlastní polohu v geometrickém prostoru a používá malý soubor vzorových vzdáleností, aby aktuální

síťová vzdálenost mohla být počítána jako funkce bodové geometrické vzdálenosti. Možné aplikace tohoto síťového souřadnicového systému jsou:

1. Peer - to - peer (P2P) sítě pro sdílení souborů a aplikace pracující s komunikací typu multicast: informace o souřadnicích mohou být použity tak, aby umožnily stanicím stahovat soubory z „nejbližšího“ peer bodu, který má kopii souboru. V mnoha těchto aplikacích je každá stanice logicky připojena k malému počtu jiných účastníků pro vytvoření překrývající se síťové struktury. Komunikace mezi uzly obvykle následuje logické linky. Umělé souřadnicové systémy mohou pomoci zlepšit úspěšnost při vyhnutí se logickým linkám s vysokou hodnotou RTT.
2. Distribuční služby: souřadnice mohou být použity u služeb, které přímo přesměrovávají zákazníka k „nejbližšímu“ obsahovému serveru, aby se minimalizovalo zpoždění.



Obr. 1.2: Propojení členů v P2P sítích

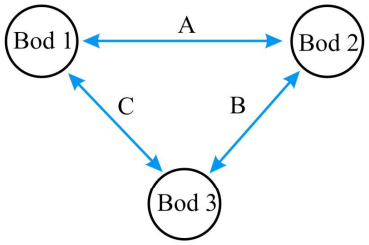
1.3 Požadavky na souřadnicové systémy

Základním požadavkem při návrhu umělého souřadnicového systému je, aby navrhovaný systém popisoval pozice stanic v Internetu s minimální chybou a byl schopen vystihnout všechna specifika dané sítě a nabízel dostatečný prostor pro rozmístění všech stanic. Dále si musí poradit s těžkostmi představovanými internetovým směrováním, dobou přenosu a řazení do front. Systém musí být určen pro velké množství stanic. Preferována je decentralizovaná implementace, která je spolehlivější vůči výpadkům a koresponduje s distribuovanou formou většiny aplikací. Při komunikaci je kladen důraz na minimální zatížení provozu, ideální umělý souřadnicový systém by neměl představovat žádnou dodatečnou zátěž pro síťový provoz a všechny informace by měl být schopen získat z existující komunikace mezi aplikacemi. Důležitým požadavkem je schopnost přizpůsobit se změnám sítě, které mohou nastat například z důvodu zvýšení provozu nebo rekonfigurace sítě. Systém by měl být schopný nastavit souřadnice stanic a pravidelně reagovat na tyto změny. Volba správného souřadnicového systému má významný vliv na

přesnost predikce, neboť při špatné volbě souřadnicového systému může dojít k tomu, že ani ověřený algoritmus nebude schopen predikovat souřadnice stanic s požadovanou přesností.

1.4 Souřadnicové systémy pro bezdrátové sítě

Pro bezdrátové sítě se používají souřadnicové systémy jako AFL (Anchor-Free Distributed Localization) [6] nebo ABC [7]. Jedná se o decentralizované systémy, které se částečně odlišují od systémů, jako je Vivaldi. Specifická vlastnost bezdrátových sítí je, že zpoždění nebo fyzická vzdálenost odpovídá u těchto systémů vzdálenosti dané zeměpisnou polohou. Například systém AFL je založen na principu potencionální energie a je používán pro Cricket senzorové systémy. Cricket senzorové systémy [7] užívají ultrazvukové signály na měření vzdáleností mezi senzory, které se následně používají při určení souřadnic. Vzdálenost d je určena jako podíl rychlosti v šíření ultrazvukového signálu v prostoru a času t potřebného k přenosu signálu mezi dvěma senzory. Platí, že $d = v \times t$. Další systém ABC je založen na práci se známými souřadnicemi z fixního souboru základních stanic tvořících jádro sítě. Každá stanice pak může zjistit svoje umístění pomocí šířených souřadnic používáním techniky trojúhelníkové nerovnosti, viz obr. 1.1.



Z trojúhelníkové nerovnosti platí:
A	<=	B	+	C
B	<=	A	+	C
C	<=	A	+	B

Obr. 1.3 Trojúhelníková rovnost

2 Algoritmy pro určení polohy stanic v síti a predikci zpoždění

2.1 Centralizované algoritmy

Umělé souřadnicové systémy využívající centralizované algoritmy vyžadují pro svoji činnost pevnou infrastrukturu, která zahrnuje centralizovanou komponentu (sadu fixních referenčních stanic), nebo požadují globální znalost stanic. Fixní referenční stanice představují jednu skupinu uzlů celého systému a vytvářejí pevný referenční rámec pro druhou skupinu uzlů, kterou tvoří stanice připojující se do sítě a využívající službu odhadu vzdáleností. Tyto systémy byly koncipovány jako celosvětová služba, podobně jako DNS (Domain Name System) nebo NTP (Network Time Protocol). Nevýhodou těchto systémů je, že nevyužívají veškerou komunikaci mezi stanicemi, ale pro aktualizace souřadnic každé stanice v prostoru má význam pouze měření a komunikace s fixními referenčními uzly. Mezi tyto systémy patří například GNP, který jako jeden z prvních systémů demonstroval možnost vypočítat umělé souřadnice tak, aby předpovídaly korektně hodnotu zpoždění. GNP [4] je založen na malém počtu (5 - 20) fixních referenčních uzlů (landmarks), další uzly si vypočítají svoje souřadnice na základě měření zpoždění k fixním referenčním uzlům. Fixní referenční uzly získávají na základě vzájemné komunikace vzdálenost ke všem ostatním fixním referenčním uzlům a na základě získaných informací si vytvoří plnou matici vzdáleností $\mathbf{L} = \mathbf{L}_{i,j}$, kde $i, j = 1..N$. Parametr N představuje počet všech fixních referenčních uzlů. Následně je každému fixnímu referenčnímu uzlu přiřazen bod $\boldsymbol{C}_i$ v 2-rozměrném souřadnicovém prostoru. Konečné přiřazení je provedeno jako výsledek minimalizace chyby:

$$E = \sum_i \sum_j \varepsilon(\mathbf{L}_{i,j}, |\, \mathbf{C}_i \mathbf{C}_j \,|) \quad , \tag{1}$$

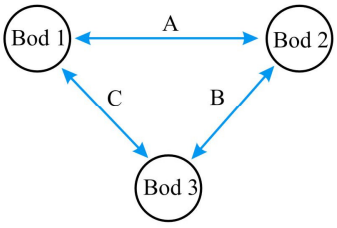
kde $\varepsilon(\mathbf{L}_{i,j}, |\, \mathbf{C}_i \mathbf{C}_j \,|)$ představuje chybovou funkci např. kvadratickou chybu. Hodnota funkce $|\, \mathbf{C}_i \mathbf{C}_j \,|$ představuje hodnotu zpoždění mezi dvěma fixními referenčními uzly o souřadnicích $\mathbf{C}_i$ a $\mathbf{C}_j$ ve zvoleném souřadnicovém prostoru vyjádřenou jejich geometrickou vzdáleností. Každý uzel využívající tyto služby postupuje stejným způsobem. Nejprve změří vzdálenost k několika fixním referenčním uzlům, získá jejich souřadnice a na základě těchto údajů určí svoje vlastní souřadnice v systému. Odhad zpoždění k libovolnému uzlu se pak určí jako přímá vzdálenost mezi těmito uzly ve zvoleném souřadnicovém prostoru. Toto měření představuje pouze velice malé zatížení sítě, neboť se jedná o malý počet fixních referenčních uzlů. Přesnost předpovědi zpoždění u tohoto systému je významně ovlivněna počáteční volbou fixních referenčních uzlů. Dalším představitelem těchto systémů je například systém NPS (Network Positioning System) [4], který představuje jednu z verzí GNP. NPS zahrnuje hierarchický systém pro snižování zátěže na mezních uzlech, strategie pro zmírnění efektu záludných uzlů a práce s NAT (Network Address Translation). Záludné uzly představují stanice v síti, které mohou být do sítě nasazeny úmyslně útočníkem s cílem poškození správné funkčnosti celého systému. NPS také demonstruje, jak se souřadnicový systém založený na fixních referenčních uzlech může přizpůsobit změně stavu sítě během pravidelné komunikace s fixními referenčními uzly a přepočítat souřadnice. Lighthouse [8] je rozšíření GNP, stejně jako GNP používá zvláštní sadu

referenčních uzlů. Rozdílný je však postup při začlenění nové stanice do systému. Stanice, která se připojuje k síti, se nemusí ptát na globální referenční uzly. Namísto toho se nová stanice může ptát nějakého z nynější sady uzlů, aby našla svoje souřadnice vztažené k této sadě. Následně transformuje tyto souřadnice do souřadnic vztaženým ke globálním referenčním uzlům. Mezi další představitele centralizovaných systémů patří systém IDMaps (Internet Distance Map Service) [9]. Tento systém vytváří síť serverů označovaných jako tracery a servery označované jako Hops. Hops servery na základě informací získaných od traceru poskytují vlastní službu odhadu zpoždění. Zpoždění mezi dvěma uživateli v síti je pak dána jako součet vzdáleností: vzdálenost stanice dotazujícího se uživatele k nejbližšímu traceru, vzdálenost stanice druhého uživatele k nejbližšímu traceru a vzdálenost mezi těmito tracery. Zatížení sítě bylo sníženo tím, že pravidelné měření bylo delegováno na podstatně menší počet k tomu určených serverů (traceru). Umístěním traceru ke každé síti (adresnímu prostoru AP), nebo skupině sítí (autonomnímu systému AS) lze dosáhnout významné redukce počtu měření. Obě varianty mají své klady a zápory. V případě autonomních systémů je výhodou jejich malý počet. Nevýhodou je, že se značně liší svým rozsahem a rozložením. V případě adresního prefixu se nám významně snížil počet uzlů, ale zůstává však velmi vysoký. Standardně jsou tracery v systému spojeny s několika AP a navzájem mezi sebou pomocí tzv. virtuálních spojů. K dosažení požadované přesnosti odhadu zpoždění musí být tracery umístěny v souladu s topologií sítě Internet.K umístění traceru se používá optimalizační úloha využívající algoritmus k-HST a minimum K-center z teorie grafů [10]. K dostatečné přesnosti odhadu při správném rozmístění traceru stačí, aby jejich podíl byl 0,2 % ze všech uzlů v síti Internet.

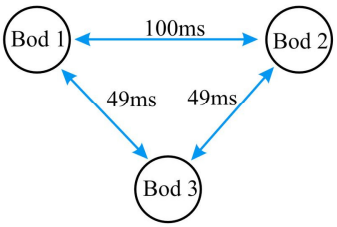
2.2 Decentralizované algoritmy

Umělé souřadnicové systémy založené na decentralizovaných algoritmech, na rozdíl od centralizovaných algoritmů, nespoléhají na explicitně určené součásti infrastruktury. Nevyžadují, aby nějaké stanice v systému působily jako fixní referenční uzly. Příklady takových systémů jsou Vivaldi [11] nebo PIC (Practical Internet Coordinates) [12]. U těchto systémů může každá stanice fungovat jako referenční uzel pro libovolné množství dalších stanic. Přesnost a stabilita těchto umělých souřadnicových systémů je založena na předpokladu, že množina referenčních uzlů, na kterých se umělé souřadnice počítají, správně spolupracuje v systému. Důležitou vlastností těchto systémů je velká dynamičnost, kdy dokážou rychle reagovat na změny v systému, například připojování a odpojování stanic. Většina z navržených distribuovaných souřadnicových systémů dosahuje chyby při předpovědi zpoždění méně než 10 % jako Vivaldi [11]. Takové přesnosti předpovědi lze dosáhnout u souřadnicových systémů založených na centralizovaných algoritmech již pomocí malé sady fixních referenčních uzlů. U distribuovaných systémů je situace mnohem složitější, protože žádná stanice nemůže působit jako fixní referenční uzel pro všechny další stanice v systému. Tento fakt má za následek u souřadnicových systémů založených na decentralizovaných algoritmech zvýšenou zranitelnost na útok zevnitř vedený útočníky, kteří se do tohoto systému infiltrují. Právě nedostatečná ochrana proti útokům zevnitř je významnou překážkou v rozšíření a většího nasazení těchto systémů v praxi. Například u algoritmu Vivaldi byla prokázána náchylnost k různým formám útoku. Některé provedené studie [11, 13] ukázaly, že když 30 % stanic lže o jejich souřadnicích, tak

přesnost algoritmu Vivaldi je snížena natolik, že systém je již zcela nepoužitelný. Když se útočníci předem domluví, tak již 5 % záludných uzlů má značný dopad na správnou funkčnost systému. Algoritmus Vivaldi používá každá stanice a získává tak svoje souřadnice a následně může fungovat jako referenční uzel pro další příchozí stanice. Použití zahrnuje nepřetržité ověřování chyb přesnosti souřadnic dané stanice. Tento přístup pomáhá přizpůsobit se změnám stavu sítě, nebo části sítě. Při vyvíjení algoritmu Vivaldi bylo zjištěno, že ve variantě s konstantním časovým krokem může při zvolení větší hodnoty dojít k oscilaci souřadnic. Tento problém odstraňuje částečně varianta s proměnným časovým krokem. PIC je podobný algoritmu Vivaldi. Vzhledem k tomu, že PIC používá na uzlu *Simplexní minimalizační proceduru* [12], nezdá se být vhodný v systémech s dynamicky se měnící situací. Výhodou PIC oproti Vivaldi je, že brání bezpečnost souřadnicového systému proti záludným uzlům používáním testu založeného na trojúhelníkové nerovnosti, viz obr. 2.1. Vivaldi se brání pouze vysoce chybovým uzlům.



Z trojúhelníkové nerovnosti platí:
A	<=	B	+	C
B	<=	A	+	C
C	<=	A	+	B



V reálné síti dochází k porušení této nerovnosti:
|100|<=|49|+|49|
|100|<=|98|

Obr. 2.1: Porušení trojúhelníkové rovnosti v reálných sítích

2.3 Systémy založené na potencionální energii

Několik systémů užívá princip potencionální energie pružiny pro nalezení konfigurace s minimální energií. Použití principu potencionální energie bylo u algoritmu Vivaldi inspirováno algoritmem na znovuobnovení trojrozměrného modelu povrchu ze souboru neorganizovaných bodů, který popsal ve své práci Hugues Hoppea [14]. Představil sílu (tuhost) pružiny jako průvodce pro optimalizaci rekonstruovaného povrchu. Princip potencionální energie pružiny se používá pro tvorbu grafických 2D prezentací komunikačních cest. Výsledky uvolnění nejsou použity na předpověď zpoždění nebo cesty komunikace. Shavitt a Tankel navrhli systém „velkého třesku“ [15] simulující potenciální pole podobné hmota-pružina u algoritmu Vivaldi. Jejich simulační model momentu hybnosti každé částice představuje tření. Cílem simulace je konvergence k stabilnímu stavu. Systém „velkého třesku“ je komplikovanější než Vivaldi a požaduje globální znalost systému.

3 Algoritmus Vivaldi

3.1 Základní popis algoritmu

Algoritmus Vivaldi představuje decentralizovaný umělý souřadnicový systém. Samotný algoritmus existuje ve variantách s konstantním nebo proměnným časovým krokem. Tento algoritmus může být implementován do různých *n*-rozměrných souřadnicových prostorů. V dalším textu bude vysvětlena volba 2-rozměrného prostoru s třetím parametrem výškou. Princip algoritmu Vivaldi spočívá v tom, že přiřazuje každé stanici umělé souřadnice v souřadnicovém prostoru. Souřadnice se pokouší přiřadit tak, aby vzdálenost mezi dvěma stanicemi v tomto prostoru přesně předpovídala zpoždění mezi nimi. Vzhledem k tomu, že zpoždění v síti je dáno zpožděním v páteřní a přístupové části, kde porušuje trojúhelníkovou nerovnost, není možné pro přesný souřadnicový systém použít 2-rozměrný souřadnicový prostor. Algoritmus se pokouší najít souřadnice, které minimalizují chybu předpovědí. Velikost kvadratické chyby v souřadnicovém systému je určena následujícím vzorcem [11]:

$$E = \sum_{i} \sum_{j} (\mathbf{L}_{i,j} - \| \mathbf{x}_i - \mathbf{x}_j \|)^2 , \tag{2}$$

kde $\mathbf{L}_{i,j}$ je aktuální zpoždění mezi uzly *i* a *j*, $\mathbf{x}_i$ je souřadnice přiřazená uzlu *i* a $\mathbf{x}_j$ je souřadnice přiřazená uzlu *j*. Základem algoritmu Vivaldi je jednoduchý centralizovaný algoritmus, který může minimalizovat rovnici 2. Tento algoritmus postupně upravuje souřadnice jednotlivých stanic s cílem minimalizovat kvadratickou chybu *E*. Vivaldi představuje distribuovanou verzi tohoto algoritmu. Chybová funkce, kterou chceme minimalizovat, je dána sumou energií přes všechny skoky, které jsou provedeny mezi jednotlivými stanicemi v souřadnicovém prostoru. Vzhledem k tomu, že kvadratická chybová funkce je rovna energii skoků, může být minimalizována při simulaci změny pohybu uzlů, která je dána potenciální energií pružin. Úskalím minimalizace této chyby je, že systém nemusí vždy najít správné řešení (globální minimum funkce), ale může skončit v lokálním minimu chybové funkce. Podrobnější popis centralizovaného algoritmu vychází z definice silového vektoru $\mathbf{F}_{i,j}$, který představuje sílu vynaloženou na uzlu *i* při skoku mezi uzly *i* a *j*. Z Hookeova zákonu můžeme ukázat, že $\mathbf{F}_{i,j}$ je [11]:

$$\mathbf{F}_{i,j} = (\mathbf{L}_{i,j} - \| \mathbf{x}_i - \mathbf{x}_j \|) \times \mathbf{u}(\mathbf{x}_i - \mathbf{x}_j) . \tag{3}$$

Skalární veličina ($\mathbf{L}_{i,j}$ - $\| \mathbf{x}_i - \mathbf{x}_j \|$) představuje posunutí z poslední pozice do nové pozice. Tato veličina představuje velikost síly vykonané skokem. Jednotkový vektor $\mathbf{u}(\mathbf{x}_i$ - $\mathbf{x}_j)$ udává směr síly na uzlu *i*. Výsledná síla na uzlu *i* ($\mathbf{F}_i$) je dána jako suma sil z dalších uzlů [11]:

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{i,j} . \tag{4}$$

Pro simulaci změny pozice algoritmus zvažuje malé časové intervaly. V každém intervalu algoritmus pohybuje každým uzlem o malou vzdálenost v souřadnicovém prostoru směrem $\mathbf{F}_i$ a potom přepočte všechny síly. Nové souřadnice uzlu *i* na konci časového intervalu jsou [1]:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{F}_i \times t \,, \tag{5}$$

kde *t* je délka časového intervalu. Velikost parametru *t* určuje, jak daleko se uzel pohybuje v každém časovém intervalu. Nalezení vhodného *t* je obzvlášť důležité při implementaci algoritmu Vivaldi. Pseudokód představující centralizovaný algoritmus [11] je zobrazen na obr. 3.1. Dokud systém konverguje k souřadnicím, které minimalizují kvadratickou chybu, probíhá výpočet v cyklu. Nejprve se pro každý uzel *i* v systému vypočítá síla na každý skok připojený

Vstup: matice zpoždění a počátečních souřadnic
Výstup: mnohem přesnější souřadnice x
Počítání souřadnice ($\mathbf{L}_{i,j}$, $\mathbf{x}_i$, ε*)*
***while** (chyba (*$\mathbf{L}_{i,j}$, $\mathbf{x}_i$*) >* ε*)*
***opakovat pro každé** i*

$$\mathbf{F}_i = 0$$

***opakovat pro každé** j*
Výpočet chyby/síly skoku. (1)

$$e = \mathbf{L}_{i,j} - \| \mathbf{x}_i - \mathbf{x}_j \|$$

Přičte vektor síly skoku k celkové síle. (2)

$$\mathbf{F}_i = \mathbf{F}_i + e \times \mathbf{u}(\mathbf{x}_i - \mathbf{x}_j)$$

Změna souřadnice bodu o malý krok ve směru vektoru síly (3)

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{F}_i \times t$$

Obr. 3.1 Pseudokód představující centralizovaný algoritmus

k uzlu *i* (řádek 2). Následně se tato síla přičte k celkové síle na uzlu *i* (řádek 3). Poté, co jsou všechny síly sečteny, se uzel *i* pohybuje malou vzdáleností ve směru síly $\mathbf{F}_i$ (řádek 3). Centralizovaný algoritmus počítá novou souřadnice pro všechny uzly s definovaným zpožděním. Vivaldi představuje rozšíření tohoto algoritmu. Rozšíření spočívá v tom, že každý uzel *i* spojitě vypočítá a nastaví svoje souřadnice založené jen na měření zpoždění od sebe k několika dalším uzlům a zjištění informací o jejich aktuální poloze v systému. V algoritmu Vivaldi každý uzel simuluje svůj vlastní pohyb v skokovém systému. Každý uzel udržuje informace o vlastních aktuálních souřadnicích, počínaje původními souřadnicemi, které může být přiděleny náhodně, nebo dle libovolného klíče. Kdykoliv uzel komunikuje s dalším uzlem, tak měří hodnotu zpoždění k tomuto uzlu a také se učí jeho aktuální souřadnice. Z těchto uzlů si následně vytváří vlastní referenční seznam. Každý uzel v systému komunikuje s dalšími uzly a neustále upravuje svoji polohu na základě tohoto algoritmu. Když se uzel *i* se souřadnicemi $\mathbf{x}_i$ dozví o uzlu *j,*

o souřadnicích $\mathbf{x}_j$ a získá hodnotu zpoždění mezi nimi, tak následně aktualizuje svoje souřadnice do nové polohy dle následující rovnice [11]:

$$\mathbf{x}_i = \mathbf{x}_i + \delta \times (rtt - \| \mathbf{x}_i - \mathbf{x}_j \|) \times \mathbf{u}(\mathbf{x}_i - \mathbf{x}_j) . \quad (6)$$

Toto pravidlo je identické k individuálním silám vypočítaným ve vnitřním cyklu centralizovaného algoritmu. Protože všechny uzly startují ze stejné pozice, tak musí dojít k jejich oddělení. Pokud je normála vypočtena na základě aktuální polohy rovna nule, tak se musí zvolit jednotkový vektor v náhodném směru. Dva uzly zabírající stejné umístění budou skokem tlačeny od sebe v jakémkoli směru.

