Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Experimentální využití algoritmu AntNet v prostředí aktivních sítí

Plzeň 2012

Zdeněk Vacek

Originál zadání

PROHLÁŠENÍ

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. května 2012

Zdeněk Vacek

ABSTRACT

This thesis presents an experimental use of the AntNet routing algorithm to perform a so-called worse-path routing. The AntNet algorithm was proposed for IP networks with the use of Mobile Agents to benefit from a programmable environment. I adapted the AntNet algorithm for a programmable-network server called Smart Active Node. This server adheres to the paradigm of Active Networks. As an experimental implementation of Quality of Service, I modified the AntNet algorithm to find worse-routes in the network. In theory, this should make possible to re-route non-real time traffic to less-congested links to reduce an overall packet-drop ratio.

PODĚKOVÁNÍ

Rád bych poděkoval mým rodičům za jejich nezměrnou podporu a pomoc v celém průběhu mého studia.

Rovněž bych chtěl poděkovat vedoucímu mé diplomové práce Ing. Tomáši Koutnému, Ph.D. za jeho vstřícný přístup, podnětné návrhy a čas, jenž mi během mé práce věnoval.

Dále pak kolegovi Ing. Vladimíru Aubrechtovi, který mi během vývoje nejednou svojí činností pomohl a to především všelijakými úpravami celého projektu SAN Serveru.

Na závěr bych rád poděkoval všem mým přátelům a kolegům a to za podporu a povzbuzování během studia, bez kterých bych nejspíš již dávno nepatřil do řad studentů Západočeské univerzity.

OBSAH

1 Úvod **1**

2 Teorie a Analýza **2**

2.1 Směrování, proč a jak 2

2.1.1 Co je to směrování 2

2.1.2 Co a k čemu je směrovací algoritmus 2

2.2 Grade32 3

2.3 Smart Active Node 4

2.3.1 Představení projektu SanC++ 4

2.3.2 Prostředí aktivních programovatelných sítí 4

2.3.3 Struktura projektu 5

2.3.4 Hlavní implementační rozdíly verzí SANu 5

2.3.5 Stav projektu před touto prací 7

2.4 Proč zrovna AntNet 8

3 AntNet **9**

3.1 Základní pojmy 9

3.2 Princip algoritmu 10

3.2.1 Vysílání kapsule 12

3.2.2 Doručování kapsule 13

3.2.3 Návrat kapsule 16

3.2.4 Metriky 16

3.2.5 Shromažd'ování směrovacích záznamů 17

3.2.6 Aktualizace směrovacích tabulek 19

3.2.7 Odebírání směrovacích záznamů 19

3.3 Další vlastnosti algoritmu 20

3.4 Rozšíření AntNetu 20

4 Směrování horší cestou **22**

4.1 Analýza 23

4.2 Potřebné úpravy 25

4.2.1 Problematika výchozích bran 26
4.2.2 Aktualizování směrovacích tabulek uzlu 27
4.3 Experiment 28
4.3.1 Testovací topologie 1 29
4.3.2 Testovací topologie 2 30
4.3.3 Testovací nástroj TraceRoute 31
4.3.4 Přidané příkazy uzlu 32

5 Implementace 34
5.1 AntNet.cpp a AntNet.h 35
5.2 MasterCode.cpp a MasterCode.h 35
5.2.1 Atributy třídy 35
5.2.2 Start() 37
5.2.3 Stop() 37
5.2.4 Initialization() 37
5.2.5 InitRoutingAndNeighborTable() 37
5.2.6 FillApplicationRoutingTable() 37
5.2.7 FillNeighborTable() 38
5.2.8 HoldTimerThreadLoop() 38
5.2.9 ProcessAppRouteTable() 38
5.2.10 ProcessNodeRouteTable() 38
5.3 SlaveCode.cpp a SlaveCode.h 38
5.3.1 Atributy třídy 38
5.3.2 ProcessCapsule() 39
5.3.3 BackwardAntProcessing() 39
5.3.4 ForgetAddressFrom() 39
5.3.5 ForwardAntProcessing() 40
5.3.6 InCycle() 40
5.3.7 IsAddressOfThisServer() 40
5.3.8 ProcessReturnedCapsule() 40
5.3.9 MergeRoutingRecords() 41
5.3.10 UpdateTablesInStorage() 41
5.3.11 UpdateServerRoutingTables() 41
5.4 AntNetCapsule.cpp a AntNetCapsule.h 41
5.4.1 Atributy třídy 42
5.4.2 AddAddress() 43
5.4.3 AddRouteRecord() 43
5.4.4 Serialize() a DeSerialize() 44
5.5 AntNetRoutingTable.cpp a AntNetRoutingTable.h 44
5.5.1 Atributy třídy 44
5.5.2 AddDirectNeighborRouteRecord() 45
5.5.3 AddRouteRecord() 45
5.5.4 Serialize() a DeSerialize() 45
5.5.5 RemoveRouteRecord() 45
5.6 Utils.cpp a Utils.h 46

5.6.1 ConvertVectorRouteRecord() a ConvertVectorRouteRecord() . . . 46
5.6.2 CompareRouteRecords() . . . 46
5.6.3 CompareRouteRecordsByMetrics() . . . 46
5.6.4 CompareRouteRecordsByNetwork() . . . 46
5.6.5 CompareRouteRecordsForUpdate() . . . 47
5.6.6 GetNextInputAddress() . . . 47
5.6.7 GetRouteRecordByAddress() . . . 47
5.6.8 MergeRouteRecordsToNetwork() . . . 48
5.6.9 ReadTablesFromStorage() . . . 48
5.6.10 SelectRandomAddress() . . . 48
5.6.11 WriteTablesToStorage() . . . 48
5.7 DebugUtils.cpp a DebugUtils.h . . . 49
5.7.1 PrintAddressTable() . . . 49
5.7.2 PrintCapsuleData() . . . 49
5.7.3 PrintInterfaces() . . . 49
5.7.4 PrintRoutingTable() . . . 49
5.8 Defines.h . . . 50
5.8.1 Ladící makra . . . 50
5.8.2 Programové konstanty . . . 53
5.8.3 Návratové chybové kódy . . . 53
5.9 Další důležité soubory . . . 54
5.9.1 Typedefs.h . . . 54
5.9.2 IRouteRecord.h a RouteRecord.h . . . 54
5.9.3 ISerializable.h, Serializable.cpp a Serializable.h . . . 54
5.9.4 CCapsule.cpp a CCapsule.h . . . 55

6 Uživatelský manuál 56
6.1 Struktura konfiguračních souborů . . . 56
6.1.1 farmer.cfg . . . 57
6.1.2 worker.cfg . . . 57
6.1.3 worker_AntNet.cfg . . . 58
6.2 Spuštění . . . 58
6.2.1 SAN . . . 58
6.2.2 AntNet . . . 58

7 Dosažené výsledky 60
7.1 *AntNet* . . . 60
7.1.1 Testovací topologie 1 . . . 60
7.1.2 Testovací topologie 2 . . . 68

8 Závěr 70

Přehled zkratek 71

Použitá terminologie 72

Seznam obrázků **74**

Literatura **76**

Přílohy **77**

A) Příklad konfigurace SAN_Farmer 77

B) Příklad konfigurace SAN_Worker 80

KAPITOLA 1

Úvod

Dokument je rozčleněn do následujících kapitol. Druhá kapitola seznamuje čtenáře s pojmem směrování v počítačových sítích, vysvětluje co je to směrovací algoritmus, představuje projekt Smart Active Node (dále v textu *SAN*[1]) a obhajuje výběr implementovaného směrovacího algoritmu AntNet.

Popis algoritmu je obsahem třetí kapitoly, zatímco 4.kapitola dokumentuje provedené práce a vytvořené zdrojové kódy. Pátá kapitola rozebírá experiment se směrováním horší cestou, takže se zabývá analýzou potřebných rozšíření, dokumentací jejich realizací a prezentováním dosažených výsledků. Obsah 6.kapitoly tvoří dokumentace potřebné konfigurace, uživatelský manuál a přezentace výsledků implementace základního směrovacího algoritmu. Poslední číslovaná sedmá kapitola diskutuje a hodnotí dosažené výsledky.

Zbylé nečíslované kapitoly obsahují soupis referencí na použité zdroje, seznam vysvětlivek používaných termínů a přílohu v podobě ukázky konfiguračního souboru.

[1]SAN - Smart Active Node, experimentální projekt serveru aktivní programovatelné počítačové sítě (další informace na [1]).

KAPITOLA 2 Teorie a Analýza

2.1 Směrování, proč a jak

2.1.1 Co je to směrování

Směrování nebo-li routování je určení směru vysílání zpráv v počítačových sítích, takže si lze pod pojmem směrování představit hledání optimální cesty v dané síti mezi komunikujícími uzly. Vyslaná zpráva je obecně doručována z výchozího do cílového uzlu přes libovolný předem neznámý počet dalších uzlů. Tyto uzly si danou zprávu postupně předávají až do cílového uzlu, přičemž se každý uzel rozhoduje, jakým směrem jí pošle dále. Pokud by během kroků tohoto mechanismu nefungovalo žádné směrování, tak by vyslaná zpráva mohla být doručena na cílový uzel ve zbytečně dlouhém čase nebo by nemusela být doručena nikdy.

Činnost směrování tedy provádí každý uzel a představuje výběr směru pro jeden následující krok v cestě zprávy, nikoliv pro celou cestu. Vybraný směr, kterým zpráva z daného uzlu půjde sítí dál, je vybírán jako optimální na základě různých parametrů (např. datová propustnost, momentální vytížení, priorita, druh vedení, typ komunikačního protokolu apod.).

2.1.2 Co a k čemu je směrovací algoritmus

Počítačovou síť lze popsat grafem, kde uzly odpovídají síťovým prvkům a serverům, zatímco hrany odpovídají jejich propojení. Aby mezi sebou jednotlivé uzly mohly komunikovat, je třeba mechanismu, který najde cestu z jednoho uzlu na druhý - směrování.

Cest může existovat vícero, každá z nich je ohodnocena směrovací metrikou, podle které se pak uzel rozhodne, jakou z dostupných cest použije. Ale aby se zdrojový uzel vůbec mohl začít takto rozhodovat, tak musí znát právě onu topologii a případně nějaké další potřebné informace. Bez získání informací o topologii nelze posílané zprávy jakkoli směrovat sítí a stanice tak

mohou komunikovat pouze na slepo. Tj. zcela nahodnými směry a nebo například prostřednictvým záplavové metody, což je většinou velice neefektivní.