Uzel i změří RTT k uzlu j a zjistí souřadnice tohoto uzlu
simple_vivaldi(*RTT, $\mathbf{x}_j$*)
Vypočítá se chyba tohoto vzoru. (1)
$$e = RTT - \| \mathbf{x}_i - \mathbf{x}_j \|$$
Najde se směr působící síly. (2)
$$\mathbf{dir} = \mathbf{u}(\mathbf{x}_i - \mathbf{x}_j)$$
Vektor síly je úměrný chybě. (3)
$$\mathbf{F}_i = \mathbf{dir} \times e$$
Dojde ke změně souřadnice uzlu malým krokem ve směru síly **F**. *(4)*
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{F}_i \times \delta$$

Obr. 3.2: Pseudokód představující distribuovaný algoritmus

Metoda *simple_vivaldi( )* je volána vždy, když je k dispozici nově změřené zpoždění ke vzdálenému uzlu a jsou zjištěny jeho aktuální souřadnice. Nejprve se vypočítá chyba na základě polohy aktuálního uzlu *i* a polohy referenčního uzlu *j* při znalosti zpoždění (řádek 1). Následně se určí velikost a směr síly, kterou se bude uzel pohybovat (řádek 2 a 3). Nakonec se uzel pohybuje z aktuální polohy do nové polohy směrem a velikostí síly $\mathbf{F}_i$ při používání konstantního časového kroku δ. Uzly systému neustále aktualizují svoji polohu, dokud nedojde k ustálení systému.

3.2 Volba časového kroku

Hlavním problémem v zavádění algoritmu Vivaldi je zajištění konvergence do souřadnic, které správně předpovídají zpoždění. Rychlost konvergence podléhá velikosti časového kroku δ. Velké hodnoty způsobí rychlou konvergenci, ale může dojít i k nastavení souřadnice ve velkém schodku. Pokud všechny uzly užívají velké hodnoty časového kroku δ, nemusí systém zkonvergovat. Namísto ustáleného stavu, který dobře předpovídá zpoždění, dojde ke kmitání a selhání konvergence k přesným souřadnicím. Velká hodnota δ způsobí uzlům, že přeskakují mezi nízkými energetickými hladinami, do kterých se však kvůli volbě δ nemohou dostat. Další

problém souvisí s tím, jak si algoritmus poradí s uzly, které mají vysokou chybu ve svých souřadnicích. Tyto uzly mohou výrazně zvýšit nepřesnost predikce. Tato slabost by mohla být využita k útoku na aplikace využívající tento systém. Když uzel *i* komunikuje s jiným uzlem *j*, který má souřadnice, které předpovídají zpoždění špatně, tak každá aktualizace pozice uzlu *i* založená na těchto souřadnicích způsobí, že se zvýší chyba předpovědi namísto, aby došlo k jejímu snížení. Proměnný časový krok δ byl zaveden z důvodu požadavku rychlé konvergence a vyhnutí se četnému kmitání. Proměnný časový krok δ závisí na tom, jak si je uzel jistý svými souřadnicemi. Když se uzel učí svoji polohu, tak je odhad jeho místa v sítí značně nepřesný (například, když se uzel poprvé připojuje). V tomto případě se nastavuje vysoká δ, která způsobí, že se uzel pohybuje rychle k přibližně správné pozici. Čím více se uzel přibližuje ke správné pozici v prostoru, tím se zmenšuje proměnný časový krok δ. Malá δ slouží k přesnému nalezení pozice. Jednoduchá přizpůsobivá δ může užívat stálý zlomek (c_c < 1) uzlové odhadované chyby [11]:

$$\delta = c_c \times e_i , \tag{7}$$

kde e_i představuje hodnotu chyby daného uzlu *i*. δ představuje zlomek cesty uzlu, který je povolen k pohybu vůči ideální pozici pro aktuální vzorek. Jestliže předpovídaná chyba uzlu [1] je mezi $\pm$ 5 %, pak se nebude pohybovat víc než 5 % směrem vůči správné pozici. Naopak jestliže je jeho chyba velká $\pm$ 100 %, pak se bude pohybovat rychle celou cestu do správné pozice. Problém s nastavením δ je, aby předpověď chyby vzala v úvahu také přesnost souřadnic vzdáleného uzlu. Pokud se přesnost souřadnic vzdáleného uzlu pohybuje kolem $\pm$ 50 %, pak by měla být dána menší spolehlivost než při vzdáleném slabém uzlu s přesností na $\pm$ 5 %. Vivaldi používá pro výpočet časového kroku δ následující vzorec [11]:

$$\delta = c_c \times \frac{e_i}{e_i + e_j}, \tag{8}$$

kde e_i představuje hodnotu chyby daného uzlu *i* a e_j představuje hodnotu chyby druhého uzlu *j*, vůči kterému aktualizuje svoji aktuální polohu. Použití této δ zaručí, že přesný uzel po komunikaci s nepřesným uzlem se bude pohybovat pouze málo. Naopak nepřesný uzel po komunikaci s přesným uzlem se bude pohybovat více. Počítání proměnného časového kroku δ tímto způsobem poskytuje vlastnosti, které si přejeme: rychlá konvergence, nízké kmitání a odolnost proti vysoce chybovým uzlům. Při implementaci proměnného časového kroku δ musí mít každý běžící uzel odhad, jak přesné jsou jeho souřadnice.

3.3 Algoritmus s proměnným krokem

Pseudokód této varianty algoritmu [11] je vidět na obr. 3.3. Metoda *vivaldi (rtt,* $\mathbf{x}_j$*,* e_j*)* počítá váhu vzoru založenou na chybě vlastních souřadnic a chybě souřadnic vzdáleného bodu (řádek 1). Algoritmus musí také sledovat místní relativní chybu. To dělá používání váženého klouzavého průměru (řádky 2 a 3). Zbytek algoritmu Vivaldi je identický s jednoduchou verzí. Vivaldi s proměnným krokem δ je opět plně distribuovaný a procedura běží na každém uzlu

v síti. Vzhledem k tomu, že každý uzel neustále provádí aktualizaci své polohy na základě informací získaných z komunikace mezi uzly, dokáže se dynamicky přizpůsobit změnám v stavu sítě.

Získána nová informace, uzel i změřil potřebné RTT k uzlu j, získal informace o
Souřadnice uzlu j a odhad chyby její e_j
Znalost vlastní souřadnice $\mathbf{x}_i$ a odhadu chyby e_i.
Konstanty používané jako ladící parametry c_e a c_c
vivaldi (rtt, $\mathbf{x}_j$, e_j)
Výpočítat váhu chyb. (1)

$$w = e_j /(e_i + e_j)$$

Vypočítat relativní chybu tohoto vzoru. (2)

$$e = |\| \mathbf{x}_i - \mathbf{x}_j \| - rtt | / rtt$$

Úprava váženého klouzavého průměru místní chyby. (3)

$$e_{i+1} = e_s \times c_e \times w + e_i \times (1 - c_e \times w)$$

Aktualizace souřadnice uzlu. (4)

$$\delta = c_c / w$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \delta \times (rtt - \| \mathbf{x}_i - \mathbf{x}_j \|) \times \mathbf{u}(\mathbf{x}_i - \mathbf{x}_j)$$

Obr. 3.3: Pseudokód algoritmu Vivaldi s proměnným krokem

3.4 Volba umělého souřadnicového prostoru

Algoritmus Vivaldi si může vybrat libovolnou strukturu souřadnic a metriku, která určuje předpovídané zpoždění daných dvou souřadnic. Platí však, že souřadnice by měly být kompaktní a mělo by být snadné vypočítat zpoždění předpovídané pro dané dvě souřadnice. Nejjednodušší volba je použít *n*-rozměrné souřadnice se standardní euklidovskou metrikou. Další množnosti jsou pak kulovité, prstencové, hyperbolické a další koordinační struktury, které již byly navrhovány a testovány [13]. Tyto souřadnicové systémy užívají alternativní funkce pro výpočet vzdáleností v naději, že předpovídají zpoždění lepé. V následujících podkapitolách je podrobněji popsáno několik možných souřadnicových prostorů. V případě nasazení systému v síti Internet je základním požadavkem na souřadnicový prostor schopnost se vyrovnat s trojúhelníkovou nerovností, ke které v sítí dochází.

3.4.1 Euklidovský prostor

Pro tento souřadnicový prostor platí následující známé rovnosti:

$$\begin{aligned} [x_1,....,x_n] - [y_1,...,y_n] &= [x_1 - y_1,...,x_n - y_n], \\ \| [x_1,....,x_n] \| &= \sqrt{x_1^2 + ... + x_n^2}, \\ \alpha \times [x_1,....,x_n] &= [\alpha x_1,....,\alpha x_n]. \end{aligned} \tag{9}$$

V případě použití euklidovského souřadnicového prostoru je důležitá volba jeho rozměru. Několik studií [13] již prokázalo, že nemá význam použít složitější prostor než 2-rozměrný, nebo 3-rozměrný. 2-rozměrný souřadnicový prostor by byl zcela dostačující v případě, že zpoždění v síti by bylo dáno pouze zeměpisnou vzdáleností. Kdyby zeměpisná vzdálenost byla jediným faktorem ve zpoždění, tak by tento prostor byl zcela dostačující. Tento požadavek však porušuje fakt, že velká část zpoždění vzniká až v přístupové části sítě. Přidáním dalšího rozměru se přesnost zlepšuje pouze mírně, ale vzrůstá znatelně složitost výpočtů a požadavky na komunikaci.

3.4.2 Kulový prostor

Tento prostor se pokouší modelovat vzdálenosti mezi body na povrchu koule. Původní myšlenka souvisela s představou, že koule představuje Zemi a vzhledem k tomu by tento prostor měl být přesnější. Přesnost tohoto modelu je však srovnatelná s 2-rozměrným euklidovským prostorem. Během testů [15] bylo zjištěno, že v rozsáhlejších sítích je chybovost kulového prostoru naopak větší. Důvodem zvýšení této chyby je vnější faktor ovlivňující provoz v síti Internet. Bylo zjištěno, že skutečná síť "neomotává" svoje cesty skrz Internet jako kolem kulového prostoru. Při studiu internetových cest z východní Asie bylo například zjištěno, že pouze malá část spojení do Evropy jde přímo, ale z větší části je většina paketů směrována do Evropy směrem na východ. Kulovitý model však vybírá kratší cestu, která však v těchto případech představuje větší chybu při porovnání skutečného a předpovídaného RTT. Následujíc rovnice umožňuje výpočet vzdálenosti mezi dvěma body na povrchu koule:

$$\Delta\hat{\sigma} = \arccos(\cos\phi_s \cos\phi_f \cos\Delta\lambda + \sin\phi_s \sin\phi_f) \text{ [°]},$$

$$d = r \times \Delta\hat{\sigma}, \tag{10}$$

kde ϕ_s a ϕ_s představují geografickou délku, λ_s a λ_f představují geografickou šířku, r poloměr kružnice a *d* je vzdálenost mezi body.

3.4.3 2-rozměrný euklidovský prostor s výškou

Tento souřadnicový prostor vychází z 2-rozměrného euklidovského prostoru rozšířeného o další parametr výšku. Nejedná se však o klasický 3-rozměrný prostor, což je dáno rozdílným způsobem výpočtů vektorových funkcí v tomto prostoru, jak ukazuje obr. 3.4. V tomto prostoru platí pro operace s vektory rovnice v následující podobě:

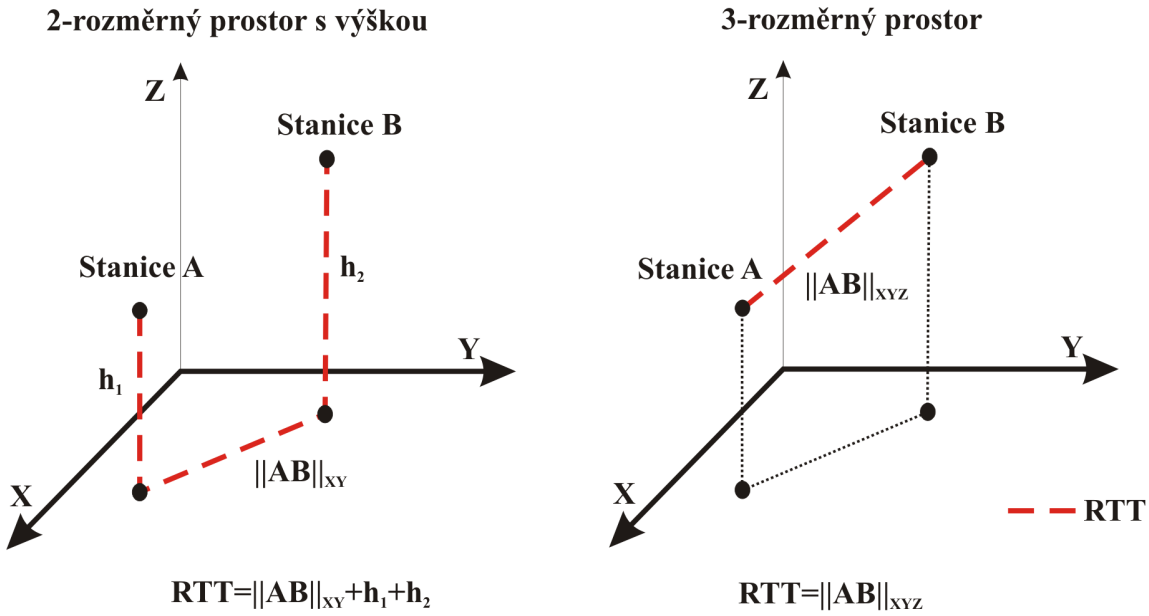
$$[x, x_h] - [y, y_h] = [(x - y), x_h + y_h],$$

$$\| [x, x_h] \| = \| x \| + x_h, \tag{11}$$

$$\alpha \times [x, x_h] = [\alpha x, \alpha x_h].$$

Euklidovská část modelu představuje páteřní síť se zpožděními úměrnými k zeměpisné vzdálenosti, zatímco výška představuje čas, který je potřebný k přenosu paketu v přístupové části

sítě. Příčinou zpoždění v přístupové části je většinou malá šířka přenosového pásma a s ní související možnost vzniku front nebo zahlcení. Hodnota předpovídaného zpoždění v tomto souřadnicovém prostoru je dána jako součet vzdáleností výšek obou uzlů nad rovinou, což představuje zpoždění v přístupové síti, a vzdálenosti jejich cesty v euklidovském prostoru. Toto je základní rozdíl mezi vektorem s výškou a euklidovskými prostory s *n*–rozměry. Každý uzel má pozitivní prvek výšku ve svých souřadnicích, kterou může dle potřeby měnit. Tento parametr slouží hlavně v případech, kdy se jeden bod nachází mezi více uzly, ale jeho vzdálenost od všech uzlů v klasickém euklidovském prostoru způsobuje, že se neustále pohybuje. V normálním euklidovském prostoru platí, že uzel, který se nalézá příliš blízko u dalšího uzlu, se od tohoto uzlu vzdálí. Ale uzel, který se nachází příliš blízko u uzlů na všech stranách, nemá kam jít a bude neustále kmitat dokola. Vektor s výškou však umožní, že uzel tlačen uzly ze všech stran upraví svoji výšku a tím se vyrovnají síly, které na něj působí. Již provedené zkoušky [11] dokazují, že vektor s výškou dosahuje větší přesnosti než 2-rozměrné a 3-rozměrné euklidovské souřadnicové systémy.



Obr. 3.4: Rozdíl mezi 2-rozměrným prostorem s výškou a 3-rozměrným prostorem

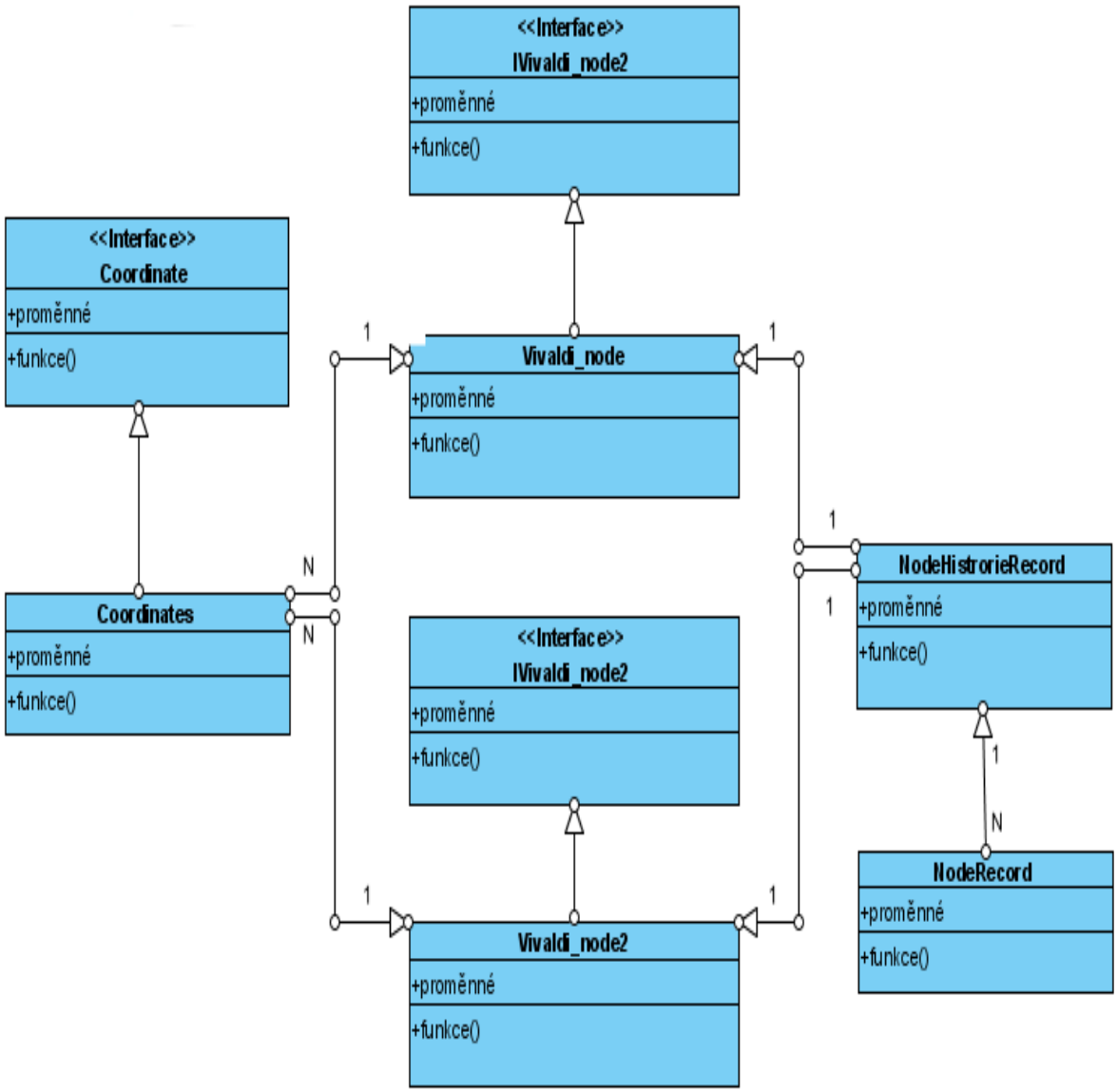
4 Vyvinutá knihovna implementující algoritmus Vivaldi

Celý simulační program, včetně mnou implementovaná knihovna Vivaldi-lib představující algoritmus Vivaldi, je navržen v prostředí moderního objektového programovacího jazyka JAVA. Prostředí JAVA bylo zvoleno hlavně proto, že poskytuje širokou podporu grafických nástrojů a knihoven, které jsou potřebné pro grafickou nástavbu celého simulačního programu a z důvodu nezávislosti na platformě. Jazyk JAVA a v něm vytvořené knihovny a programy lze spustit na libovolném operačním systému za předpokladu, že je na něm instalován interpret zkompilovaných javovských programů JVM (Java Virtual Machina). Vzhledem k tomu, že vytvořená knihovna implementující algoritmus Vivaldi bude sloužit pouze pro simulační systém a nebude nasazena v reálné síti nebo jako součást nějaké síťové aplikace, tak je jazyk JAVA velice vhodný vzhledem k přenositelnosti mezi různými operačními systémy. V případě, že by tato knihovna měla být používána v reálném síťovém provozu, tak by bylo výhodnější použít k implementaci programovací jazyk C. V tomto projektu je momentálně řešena implementace algoritmu Vivaldi s využitím 2-rozměrného prostoru s třetím parametrem výškou a to s konstantním a proměnným časovým krokem. Vzhledem k požadavku implementace dvou variant algoritmu s konstantním a proměnným časovým krokem, které však mají větší část algoritmu společnou, byla základní struktura programu vytvořena pomocí rozhraní, které slouží k definici společné části. Vytvořené třídy, které toto rozhraní implementují, se liší pouze v definici některých funkcí. Toto řešení umožňuje jednoduchou úprava programu, např. v případě požadavku na změnu souřadnicového prostoru. V příloze B je znázorněna implementace knihovny Vivaldi-lib do simulačního programu VIVALDIMONITOR [2], který ve spolupráci vytvořil Bc. Jaroslav Švéda.

4.1 Architektura knihovny

Obr. 4.1 představuje schéma struktury navržené knihovny, která implementuje obě varianty algoritmu Vivaldi v 2-rozměrném souřadnicovém systému s třetím parametrem výškou. Úplné schéma knihovny Vivaldi-lib je zobrazeno v příloze A. Knihovnu tvoří šest tříd a tři rozhraní. Dále jsou součástí této knihovny dvě třídy Tutorial_static a Tutorial_dynamic, které slouží pro základní testování knihovny a ukázku práce s objekty jednotlivých tříd. Jednotlivé třídy byly navrhovány vždy tak, aby umožnily jednoduchou změnu nebo rozšíření systému například do jiného souřadnicového systému, což je možné provést změnou definice jediné třídy. Základem této knihovny jsou dvě třídy Vivaldi_node a Vivaldi_node2, které představují samotný uzel sítě a implementují v něm danou variantu algoritmu Vivaldi. Obě tyto funkce implementují rozhraní Thread a jsou navrženy jako spustitelné. Implementace těchto tříd umožňuje nástavbovému simulačnímu software vytvořit a spouštět objekty těchto tříd na pozadí. Dále je tu třída Coordinates, jejímž hlavním úkolem je implementace vektorových funkcí pro zvolený 2-rozměrný souřadnicový prostor s výškou. V případě požadavku změny souřadnicového systému stačí provést pouze změny v této třídě. Další dvě třídy jsou navrženy pro ukládání a předávání informací nadstavbovému systému. Třída NodeRecord slouží k uložení aktuálního stavu uzlu sítě a nabízí funkce pro práci s informacemi o uzlu. Tato třída stejně jako třída NodeHistorieRecord je společná pro oba typy uzlů, jak s proměnným, tak konstantním časovým krokem. Třída

NodeHistorieRecord slouží k ukládání historie všech potřebných záznamů každého uzlu. Jednotlivé záznamy v této třídě jsou objekty třídy NodeRecord, které se vytvářejí vždy po provedení aktualizace pozice každého uzlu sítě.