Směrovací záznam může administrátor vkládat ručně, za pomocí tzv. statických cest. Tento způsob se však z principu neumí vyrovnat s dynamickými změnami topologie sítě, např. výpadek spojení. Zároveň ani s rostoucím počtem uzlů a jejich propojení, protože roste složitost nutných úprav. Jako řešení se používají směrovací algoritmy, které sestaví příslušné směrovací záznamy a provedou jejich modifikaci při změně topologie. Samotný algoritmus reprezentuje mechanismus, resp. posloupnost kroků umožňující uzlu získat informace o „logické topologii[1]“ okolní sítě.

Cílem algoritmu je nalezení topologie okolní sítě a hlavně udržování v daném čase aktuálních směrovacích záznamů o její současné podobě v směrovacích tabulkách jednotlivých uzlů. Aktuálnost pouze v daném čase je dáná tím, že sít' se v čase všelijak mění. Mohou být průběžně připojovány nové uzly nebo naopak současné odpojovány nebo může docházet k výpadkům nebo vytvořením nových komunikačních linek.

Naplňování tohoto cíle spočívá v neustálem vysílání nových kapsulí. Jejich doručováním po různých náhodných trasách, dochází k neustálému dohledu nad znamými linkami a zároveň k poznávání nových spojení. Dohled nad známými linkami spočívá v neustálem obnovování informací o jejich momentálním stavu. V případě nějakého výpadku k obnovení informace o tomto spojení nedojde a příslušný směrovací záznam tak po určité době (dané časovačem) pozbude platnosti.

2.2 Grade32

Tento projekt představuje předchůdce projektu *Smart Active Node*. Projekt byl určen pro provádění komplikovaných výpočtů, dále sloužil ke studiu a experimentům s programovalnou počítačovou sítí.

Rozdíl programovatelné sítě proti síti klasické spočívá v tom, že uzel v klasické síti nezkoumá po přijetí zprávy její obsah, pouze ji podle informací v jejím záhlaví pošle dále. Zatímco uzel v programovatelné síti může nad přijatou zprávou vykonávat libovolný programový kód. Tato vlastnost právě přináší do oblasti počítačových sítí nové možnosti (například v oblasti bezpečnosti počítačových sítí).

Obsah projektu tvoří sít'ový server napsaný především v programovacím jazyce pascal. Jednotlivé instance tohoto serveru poté tvoří experimentální sít'. Samotný návrh včetně implementace obsahuje množství vyzkoušených a hlavně odladěných programátorských kontrukcí a standardů, které byly předlohou vnitřní architektuře veškerých verzí následného projektu *Smart Active Node*.

[1]Logická topologie - Virtuální uspořádání sítových uzlů vystihující přenosy dat. Oproti fyzické topologii (tj. fyzickému propojení uzlů) se může výrazně lišit (viz VLAN), ale ne nutně.

Na každém uzlu běží množina aktivních aplikací, které mezi sebou komunikují prostřednictvím zpráv, kapsulí. Tyto kapsule jsou obdobou standardních paketů z klasických sítí. Liší se akorát svojí vnitřní strukturou. Aktivní uzel přidává do kapsule proti paketu navíc mimo jiné identifikátor konkrétní aktivní aplikace, která jej vytvořila. Tento identifikátor spolu s implementovaným distribučním mechanismem aktivních aplikací zajišt'uje zpracovánání kapsule na všech navštívených uzlech podle naprosto totožné verze zdrojových kódů.

Kvůli přechodu na modernější programovací jazyky bylo od dalšího vývoje tohoto projektu postupně upuštěno a začala se vyvíjet nejprve Java a nyní C++ verze projektu *Smart Active Node*.

2.3 Smart Active Node

2.3.1 Představení projektu SanC++

Projekt *Smart Active Node* je již třetím vývojovým krokem experimentální implementace aktivního programovatelného serveru kódovaného v jazyce C++ vyvíjeného na ZČU[2], přičemž se vývojový team snaží o platformní nezávislost zdrojových kódů.

Jejich poslední verze vychází z konceptu předchůdce *Grade32* [3] a navazuje i na minulou implementaci projektu SAN kódovanou v programovacím jazyce Java. Vývoj v Javě byl zastaven především z důvodů dosažení rychlostních limitů programovacího jazyka, protože byl interpretovaný „Java bytecode[4]“ zkrátka příliš pomalý. Přesto, že byl vývoj obou projektů zastaven, tak doposud jejich zdrojové kódy slouží jako bohatá inspirace vývojařům současné C++ verze. Osobně jsem z těchto zdrojových kódů také čerpal.

2.3.2 Prostředí aktivních programovatelných sítí

Hlavní rozdíl mezi pasivní a aktivní programovatelnou sítí spočívá v tom, že pasivní uzel nijak neanalyzuje ani nezpracovává kód obsažený v přijaté zprávě, pouze jej podle sít'ových hlaviček směrují sítí. Oproti tomu aktivní uzel může být kód této zprávy spouštěn, zpracováván a vykonávat libovolný programový kód. Slovo „může“ zde má opodstatněný význam, jelikož v případě, že se nebude na mezilehlých aktivních uzlech nic vykonávat, tak sít' defakto degraduje zpět na sít' pasivní.

Schopnost vykonávat na sít'ových uzlech jakýkoli kód přináší do oblasti vývoje počítačových sítí spoustu nových možností. Příkladem lze uvést snadnou aktualizaci používaných protokolů a správu sítě. To umožňuje kupříkladu

[2] ZČU - Západočeské univerzita v Plzni.

[3] Grade32 - experimentální implementace aktivního programovatelného serveru, předchůdce projektu SAN napsaný v jazyce Pascal.

[4] Java bytecode - Přeložený zdrojový kód jazyka Java do instrukcí pro JVM (Java virtual machine).

i snadné aktualizace směrovacích algoritmů za provozu a to pouhou změnou programového kódu. U pasivních sítí není tento druh operací realizovatelný, jelikož zde jsou směrovací algoritmy svázány existujícímy standardy a jejich změna je minimálně časově náročná.

2.3.3 Struktura projektu

Jak již bylo napsáno dříve programovacím jazykem současné verze projektu *SAN*, především tedy okruhu vlivu této práce, je programovací jazyk *C++*. A i přes nemalou snahu o udržení platformní nezávisloti jsou zdrojové kódy združeny coby projekt (resp. celé „solution[5]“) poslední verze vývojového nástroje MS Visual Studia. Konkrétně se, v okamžiku psaní této práce, jedná o verzi MS Visual Studio 11 Beta vydanou až 29. února 2012 a s ním spojenou poslední verzi překladače. Důvodem k volbě těchto prostředku byl standard jazyka C++, který přináší standardizované konstrukce pro vícevláknové výpočty. Navzdory použití beta verze vývojových prostředků se jeho volba příznivě projevila na meziplatformní přenositelnosti kódu.

Programovatelný aktivní server ve své podstatě tvoří dvojice hlavních podprojektů *SAN_Server* a *SAN_Worker*. Kde podprojekt *SAN_Server* (dále v textu jako *SAN_Farmer*) představuje proces řídící chod celého uzlu a instance podprojektu *SAN_Worker* reprezentují procesy na vykonávání jednotlivých dílčích úkolů (např. vykonávání kódu aktivní aplikace). Oba podprojekty využívají společnou knihovnu *san.lib* z podprojektu *SAN* a vzájemně spolu tato dvojice komunikuje prostřednictvím mechanismu RPC[6] implementovaného v další knihovně *rpc.lib*. V projektovém balíku je pak ještě několik dalších podprojektů, ale za zmíňku v kontextu této práce stojí především složka aktivních aplikací a podprojekty *Ping*, *AntNet* a *TraceRoute*. Tyto podprojekty představují v rámci této práce klíčovou roli. Podprojekt *Ping* bylo potřeba před započetím implementace směrovacího algoritmu *AntNet* plně zprovoznit pro zjišťování odezvy ostatních serverů. Až poté mohl být efektivně zahájen proces implementace podprojektu *AntNet*. Nakonec byla ještě doimplementován podprojekt *TraceRoute* na ověření základní funkčnosti směrovacího algoritmu.

Základní informace o celém projektu SAN jsou zveřejněné na webu projektu [1] a veškeré zdrojové kódy jsou uloženy na veřejně dostupném SVN uložišti [2].

2.3.4 Hlavní implementační rozdíly verzí SANu

Jako nejzásadnější rozdíl, C++ verze oproti všem předchozím, lze označit jinou vnitřní architekturu (tj. strukturu a názvy tříd, rozhraní i typů jednotlivých proměnných) včetně odlišných programátorských konstrukcí používa-

[5]Solution - Programový prostror řešení celého projektu, který obsahuje jednotlivé dílčí podprojekty představující jeho samostatné komponenty.

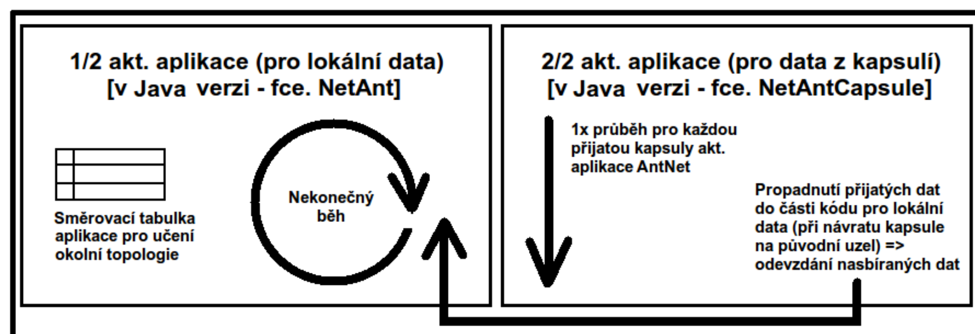
[6]RPC - Remote Procedure Call, mechanismus volání vzdálených procedur.

ných Java verzi. Tento rozdíl činí představu o pouhé reimplementaci z předchozích verzí zcela nereálnou, takže musela být celá implementace provedena kompletně od začátku.

Jako jiný rozdíl, který nemalým způsobem ovlivnil implementaci směrovacího algoritmu, můžu identifikovat například rozdílný způsob přístupu k síti a identifikace sít'ových rozhraní serverů. Současná C++ verze se narozdíl od minulých verzí snaží napodobit již známé machanismy klasického IP protokolu (tj. identifikace sít'ových uzlů pomocí adresy a sít'ové masky, dále pak i obdobnou strukturou směrovacích tabulek), zatímco předchozí verze SANu používaly vlastní řešení. uvažovaly pouze jednoznačnou adresu.

Následující rozdíl ovlivnuje především rychlost vývoje aktivních aplikací, což se týká i implementace v této práci. V předchozích verzích byl totiž před začátkem programování aktivních aplikací hotový mechanismus distribuce aktivních aplikací. Tento mechanismus vždy po spuštění nové verze aktivní aplikace na libovolném serveru rozšířil její kód na ostatní servery v síti. Což u vývoje aplikací, které pro správný chod nebo jen testování potřebují mít spuštěno více instancí, vedlo kvůli nižší četnosti restartování všech instancí k jasné úspoře času. Ale nastěstí šel tento rozdíl, alespoň částečně obejít a urychlit hromadné spouštění vytvořením sady spouštěcích skriptů.