Obr. 4.1: UML diagram knihovny implementující algoritmus Vivaldi

4.2 Definice a implementace jednotlivých tříd

4.2.1 Třída Coordinates

Třída Coordinates implementuje rozhraní Coordinate. Hlavním úkolem této třídy je zobrazení uzlu ve 2-rozměrném souřadnicovém prostoru s výškou a implementace funkcí potřebných pro práci s vektory. Popis rozhraní Coordinate a souhrn funkcí, které nabízí, je uveden níže včetně stručného popisu funkce na obr. 4.2.

```
public interface Coordinate

/*  Funkce getCoordinate vrací  vektor [x,y,h] aktuálních souřadnic bodu */
public double[ ] getCoordinate();

/* Funkce set Coordinate slouží k nastavení hodnot souřadnic, vstupním parametrem je vektor
souřadnic [x,y,h] */
public void setCoordinate(double point[]);

/*  Funkce plus provede součet dvou souřadnic (this + a), vrací vektor výsledného součtu
aktuálního vektoru a vektoru a */
public Coordinates plus(final Coordinates a);

/* Funkce minus provede rozdíl dvou souřadnic (this + a), vrací vektor výsledného rozdílu
aktuálního vektoru a vektoru a */
public Coordinates minus(final Coordinates a);

/*  Funkce leftDotProduct vrací vektor násobeny konstantou alpha zleva*/
public Coordinates leftDotProduct(double alpha);

/*  Funkce rightDotProduct vrací vektor násobeny konstantou alpha zprav*/
public Coordinates rightDotProduct(double alpha);

/* Funkce norm() vrací hodnotu normály vektoru*/
public double norm();

/* Funkce citary() provede výpočet a určení jednotkového vektoru (this/||this|| = this * 1/||this||)
vraci jednotkový vektor */
public Coordinates toUnitary();

/* Funkce calculateRTT vrací hodnotu RTT spočtenou mezi dvěma uzly v souřadnicovém
systému */
public double calculateRTT(Coordinates a);
```

Obr. 4.2: Definice rozhraní Coordinate

Samotná třída Coordinates definuje proměnné a metody potřebné pro reprezentaci a práci se souřadnicemi ve 2-rozměrném prostoru s výškou. Třída používá proměnné uvedené v tab. 4.1.

Tab. 4.1: Základní parametry třídy Coordinate

Název proměnné	Typ	Význam proměnné
coords [ ]	double	udržuje informaci o poloze
time		udržuje informace o čase, kdy byla souřadnice vytvořena
rtt		udržuje informaci o velikosti RTT

4.2.2 Třída NodeRecord

Objekty této třídy slouží k uložení informací o daném uzlu po každém provedení aktualizace polohy. Každý takový záznam se ukládá do proměnné NodeHistorieRecord, která udržuje všechny záznamy pro konkrétní uzel. Tab. 4.2 popisuje základní parametry této třídy včetně jejich významu.

Tab. 4.2: Základní parametry třídy NodeRecord

Název proměnné	Typ	Význam proměnné
coords [ ]	double	udržuje informaci o poloze
time		udržuje informace o čase, kdy byla souřadnice vytvořena
magnitude		slouží k uložení informace o aktuální velikosti silového vektoru
error		slouží k uložení hodnoty chyby mezi aktuální skutečnou a predikovanou hodnotou RTT
delta		udržuje aktuální hodnotu časového kroku
ei		udržuje aktuální hodnotu vlastní chyby
w		udržuje aktuální hodnotu váhového vektoru (používá se pouze u varianty s proměnným časovým krokem)

4.2.3 Třída NodeHistorieRecordrecord

Objekty této třídy slouží každému uzlu pro zaznamenání všech stavů, ve kterých se každý uzel nacházel. Tato třída obsahuje proměnné, viz tab. 4.3. Třída obsahuje pouze jedinou funkci *getLastRecord( )*.Tato funkce vrací poslední vložený záznam. Vzhledem k tomu, že každý uzel běží jako nezávislé vlákno a do seznamu neustále ukládá po každém výpočtu nový záznam na stejnou pozici, ze které tato funkce tento záznam předává nástavbové aplikaci, bylo nutné z důvodu bezpečnosti přístupu tuto metodu zvolit jako synchronizovanou.

Tab. 4.3: Základní parametry třídy NodeHistorieRecordrecord

Název proměnné	Typ	Význam proměnné
name [ ]	double	udržuje jméno uzlu
historiesPoints	Vector<NodeRecord>	udržuje záznamy o všech stavech uzlu

4.2.4 Třída Vivaldi_node

Třída Vivaldi_node implementuje rozhraní IVivaldi_node. Popis rozhraní IVivaldi_node a souhrn funkcí které nabízí, je uveden včetně stručného popisu funkcí na obr. 4.3. Objekt této třídy představuje uzel sítě, na kterém běží samotný algoritmus Vivaldi s konstantním časovým krokem. Objekt této třídy nabízí dva typy konstruktorů. První konstruktor nabízí možnost vytvořit statický objekt. Jedná se o uzel, který zůstává trvale na předem definovaném místě v prostoru a slouží pouze jako referenční uzel pro libovolné aktivní uzly. Tento uzel není spuštěn jako samostatné vlákno. Druhý konstruktor vytváří dynamický objekt, který je již spouštěn v podobě samostatného vlákna a běží na něm algoritmus Vivaldi. Dynamické objekty představují reálné uzly sítě a používají se pro simulace.

```
public interface IVivaldi_node {
   /*funkce addPoint (Vivaldi_node a)
     * tato funkce slouží k přidáni nového bodu do seznamu referenčních
     * bodů a automaticky spočítá RTT jako vzdálenost mezi souřadnicemi
     * nového bodu vůči koncovému bodu (endPoint)*/
   public void addPoint( Vivaldi_node a);

   /*funkce addPoint (Vivaldi_node a, Double rtt)tato funkce slouží k
     přidáni nového bodu do seznamu referenčních bodu a nastaveni
     explicitního RTT*/
   public void addPoint( Vivaldi_node a, Double rtt);

   /*funkce removePoint(int a)tato funkce slouží k odstranění referenčního
     bodu ze seznamu referenčních bodů na pozici "a" a automatickému
     zrušení informace o RTT */
   public void removePoint (int a);

   /*funkce updateRTT(int a, Double rtt)tato funkce slouží k změně hodnoty
     RTT vůči danému referenčnímu bodu dojde k přepsání hodnoty RTT na
     patřičné pozici v seznamu referenstRTT */
   public void updateRTT (int a, Double rtt);

   /*Funkce execute() představuje implementaci algoritmu Vivaldi
     * Vstupními parametry jsou skutečná hodnota RTT vůči referenčnímu
     * bodu a jeho souřadnice. Výsledkem této metody je úprava aktuální
     * pozice a zapsání informace do historie daného bodu */
   public void execute(double rtt, final Coordinates x_j);
```

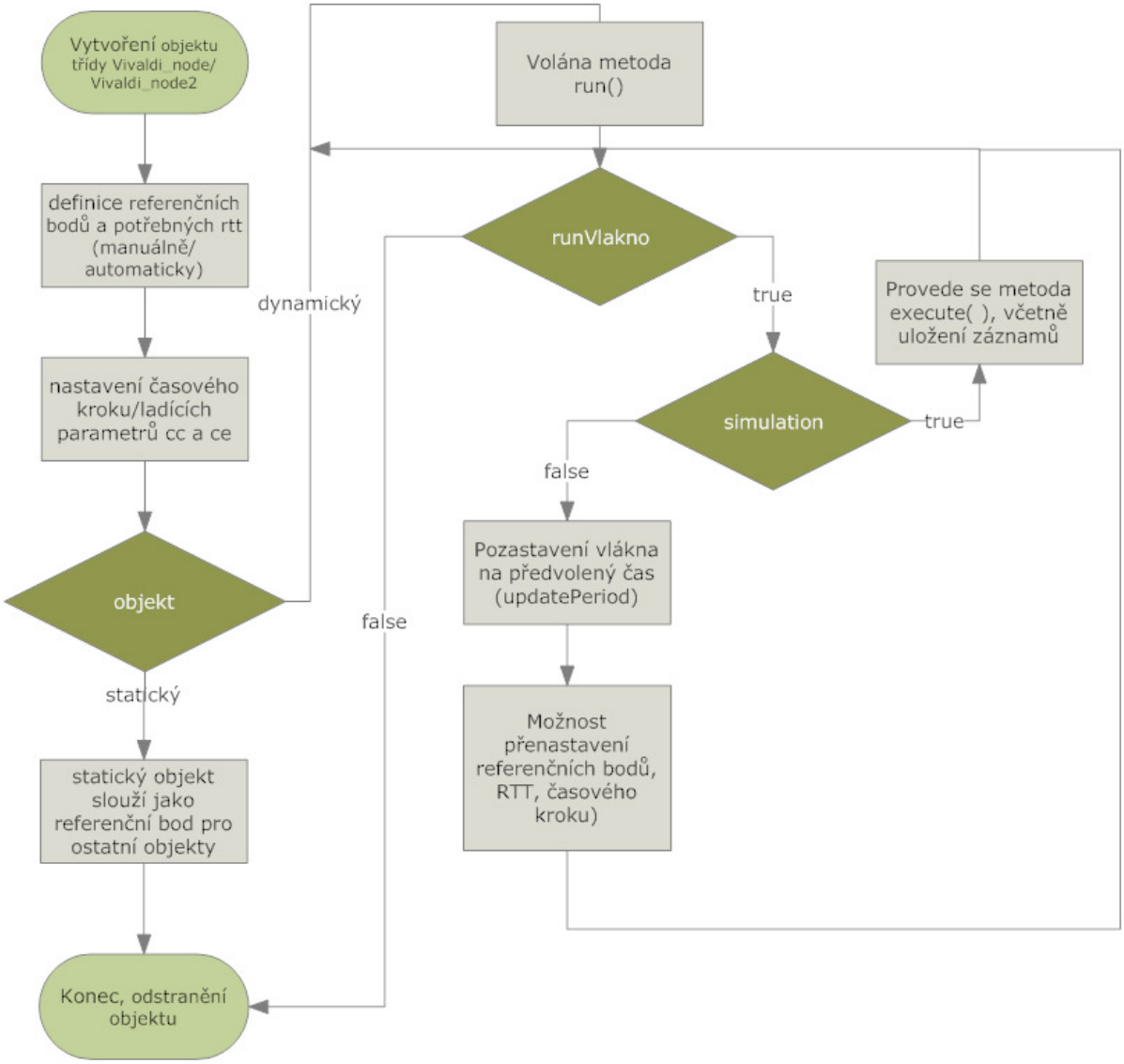
Obr. 4.3: Definice rozhraní IVivaldi_node

Nejdůležitější parametry třídy Vivaldi_node včetně popisu jejich významu jsou uvedeny v tab. 4.4. Samotný zdrojový kód třídy Vivaldi_node je uveden v příloze C.

Tab. 4.4: Základní parametry třídy Vivaldi_node

Název proměnné	**Typ**	**Význam proměnné**
count	int	udržuje počet instancí vláken
point	coordinates	udržuje aktuální polohu uzlu
endPoint		udržuje polohu koncového uzlu
delta	double	udržuje aktuální hodnotu časového kroku
name	string	udržuje aktuální hodnotu váhové vektoru
simulation	boolean	Určuje, kdy běží simulace
runVlakno		určuje platnost metody run( )
record	NodeHistorieRecord	udržuje informace o referenčních bodech
referentsPoints	Vector<Vivaldi_node>	udržuje informace o příslušných známých bodech
referentsRTT	Vector<Double>	udržuje informace o příslušných hodnotách RTT bodu vůči známým referenčním bodům

Činnost třídy Vivaldi_node a princip funkce metody run( ) je popsán vývojovým diagramem, viz obr. 4.4. Vzhledem k tomu, že se jedná o decentralizovaný algoritmus, tak každý uzel, jenž je představován objektem třídy Vivaldi_node, běží jako nezávislé vlákno. Vzhledem k tomu, že JAVA neumožňuje vlákna zastavit, tak vlákno běží, dokud není zrušen celý objekt. K zastavení činnosti vlákna po ukončení simulace je použit časovač, který vlákno uspí po stanovený čas.



Obr. 4.4: Životní cyklus objektu třídy Vivaldi_node/Vivaldi_node2

4.2.5 Třída Vivaldi_node2

Třída Vivaldi_node2 implementuje rozhraní IVivaldi_node2. Popis rozhraní IVivaldi_node2 a souhrn funkcí, které nabízí, je uveden níže na obr. 4.5 včetně stručného popisu funkcí. Objekt této třídy představuje uzel sítě, na kterém běží samotný algoritmus Vivaldi s proměnným časovým krokem. Objekt této třídy nabízí opět dva typy konstruktorů. První konstruktor nabízí možnost vytvoření statického objektu, druhý konstruktor vytváří dynamický objekt. Použití

těchto uzlů je totožné jako použití v případě objektu třídy Vivaldi_node. Nejdůležitější parametry třídy Vivaldi_node2 včetně popisu jejich významu jsou uvedeny v tab. 4.4. Samotný zdrojový kód třídy Vivaldi2_node je uveden v příloze D.

```
public interface IVivaldi_node2 {

    /* funkce addPoint (Vivaldi_node2 a) tato funkce slouží k přidání nového
     * bodu do seznamu referenčních bodů a automaticky spočítá RTT jako
     * vzdálenost mezi souřadnicemi nového bodu vůči koncovému bodu*/
   public void addPoint( Vivaldi_node2 a);

    /* funkce addPoint (Vivaldi_node a, Double rtt)
     * tato funkce slouží k přidání nového bodu do seznamu referenčních
     * bodu a nastavení explicitního RTT */
   public void addPoint( Vivaldi_node2 a, Double rtt);

    /* funkce remowePoint(int a)
     * tato funkce slouží k odstranění referenčního bodu ze seznamu
     * referenčních bodů na pozici "a" a automatickému zrušení informace o
     * RTT */
   public void remowePoint (int a);

    /* funkce updateRTT(int a, Double rtt)
     * tato funkce slouží k změně hodnoty RTT vůči danému referenčnímu bodu
     * dojde k přepsání hodnoty RTT na patřičné pozici v seznamu referenstRTT
     */
public void updateRTT (int a, Double rtt);

    /* Funkce execute() představuje implementaci algoritmu Vivaldi
     * Vstupními parametry jsou skutečná hodnota RTT vůči referenčnímu
     * bodu, jeho souřadnice a jeho vlastní chyba. Výsledkem teto metody je
     * úprava aktualizované pozice a zapsání informace do historie daného bodu
     */
   public void execute(double rtt, final Coordinates x_j,final double ei);
```

Obr. 4.5: Definice rozhraní IVivaldi_node2

Tab. 4.5: Základní parametry třídy Vivaldi_node2

Název proměnné	**Typ**	**Význam proměnné**
count	int	udržuje počet instancí vláken
point	coordinates	udržuje aktuální polohu uzlu
endPoint		udržuje polohu koncového uzlu
delta	double	udržuje aktuální hodnotu časového kroku
name	string	udržuje aktuální hodnotu váhového vektoru (používá se pouze u varianty s proměnným časovým krokem)
simulation	boolean	Určuje, kdy běží simulace
runVlakno		určuje platnost metody run( )
record	NodeHistorieRecord	udržuje informace o referenčních bodech
referentsPoints	Vector<Vivaldi_node>	udržuje informace o příslušných známých bodech
referentsRTT	Vector<Double>	udržuje informace o příslušných hodnotách RTT bodu vůči známým bodům

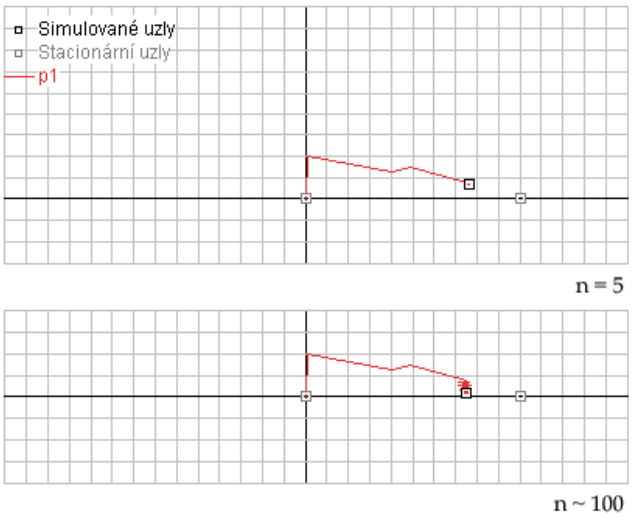
Činnost třídy Vivaldi_node2 a princip funkce metody run( ) je popsán vývojovým diagramem, viz obr. 4.4. Vzhledem k tomu, že se jedná o decentralizovaný algoritmus, tak každý uzel, jenž je představován objektem třídy Vivaldi_node2 běží jako nezávislé vlákno. Jelikož JAVA neumožňuje vlákna zastavit, tak vlákno běží, dokud není zrušen celý uzel. K zastavení činnosti vlákna po ukončení simulace je použit časovač, který vlákno uspí po stanovený čas.

4.3 Popis provedených simulací

Při základním testování konvergence vytvořené knihovny Vivaldi-lib bylo provedeno několik jednoduchých testů na obou variantách algoritmu. Pro jednoduchost byly v simulacích použity nehybné, stacionární body, sloužící pro ukotvení vytvořené struktury v prostoru. Jednalo se pouze o simulace základního chování uzlů implementujících algoritmus Vivaldi. Cílem simulací bylo otestovat a potvrdit funkčnost této knihovny a správnost implementace tohoto algoritmu. Všechny vzorové simulace jsou popsány a doplněny sérií grafů ilustrujících průběh chování sítě pro zvolenou konfiguraci.

4.3.1 Konvergence sítě s třemi uzly

Nejjednodušším testem konvergence byla sestava tří uzlů ležících na přímce. Byla testována se dvěma stacionárními uzly, kdy třetí uzel měl nastaveny pouze hodnoty zpoždění vůči oběma bodům. V této simulaci došlo k ustálení třetího uzlu přesně tam, kde měl skončit nezávisle na tom, kde byly zvoleny jeho počáteční souřadnice. Průběh této simulace lze vidět na obr. 4.6. V případě konfigurace, kdy body neležely na přímce, došlo k ustálení třetího bodu v jednom ze dvou osově souměrných řešeních. Ve variantě se všemi pohyblivými body došlo k ustálení bodů v prostoru tak, že ve výsledné poloze odpovídala vzájemná poloha bodů hodnotám zpoždění mezi nimi. Vzhledem k tomu, že nebyl žádný bod stacionární, body se při různých hodnotách startova-

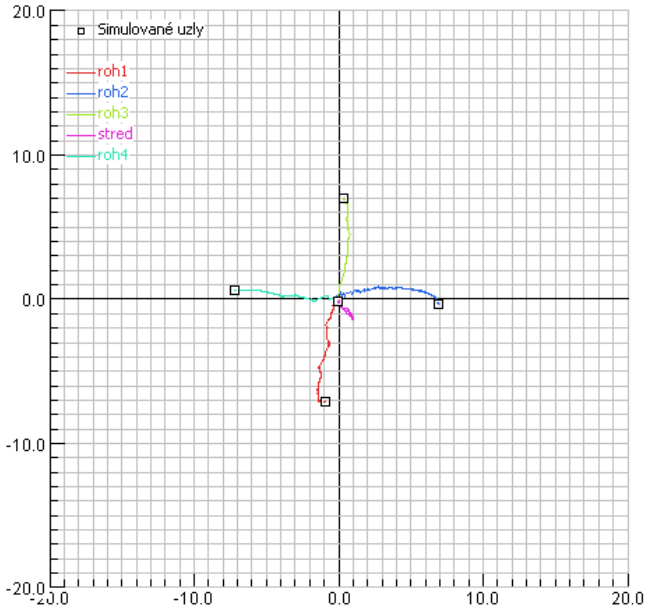


Obr. 4.6: Průběh konvergence jednoho bodu (Vivaldi, nalezení pozice stanice)

cích poloh nacházely ve výsledku na rozdílných souřadnicích, ale vzdálenosti mezi nimi v prostoru odpovídaly hodnotám jejich vzájemných zpoždění. Volba počátečních hodnot a množství stacionárních bodů měly vliv pouze na rychlost konvergence.

4.3.2 Konvergence sítě s pěti uzly

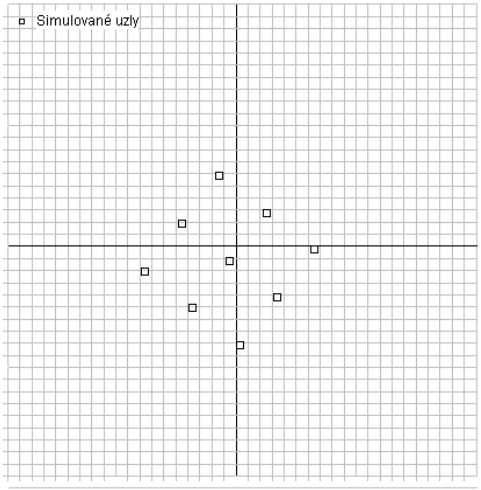
V dalších testech byla použita varianta s pěti uzly tvořícími čtverec o straně dlouhé 10 a jeho střed v různé konfiguraci systému. Všechny body startovaly z počátku souřadné soustavy. Simulace potvrdily správnou funkčnost knihovny a konvergenci algoritmu. Průběh simulace lze vidět na obr. 4.7.