Ve vnitřním způsobu spouštění aktivních aplikací je v C++ verzi patrný další rozdíl. Předchozí verze fungovaly tak, že byla aktivní aplikace rozdělena jakoby na dvě poloviny. Přičemž první polovina aplikace představovala kód pro lokální data, který mohl běžet např. v nekonečné smyčce a neustále vysílat data. Zatímco druhá polovina reprezentovala kód pro data z přijatých zpráv. Když se zpráva vrátila na původní uzel, tak data ze zprávy propadla do kódu první poloviny aktivní aplikace, který s nimy mohl libovolně nakládat. Třeba se tak postupně učit topologii okolní sítě (což byl zjednodušsený princip implementace směrovacího algoritmu *AntNet* z předchozích verzí projektu *SAN* viz *Obrázek 2.1*).



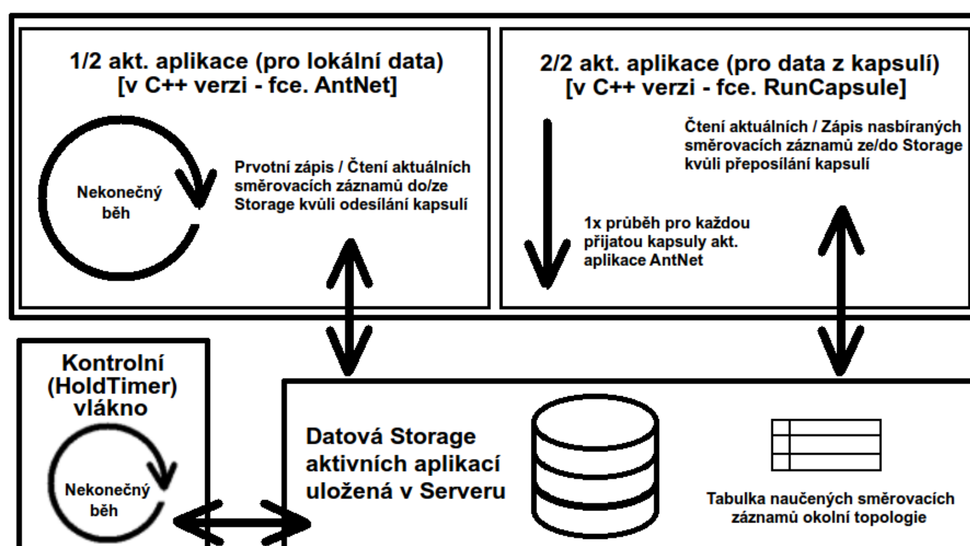
Obrázek 2.1: Schéma běhu aktivní aplikace (Java verze)

V C++ verzi je tomu v současnosti jinak, aplikace je sice také rozdělena, ale každá z částí by měla podle současného návrhu vždy jednou proběhnout a skončit. Skončení první části aplikace místo nekonečného běhu má ovšem

za následek to, že do ní nemůžou po návratu zprávy na původní uzel propadnout žádná data. Protože už tato část neběží, takže zatím ani nevznikla potřeba tento propadávací mechanismus naprogramovat. Jenže bez možnosti uchovávat přijatá data se z principu nikdy nemůže aktivní aplikace naučit topologii okolní sítě a měla by vždy pouze parciální informace z jediné zprávy.

Z tohoto chybného stavu vedla dvě východiska, buďto kompletně předělat machanismus spouštění po vzoru předchozích verzí nebo snadnější varianta v podobě implementace datového uložiště (*Storage* [7]), která zvítězila. Takže má každá aktivní aplikace svoje datové uložiště, do kterého mohou přistupovat obě části aktivní aplikace a učit se tak topologii celé sítě (viz *Obrázek 2.2*).

Na tomto obrázku je zachyceno i nově vytvořené kontrolní vlákno směrovacího algoritmu (bližší informace o vláknu viz oddíl 3.4), které s nově vytvořenou datovou storage také pracuje.



Obrázek 2.2: Schéma běhu aktivní aplikace (C++ verze)

2.3.5 Stav projektu před touto prací

Ve všech předchozích verzích aktivního serveru *SAN*, byla vždy před začátkem implementace směrování funkční aktivní aplikace *Ping*, která vlastně demonstrovala všechny potřebné dovednosti kódu aktivní aplikace nutné pro ostatní aktivní aplikace (především pak ukazovala mechanismus adresování, zápis/čtení dat do/z zprávy a její příjem s odesíláním).

Jenže současná verze tuto funkčnost postrádala a bylo ji potřeba i nadrámec zadání této práce teprve doplnit. Což i s výskytem a objevením několika nečekaných chyb, dále i kvůli horší orientaci v cizím kódu s minimálním výskytem komentářů nevyhnutelně vedlo k dalším časovým zpožděním.

[7] Storage - Datové uložiště SAN_Serveru pro aktivní aplikace.

Mezi nutné práce vykonané před započetím vývoje je potřeba započítat i návrh rozšíření kofiguračních souborů pro nastavení směrovacích záznamů, k tomu příslušnou úpravu parseru a nakonec i vytvoření konfigurací pro více instancí. K čemuž zároveň patří zautomatizování konfiguračních procesů implementováním nějakého směrovacího algoritmu, což je mimo jiné právě cílem této práce.

2.4 Proč zrovna AntNet

Pro směrování existuje řada nejrůznějších algoritmů, jako například *RIP*[3], *EIGRP*[4], *OSPF*[5], *IS-IS*[6], a *BGP*[7].

Bohužel však žádný z předchozích algoritmů nebyl přímo navržen pro prostředí aktivních počítačových sítí, tak jako vybraný algoritmus. Jejich imlementace byla zbytečně komplikovaná i zdlouhavá a navíc by výsledek měl být dokonce o něco pomalejší. Proto byl i vzhledem k jeho úspěšným předchozím implementacím a množství nasbíraných zkušeností z minulých časů byl vybrán algoritmus *AntNet*. Jeho návrhem pro aktivní sítě je dána snažší implementace, navíc i menší komplikace následných úprav pro implementaci „směrování horší cestou [9] (tzv. *worse path routing*)“.

KAPITOLA 3 AntNet

První část této kapitoly je podrobný popis směrovacího algoritmu AntNet, a to v podobě, ve které byl již dříve několikrát úspěšně implementován. Jeho poněkud obecnější a méně propracovaná verze se nachází v 3.kapitole *Teoretická část*, kterou obsahuje bakalářská práce [8] věnující se implementaci základní verze tohoto algoritmu do Java verze projektu SAN.

Hlavní výhodou směrovacího algoritmu *AntNet* je oproti ostatním algoritmům skutečnost, že byl navržen přímo pro aktivní programovatelnou sít', což podstatně zjednodušuje jeho implementaci. Princip jeho funčnosti spadá do třídy „*Distance Vector*“ směrovacích algoritmů. To znamená, že algoritmus reprezentuje směr cesty jako vektor směru (adresa na následujícího uzlu při doručování zprávy sítí) a vzdálenosti nebo-li kvality linky (metrika). Před dalším popisováním principu funčnosti je vhodné definovat základní pojmy, které budou dále častokrát využívány.

3.1 Základní pojmy

Prvním základním pojmem je **kapsule**, jedná se o zprávu posílanou v aktivních sítích, která je obdobou paketu (tj. zprávy přenášené v klasické pasivní síti). Hlavní rozdíl mezi nimy představuje vlastnost, že paket po dosažení konečného cílového uzlu nebo vypršení jeho *HopCount*[1] zaniká, zatímco kapsule existuje dál. Může pak být dále plněna daty, přeadresována a odeslána dále. Kapsule narodíl od paketu mimo jiné obsahuje identifikátor zdrojové aktivní aplikace z níž byla vyslána. Pomocí tohoto identifikátoru je na všech uzlech kontrolováno, že přijatou kapsuli zpracovává stejná verze aktivní aplikace, která jí vytvořila. Navíc v minulých verzích SANu byl již implementován distribuční mechanismus, který si byl případnou chybějící aktivní aplikaci vyžádat od ostatních uzlů.

[1]HopCount - je obdoba parametru TTL (Time To Live), používaného v klasických sítích. Udává délku života kapsule, resp. maximální počet přeskoků mezi uzly, jenž může kapsule absolvovat. Je to jedno z bezpečnostních opatření sítě proti jejímu zahlcení.

Druhým základním pojmem je **uzel**. Uzel je instance projektu SAN, kterou v okamžiku běhu představuje skupina jednoho řídícího procesu *SAN_Farmer* a více pracovních procesů *SAN_Worker*. Každá instance je spouštěna s parametry načítanými z příslušných konfiguračních souborů (viz kapitola 6).

Dále bych chtěl ujasnit v textu používané typy uzlů. **Výchozí uzel** je instance SANu, ze které byla kapsule odeslána do sítě a pro jednu konkrétní kapsuli je vždy jen jeden. Pro tuto kapsuli rovněž existuje pouze jediný **konečný cílový uzel**, což je teké instance SANu, které výchozí uzel odeslanou kapsuli adresuje. Mezi každou dvojicí těchto uzlů se v obecném případě může vyskytovat 0 až N **mezilehlých uzlů**. Toto jsou také instance SANu, ale těmito instancemi kapsule, vyslaná z výchozího uzlu a adresované na konečný cílový uzel, pouze prochází. Ono to nikdy není pouhé procházení, ale o tom až níže. Posledním často zmiňovaným typem uzlu je **přímý sousední uzel**. Relace „přímého sousedství“ platí pro každou dvojici sít'ových uzlů, mezi kterými je minimální vzdálenost (tj. není mezi nimy žádný jiný uzel).

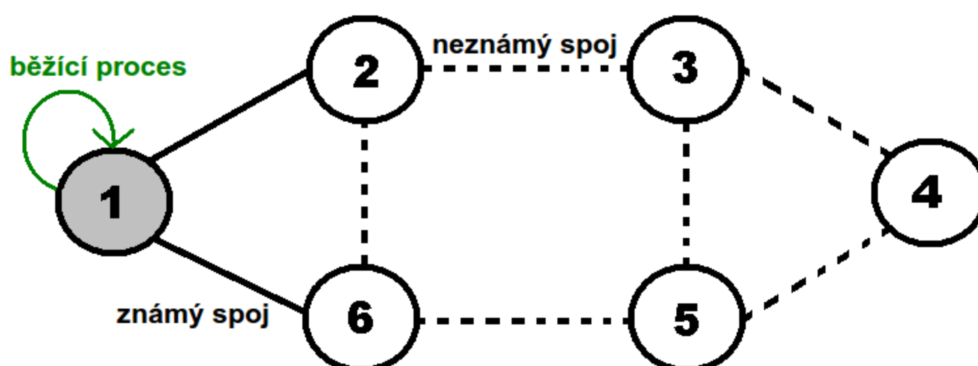
Nyní je ještě zapotřebí objasnit typy směrovacích tabulek zmiňovaných v tomto dokumentu. První je **směrovací tabulka aplikace**. Nachází se přímo ve vytvořeném programu (tj. v kódu implementace směrovacího algoritmu *AntNet*) a slouží pro ukládání směrovacích záznamů ze všech vrácených kapsulí (tj. informace o topologii celé okolní sítě). Zatímco **směrovací tabulka uzlu** je uložena v sít'ové části uzlu (tj. v procesu *SAN_Farmer*) a propaguje se do ní pouze podmnožina směrovacích záznamů „nejvýhodnějších spojení[2]“ z směrovácí tabulky aplikace. Toto jsou všechny typy směrovacích tabulek v základní verzi algoritmu. Do rozšířené verze algoritmu s směrováním horší cestou bylo z principu nutné přidat druhou směrovací tabulku serveru, ale této problematice je věnována celá kapitola 4 dále v textu.

Posledním a nejvíce důležitým pojmem je **SAN adresa**. Jedná se o datový typ, jehož struktura obsahuje čtveřici 64-bitový čísel. Tento datový typ je pak používán napříč celým projektem například jako sít'ový indentifikátor uzlu nebo jako jednotlivé položky směrovacích záznamů (tj. identikátor cílové sítě, sít'ová maska a brána).

3.2 Princip algoritmu

Jednotlivé sít'ové uzly tvoří instance stejného programu běžící jako nezávislé procesy. Principu směrovacího algoritmu bude nejsnazší popisovat pouze z pohledu konkrétního uzlu (viz *Obrázek 3.1*).

[2]Nejvýhodnější spojení - spojení s nejnižšími náklady (tj. s výhodnější metrikou).



Obrázek 3.1: Běžící proces směrovacího algoritmu

Další důležitou připomínkou je, že chod algoritmu je cyklický. Nepokouší se poznat a naučit celou okolní topologii v jediném kroku, ale naopak. V každé svojí iteraci dělá malé kroky dál sítí a v každém kroku poznává něco nového. Díky používanému principu nahodilosti zkoumání okolí algoritmus nemusí okamžitě odhalit tu nejvýhodnější cestu mezi konkrétními uzly, ale při poznávání okolní topologie dalšími iteracemi tato cesta postupně konverguje k té nějvýhodnější možné. Algoritmus ve svém principu kopíruje svět mravenců a pro dynamicky se měnící sít' (ve které se uzly různě připojují a odpojují) musí, především kvůli platnosti dat ve směrovacích tabulkách, pracovat permanentně. Tyto vlastnosti by měly zaručovat flexibilitu a krátkou reakční dobu na změny.

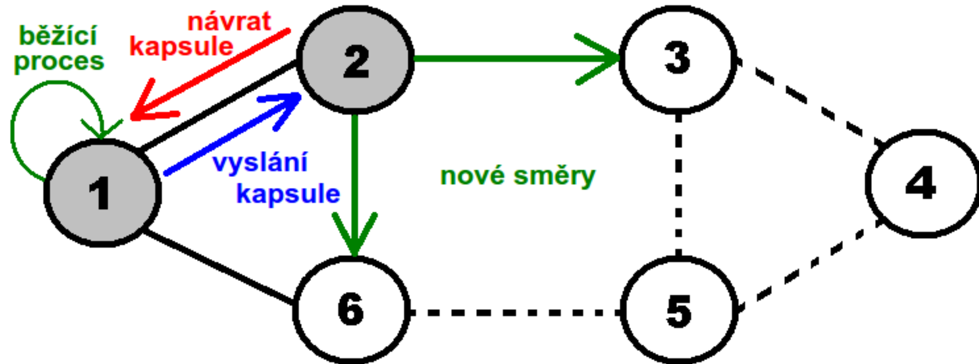
Na začátku zná každý uzel pouze množinu svých přímých sousedních uzlů (v modelovém případě na *Obrázku 3.1* výchozí uzel 1 zná pouze své přímé sousední uzly 2 a 6). A po spuštění směrovacího algoritmu vyšle do sítě výchozí uzel (v modelovém případě uzel 1) kapsuli na adresovanou libovolnému momentálně známému uzlů (tj. bud' uzlu 2 nebo 6, v popisovaném případě uzlu 2). Při správné funkčnosti algoritmu se kapsule vrací zpět na výchozí uzel po totožné cestě, kterou byla doručena na konečný cílový uzel. Návrat po totožné cestě je zařízen pomocí zaznamenávání adres navštívených uzlů při doručování konečnému cílovému uzlu a jejich využívání během na zpátěční cestě při vracení zpět výchozímu uzlu.

Po doručení kapsule na konečný cílový uzel je kapsule otočena a naplněna platnými směrovacími záznamy na přímé soudední uzly tohoto uzlu (v modelovém případě na uzly 3 a 6, viz *Obrázek 3.2*). Dále je jí nastavena nová zdrojová i cílová adresa a uzel jí odešle zpět. Zdrojová adresa je adresa aktuálního uzlu a cílová je nastavována podle seznamu dříve navštívených uzlů. Konkrétně vždy na poslední adresu v seznamu tak, aby byla dodržena totožná cesta a tento seznam je po každém použití poslední adresy zkracován.

Na mezilehlých uzlech je při první cestě do kapsule postupně ukládán seznam navštívených uzlů a při návratu jsou pak do kapsule postupně shromažd'ovány směrovací záznamy.

Nasbíranými záznamy si po návratu kapsule na výchozí uzel (v modelo-

vém případě uzel 1) aktualizuje svoji vlastní směrovací tabulku a učí se tak o topologii sítě za konečným cílovým uzlem, kterému na počátku adresoval kapsuli (v modelovém případě za uzlem 2).



Obrázek 3.2: Modelový příklad vyslané kapsule z uzlu 1 na uzel 2

V následujících iteracích jsou obdobným způsobem posílány kapsule i na ostatní uzly a výchozí vysílací uzel se může po návratu každé kapsule naučit o okolní síti něco nového.

Jednotlivé kroky a detailnější rozbor celého algoritmu jsou rozepsány do několika následujících oddílů. První oddíl popisuje jak jsou z výchozího uzlu vysílány jednotlivé kapsule a jakým způsobem jsou adresovány cílovým uzlům. Ve druhém oddílu je vysvětleno doručování kapsule po síti. Následující oddíl objasňuje používané principy při návratu kapsule (tj. hlavně mechanismus kopírování předchozí cesty kapsule) a v posledním oddílu rozebírám shromažďování směrovacích záznamů.

3.2.1 Vysílání kapsule

Prvním krokem algoritmu je vytvoření nové kapsule, které následně musí nastavit základní trojici adres. Nejprve nastaví adresu konečného cílového uzlu. Tuto adresu algoritmus určí výběrem náhodné adresy ze seznamu znamých uzlů (tj. z množiny všech aktivních směrovacích záznamů). Jako druhá se nastavuje adresa uzlu, kterému bude kapsule poslána v prvním kroku algoritmu. Selekce této adresy je realizována opět náhodným výběrem, ale tentokrát pouze ze seznamu známých přímých sousedních uzlů (tj. z množiny aktivních směrovacích záznamů ukazující na cílový uzel ve stejné subsíti [3]). Jako poslední je třeba nastavit zdrojovou adresu kapsule, ta je určována na základě adresy přímého souseda, která byla zrovna nastavena. Zdrojovou adresou kapsule je tedy adresa rozhraní uzlu, jehož pomocí je k tomuto přímému sousednímu uzlu připojen.

[3] Subsíť - podmnožina uzlů celé sítě, kterou lze z vnějšího pohledu adresovat stejnou síťovou adresou.

Po nastavení těchto adres jsou do kapsule zapsána data potřebná pro správný chod algoritmu (popis těchto dat se nachází v následující kapitole 5, jelikož se jedná o technické a implementačně závislé věci). V tomto kroku se jedná především o definování základní struktury kapsule, kterou posléze budou plnit až ostatní uzly, které přijdou s kapsulí do styku. Dále již dochází k samotnému odesílání kapsule.

3.2.2 Doručování kapsule

Nasledující oddíl lze rozdělit do dvou hlavních částí a to podle toho směru, kterým je zrovna kapsule doručována. První část zahrnuje směr kdy je kapsule doručována na konečný cílový uzel, zatímco druhá část popisuje směr opačný, tj. doručování z konečného cílového uzlu zpět na uzel výchozí.

a) z výchozího na konečný cílový uzel

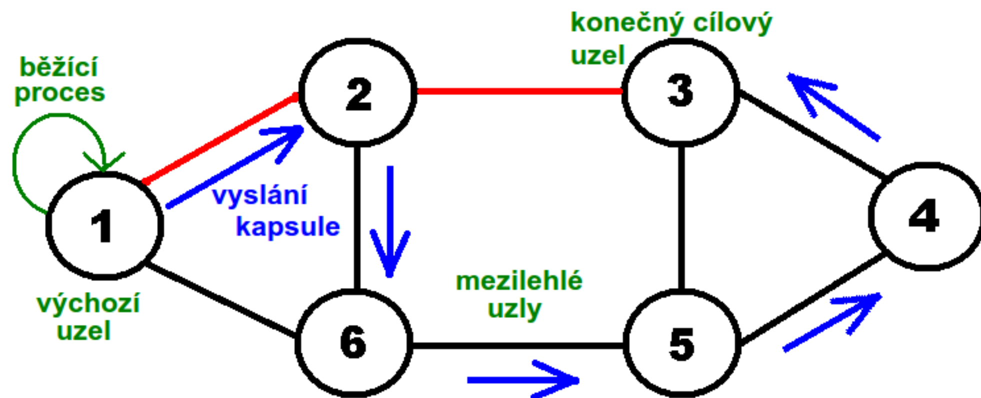
Každá odeslaná kapsule je sítí doručována tak, že je po svém přijmutí libovolným uzlem zpracována podle kódu aktivní aplikace a odeslána na další uzel. Součástí tohoto zpracování je v první řadě výběr a nastavení nové adresy přímého sousedního uzlu, jemuž bude v následujícím kroku kapsule předáná. Selekce této adresy je opět realizována náhodným výběrem, což ve finále vede k požadovanému efektu „doručování kapsulí náhodnými cestami (nedeterminismus[4])“.

Pro snadnější pochopení je celé doručování ilustrováno níže na příkladu (viz *Obrázek 3.3*).