Obr. 4.7: Průběh konvergence pěti bodů (Vivaldi, nalezení pozice stanice)

4.3.3 Konvergence sítě s devíti uzly

Poslední test byl proveden s devíti uzly. V tomto testu bylo zachyceno chování uzlů ve dvou simulacích. V prvním případě každý uzel znal spoje ke všem ostatním, v druhém případě měl určeny 3 protější uzly náhodným výběrem. V obou případech uzly konvergovaly do správné výsledné polohy. Průběh simulace lze vidět na obr. 4.9.

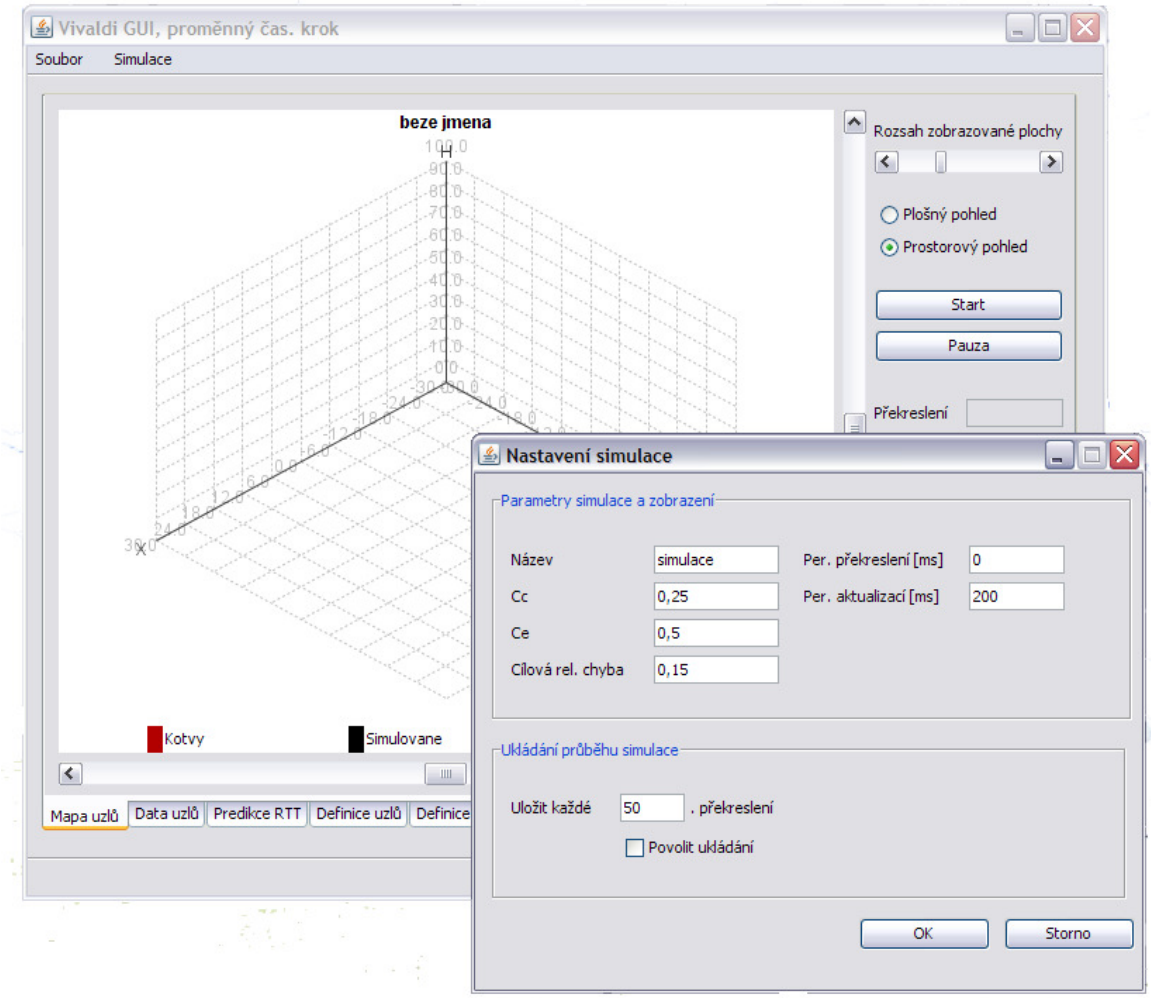


Obr. 4.8: Průběh konvergence devíti bodů (Vivaldi, nalezení pozice stanice)

4.4 Základní popis simulačního programu VIVALDI MONITOR

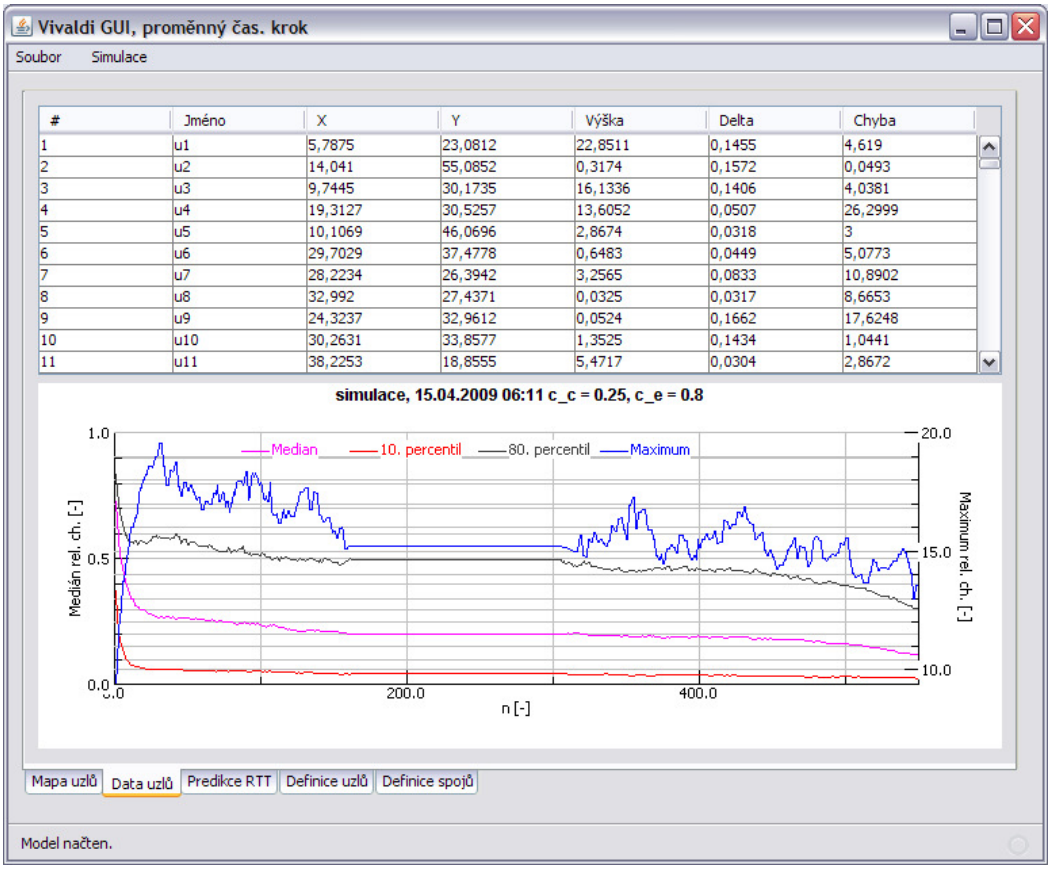
Aplikace VIVALDIMONITOR [2] tvoří grafické rozhraní pro výše popsanou knihovnu. Tato aplikace umožňuje pozorovat činnost sítě uzlů implementujících algoritmus Vivaldi jako distribuovanou decentralizovanou vzdálenostní službu. Tato aplikace, jejíž vývoj byl hlavním cílem práce spolupracovníka Bc. Jaroslava Švédy, je poskytnuta ve dvou variantách - první variantou je VIVALDIMONITOR s proměnným časovým krokem, využívající objekty Vivaldi_node2, druhou variantou je VIVALDIMONITOR F s konstantním časovým krokem, využívající objekty Vivaldi_node. Základní grafické rozhraní aplikace VIVALDIMONITOR je zobrazeno na obr. 4.9. Znázornění činnosti vyžaduje pečlivou volbu veličin, co nejlépe popisující aktuální stav sítě i její vývoj. Z tohoto důvodu aplikace nabízí několik různých pohledů na zaznamenané vzorky veličin uzlů sítě. Jde jmenovitě o zobrazení aktuálních souřadnic uzlů prostřednictvím grafu, výpis aktuálního vzorku údajů všech uzlů v tabulce, dále pak grafické zobrazení předchozího vývoje souřadnic uzlů, časová závislost relativní chyby predikce vzdálenosti a výšky uzlu. Samozřejmostí je i možnost načtení předpřipraveného modelu sítě ze souboru, jeho úprava a uložení. Je možný i jednoduchý export záznamů jednotlivých uzlů i sítě jako celku v podobě skriptu prostředí Matlab, do souboru lze uložit i přehled spojů společně s jejich predikcemi a chybou predikce. Aplikace byla napsána v Javě z důvodu přenositelnosti na jiné operační systémy. Jako základ pro aplikaci byl použit aplikační framework založený na GUI knihovně Swing.

Aplikace je pro jednoduchost ovladatelná z jejího hlavního okna. Toto hlavní okno obsahuje sadu několika karet, které velmi efektivně zpřístupňují většinu funkcí, jak ukazuje obr. 4.10. Tato sada karet obsahuje dvojici karet pro zobrazení aktuálního rozložení uzlů probíhající simulace „Rozložení uzlů“ a „Data uzlů“. Za ní následuje karta „Predikce RTT“, která umožňuje zhodnocení přesnosti predikce latence všech spojů. Poslední dvě karty pak soustřeďují veškeré rozhraní pro definice topologie sítě, tj. uzlů a spojů mezi nimi včetně jejich vlastností, jako je poloha, metrika (RTT) a směr spojů.



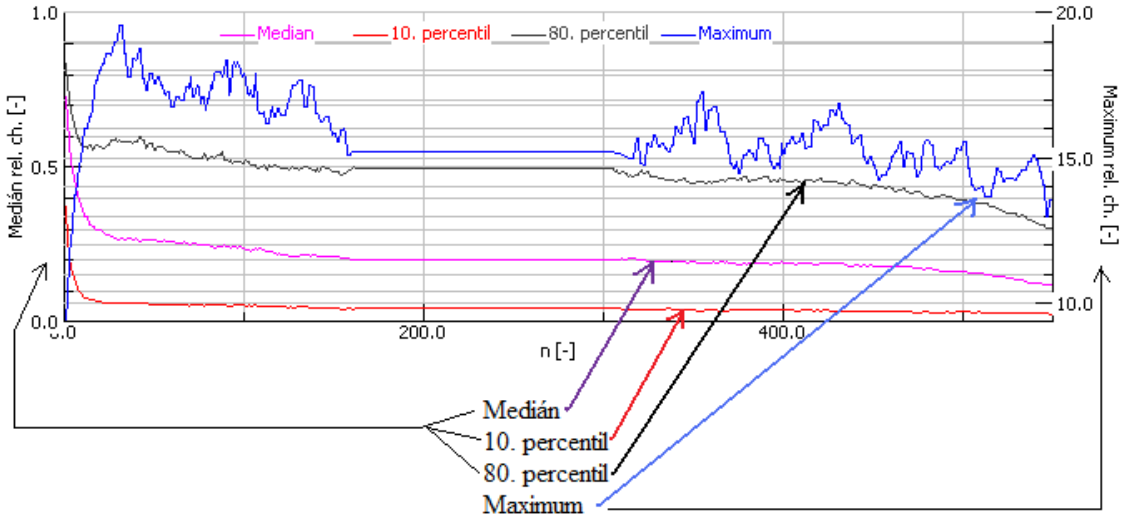
Obr. 4.9: Ovládací rozhraní simulačního software

Hlavní rozhraní je doplněno dvěma nabídkami ve standardní nabídkové liště pod titulkovým pruhem okna. Jde především o standardní nabídku „Soubor“, umožňující import a export modelů a dat, a menu „Simulace“, umožňující nastavit parametry simulace a také uložit aktuální obsah grafů rozložení sítě a vývoje predikční chyby jako obrazové soubory.



#	Jméno	X	Y	Výška	Delta	Chyba
1	u1	5,7875	23,0812	22,8511	0,1455	4,619
2	u2	14,041	55,0852	0,3174	0,1572	0,0493
3	u3	9,7445	30,1735	16,1336	0,1406	4,0381
4	u4	19,3127	30,5257	13,6052	0,0507	26,2999
5	u5	10,1069	46,0696	2,8674	0,0318	3
6	u6	29,7029	37,4778	0,6483	0,0449	5,0773
7	u7	28,2234	26,3942	3,2565	0,0833	10,8902
8	u8	32,992	27,4371	0,0325	0,0317	8,6653
9	u9	24,3237	32,9612	0,0524	0,1662	17,6248
10	u10	30,2631	33,8577	1,3525	0,1434	1,0441
11	u11	38,2253	18,8555	5,4717	0,0304	2,8672

Obr. 4.10: Grafické zobrazení základních údajů simulace

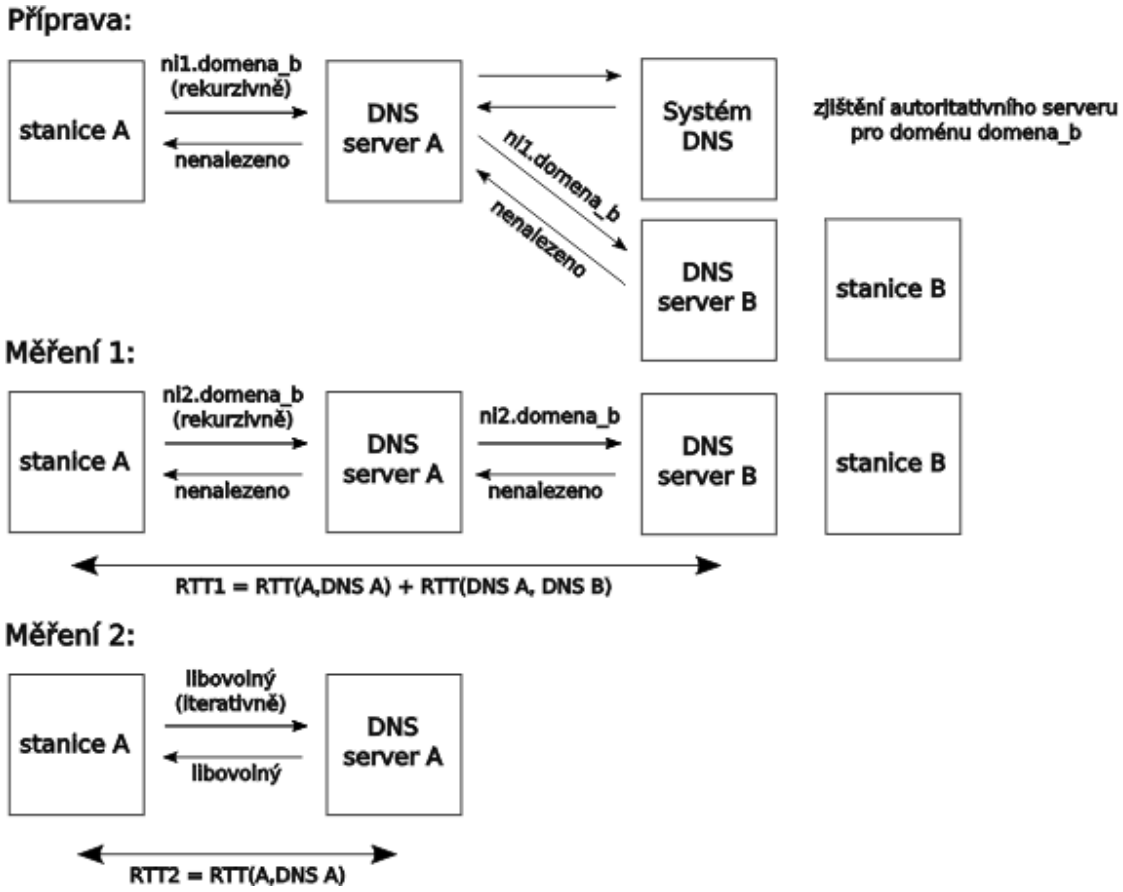


Obr. 4.11: Význam jednotlivých průběhů základních parametrů sítě v grafickém zobrazení

5 Nalezení pozice v umělých a experimentálních sítích

Poslední část diplomové práce byla zaměřena na základní otestování celého programu a samotného algoritmu na sítích většího rozsahu, které byly tvořeny uzly implementující algoritmus Vivaldi. Pomocí vytvořeného simulačního programu [2] byla provedena řada simulací zaměřených na zjištění základního chování obou variant algoritmu Vivaldi. První část simulací se zabývala chováním obou algoritmů vzhledem k nastavení „ladících" parametrů, které ovlivňují chování jednotlivých metod. U varianty s konstantním časovým krokem je hlavním parametrem volba velikosti samotného časového kroku δ, který se používá v rozsahu (0,1). U varianty s proměnným časovým krokem se jedná o parametry c_c a c_e, které mají vliv na chování samotného algoritmu. Simulace pro oba typy algoritmů byly vždy prováděny na stejném vstupním souboru dat a pro stejný časový úsek. Tento úsek představuje hodnota počtu aktualizací každého bodu, aby bylo možné porovnat nejen chování obou algoritmů na základě nastavení jednotlivých parametrů, ale taky proto, aby bylo možné porovnat oba algoritmy navzájem. Cílem bylo zjistit, jak se oba algoritmy liší z hlediska rychlosti konvergence a přesnosti predikce RTT. Jednotlivé simulace jsou charakterizovány čtyřmi základními parametry, které jsou medián relativní chyby celé sítě, 10. percentil relativní chyby sítě (hodnota relativní chyby sítě, která se po seřazení všech hodnot od nejmenší po největší nachází na pozici odpovídající 1/10 všech hodnot od nejmenší hodnoty), 80. percentil relativní chyby sítě (hodnota relativní chyby sítě, která se po seřazení všech hodnot od nejmenší po největší nachází na pozici odpovídající 8/10 všech hodnot od nejmenší hodnoty) a maximální absolutní relativní chyba uzlů v síti v daný okamžik. První část simulací byla prováděna na umělých sítích o velikosti 15×15 a 20×20 uzlů. Tyto sítě byly uspořádány ve čtvercové struktuře, aby bylo možné určit úplnou matici vzdáleností mezi všemi uzly. Z této matice bylo vybráno náhodně 5, 10, 15 nebo 20 vazeb pro každý uzel, které sloužily jako vstupní data pro simulace. Pro výběr vazeb byl použit jeden z nástrojů, který je součástí simulačního programu. Po provedené simulaci byly na základě rozmístění uzlů dopočítány všechny zbylé vazby, aby bylo možné vytvořit opět plnou matici vzdáleností a tu porovnat s generující maticí. Na základě těchto údajů pak byly určovány chyby u jednotlivých spojů. Stejným způsobem pak byla zpracována i data ze sítě PlanetLab označené jako sada King1. Druhá část simulací byla provedena právě na této sadě dat. Základní sada měla velikost 1740 uzlů a tato data byla získána v červnu 2004 metodou King[16], viz obr. 5.1 [2]. Tato metodika predikce zpoždění je založena na předpokladu přítomnosti alespoň jednoho DNS serveru blízko každé stanice v Internetu z hlediska zpoždění. Při platnosti tohoto předpokladu je možné přesně předpovědět zpoždění mezi dvěma stanicemi A, B měřením zpoždění mezi jim nejbližšími DNS servery (DNS A, DNS B). Toto zpoždění nelze měřit přímo. Měření se provádí pomocí běžných DNS dotazů. Když stanice A potřebuje odhadnout vzdálenost ke stanici B, tak provede dvě měření. První měření je provedeno rekurzivním DNS dotazem na server DNS B na libovolné doménové jméno v jeho doméně. Pokud nemá DNS A výsledek dotazu v cache, požadavek je přeposlán serveru DNS B, který pošle výsledek dotazu zpět serveru DNS A. Ten pak výsledek přepošle zpět stanici A. Doba zpracování tedy bude odpovídat zpoždění k DNS B. Následně je zjištěno zpoždění k DNS A, nejlépe iterativním DNS dotazem, který musí být zpracován pouze na základě cache a záznamů daného serveru. Zpoždění mezi oběma servery je

pak dáno rozdílem obou měření. Z celé sady 1740 uzlů bylo náhodně vybráno 80 a jejich vzájemné RTT byly použity jako základ pro simulace. Z tohoto základního modelu byly posléze pomocí nástroje MNVyberSousedy[2] vytvořeny modely s různými počty vazeb na uzel. Se spolupracovníkem jsme provedli dvě nezávislé testování, jejichž výsledky jsou popsány v závěru této kapitoly.



Obr. 5.1: Princip zjištění RTT pomocí metody King1 v síti PlanetLab

5.1 Simulace v umělých sítích s konstantním časovým krokem

Byly provedeny simulace pro sítě velikosti 15×15 a 20×20 uzlů. Pro obě sítě byly simulace vždy provedeny za situace, že každý uzel znal 5, 10, 15 nebo 20 náhodně zvolených sousedů. Tab. 5.1 a 5.2 zobrazuje závislost chování sítě v závislosti na volbě parametru δ. Současně data v této tabulce umožňují porovnat vliv počtu vazeb jednotlivých uzlů na průběh jednotlivých charakteristik. Hodnoty v tabulce byly změřeny po cca 1000 aktualizacích. Průběhy těchto měření jsou zobrazeny na obr. 5.2 až 5.7. Z těchto simulací lze vidět, že optimální hodnota parametru δ je kolem hodnoty 0,3. Při této hodnotě jsou nejoptimálnější průběhy všech čtyř sledovaných parametrů. Toto chování se projevilo stejně i na modelu 20×20 uzlů a souhlasí s výsledky jiných studií [6]. Při volbě hodnoty blížící se k 1, síť sice na začátku konverguje rychleji, ale v okamžiku, kdy se již přiblíží blízko své optimální poloze, může docházet k oscilacím kolem této polohy a současně tak i k větší chybě, než při menším kroku. Naopak

volba parametru δ kolem hodnoty 0,1 způsobuje pomalejší konvergenci a je tu i riziko, že bod uvízne v místě s lokální minimální chybou.