`Příklad:`

V tomto příkladě je doručována kapsule z výchozího uzlu (1) na konečný cílový uzel (3). Jenže místo přímé „červené“ cesty (1, 2 a 3) může být kapsule doručena jinou náhodnou „modrou“ cestou (např. 1, 2, 6, 5, 4 a 3) a navštívit tak spoustu jiných uzlů. Díky delší cestě se narozdíl oproti té přímé červené po návratu kapsule výchozí uzel dozví o uzlech 4 a 5.

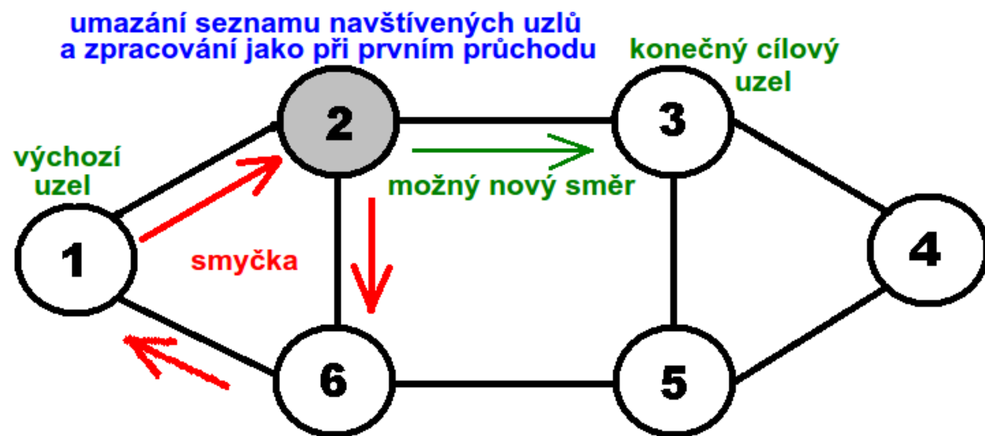
[4]Nedeterminismus - Prvek náhody, není možné předem určit kudy bude kapsule kapsule doručována.



Obrázek 3.3: Příklad možného průchodu kapsule sítí

Výhodou tohoto mechanismu je objevování všelijakých propojení mezi uzly a tím pádem i rychlejšího poznávání topologie okolní sítě. Na druhou stranu nevýhoda tohoto mechnismu spočívá v tom, že kapsule může během doručování opakovaně procházet smyčkou (např. přes uzly 1, 2 a 6) než půjde novým směrem (viz *Obrázek 3.4*). Přitom každý průchod již navštíveným uzlem vede k ukládání duplicitních dat. Z důvodu eliminace vzniku těchto duplicit je při doručování kapsule na každém mezilehlém uzlu od začátku procházen seznam navštívených uzlů a kontrolován výskyt adresy onoho mezilehlého uzlu. Přičemž nalezením svojí adresy uzel zjistí, že kapsule byla doručena po nějaké smyčce. Pro zamezení ukládání duplicitních dat pak umaže zbytek seznamu navštívených uzlů (tj. obsah od své adresy dál) a dále pokračuje jako kdyby byla kapsule doručena poprvé.

Poslední důležitou součástí zpracovacího procesu kapsule na mezilehlém uzlu je při tomto směru kapsule ukládání adresy navštíveného uzlu do kapsule. Tímto v kapsuli postupně vznikne seznam navštívených uzlů, který je pak používán při zpáteční cestě kapsule. Jelikož se pro správnou funkčnost algoritmu musí kapsule vracet po totožné cestě.



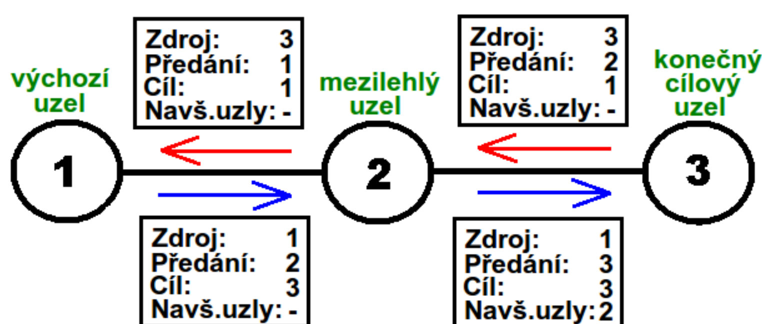
Obrázek 3.4: Příklad průchodu smyčkou při doručování kapsule

b) z konečného cílového uzlu zpět na uzel výchozí

V této části je především popsán mechanismus kopírování cesty kapsule, ale při tomto doručovacím směru na uzlech dochází ještě ke shromažďování směrovacích záznamů. Ale to je vysvětleno až v oddíle 3.2.5.

Z principu algoritmu je kapsule na konečný cílový uzel směrována nedeterministickým způsobem, ale její návrat už takto probíhat nemůže, protože by se takové kapsule původní uzel nemusel dočkat. Pro snadnější pochopení je tento princip níže ilustrován následujícím příkladem (viz *Obrázek 3.5*). V tomto příkladu je opět posílána kapsule mezi uzly 1 a 3.

Ke kopírování cesty je postupně využíván seznam navštívených uzlů, který je uložen a přenáší se v kapsuli. Uzel, který při zpáteční cestě zrovna zpracovává kapsuli, kontroluje délku tohoto seznamu. Pokud se v něm nachází nějaké adresy, tak uzel poslední adresu použije a ze seznamu ji odstraní. Podstata využití této adresy spočívá v tom, že se jedná o adresu dalšího přímého sousedního uzlu, kterému bude předána kapsule. Tento postup je opakován od konečného cílového uzlu až po odebrání posledního záznamu. Nulová délka seznamu navštívených uzlů pak znamená, že kapsule je před svým posledním skokem a ten je určen adresou zdrojového případně konečného cílového uzlu.



Obrázek 3.5: Ukázka nastavení kapsule při návratu

3.2.3 Návrat kapsule

Po doručení kapsule konečnému cílovému (uzlu 3) také dojde k jejímu zpracování podle programového kódu. Ale na tomto uzlu musí dojít v první řadě k otočení kapsule (tj. záměny adres zdrojového a koncového cílového uzlu). Dále pak právě na tomto uzlu startuje mechanismus třídění a shromažďování směrovacích záznamů, který probíhá i na všech navštívených mezilehlých uzlech. Samotné sbírání vybraných směrovacích záznamů je složitější proces, takže je mu věnován samostatný oddíl 3.2.5. Vybrané záznamy jsou nakonec zapisovány do vnitřní struktury kapsule.

Dalším krokem je pak nastavení adresy uzlu, kterému bude kapsule dále předána (tj. adresy posledního navštíveného uzlu) a její samotné odeslání. Tato adresa je uložena v kapsuli na konci seznamu navštívených uzlů (viz druhá část předchozího oddílu 3.2.2).

Když se naplněná kapsule vrátí zpět na domácí uzel, tak dochází k aktualizování záznamů lokální směrovací tabulky záznamy z přijaté kapsule a uzel tak získává aktuálnější představu o topologii okolní sítě. Tato problematika ovšem také není zcela triviální, proto je jí opět věnován samostatný oddíl 3.2.6.

3.2.4 Metriky

Libovolný směrovací algoritmus musí být při své práci schopen vzájemně od sebe rozlišit jednotlivé komunikační cesty včetně jednotlivých linek v dané topologii. Tomuto účelu slouží vlastnost linky zvaná metrika. Ve své podstatě se vždy jedná o číselnou hodnotu, kterou může určovat široké spektrum atributů tak, aby co nejlépe vystihovala jejich různorodost.

Mezi používané atributy patří mimo jiné hardwarové vlastnosti uzlu (např. kvalita síťového rozhraní daná rozmanitou hardwarovou podporou). Dále pak typ přenosového média (např. UTP[5], STP[6] či optický kabel) a s ním spojená šířka používaného frekvenčního pásma (např. 100-1000Mhz) i podporovaná

[5]UTP (Unshielded Twisted Pair) - nestíněná kroucená dvojlinka
[6]STP (Shielded Twisted Pair) - stíněná kroucená dvojlinka

přenosová rychlost (např. 10-1000Mb/s). Mimo to hodnotu metriky mohou ovlivňovat i administrativní vlastnosti jako například typ používáného komunikačního protokolu, počet skoků kapsule na doručovací cestě nebo aktuální zatížení dané linky.

Přitom v praxi používané směrovací algoritmy využívají z hlediska praktičnosti a ekonomiky věci pouze některé ze zmíněných atributů (např. RIP - počet skoků nebo OSPF - přenosová rychlost). Skládání atributů pomocí matematických vztahů a výpočet složitějších metrik není tak častý (např. EIGRP), protože často opakované komplikované matematické operace stojí výpočetní čast a daný směrovací protokol proti ostatním znevýhodňují.

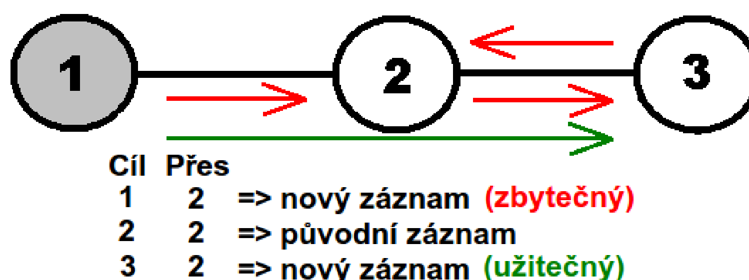
Projekt *SAN* ve svém návrhu rozlišuje dvojici metrik. První je tzv. ***Protokolová metrika*** sloužící k rozlišení linek na základě používaného přenosového protokolu, ale konkrétní ohodnocení jednotlivých protokolů zatím nebylo nikde stanoveno. Druhou metrikou pak je tzv. ***Administrativní metrika***, která vzájemně odlišuje linky podle počtu skoků, kvality linky a jejího aktuálního vytížení, ale ani postup jejího stanovení pro konkrétní linky zatím nebylo nikde definován.

Implementace směrovacího algoritmu *AntNet* s těmito metrikami pracuje pouze obecně jako s čísly a do jejich výpočtu nijak nezasahuje. Jediné co je pevně dané programovým kódem je interpretace těchto čísel. Vytvořená implementace považuje linku s menší hodnotou metriky za výhodnější.

3.2.5 Shromažďování směrovacích záznamů

Proces poznávání okolní topologie je založen na postupném sbírání směrovacích záznamů ze všech při zpáteční cestě navštívených uzlů.

Kvůli udržení maximální efektivity směrovacího algoritmu dochází pouze ke sbírání směrovacích záznamů na přímé sousední uzly a to ještě neúplně všech. Informace, že se uzel může dostat na daný uzel smyčkou přes následující uzel (viz *Obrázek 3.6*) je naprosto zbytečná. A vzledem k tomu, že tyto záznamy z principu vznikají při každém skoku kapsule, vede jejich filtrování ke zmenšení objemů přenášených dat a tím pádem i ke zvýšení rychlosti celého algoritmu.