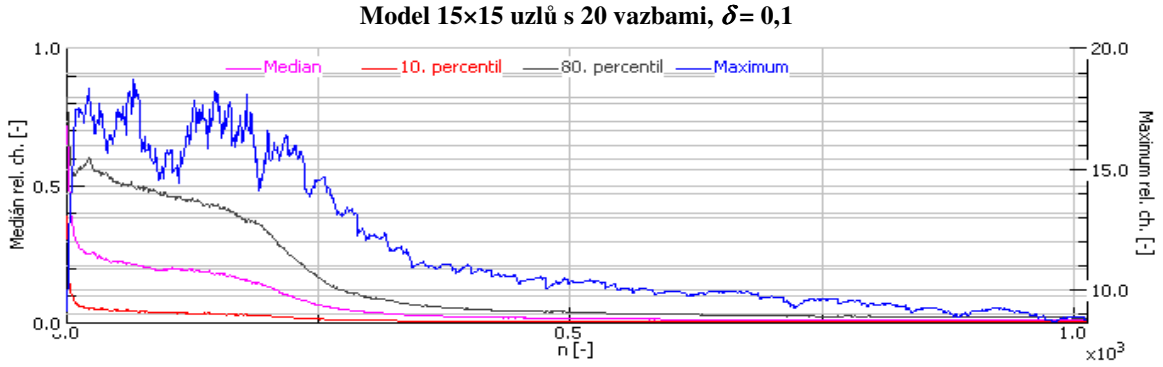
Tab. 5.1: Vliv parametru δ a počtu vazeb na přesnost predikce v sítí 20×20 uzlů

Model sítě 20 x 20						
Počet vazeb	**20**	**10**	**20**	**10**	**20**	**10**
Delta [-]	0,8		0,3		0,1	
Median [%]	0,245	0,262	0,252	0,243	0,596	0,581
Max abs. chyba [s]	0,910	1,822	3,371	2,991	6,231	7,017

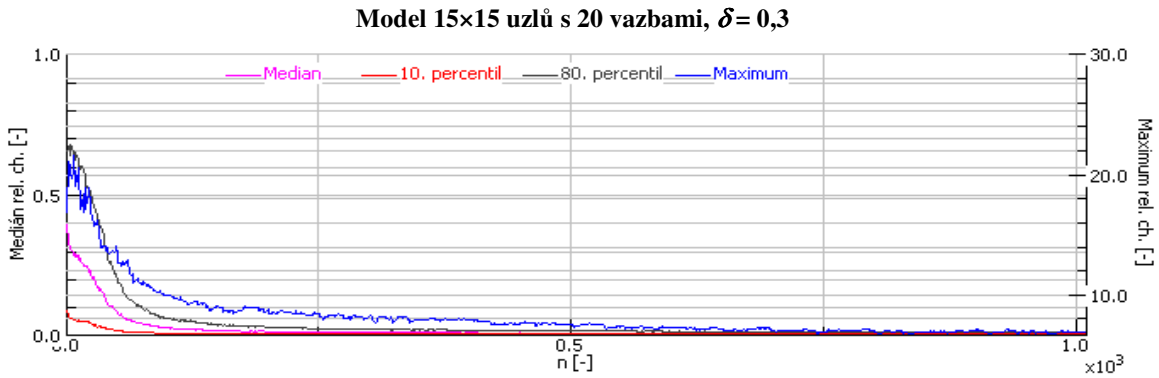
Tab. 5.2: Vliv parametru δ a počtu vazeb na přesnost predikce v sítí 15×15 uzlů

Model sítě 15 x 15						
Počet vazeb	**20**	**10**	**20**	**10**	**20**	**10**
Delta [-]	0,8		0,3		0,1	
Median [%]	0,295	0,272	0,284	0,268	0,624	0,668
Max abs. chyba [s]	0,80132	0,72723	1,55379	1,683	4,36305	3,87175

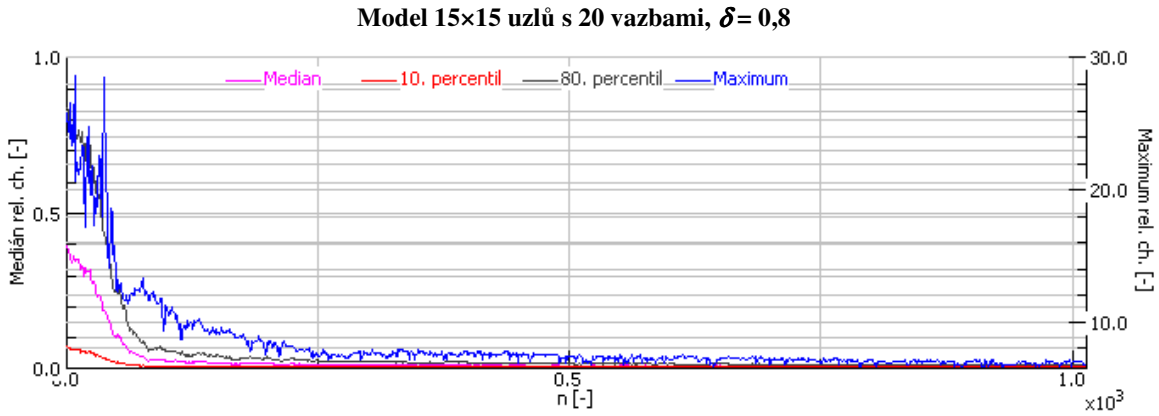
Z hlediska počtu vazeb každého uzlu na chování celé sítě nelze říct přesně ideální počet. Z provedených simulací se za optimální hodnotu jevil v našem případě počet 10 vazeb. Průběhy simulace závislosti rychlosti konvergence a přesnosti predikce zobrazují obr. 5.2 až 5.7. Významnou roli zde však hraje náhodnost výběru sousedů, která může způsobit, že i při malém počtu sousedů může být rychlost a přesnost konvergence lepší než při větším počtu. Obecně však platí, že větší přesnost a rychlost predikce se dosahuje při větším počtu referenčních stanic. Hlavní roli zde hraje přesnost souřadnice u každého uzlu. Z provedených simulací je patrné, že počet vazeb má vliv na rychlost konvergence celé sítě, ne však na přesnost predikce. Tato varianta na rozdíl od varianty s proměnným časovým krokem nebere v případě aktualizace uzlu *i* v úvahu vlastní relativní chybu druhého uzlu *j*, vůči kterému svoji pozici upravuje. Tento fakt může způsobit, že v případě výskytu několika (cca 5 %) vysoce chybných uzlů již systém nemusí konvergovat do optimálního stavu.



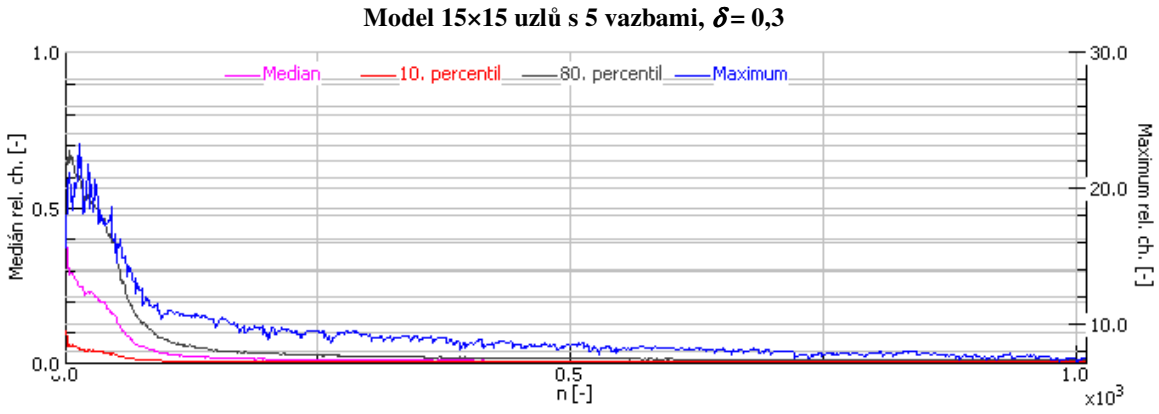
Obr. 5.2: Vliv konstantního časového kroku při δ= 0,1 (Vivaldi)



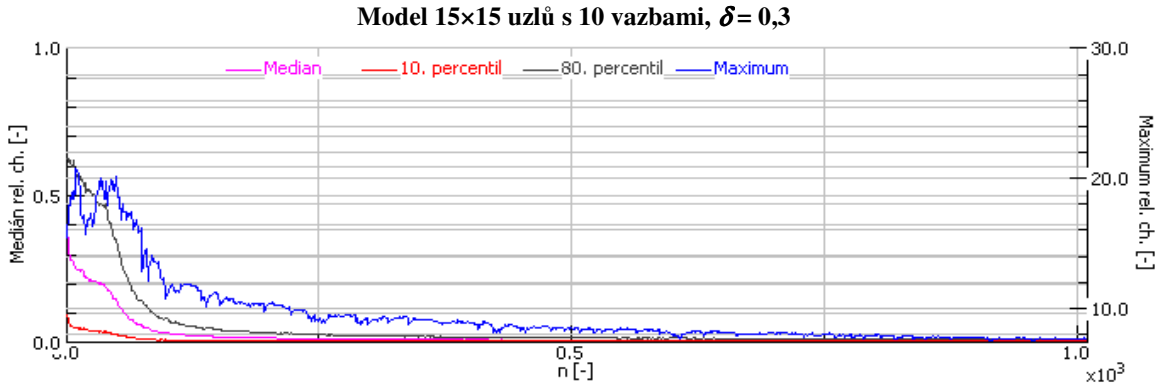
Obr. 5.3: Vliv konstantního časového kroku při $\delta = 0{,}3$ (Vivaldi)



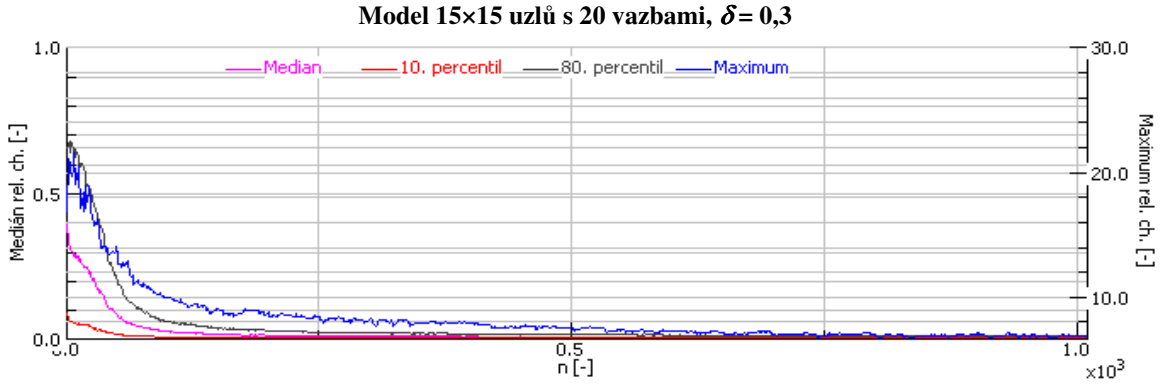
Obr. 5.4 : Vliv konstantního časového kroku při $\delta = 0{,}8$ (Vivaldi)



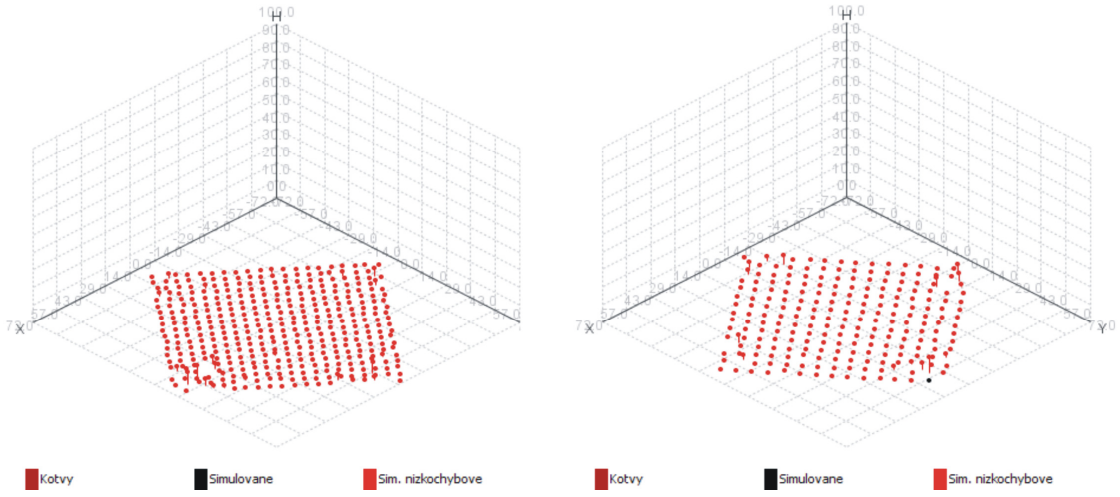
Obr. 5.5: Vliv počtu vazeb na rychlost a přesnost konvergence při 5 vazbách, $\delta = 0{,}3$ (Vivaldi)



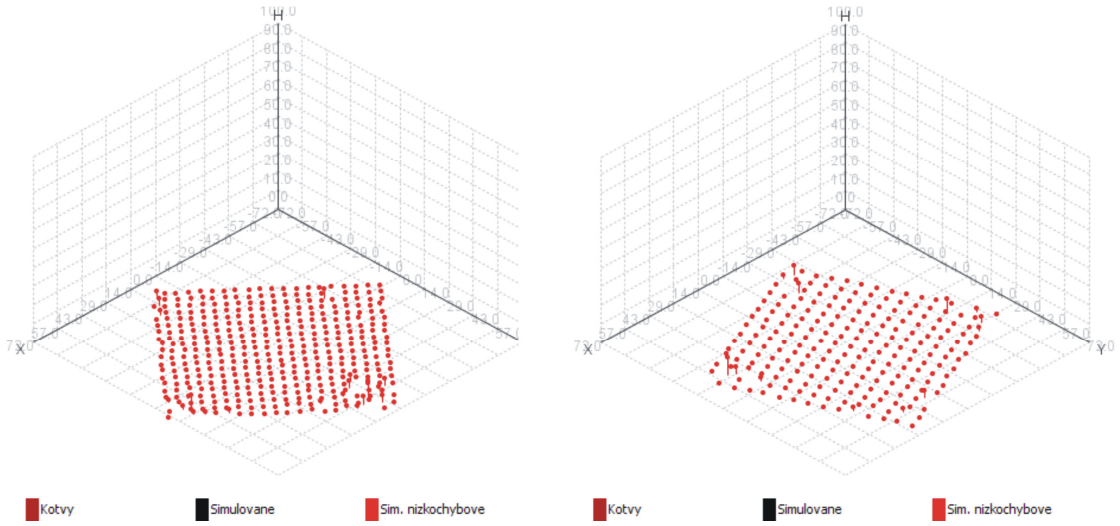
Obr. 5.6 : Vliv počtu vazeb na rychlost a přesnost konvergence při 10 vazbách, $\delta = 0{,}3$ (Vivaldi)



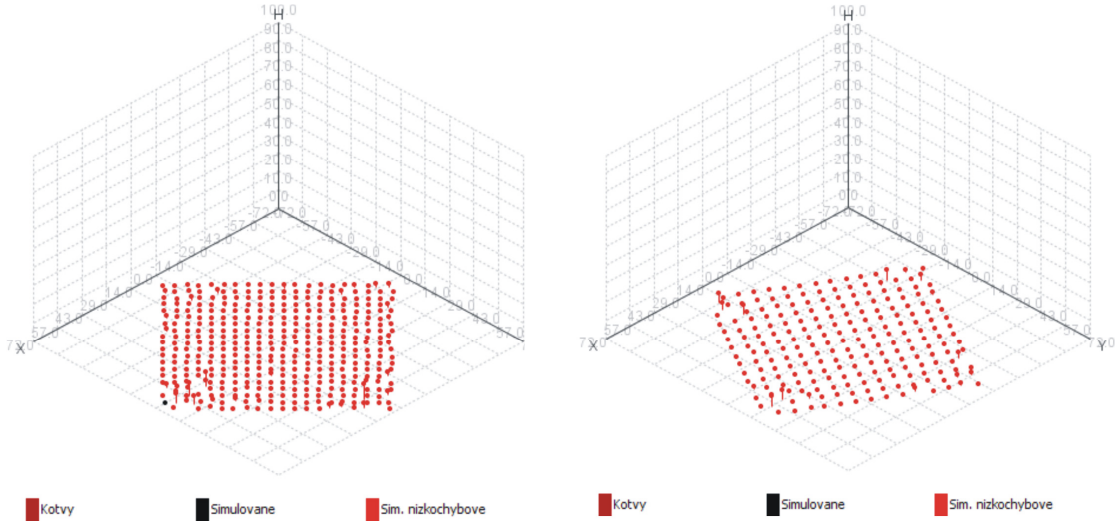
Obr. 5.7 : Vliv počtu vazeb na rychlost a přesnost konvergence při 20 vazbách, $\delta = 0{,}3$ (Vivaldi)



Obr. 5.8 : Rozmístění uzlů sítě při $\delta = 0{,}3$ po 1000 aktualizacích při 5 vazbách (Vivaldi)



Obr. 5.9 : Rozmístění uzlů sítě při $\delta = 0{,}3$ po 1000 aktualizacích při 10 vazbách (Vivaldi)



Obr. 5.10 : Rozmístění uzlů sítě při $\delta = 0{,}3$ po 1000 aktualizacích při 20 vazbách (Vivaldi)

Obr. 5.8 až 5.10 zobrazuje rozmístění uzlů sítě v prostoru po provedení cca 1000 aktualizací pro sítě 20×20 uzlů (vpravo) a 15×15 uzlů (vlevo) v závislosti na počtu vazeb. Červeně vyznačené uzly mají chybovost menší než 1 %.

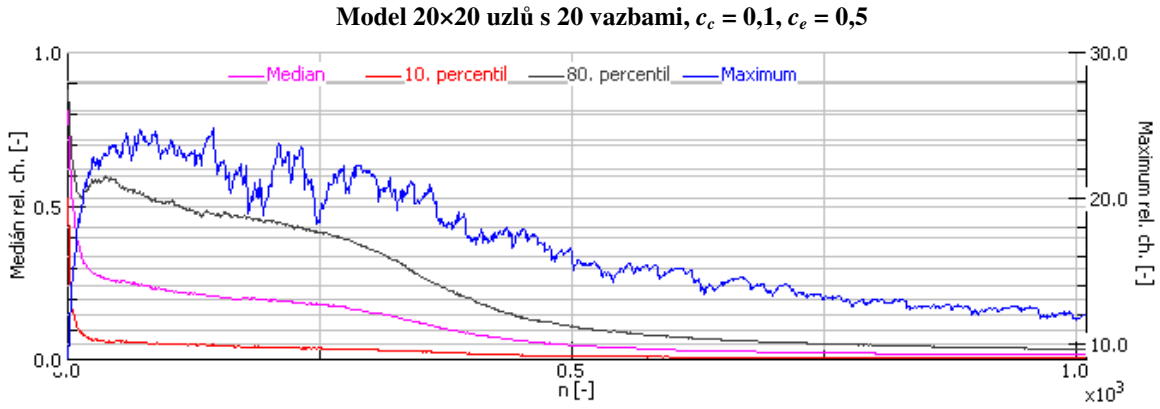
5.2 Simulace v umělých sítích s proměnným časovým krokem

Simulace s proměnným časovým krokem byly prováděny na stejných sítích o velikosti 15×15 a 20×20 uzlů jako simulace s konstantním časovým krokem. Opět byly provedeny simulace za situace, že každý uzel znal 5, 10, 15 nebo 20 náhodně zvolených sousedů. Při simulacích byl sledován vliv velikosti ladících parametrů c_c a c_e ovlivňující velikost proměnného časového kroku na základní charakteristiky, charakterizující rychlost konvergence a přesnost predikce

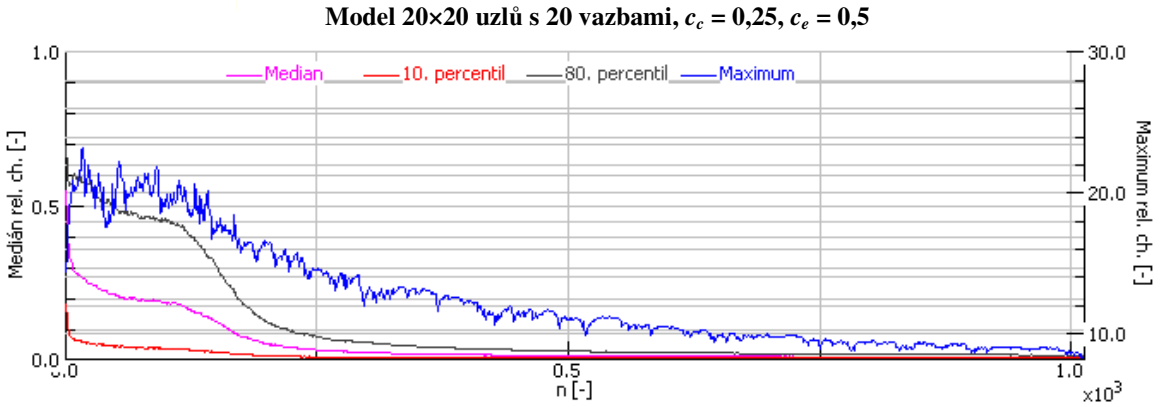
zpoždění. Tab. 5.3 zobrazuje výsledky měření. Hodnoty v tabulce byly změřeny po cca 1000 aktualizacích. Z naměřených hodnot je patrné, že optimální hodnota parametru c_c je při hodnotě blízké 1. Tento fakt platí na datech z umělých sítí, v reálných sítích je za optimální variantu považována hodnota konstanty c_c = 0,25 [11]. Vyšší hodnota vede sice k rychlejší konvergenci, ale může způsobit oscilaci uzlu kolem optimální polohy a tím i vyšší chybu predikce zpoždění. Optimální hodnota druhého ladícího parametru c_e se pohybuje kolem hodnoty 0,5. Průběhy části těchto měření ilustrují obr. 5.11 až 5.17.

Tab. 5.3: Vliv parametru c_c , c_e a počtu vazeb na přesnost predikce

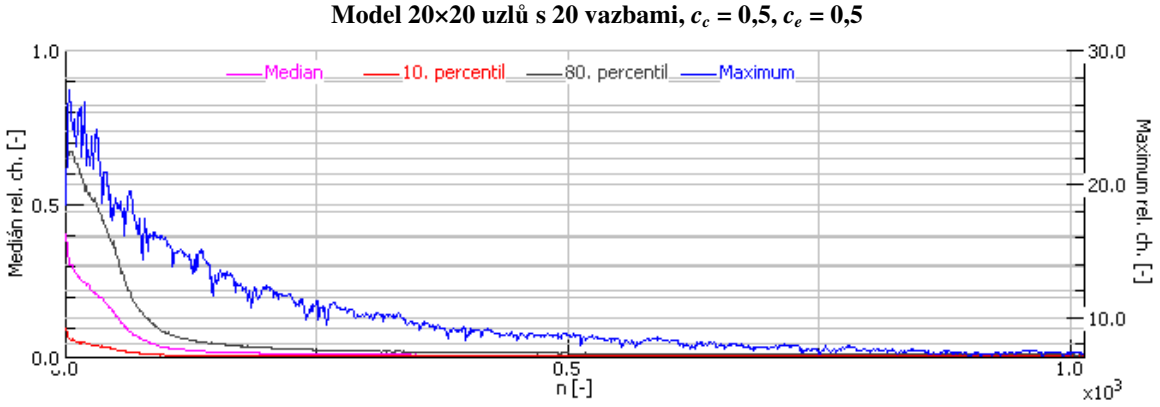
Model 20×20 uzlů s 20 vazbami						
c_c **[-]**	**0,8**			**0,5**		
c_e **[-]**	**0,9**	**0,5**	**0,1**	**0,9**	**0,5**	**0,1**
Median [%]	0,141	0,144	0,167	0,200	0,210	0,209
Max abs. chyba [s]	1,806	2,240	2,098	3,132	2,585	2,670
10. percentil [-]	0,00020	0,00022	0,00026	0,00032	0,00033	0,00034
c_c **[-]**	**0,25**			**0,1**		
c_e **[-]**	**0,9**	**0,5**	**0,1**	**0,9**	**0,5**	**0,1**
Median [%]	0, 421	0, 412	4630	12850	1,301	1,400
Max abs. chyba [s]	5,465	4,540	5,482	9,156	9,559	9,505
10. percentil [-]	0,00069	0,00067	0,00075	0,00223	0,00231	0,00254



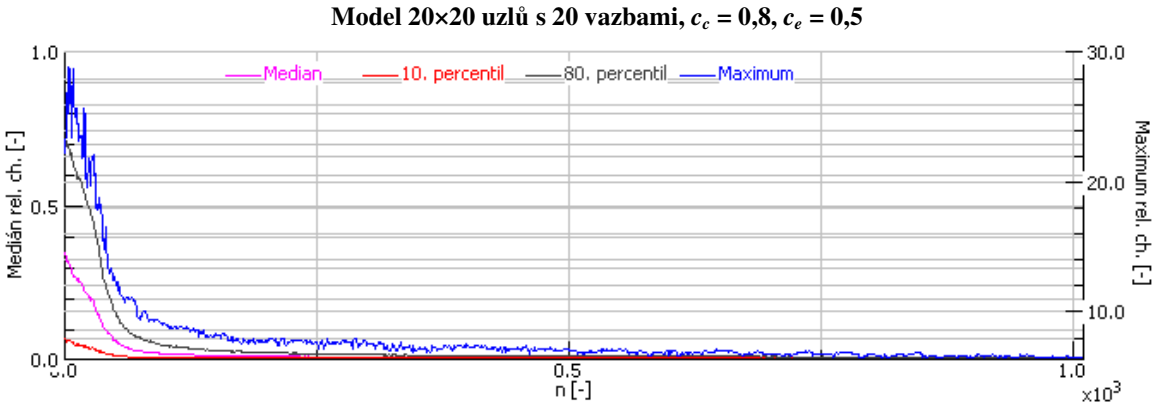
Obr. 5.11: Vliv parametru c_c na konvergenci a přesnost predikce RTT při c_c = 0,1 (Vivaldi)



Obr. 5.12: Vliv parametru c_c na konvergenci a přesnost predikce RTT při c_c = 0,25 (Vivaldi)



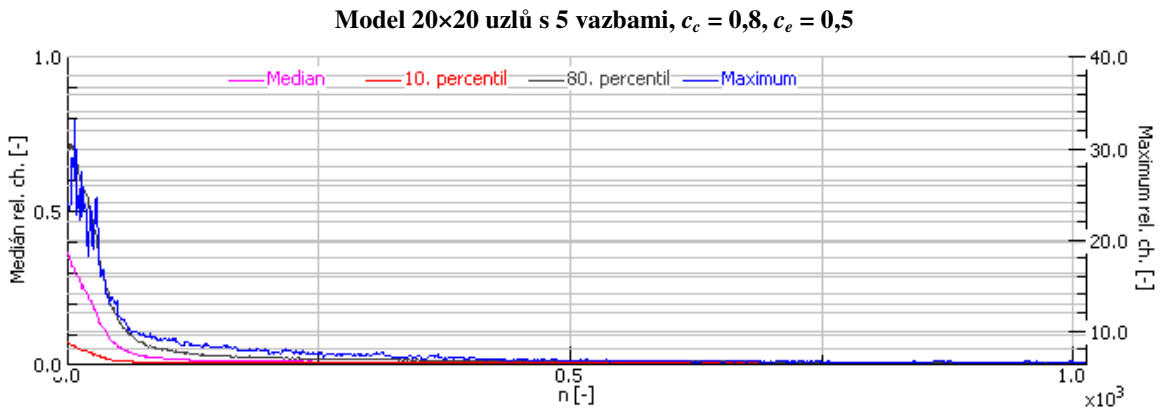
Obr. 5.13 : Vliv parametru c_c na konvergenci a přesnost predikce RTT při c_c = 0,5 (Vivaldi)



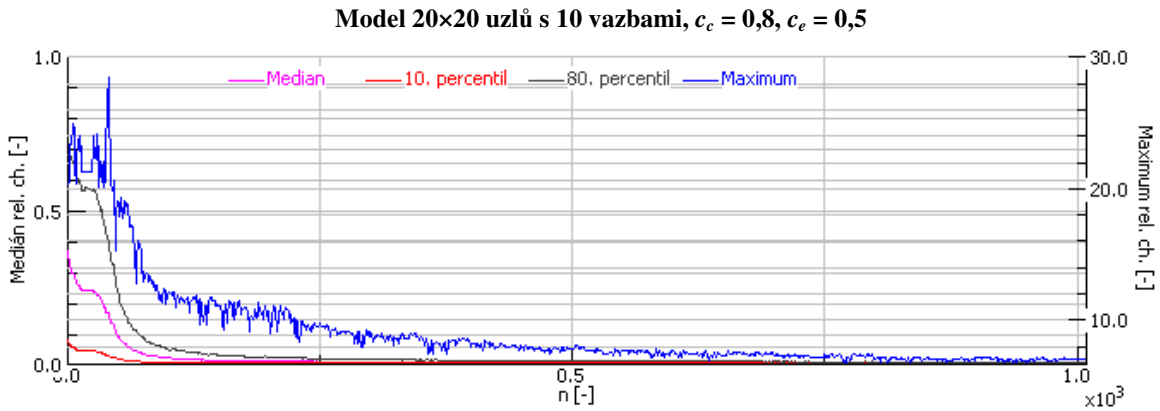
Obr. 5.14 : Vliv parametru c_c na konvergenci a přesnost predikce RTT při c_c = 0,8 (Vivaldi)

Z hlediska počtu vazeb každého uzlu na chování celé sítě je situace stejná jako u varianty s konstantním časovým krokem. Vzhledem k náhodnosti výběru sousedů nelze stanovit přesně

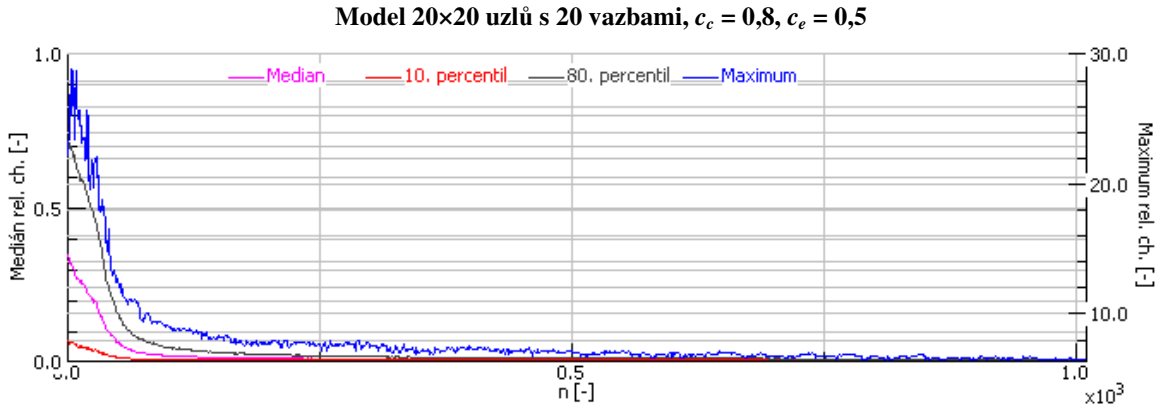
ideální počet. Z provedených simulací se za optimální hodnotu jevil v našem případě počet 20 vazeb. Z provedených simulací je patrné, že počet vazeb má vliv na rychlost konvergence celé sítě, ne však na přesnost predikce. Tato varianta na rozdíl od varianty s konstantním časovým krokem již bere v případě aktualizace uzl *i* v úvahu vlastní relativní chybu druhého uzlu *j*, vůči kterému svoji pozici upravuje. Tento fakt umožňuje vyšší odolnost sítě vůči výskytu chybných uzlů. Není však ochranou proti úmyslnému útoku na tuto síť. Problematikou odolnosti algoritmu Vivaldi vůči úmyslným útokům se zabývá studie [11]. Následující obrázky ukazují průběh chování sítě v závislosti na počtu vazeb u jednotlivých uzlů. Zobrazené průběhy jsou pro síť 20×20 uzlů při proměnném časovém kroku a při $c_c = 0{,}8$ a $c_e = 0{,}5$. Stejné chování sítě v závislosti na nastavených parametrech se projevilo i na síti 15×15 uzlů.



Obr. 5.15 : Vliv počtu 5 vazeb na rychlost a přesnost konvergence při $c_c = 0{,}8$, $c_e = 0{,}5$ (Vivaldi)



Obr. 5.16: Vliv počtu 10 vazeb na rychlost a přesnost konvergence při $c_c = 0{,}8$, $c_e = 0{,}5$ (Vivaldi)



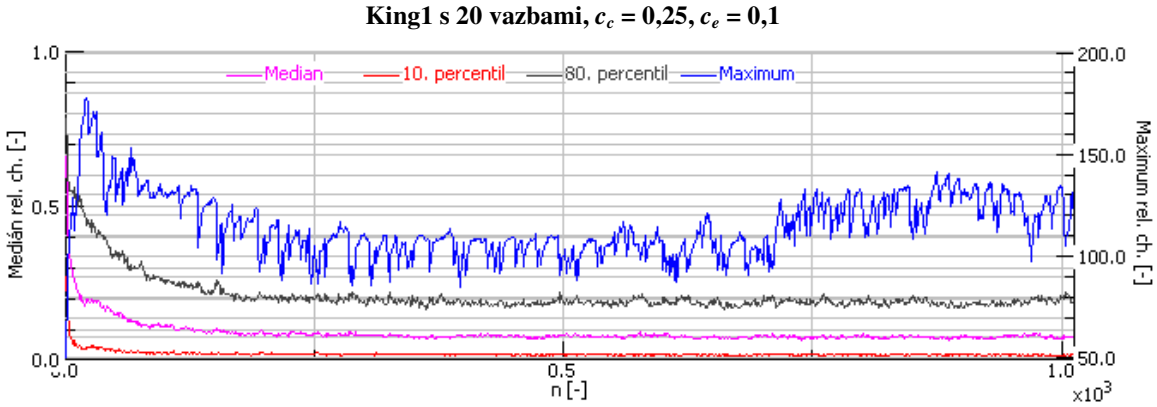
Obr. 5.17: Vliv počtu 20 vazeb na rychlost a přesnost konvergence při $c_c = 0{,}8$, $c_e = 0{,}5$ (Vivaldi)

5.3 Nalezení pozice v experimentálních sítích s konstantním a proměnným časovým krokem

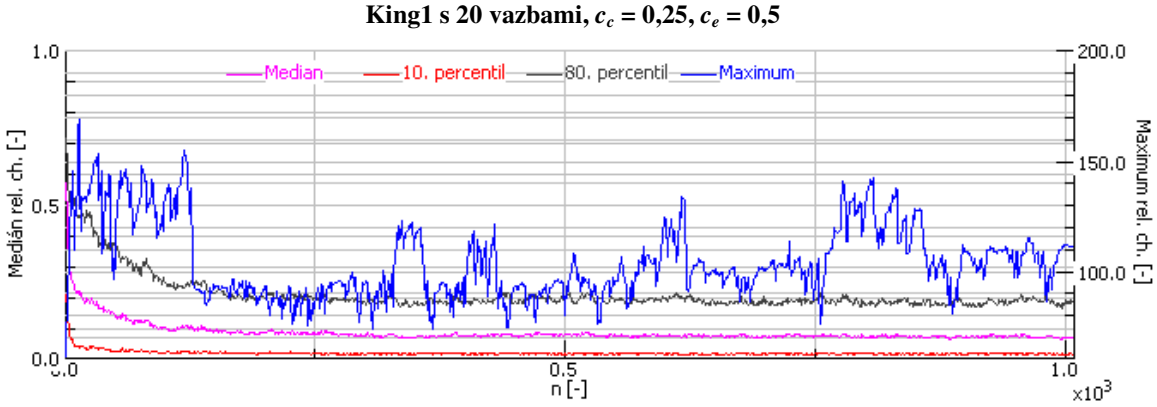
K simulaci na reálných datech byla použita část dat získaných z měření v experimentální síti PlanetLab [16]. Z celé sítě bylo vybráno 80 uzlů s plnou maticí vzájemných vzdáleností. Tato data byla podrobena simulacím při rozdílných nastaveních parametrů c_c a c_e při použití proměnného časového kroku a při rozdílném nastavení parametru δ a při konstantním časovém kroku. Všechny simulace byly provedeny při různém počtu vazeb u jednotlivých uzlů. Tab. 5.4 zobrazuje hodnoty základních parametrů simulace po provedení cca 1000 aktualizací při použití varianty s proměnným časovým krokem. Z provedených simulací je patrné, že přesnost predikce je nejlepší při volbě parametru c_c kolem hodnoty 0,25. Relativní chyba predikce RTT se v tomto případě pohybuje zhruba do 10 %, což odpovídá závěrům z jiných studií [11,13], která se touto problematikou zabývaly. Naměřené hodnoty opět potvrzují, že na přesnost predikce má minimální vliv počet vazeb u každého uzlu.

Tab. 5.4: Vliv parametru c_c a počtu vazeb na přesnost predikce v experimentální síti při kont. c_e

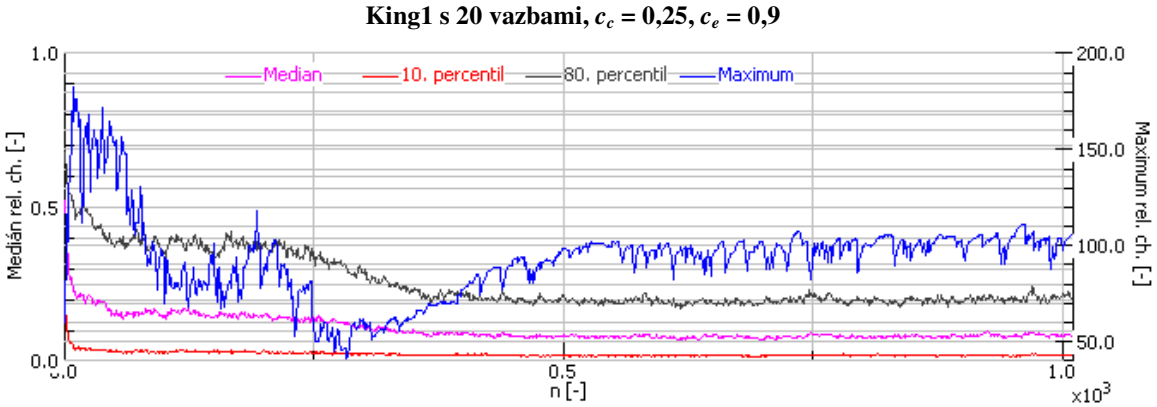
	Medián[%]		
Počet vazeb	**c_c = 0,25**	**c_c = 0,5**	**c_c = 0,9**
10	9,97	10,70	11,00
20	9,67	9,55	11,25
50	9,55	9,84	11,36
Pozn.: Všechny simulace byly prováděny při c_e = 0,5			



Obr. 5.18: Vliv parametru c_e na rychlost a přesnost konvergence při c_e = 0,1(Vivaldi)

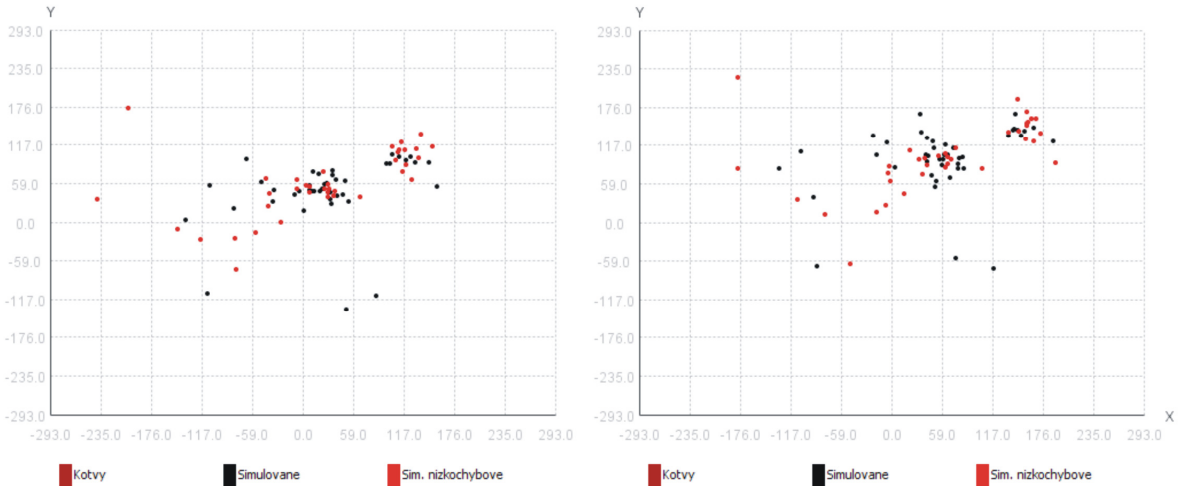


Obr. 5.19: Vliv parametru c_e na rychlost a přesnost konvergence při c_e = 0,5 (Vivaldi)

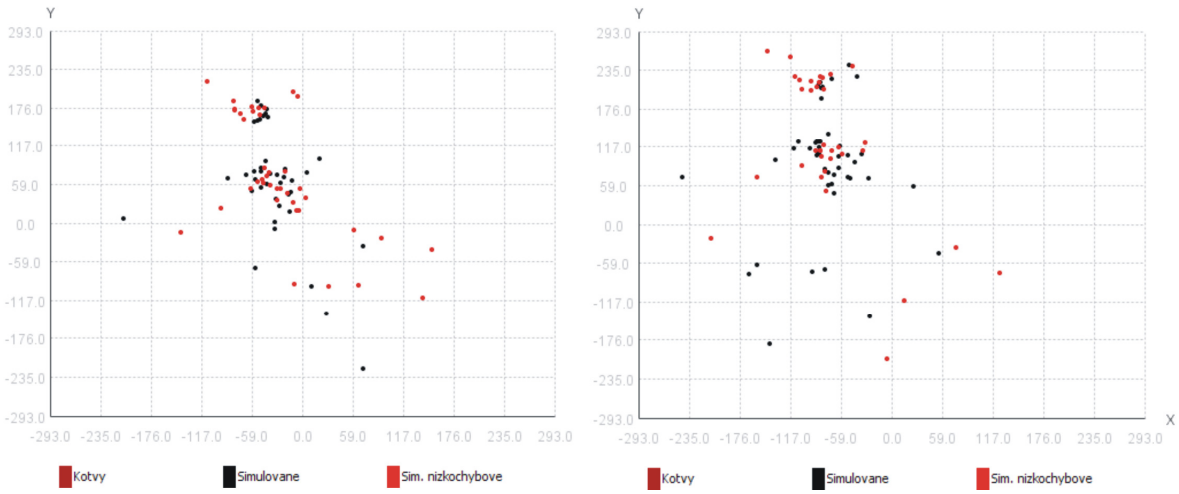


Obr. 5.20: Vliv parametru c_e na rychlost a přesnost konvergence při c_e = 0,9 (Vivaldi)

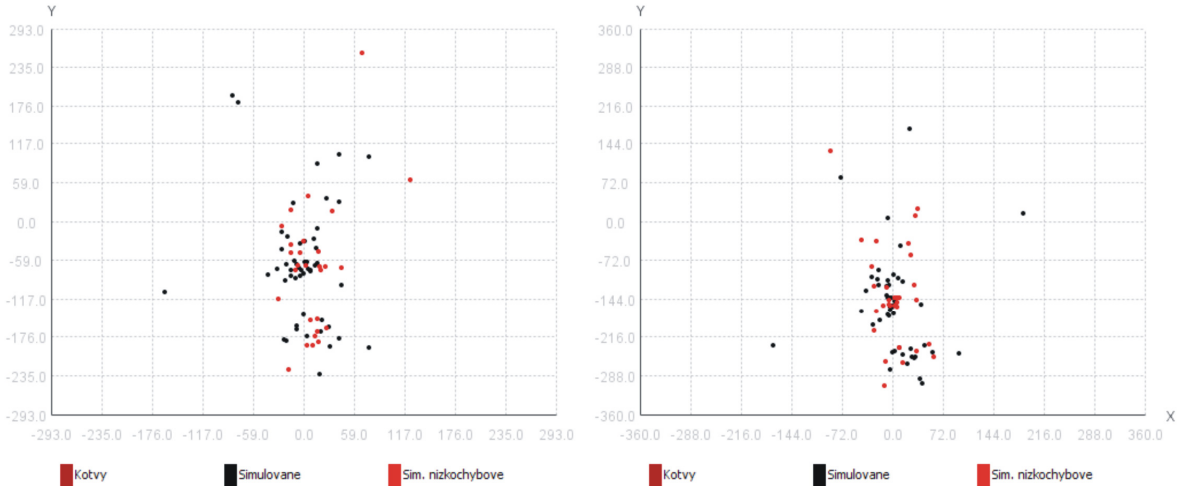
Následující obr. 5.21 až 5.23 ukazují rozmístění jednotlivých uzlů v souřadnicovém prostoru, které odpovídají simulacím zobrazeným na obr. 5.18 až 5.20. Červeně zvýrazněné uzly představují stanice s chybovostí menší než 10 %. Obrázek vlevo zachycuje situaci po cca 1000 aktualizacích, obrázek vpravo zachycuje situaci po cca 2000 aktualizacích. Z obrázků je vidět, že část uzlů se shlukuje do větších shluků, což souvisí s tím, že tyto body se nacházejí ve stejné oblasti. Shluky souvisejí i s geografickým rozmístěním, protože tyto uzly jsou spojeny pomocí vysokokapacitních rychlých linek, ale jsou umístěny po celém světě. Uzly umístěné na stejném kontinentu představují jeden z mnoha shluků.