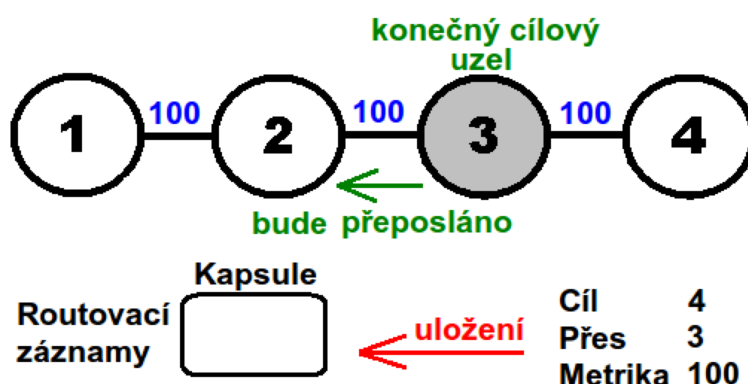


Obrázek 3.6: Zbytečný směrovací záznam pro uzel 1.

Před startem části algoritmu určené ke sbírání je nutné zjistit adresu uzlu,

kterému bude kapsule dále přeposlána. Tato adresa je důležitá právě kvůli vyfiltrování zbytečného záznamu.

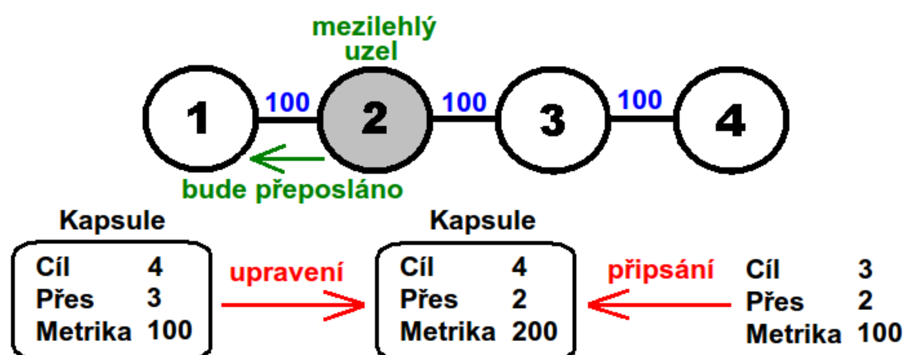
Jako první do kapsule zapisuje směrovací záznamy konečný cílový uzel. Algoritmus si získá množinu aktivních záznamů na přímé sousední uzly. Poté v této množině hledá zbytečný záznam kvůli jeho vynechání. Algoritmicky je možné tento záznam identifikovat podle shodné cílové adresy s adresou uzlu, kterému bude kapsule přeposlána. Ostatní záznamy mohou být uloženy do kapsule (viz *Obrázek 3.7*). Ale vzhledem k tomu, že obecný směrovací záznam obsahuje řadu všelijakých parametrů, jsou ukládané záznamy patřičně redukovány, což vede k dalším úsporám a urychlení. Hlavní myšlenkou redukce záznamů je přenášení pouze užitečných informací. Za užitečné informace lze z konkrétních směrovacích záznamů považovat údaje o cíly záznamu. Dále pak o bráně, kterou se na daný cíl dostanu a ohodnocení příslušného spojení (tj. metriky [7]). Zbylé informace z směrovacích záznamů (např. časovač platnosti, ID záznamu nebo adresa zdrojového rozhraní) jsou stejně platné pouze pro daný uzel, takže si je ostatní uzly musí nastavit podle sebe.



Obrázek 3.7: Zápis směrovacích záznamů na konečném cílovém uzlu.

Každý následující uzel, který bude zapisovat další záznamy (tj. každý mezilehlý uzel), musí nejprve projít stávající již v kapsuli uložené záznamy. Přičemž musí tyto záznamy upravit po vzoru směrovacích záznamů na přímé sousední uzly. Každému záznamu musí být změněn atribut brány na adresu tohoto uzlu a příslušně navýšena metrika. Toto navýšení bude mít hodnotu metriky spojení, po kterém kapsule přišla na tento uzel (viz *Obrázek 3.8*). Až poté smí tento uzel připsat nové záznamy na své přímé sousední uzly, což provádí podle stejného postupu jako konečný cílový uzel (viz předchozí odstavec).

[7]Metrika spojení - Ohodnocení cesty, může mít spoustu podob (rychlost, cena, vytížení, fyzické medium, ...) a mnoho různých interpretací (např. více je lepe).



Obrázek 3.8: Zápis směrovacích záznamů na mezilehlých uzlech.

Po dokončení zpáteční cesty kapsule musí směrovací záznamy obdobným způsobem upravit i výchozí uzel. Ale tento uzel již neupravuje parametry brány, tím by přišel o nejcennější informace. Pouze všem záznamům navyšuje metriky a to opět o hodnotu metriky posledního spojení kterým byla doručena kapsule.

3.2.6 Aktualizace směrovacích tabulek

Po návratu naplněné kapsule je nejprve všemi směrovacími záznamy aktualizována směrovací tabulka aplikace. To znamená, že se pouze samotný algoritmus dozví všechny informace o okolní topologii. Až poté se současně procházejí směrovací tabulky uzlu i aplikace a do tabulky uzlu se propagují pouze záznamy „nejvýhodnějších spojení[8]".

Samotný proces aktualizování libovolné směrovací tabulky pak spočívá v tom, že se algoritmus pro každý směrovací záznam z vrácené kapsule nejprve snaží v této tabulce uzlu vyhledat záznam ukazující na stejnou cílovou síť (tj. cílová adresa záznamu bitově přenásobená síťovou maskou). Pokud je takový záznam nalezen, tak jsou tyto záznamy vzájemně porovnány podle metrik a ten výhodnější je uložen. V opačném případě je situace vyřešena pomocí přidání nového záznamu na konec tabulky.

3.2.7 Odebírání směrovacích záznamů

V algoritmu nejsou přímo definované situace určené k mazání směrovacích záznamů ze směrovacích tabulek. Jejich odebírání ze všech směrovacích tabulek je prováděno automaticky kontrolním vláknem. Toto programové proces pravidelně snižuje a kontroluje časovače platnosti jednotlivých směrovacích záznamů a terpve po jejich vyprchání směrovací záznamy maže.

Různým nastavením těchto časovačů včetně jejich pravidelného snižování a četnosti jejich kontrol lze do určité míry ovlivnit výsledné chování směrovacího algortmu, především pak rychlost jeho reakcí na změny okolní topologie.

[8] Nejvýhodnější spojení - spojení s nejnižšími náklady (tj. s výhodnější metrikou).

Na druhou stranu příliš časté kontroly, tj neustálé procházení směrovacích tabulek, povedou k plýtvání procesorového času daného uzlu.

3.3 Další vlastnosti algoritmu

Algoritmus počítá se situací, že se nějaký uzel ze sítě odpojí. Protože jinak by kapsule adresované tomuto uzlu byly neustále doručovány sítí dokud by se uzel opět nepřipojil, což by mohle vést až k zahlcení a následnému výpadku sítě. Zánik těchto „mrtvých" zpráv bez adresáta je řízen pomocí parametru *HopCount*[9]. Implementovaná verze v sobě má navíc kontrolu hodnoty toho parametru. Přičemž pokud klesne jeho velikost na polovinu své původní hodnoty, tak dochází k řízenému otočení kapsule, aby ještě byl umožněn návrat kapsule zpět na výchozí uzel.

Návraty naplněných kapsulí jsou z principu asynchronní události. Jenže při zpracování těchto událostí dochází k aktualizacím dat sdílených jak pro samotný uzel (tj. libovolné další procesy a vlákna), tak i pro části aktivní aplikace AntNet. Což může vést k nestabilitě sítě používáním neplatných dat, proto je třeba aktualizace řídit pomocí synchronizačních prostředků. V této C++ verzi představuje kritickou sekci práce se směrovací tabulkou aplikace (tj. od okamžiku přečtění dat až po jejich opětovný zápis), která je uložena v datové storage uzlu (viz *Obrázek 2.2*). Výluční přístup do této sekce je řízen základním sychnonizačním primitivem „Mutex[10]".

3.4 Rozšíření AntNetu

Nejzásadnější změnou původního návrhu AntNetu je rozšíření pro směrování horší cestou, jejíž důsledkům je věnována celá následující kapitola.

Druhé rozšíření slouží ke zvýšení efektivity práce s směrovacími záznamy (tj. byla do implementace zahrnuta obdoba koncepce supernettingu). Její konkrétní realizace spočívá v tom, že ve směrovací tabulce aplikace jsou uloženy nasbírané záznamy jednotlivě, ale do směrovacích tabulek uzlu se propagují sdružené informace o nejlepších směrech. Podobné informace (tj. záznamy na uzly ve stejné subsít'i) jsou sdružovány do jediného směrovacího záznamu na celou subsít', což vede ke snížení jejich výsledného počtu uložených záznamů. Doručení konkrétnímu uzlu pak obstarává až uzel na hranici dané sub sítě.

Poslední změna vychází z reálného provozu směrovacích algoritmů. Do rozšířeného návrhu implementace bylo přidáno kontrolní programové vlákno, které pravidelně snižuje hodnoty čítačů platnosti jednotlivých záznamů ve

[9]HopCount - je obdoba parametru TTL (Time To Live), používaného v klasických sítích. Udává délku života kapsule, resp. maximální počet přeskoků mezi uzly, jenž může kapsule absolvovat. Je to jedno z bezpečnostních opatření sítě proti jejímu zahlcení.

[10]Mutex (MUTual EXclusion) - Synchronizační prostředek paralelního programování, zabraňující současnému využívání sdílených zdrojů.

všech dostupných směrovacích tabulkách. Toto vlákno zároveň provádí jejich hodnoty a při jejím vynulování směrovací záznamy maže.

Průběh kontrolního vlákna je rovněž cyklický a každou iteraci představuje následující dvojice kroků. Prvník krokem je načtení aktuální podoby směrovací tabulky aplikace z datové storage (tj. vstup do kritické sekce), zpracování této tabulky (viz předchozí odstavec) a zápis její nové podoby zpět do storage (tj. opuštění kritické sekce). Ve druhém kroku jsou pak procházeny směrovací tabulky uzlu, ale jejich zpracování již v kontextu *AntNetu* nepředstavují kritickou sekci. Přístup k nim je totiž synchronizován samostatně na úrovni uzlu.

Dále byla do SANu doimplementována podpora deaktivace sít'ových rozhraní uzlu, která má za následek, že toto kontrolní vlákno zneplatní odpovídající směrovací záznamy. Tyto záznamy nejsou z tabulek ihned odstraňovány, nýbrž se očekává, že doslo pouze k výpadku, který bude brzy odstraněn. Poté stačí dané záznamy pouze reaktivovat, což by mělo představovat jisté urychlení při zotavování po výpadku.

KAPITOLA 4

Směrování horší cestou

Základní verze směrovacího algoritmu *AntNet* pracuje z hlediska kvality poskytovaných služeb (Quality of Services, dále jen QoS) na základě metodiky „Best-effort services“. Což znamená, že se maximálně snaží doručit každou kapsuly na cílový uzel co nejrychleji a nejefektněji. Oproti tomu implementování rozšíření pro směrování horší cestou vede na fungování podle metodiky „Differentiated services“, která představuje rozlišování kapsulí do různých kategorií jejich značkováním.

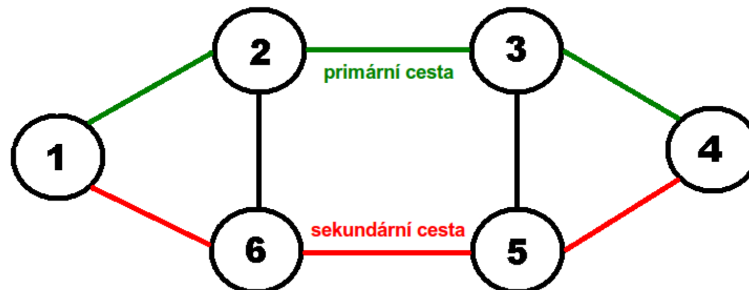
Tyto značky jsou ukládány do záhlaví jednotlivých kapsulí a na základě hodnoty těchto značek je s kapsulemi rozdílně zacházeno. Konkrétní způsob zacházení je pak dán různými předdefinovanými parametry.

Pro tento účel má projekt *SAN* ve struktuře kapsule (tj. ve třídě *CCapsule*) pole nastavovacích příznaků (tj. atribut bitových příznaků *flags*). Současná verze počítá pouze se dvojím typem kapsulí, proto je na jejich rozlišení zapotřebí jediného bitu nabývající některé z dvojice hodnot 0 nebo 1. Na základě hodnoty této značky je pak na každém uzlu pro danou kapsuly vybírána příslušná linka. Přičemž hodnota 1 označuje ***TS*** *(Time Sensitive)* kapsule citlivé na doručení v co nejlepším čase, zbylé ***NonTS*** *(Non Time Sensitive)* kapsule identifikuje hodnota 0.

Základním cílem experimentu je pokusit se v dané síťové topologii pokud je to vůbec možné najít, resp. označit dvojici cest mezi každou dvojicí uzlů. První cesta by měla být označena jako primární, která by měla být využívána především pro kapsule, jejichž priorita doručení v co nejlepším čase je vysoká (např. data streamovaného videa). Oproti tomu by mohla ve stejné topologii současně existovat druhá cesta, označena jako sekundární, které by se mělo při dostatečném obsazení primární cesty využívat pro doručování nízko prioritních kapsulí (např. data emailových zpráv). Což by v praxi mělo kvůli rozložení provozu na větší část topologie teoreticky vést k efektivnějšímu využití všech doposud používaných komunikačních linek (viz *Obrázek 4.1*).

Před začátkem implementačních úprav si bylo zapotřebí promyslet princip

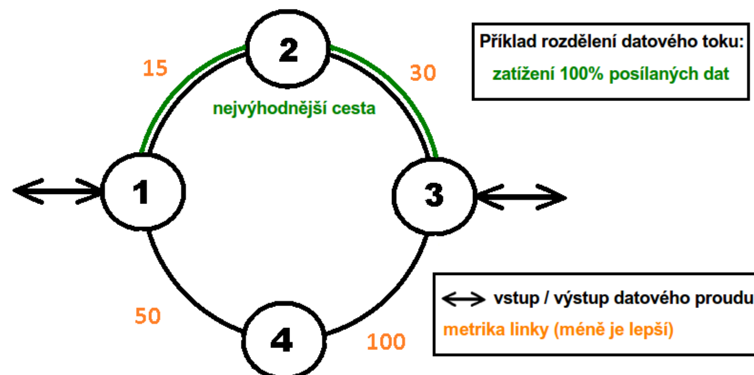
fungování konceptu (viz oddíl 4.1) a seznam nutných úprav vedoucí k jeho implementaci (viz oddíl 4.2).



Obrázek 4.1: Ukázka různých cest mezi uzly 1 a 4

4.1 Analýza

Základní verze směrování doručovala kapsule pouze nejvýhodnější cestou, kterou tudíž maximálně zatěžovala (viz *Obrázek 4.2*).



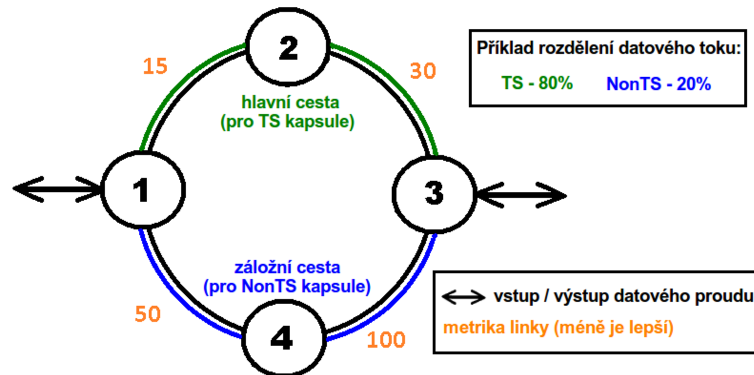
Obrázek 4.2: Pouze nejvýhodnější cesta mezi uzly 1 a 3

Koncept směrování horší cestou by teoreticky měl část tohoto zatížení odvést (přesměrovat) na cílový uzel jinými linkami. Během programování jsem přišel na to, že směrování horší cestou podle popisu z úvodního odstavce této kapitoly bude jednodušše fungovat pouze v topologii, kterou budou procházet datové proudy pouze jediným směrem, resp. tímto směrem a směrem přesně opačným (viz *Obrázek 4.3*). V tomto případě totiž bude jasně dáno rozdělení cest (pro *TS* [1] a *NonTS* [2] kapsule) podle metrik jednotlivých linek. Poměr jejich zatížení bude odpovídat rozdělení datového proudu např. 80%

[1] TS (Time Sensitive) - Vysoce prioritní kapsule citlivé na doručení v co nejlepším čase, pro jejich doručování jsou využívány nejvýhodnější linky (např. video stream).

[2] NonTS (Non Time Sensitive) - Nízce prioritní kapsule, které mohou být doručovány pomocí horších linek (např. e-mail).

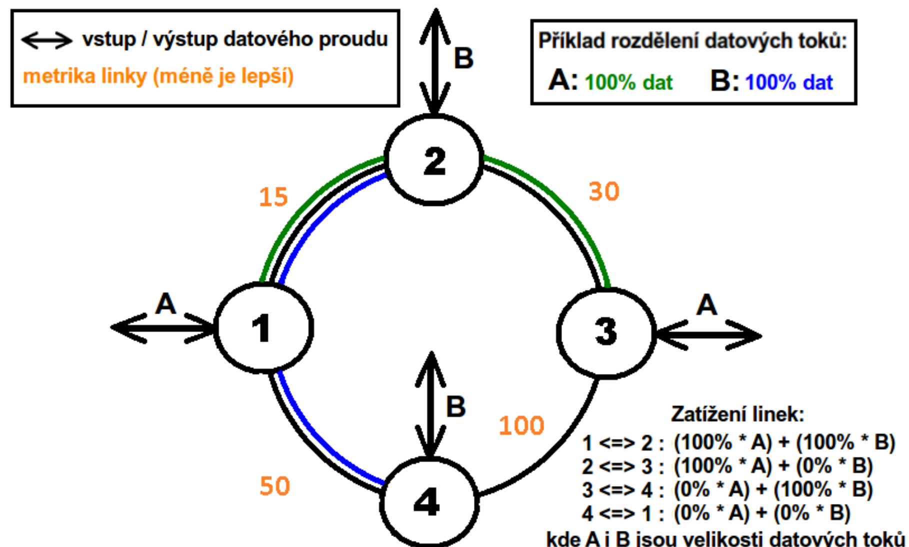
pro *TS* a zbylých 20% pro *NonTS* kapsule a takový výsledek byl a je přáním počátečních úvah.



Obrázek 4.3: Ukázka datových proudů mezi uzly 1 a 3

Hlavním cílem směrování horší cestou totiž je řídit využívání linek tak, aby jejich zatěžování co nejvíce odpovídalo rozdělení kvality těchto linek (tj. nejvýhodnější cestou doručovat primárně pouze *TS* kapsule a naopak). Při plnění hlavního cíle je tedy teoreticky očekáváno vetší zatížení kvalitnějších linek (rychlejších, levnějších, momentálně méně vytížených, ...) oproti ostatním. Jenže přítomnost dalšího datového proudu s jiným směrem (např. mezi uzly 2 a 4) situaci změní. Při tom se existenci vícera datových proudů ve skutečném provozu není možné reálně vyhnout, jelikož za datový proud lze považovat i jedinou vyslanou kapsuli.

Přidání dalšího datového proudu, např. mezi zminované uzly 2 a 4, do základní verze změní vytížení linek následujícím způsobem (viz *Obrázek 4.4*).

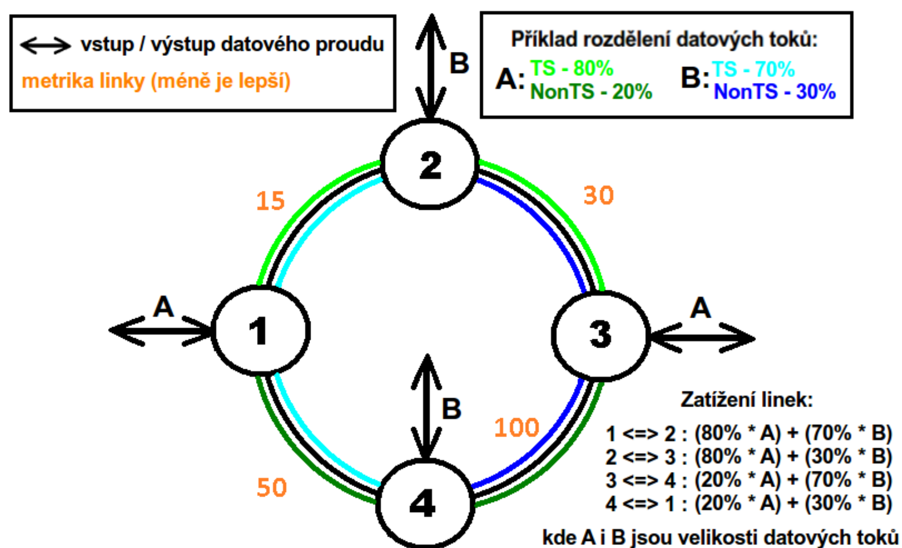


Obrázek 4.4: Ukázka dvojice dat. proudů A a B u základní verze směrování

Při shodných velikostech datových toků např. 10Mb/s bude vytížení linek 1x20Mb/s, 2x10Mb/s a jedna linka nebude zatížena vůbec.