Obr. 5.21: Rozmístění uzlů po 1000 (2000) aktualizacích při $c_c = 0{,}25$, $c_e = 0{,}1$



Obr. 5.22: Rozmístění uzlů po 1000 (2000) aktualizacích při $c_c = 0{,}25$, $c_e = 0{,}5$

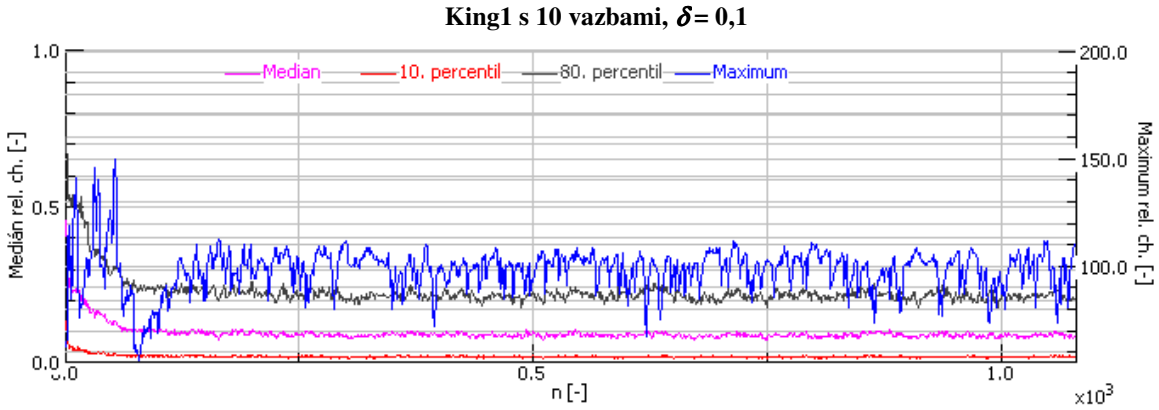


Obr. 5.23: Rozmístění uzlů po 1000 (2000) aktualizacích při $c_c = 0{,}25$, $c_e = 0{,}9$

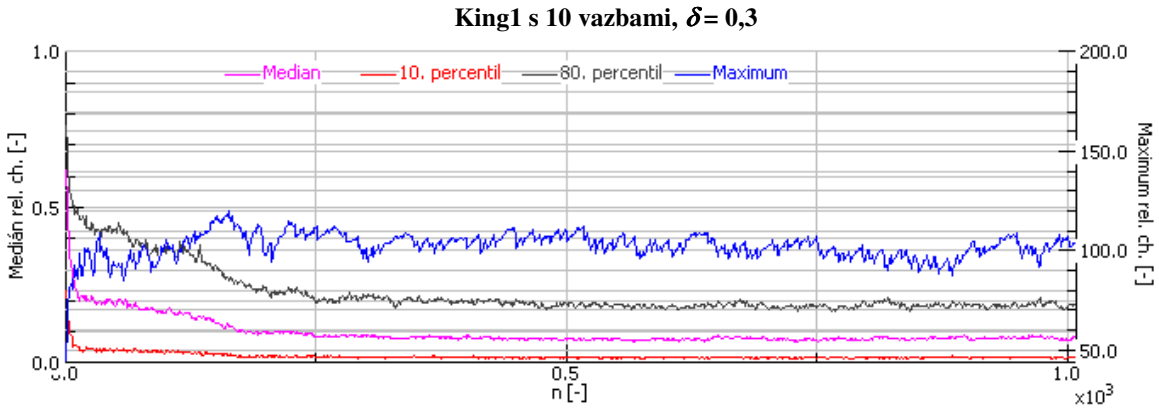
Tab. 5.5 ukazuje výsledek měření na stejných sítích jako v předchozím případě, avšak za použití varianty algoritmu Vivaldi s konstantním časovým krokem. Naměřené hodnoty základních parametrů sítě byly získány opět po provedení cca 1000 aktualizací. Z provedených simulací je patrné, že přesnost predikce je nejlepší při volbě parametru δ kolem hodnoty 0,1. Relativní chyba predikce RTT se v tomto případě pohybuje opět zhruba do 10 % Naměřené hodnoty potvrzují, že na přesnost predikce má minimální vliv počet vazeb u každého uzlu.

Tab. 5.5: Vliv parametru δ a počtu vazeb na přesnost predikce v experimentální síti

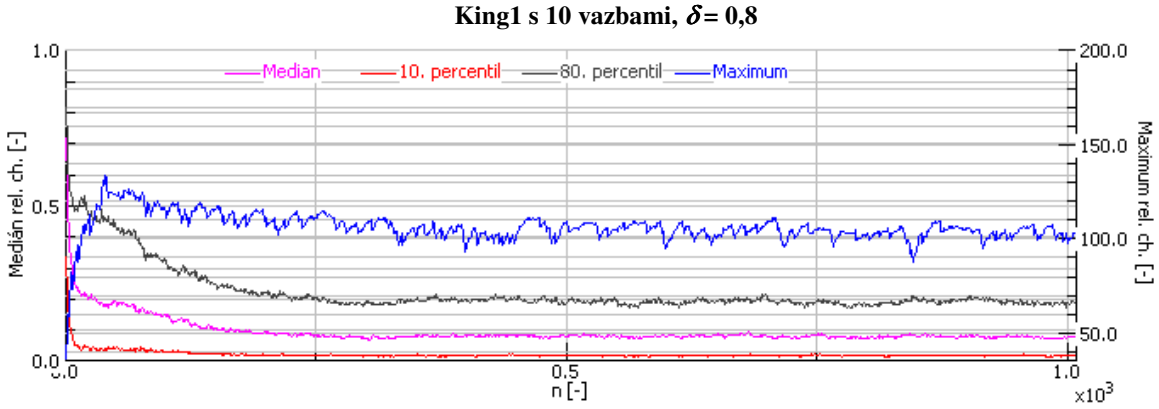
	Medián [%]		
Počet vazeb	**$\delta = 0{,}1$**	**$\delta = 0{,}3$**	**$\delta = 0{,}8$**
10	10,145	9,809	15,385
20	9,753	10,347	18,561
50	9,765	11,98	17,983



Obr. 5.24: Vliv parametru δ na rychlost a přesnost konvergence při $\delta = 0{,}1$ (Vivaldi)

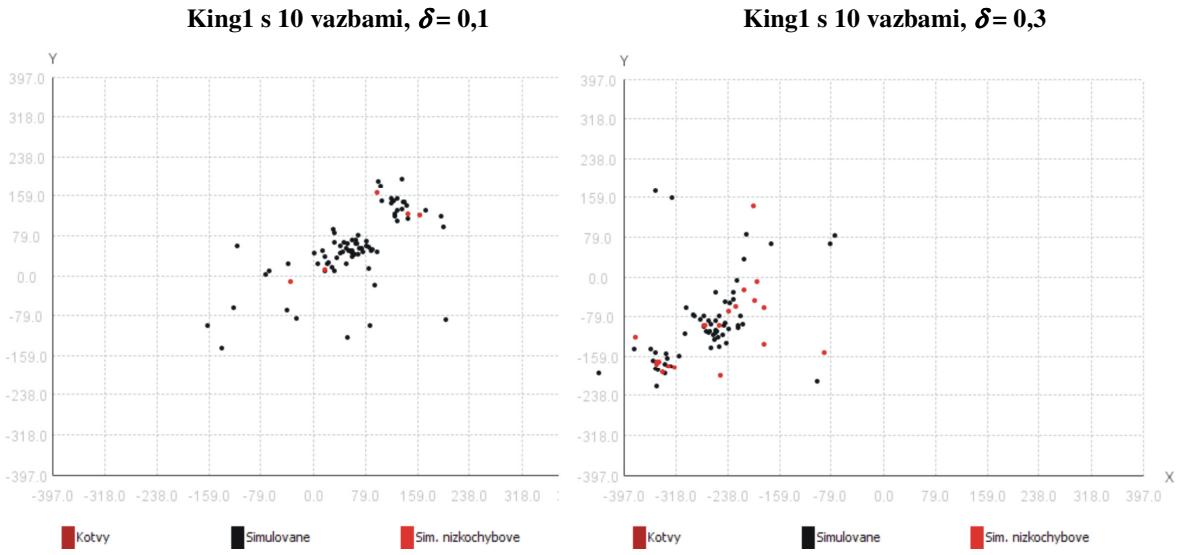


Obr. 5.25: Vliv parametru δ na rychlost a přesnost konvergence při δ = 0,3 (Vivaldi)

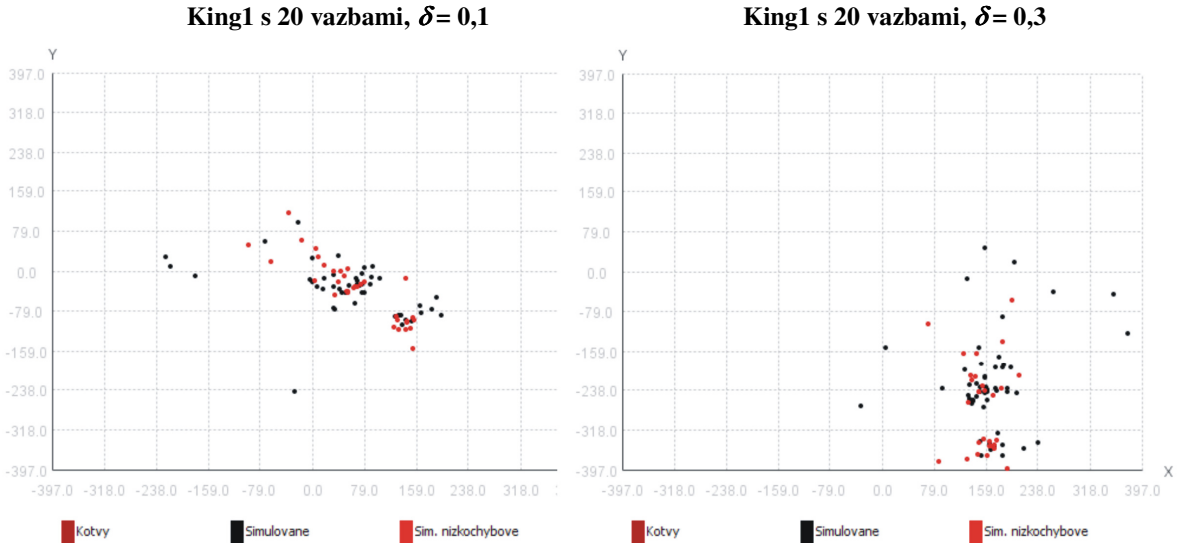


Obr. 5.26: Vliv parametru δ na rychlost a přesnost konvergence při $\delta = 0{,}8$ (Vivaldi)

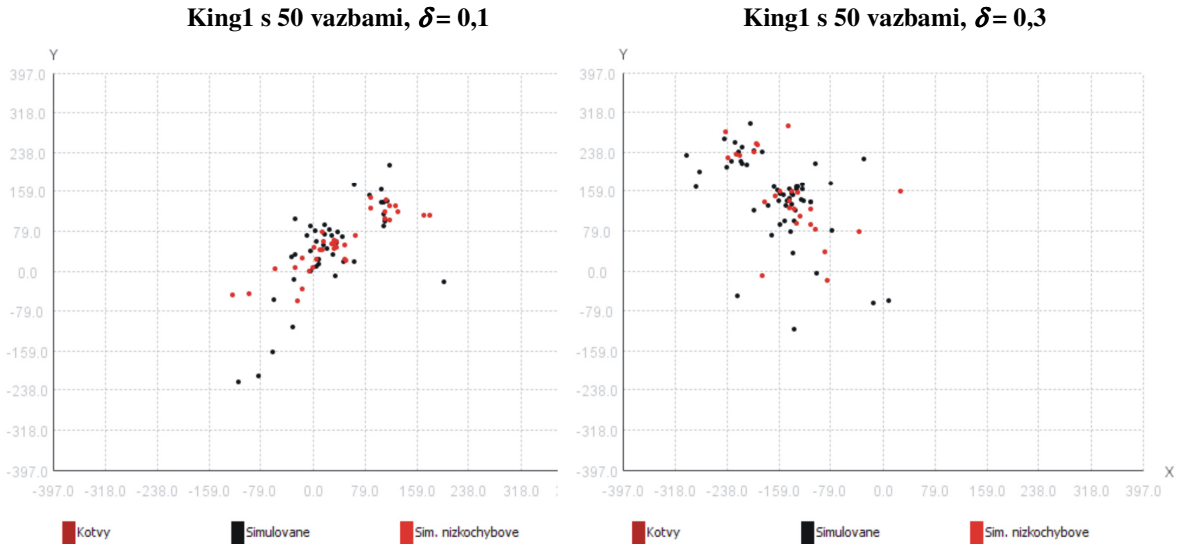
Následující obr. 5.27 až 5.29 znázorňují rozmístění jednotlivých uzlů v souřadnicovém prostoru při $\delta = 0{,}1$ vlevo a $\delta = 0{,}3$ vpravo v závislosti na počtu vazeb. Červeně zvýrazněné uzly představují stanice s chybovostí menší než 10 %. Z obrázků je vidět, že část uzlů se shlukuje do větších shluků což souvisí s tím, že tyto body se nacházejí ve stejné oblasti. Uzly umístěné na stejném kontinentu představují jeden z mnoha shluků.



Obr. 5.27: King 1 - rozmístění uzlů v závislosti na počtu vazeb, při 10 vazbách



Obr. 5.28: King 1 - rozmístění uzlů v závislosti na počtu vazeb, při 20 vazbách



Obr. 5.29: King 1 - rozmístění uzlů v závislosti na počtu vazeb, při 50 vazbách

5.4 Vyhodnocení simulací

Všechny simulace byly prováděny s oběmi variantami algoritmu Vivaldi pomocí programu VIVALDIMONITOR. První část simulací byla zaměřena na chování těchto variant na uměle vytvořených sítích. Základní vlastností těchto sítí je, že hodnota vzdálenosti mezi uzly v námi definovaném prostotu představující hodnotu RTT a při námi definovaném rozmístění uzlů je konstantní. Zatímco v případě testování a simulace na datech získaných ze sítě PlanetLab se jedná o náhodné rozmístění uzlů a hodnota RTT získaná měřením je získána na základě dlouhodobějších měření a je průměrnou hodnotou vyznačující se určitou chybou. V případě simulace na umělých sítích 15×15 a 20×20 uzlů obě varianty algoritmu konvergovaly velice rychle, zhruba během prvních 300 aktualizacích, jak je vidět na zobrazených průbězích obr 5.2 až 5.17. Uzly v síti se uspořádaly tak, že predikovaná hodnota RTT byla odhadnuta s minimální chybou menší než 1 %. V případě konstantního časového kroku byla optimální volba δ kolem hodnoty 0,3. V tomto případě konvergovala síť do optimálního rozmístění nejrychleji. V případě použití proměnného časového kroku síť nejrychleji konvergovala při nastavení ladících parametrů $c_c = 0{,}9$ a $c_e = 0{,}5$. Vliv počtu vazeb každého uzlu neměl příliš velký vliv na rychlost konvergence a přesnost predikce. Tato skutečnost je dána hlavně tím, že sousedé jsou v rámci této simulace voleni zcela náhodně a nevíme tedy, kteří se usadí dříve a kteří později. Při reálném nasazení by jako sousedé pro získání polohy byly voleny ty stanice, se kterými by komunikovaly. V případě peer - to - peer sítí by se jednalo o významné servery. V případě testů na části dat získaných z měření v síti PlanetLab pod označením King1, byla chyba predikce při nastavení nejvhodnějších parametrů zhruba do 10 %. Nepatrně lepšího výsledku dosáhla varianta s proměnným časovým krokem. Optimální volba ladících parametrů byla kolem $c_c = 0{,}25$ a $c_e = 0{,}5$. V případě varianty s konstantním časovým krokem byla ideální δ v rozmezí 0,1 až 0,3. Výsledky simulací korespondují s výsledky jiných zahraničních studií [11,13], které se touto problematikou zabývají. Potvrzují, že přesnost odhadu na základě algoritmu Vivaldi s proměnným časovým krokem se pohybuje zhruba kolem 10 %. Varianta s pevným časovým

krokem v reálné síti nedosahuje takové přesnosti jako varianta s proměnným časovým krokem, neboť v případě méně přesných informací o vzdálenosti k sousedům se síť více rozkmitává a není schopna se ustálit. Navíc na rozdíl od druhé varianty nebere uzel v případě aktualizace své polohy vůči jinému uzlu v potaz vlastní chybu souřadnic druhého uzlu. Aktualizuje svoji polohu stejně jako v případě, že komunikuje s uzlem o minimální chybě (bod, který už je ustálen), tak i v případě, že komunikuje s uzlem o vysoké chybě. Algoritmus s proměnným časovým krokem již při každé aktualizaci bere tento faktor v úvahu. Ale jak potvrzují studie [11,13], je u algoritmu Vivaldi v obou variantách nebezpečí v případě útoku zvenku, kdyby útočník do sítě úmyslně vložil uzly s falešnými údaji o svojí poloze. V takovémto případě již při 5 % falešných chybových uzlů není síť schopna zkonvergovat.

6 Závěr

Tato práce byla zaměřena na studium systémů, které slouží k odhadu pozic stanic v síti Internet a predikci jejich vzájemné vzdálenosti. Oficiálně nejsou přijaty žádné standardy, které by se touto problematikou zabývaly a stanovovaly zásady, jaké systémy a jak mají být používány. V této práci jsem se zaměřil na studium algoritmu Vivaldi. Tento algoritmus představuje distribuovaný, decentralizovaný algoritmus, který využívá umělé souřadnicové systémy k odhadům zpoždění mezi stanicemi v síti. Díky decentralizovanému charakteru tohoto algoritmu se jeví jako vhodný pro překryvné sítě typu peer-to-peer. Vyvinutá knihovna Vivaldi-lib implementuje obě varianty algoritmu a to s konstantním a proměnným časovým krokem. Jako souřadnicový systém byl zvolen 2-rozměrný prostor s třetím parametrem výškou. Tato knihovna byla současně použita v diplomové práci spolupracovníka Bc. Jaroslava Švédy. Ten vytvořil grafické ovládací rozhraní VIVALDIMONITOR pro tuto knihovnu. Tato aplikace slouží k sestavení simulace, jejímu ovládání a vyhodnocení. Správná funkčnost knihovny byla ověřena řadou simulací.

Další část této práce byla zaměřena na simulace chování implementovaného algoritmu na datech získaných z umělých a experimentálních sítí. Všechny simulace byly prováděny pomocí aplikace VIVALDIMONITOR. Provedené simulace potvrdily, že tento algoritmus se jeví jako velice vhodný pro nasazení v reálných sítích. Na datech získaných z umělých sítí byla chybovost predikce menší než 1 %. Na datech získaných z experimentálních sítích se chybovost pohybovala kolem 10 %, což koresponduje s výsledky i jiných studií [11,13]. Lepší chování a vyšší přesnost je u varianty s proměnným časovým krokem, která se lépe vypořádá i se změnami v sítí. K přesnějším výsledkům a k možnosti stanovení nastavení základních parametrů tohoto algoritmu je nutné provést řadu dalších rozsáhlejších simulací na větším množství vstupních dat včetně dat získaných z provozu v reálných sítích. Vyvinutá knihovna Vivaldi-lib a aplikace VIVALDIMONITOR může sloužit právě pro následné studium a testování samotného algoritmu.

Literatura

[1] JAROŠOVÁ, L. Měření vzdáleností stanic prostřednictvím ICMP protokolu v IP sítích. Brno: Vysoké učení technické v Brně, Fakulta Elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2001, počet stran 63.Vedoucí bakalářské práce Ing. Radim Burget.

[2] ŠVÉDA, Jaroslav. Nalezení pozice stanice v internetu pomocí umělé vytvořených souřadnicových systémů. [s.l.], 2008. 79 s. VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ. FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ.

[3] Theory.org Wiki. Bittorrent Protocol Specification v1.0 [online, cit. 14. 12. 2008]. URL: <http://wiki.theory.org/BitTorrentSpeci_cation>

[4] Ng, T. - Zhang, H. *Predicting Internet network distance with coordinates- based approaches* [online]. In INFOCOM 2002. *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings.* IEEE, Vol. 1 (2002).
URL:<http://www.cs.rice.edu/~eugeneng/papers/INFOCOM02.pdf>

[5] KaZaA media dekstop. http://www.kazaa.com/

[6] N. Priyantha, H. Balakrishnan, E. Demaine, S. Teller. *Anchor-free distribuded localization in sensor network.* Technical Report TR-892, MIT LCS, April 2003.
URL: <http://cricket.csail.mit.edu/papers/TechReport892.pdf>

[7] C. Savarese, J.M. Rabaty, J. Beutel. *Locationing in distributed ad-hoc wireless sensor network.* In *ICASSP*, strana 2037-2040, May 2001.

URL:<http://bwrc.eecs.berkeley.edu/publications/2001/Locatng_distrb_ad-hoc_wrlss_snsr_ntwks/icassp2001_final.pdf>

[8] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. *Lighthouses for scalable distributed location*. In IPTPS, 2003.

[9] FRANCIS, P. IDMaps: A Global Internet Host Distance Estimation Service.
In IEEE/ACM Trans. on Networking [online], New York : [s.n], October 2001.
[cit. 7. 12. 2008].
URL: <http://idmaps.eecs.umich.edu/papers/ton01.pdf>

[10] H. Hoppe. *Surface reconstruction from unorganized points.* Dizertační práce, Department of Computer Science and Engineering, University of Washington, 1994.
URL: <http://www.ieee-infocom.org/2003/papers/47_02.PDF>

[11] BAŠTINEC, J.- NOVÁK, M. Moderní numerické metody [online]. Brno: Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. Ústav telekomunikací, 2009.
URL: <http://www.umat.feec.vutbr.cz/ novakm/MMNM2009.pdf> [cit. 2. 5. 2009].

[12] DABEK, F., COX, R., KAASHOEK, F., MORRIS, R. *Vivaldi: a decentralized network coordinate system*. ACM SIGCOMM Computer Communication Review. Association for Computing Machinery, 2004. ISSN: 0146-4833.

[13] M. Costa, M. Astro, A. Rownstrom, P. Key. *PIC: Practical Internet coordinates for distance estimation*. In *International Konference on Distributed Systeme,* Tokyo, Japan, March 2004.
URL: < http://www.cl.cam.ac.uk/Teaching/2003/AdvSysTop/pic.pdf >

[14] David John Zage, Cristina Nita-Rotaru. *On the Accuracy of Decentralized Virtual Coordinate Systeme in Adversarial Network* [online]. ACM CCS 2007.
URL: < http://www.cs.purdue.edu/homes/zagedj/docs/ccs050-zage.pdf>

[15] Y. Shavitt and T. Tankel. *Big-bang simulation for embedding network distances in Euclidean space*. In Proc. of IEEE Infocom, April 2003.

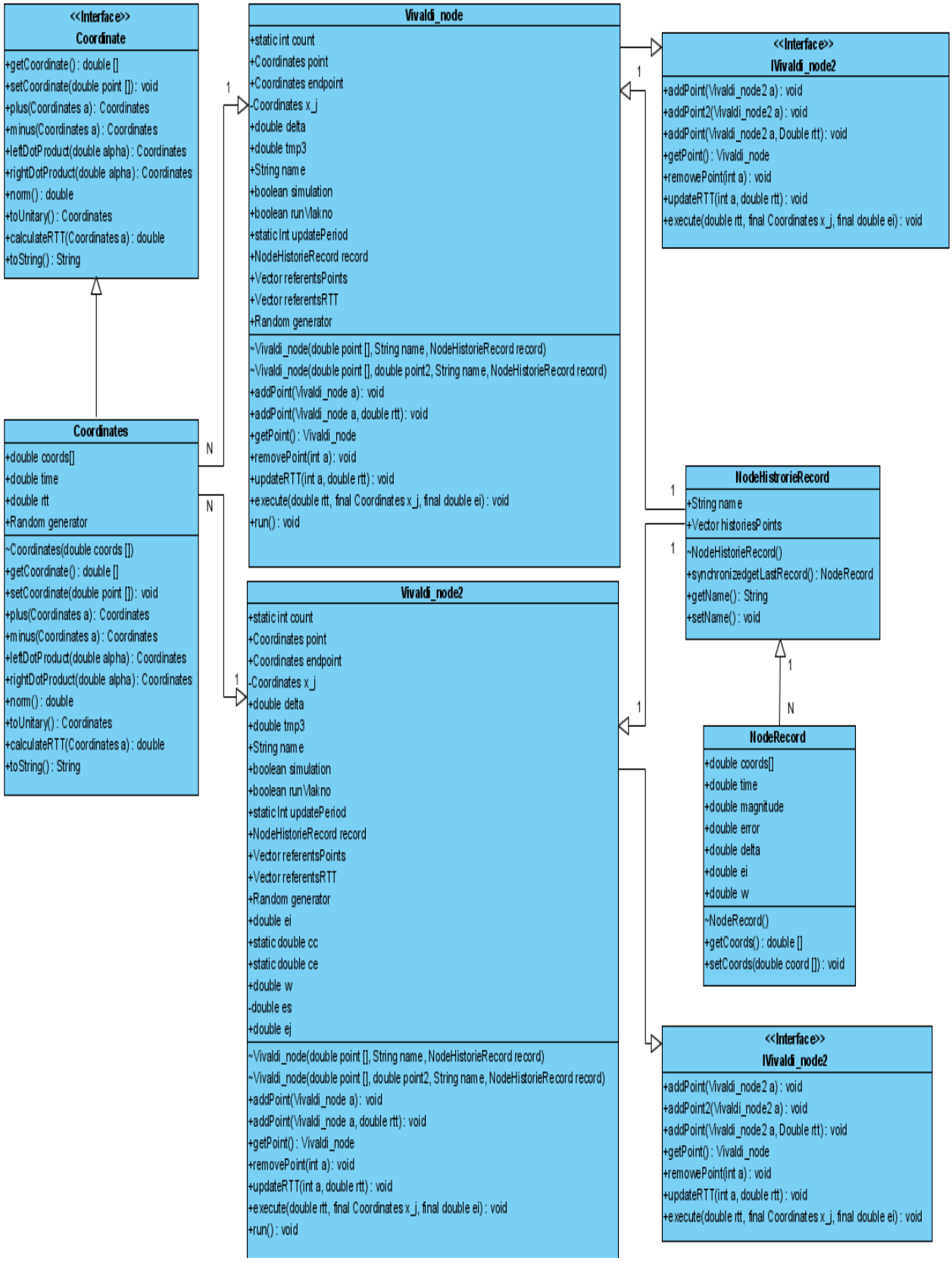
[16] The King data set [online]. Cambridge (USA) : Massachusetts Institute of Technology. Computer Science and Artificial Intelligence Laboratory, 2003.
[cit. 12. 4. 2009]

Seznam zkratek

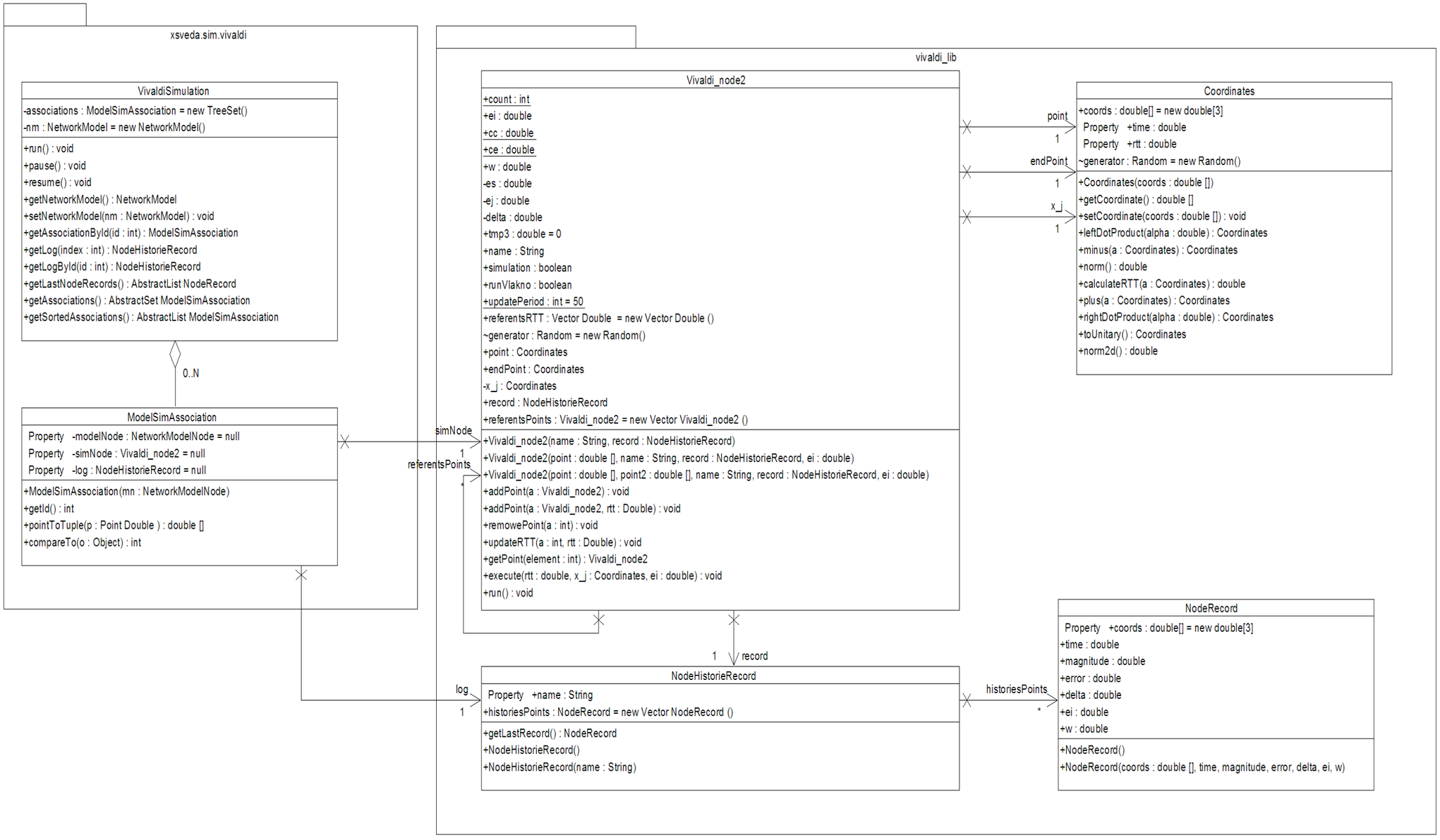
Zkratka	Anglický význam	Český význam
ABC	Assumption Based Coordinates	Decentralizovaný algoritmus pro určení polohy stanice v bezdrátových sítích.
AFL	Anchor-Free Localization	Decentralizovaný algoritmus pro určení polohy stanice v bezdrátových sítích.
DNS	Domain Name System	Hierarchický systém doménových jmen, který je realizován servery DNS a protokolem stejného jména.
GNP	Global Network Positioning	Centralizovaný syntetický souřadnicový systém pro určení polohy stanice v Internetu.
IDMAPS	Internet Distance Map Service	Centralizovaný syntetický souřadnicový systém pro určení polohy stanice v Internetu.
ICQ	I Seek You	ICQ je software pro instant messaging využívající protokolu OSCAR.
JAVA		Java je objektově orientovaný programovací jazyk.
JVM	Java Virtual Machina	Interpret zkompilovaných javovských programů.
NAT	Network Address Translation	Způsob úpravy síťového provozu přes router přepisem výchozí, nebo cílové IP adresy, často i se změnou čísla TCP/UDP portu u průchozích IP paketů.
NPS		Centralizovaný syntetický souřadnicový systém pro určení polohy stanice v Internetu.
P2P	Peer-to-peer	Peer-to-peer (doslova rovný s rovným), klient-klient je označení architektury počítačových sítí, ve které spolu komunikují přímo jednotliví klienti (uživatelé).
PIC		Decentralizovaný syntetický souřadnicový systém pro určení polohy stanice v Internetu.
Ping	Packet InterNet Groper	Program ping umožňuje prověřit funkčnost spojení mezi dvěma síťovými rozhraními v počítačové síti, která používá rodinu protokolů TCP/IP.
RTT	Round trip time	Velikost zpoždění mezi dvěma hostiteli v síti, udává se často v ms.
SKYPE		Peer-to-peer program, který umožňuje provozovat internetovou telefonii.
VOIP	Voice over IP	Internetová telefonie založená na TCP/IP.
δ		Konstantní časový krok v Algoritmu Vivaldi.

Seznam příloh

Příloha A: UML diagram knihovny Vivaldi-lib



Příloha B: UML diagram celého simulačního programu



Příloha C: Zdrojový kód třídy Vivaldi_node

```
public class Vivaldi_node extends Thread implements IVivaldi_node {
    public static int count;     // čítač počtu objektu
    public Coordinates point;    // startovací bod simulace
    public Coordinates endPoint;
    private Coordinates x_j;// koncový bod simulace
    public double delta = 0.5;   // proměnná algoritmu Vivaldi
    public double tmp3 = 0;      // časová proměnná
    public String name;          // jméno uzlu
    public boolean simulation;   // určuje zdali se provádi simulace
    public boolean runVlakno;    // urdržuje platnost metody run()
    public static int updatePeriod = 50;
    // proměnná record slouži k ukládání historie bodu
    public NodeHistorieRecord record;// = new NodeHistorieRecord() ;
    // proměnná udržující informace o referenčních bodech
    public Vector<Vivaldi_node> referentsPoints = new Vector<Vivaldi_node>();
   // proměnná udržující informace o přislušných hodnot. RTT bodu vůči referenčním bodům
    public Vector<Double> referentsRTT = new Vector<Double>();
    Random generator = new Random();
/* Definice konstruktorů  */
public Vivaldi_node(String name, NodeHistorieRecord record) {
    double point []= new double [3];
    for (int i = 0; i < point.length; i++) {
         point[i]=this.generator.nextDouble();
    }
    this.point = new Coordinates(point);
    this.endPoint= new Coordinates(point);
    this.name = name;
    this.record = record;
    this.record.name = this.name;
   record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3, 0,0,this.delta,0,0));
   this.count++;
}
public Vivaldi_node(double point[], String name, NodeHistorieRecord record) {
    if (point[2]<0.0000001) {
        point[2]=0.0000001;}
    this.point = new Coordinates(point);
    this.endPoint= new Coordinates(point);
    this.name = name;
    this.record = record;
    this.record.name = this.name;
```

```
    record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3, 0,0,this.delta,0,0));
    this.count++;
}
public Vivaldi_node(double point[], double point2[], String name, NodeHistorieRecord
record) {
     if (point[2]<0.0000001) {
          point[2]=0.0000001;
     }
     if (point2[2]<0.0000001) {
        point2[2]=0.0000001;
     }
     this.point = new Coordinates(point)
     this.endPoint = new Coordinates(point2);
     this.name = name;
     this.record = record;
     this.record.name = this.name;
     record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3, 0,0,this.delta,0,0));
    this.count++;
}
/* Definice metod a funkci */
public void addPoint(Vivaldi_node a) {
     this.referentsRTT.add(endPoint.calculateRTT(a.endPoint));
     this.referentsPoints.addElement(a);
     count++;
 }
public void addPoint(Vivaldi_node a, Double rtt) {
     this.referentsRTT.add(rtt);
     this.referentsPoints.addElement(a);
}
public void removePoint(int a) {
     if (a<referentsPoints.size()) {
     referentsPoints.remove(a);
     referentsRTT.remove(a);
     }
}
public void updateRTT(int a, Double rtt) {
     if (a<referentsPoints.size()) {
     referentsRTT.set(a, rtt);
    }
}
```

```
 public Vivaldi_node getPoint(int element) {
     return referentsPoints.get(element);
}
public void execute(double rtt, final Coordinates x_j) {
   this.x_j=x_j;
   Coordinates x_i_x_j = point.minus(this.x_j); // x_i - x_j
   double predikceRTT = x_i_x_j.norm();//| x_i - x_j ||
   double e = rtt-predikceRTT;
   Coordinates dir = x_i_x_j.toUnitary();        // u(x_i - x_j)
   double tmp1 = delta * (rtt - predikceRTT);        // delta * (rtt - || x_i - x_j||)
   Coordinates tmp2 = dir.leftDotProduct(tmp1); // delta * (rtt - || x_i - x_j||) * u(x_i - x_j)
   this.point = this.point.plus(tmp2);           // xi = x_i + delta * (rtt - || x_i - x_j||) * u(x_i - x_j)
   if (this.point.coords[2]<0.0000001) {
      this.point.coords[2]=0.0000001;
  }
  this.tmp3++;
  this.point.setTime(this.tmp3)
  synchronized (this) {
       record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3,
       x_i_x_j.norm(),e,this.delta,0,0));
  }
}
 public void run() {
    this.runVlakno = true;
    synchronized (this) {
         this.simulation = true;
    }
    while (this.runVlakno) {
         if (this.simulation) {
             int j;
             j = this.generator.nextInt((this.referentsPoints.size()));
            execute(this.referentsRTT.get(j).doubleValue(), this.referentsPoints.get(j).point);
            try { Thread.sleep(Vivaldi_node.updatePeriod)
                 } catch (InterruptedException e) {e.printStackTrace();}
         } else{ try {    Thread.sleep(5000);// vlákno se uspí}
                   catch (InterruptedException e)
                   e.printStackTrace() }
                 }
   }
   this.count--;// sníží se proměnná udržující informaci o počtu instancí
  System.out.println("konec vlákna");}}
```

Příloha D: Zdrojový kód třídy Vivaldi_node2

```
public class Vivaldi_node2 extends Thread implements IVivaldi_node2 {
   public static int count;     // čítač počtu objektu
   public Coordinates point;    // startovací bod simulace
   public Coordinates endPoint;
   public double ei; // hodnota vlastní chyby
   public static double cc; // ladící parametry cc (0..1)
   public static double ce; // ladící parametry ce (0..1)
   public double w; // váhový vektor
   private double es;// kumulovaná chyba
   private Coordinates x_j;// koncový bod simulace
   private double ej; // lokální chyba druhého bodu
   private double delta;   // proměnná algoritmu Vivaldi
   public double tmp3 = 0;      // časová proměnná
   public String name;          // jméno bodu
   public boolean simulation;   // určuje zdali se provádí simulace
   public boolean runVlakno;    // urdržuje platnost metody run()
   public static int updatePeriod = 50;
   // proměnná record slouži k ukládáni historie bodu
   public NodeHistorieRecord record;// = new NodeHistorieRecord() ;
   // proměnná udržující informace o referenčních bodech
      public Vector<Vivaldi_node2> referentsPoints = new Vector<Vivaldi_node2>();
   // proměnná udržující informace o příslušných hodnot. RTT bodů vůči referenčním bodům
   public Vector<Double> referentsRTT = new Vector<Double>();
   Random generator = new Random();
 /* Definice konstruktoru*/
public Vivaldi_node2(String name, NodeHistorieRecord record) {
      double point []= new double [3];
      for (int i = 0; i < point.length; i++) {
                        point[i]=this.generator.nextDouble();
               }
      this.point = new Coordinates(point);
      this.endPoint= new Coordinates(point);
      this.name = name;
      this.record = record;
      this.record.name = this.name;
      record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3, 0,this.delta,0,0));
      this.count++;
}
public Vivaldi_node2(double point[], String name, NodeHistorieRecord record,double ei) {
      if (point[2]<0.0000001) {
               point[2]=0.0000001;
         }
      this.point = new Coordinates(point);
      this.endPoint= new Coordinates(point);
      this.name = name;
      this.record = record;
      this.record.name = this.name;
      this.ei=ei;
      record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3,
```

```
        0,0,this.delta,this.ei,0));
        this.count++;
}
public Vivaldi_node2(double point[], double point2[], String name, NodeHistorieRecord
record,double ei) {
      if (point[2]<0.0000001) {
                  point[2]=0.0000001;
      }
      if (point2[2]<0.0000001) {
                  point2[2]=0.0000001;
      }
      this.point = new Coordinates(point);
      this.endPoint = new Coordinates(point2);
      this.name = name;
      this.record = record;
      this.record.name = this.name;
      this.ei=ei;
      record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp,this.delta,this.ei,w));
      this.count++;
}
 / * Definice metod a funkci */
public void addPoint(Vivaldi_node2 a) {
        this.referentsRTT.add(endPoint.calculateRTT(a.endPoint));
        this.referentsPoints.addElement(a);
        count++;
}
public void addPoint(Vivaldi_node2 a, Double rtt) {
        this.referentsRTT.add(rtt);
        this.referentsPoints.addElement(a);
}
 public void remowePoint(int a) {
          if (a<referentsPoints.size()) {
              referentsPoints.remove(a);
              referentsRTT.remove(a);
           }
}
 public void updateRTT(int a, Double rtt) {
          f (a<referentsPoints.size()) {
               referentsRTT.set(a, rtt);
          }
}
public Vivaldi_node2 getPoint(int element) {
        return referentsPoints.get(element);
}
public void execute(double rtt, final Coordinates x_j, final double ei) {
          this.x_j=x_j;
          this.ej=ei;
          if (this.ei!=0) {
              this.w=this.ei/(this.ei+this.ej);              // w=ei/(ei+ej)
```

```
          }else {
                this.w=0;
          }
        Coordinates x_i_x_j = point.minus(this.x_j); // x_i - x_j
        double predikceRTT = x_i_x_j.norm();  //|| x_i - x_j ||
        if ((predikceRTT-rtt)!=0) {
          this.es=(Math.abs(predikceRTT-rtt))/rtt; //es=|||xi-xj||-rttl/rtt
        }else {this.es=0;
                }
        this.ei=this.es*this.ce*this.w+this.ei*(1-this.ce*this.w);
        this.delta=this.cc*this.w;
        double e = rtt-predikceRTT;
        Coordinates dir = x_i_x_j.toUnitary();// u(x_i - x_j)
        double tmp1 = this.delta * (rtt - predikceRTT); // delta * (rtt - || x_i - x_j||)
        Coordinates tmp2 = dir.leftDotProduct(tmp1); // delta * (rtt - || x_i - x_j||) * u(x_i - x_j)
        this.point = this.point.plus(tmp2);      // xi = x_i + delta * (rtt - || x_i - x_j||) * u(x_i - x_j)
        if (this.point.coords[2]<0.0000001) {
                  this.point.coords[2]=0.0000001;
        }
        this.tmp3++;
        this.point.setTime(this.tmp3);
        synchronized (this) {
              record.historiesPoints.add(0, new NodeRecord(this.point.coords, tmp3,
              x_i_x_j.norm(),e,this.delta,this.ei,this.w));
        }
}
public void run() {
    this.runVlakno = true;
    synchronized (this) {
          this.simulation = true;
       }
   while (this.runVlakno) {
         if (this.simulation) {
             int j;
              j = this.generator.nextInt((this.referentsPoints.size()));
            execute(this.referentsRTT.get(j).doubleValue(),
                      this.referentsPoints.get(j).point,this.referentsPoints.get(j).ei);
             try {Thread.sleep(Vivaldi_node2.updatePeriod);}
            catch (InterruptedException e) { e.printStackTrace();}
         } else{  try { Thread.sleep(5000); }// vlákno se uspí
                   catch (InterruptedException e) { e.printStackTrace();}
                  }
    }
    this.count--;// sníží se proměnná udržující informaci o počtu instancí
    System.out.println("konec vlakna")
 }
}
```