Použití konceptu směrování horší cestou a při rozdělení druhého datového toku poměrem např. 70% pro *TS* a zbylých 30% pro *NonTS* kapsule nám zásadně změní poměry zatížení jednotlivých linek (viz *Obrázek 4.5*). Pro porovnání při stejných velikostech datových toků 10Mb/s nyní bude jejich zatížení rozloženo následovně 1x15Mb/s, 1x11Mb/s, 1x9Mb/s a 5Mb/s.

Již z tohoto jednoduchého příkladu je vidět jak koncept směrování horší cestou funguje. Úspěšně rozmělní datové toky i na méně výhodnější linky, což umožní vyšší využití nejvýhodnějších linek. Toto chování je níže ověřeno i praktickými pokusy na složitějších topologiích (viz oddíl 4.3).



Obrázek 4.5: Ukázka dvojice dat. proudů A a B u směrování horší cestou

4.2 Potřebné úpravy

Tento oddíl rozebírá změny celého projektu *SAN* včetně implementace směrovacího algoritmu *AntNet*, které bylo nutné provést pro zavedení konceptu směrování horší cestou.

Implementování značkování různých typů doručovaných kapsulí bylo první nutnou úpravou. Druhá realizovaná úprava ovlivnila schopnost učení se další cesty na daný konečný cílový uzel. V principu samotného směrovacího algoritmu není narozdíl od vnitřní podoby uzlu v tomto ohledu zapotřebí žádné změny, protože algoritmus ukládáním všech záznamů do své směrovací tabulky aplikace stejně poznává celou okolní topologii. Oproti tomu do vnitřní struktury byla sice ne až tak z principu, ale spíše kvůli urychlení, přidána druhá směrovací tabulka. Uzel tak obsahuje dvojici samostatných směrovacích tabulek, první pro *TS* (dále jako ***TS směrovací tabulka uzlu***) a druhou *NonTS* pro směrovací záznamy (dále jako ***NonTS směrovací tabulka***

uzlu). Získané urychlení spočívá v tom, že uzel může pro každou kapsuli procházet pouze záznamy ze směrovací tabulky odpovídající typu dané kapsule (tj. při obsazení směrovacích tabulek stejným počtem záznamů až 50%).

Ovšem důsledkem přidání nové směrovací tabulky byla nutnost upravení veškerých používaných procedur, funkcí a rozšíření seznamu služeb poskytovaných aktivním aplikacím včetně příslušných přístupových rozhraní. Následky těchto úprav se částečně dotkly i zdrojového kódu směrovacího algoritmu *AntNet*.

Implementace směrovacího algoritmu se týká především nutná třetí a nejvýraznější úprava, která spočívá v předělání mechanismu delagace směrovacích záznamů ze směrovací tabulky aplikace do obou směrovacích tabulek uzlu. Princip nového mechanismu je popsán v oddílu 4.2.2.

Mechanismus používání výchozí brány byl také upraven. Rozebrání této komplikovanější problematiky se věnuje samostatný oddíl 4.2.1.

Po dokončení všech zmíněných úprav byl vytvořen návrh několika testovacích topologií včetně výroby příslusných konfiguračních a spouštěcích souborů. Na nichž bude možné spustit rozšířenou verzi implementace směrovacího algoritmu *AntNet* a hlavně otestovat funkčnost konceptu směrování horší cestou (popis testování viz oddíl 4.3).

4.2.1 Problematika výchozích bran

V tomto projektu stejně jako v klasických sítích původně existoval záznam pouze o jediné výchozí bráně pro všechny kapsule, který byl používán následovně. Pokud uzel přijmul kapsuli s cílovou adresou, kterou tento uzel neznal, tak byla použita kapsule předána uzlu (bráně) právě s adresou výchozí brány.

Nyní však existují v konfiguraci dva záznamy výchozích bran (viz popis konfiguračních souborů v oddílu 6.1). Každý je pro různý typ (tj. 1x pro *TS* a 1x pro *NonTS*) datových kapsulí a jejich rozlišování umožňuje lepší konfigurovatelnost různých výchozích směrů různým typů kapsulí. Přičemž brány jsou používány stejným způsobem jako brána klasická.

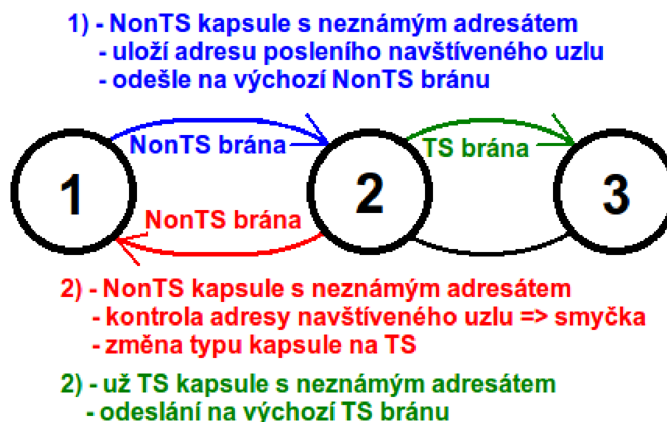
V reálných libovolně komplikovaných topologiích nejde zabránit tomu, aby žádná dvojice přímých sousedních uzlů neměla nastavené záznamy výchozí bran proti sobě (tj. do smyčky). V některých topologiích je takové nastavení dokonce jediné možné (viz *Obrázek 4.6*). Takové doručovací smyčky výchozích bran dokonce nemusí existovat jen mezi přímými sousedními uzly, nýbrž může být uzavřena přes libovolný počet dalších uzlů.

Když při takovém to nastavení přijde kapsule pro níž ani jeden z uzlů nezná cílový směr, tak si ji budou oba uzly předávat do vypršení času její platnosti a jejího zániku. Takové kapsule by nikdy nemohly být doručeny cílovým uzlům.

Obrázek 4.6: Ukázka smyčky výchozích bran

Částečné řešení pouze pro *NonTS* kapsule této problematiky ukrývá právě existence dvojího typu doručovaných kapsulí a výchozích bran. Jeho základní myšlenka vymyšleného řešení spočívá v tom, že je v doručovaných kapsulích uchováván seznam adres všech navštívených uzlů. Každý následující uzel, který bude chtít kapsuli pomocí záznamu výchozí brány dále odeslat, může pomocí tohoto seznamu zkontrolovat zda nebude doručována po smyčce.

Pokud ano, tak by mělo dojít ke zvýšení priority kapsule (tj. změně typu z *NonTS* na *TS*). Výsledkem je, že uzel bude pro stejnou kapsuli prohledávat směrovací tabulku s druhým typem záznamů, což zvyšuje pravděpodobnost nalezení vhodného záznamu. Případně bude nakonec použita výchozí brána pro druhý typ kapsule, což by mohlo zamezit generování těchto smyček (viz *Obrázek 4.7*).



Obrázek 4.7: Změna doručovacího směru kapsule

Částečnost řešení je dáná tím, že je navrženo pouze pro jediný typ kapsulí a také kvůli tomu, že ve své jednoduchosti vychází z předpokladu různého nastavení záznamů jednotlivých výchozích bran. Jenže tento překpoklad není vždy možné naplnit (např. uzel s jediným síťovým rozhraním).

4.2.2 Aktualizování směrovacích tabulek uzlu

Algoritmu při existenci jediného typu kapsule a hlavně jediné směrovací tabulky uzlu stačilo nejprve pro každý známý konečný cílový uzel vyhledat ve

směrovací tabulce aplikace záznam s nejvýhodnějším směrem (tj. s nejlepší metrikou). Poté se algoritmus pokoušel podle stejné adresy najít k tomuto záznamu odpovídající záznam v jediné směrovací tabulce uzlu. V případě nalezení takového záznamu se tyto záznamy vzájemně porovnávaly a ten výhodnější v tabulce uzlu uchován. V opačném případě byl záznam s nejlepším směrem ze směrovací tabulky aplikace přidán na konec směrovací tabulky uzlu.

Při existenci více směrovacích tabulek uzlu je situace o něco složitější. Ve směrovací tabulce aplikace je stejným způsobem pro každou známou adresu konečného cílového uzlu vyhledáván záznam s nejvýhodnější metrikou. Dále pak algoritmus stejným způsobem jako u jediné tabulky prochází nejprve *TS směrovací tabulku*, protože má vyšší prioritu a je z principu plněna přednostně. Snaží se vyhledat záznam se shodnou adresou konečného cílového uzlu, který by aktualizoval nebo *TS směrovací tabulku* rozšiřuje.

Rozdíl je patrný až v okamžiku, kdy je nejvýhodnější záznam ze směrovací tabulky aplikace méně výhodný než aktuálně platný záznam v *TS směrovací tabulce uzlu*. V tomto případě nelze už záznam použít pro *TS směrovací tabulku uzlu* a musí být ještě prohledána *NonTS směrovací tabulka uzlu*. U té je mechanismus prohledávání, porovnávání včetně případné aktualizace opět totožný jako u používání jediné tabulky (tj. aktualizace, vložení nebo vynechání). K vynechání záznamu dojde pokud je i v *NonTS směrovací tabulce uzlu* výhodnější záznam.

Přičemž při každé aktualizaci směrovacího záznamu dochází mimo úprav funkčních atributů záznamu (tj. cílová adresa, maska, brána, metrika) k obnovení čítače platnosti záznamu.

4.3 Experiment

Realizace experimentu má sloužit k praktickému ověření výše uvedených teoretických úvah a shodnocení výsledků (viz následující oddíl 7). Jeho provedení spočívalo v první řadě ve vytvoření návrhu testovacích topologií včetně jejich zanesení do příslušných konfiguračních souborů (popis vytvořených testovacích kofigurací viz níže). Dále pak ve spuštění všech uzlů včetně směrovacího algoritmu *AntNet*, čímž by mělo dojít hlavně k naplnění obou (*TS* i *NonTS*) směrovacích tabulek všech uzlů. Posledním krokem testu je vybírání dvojice náhodných uzlů z testovací topologie. Přičemž na prvním uzlu z vybrané dvojice ještě spustím vytvořený testovací nástroj *TraceRoute*[3], kterým pošlu kapsuli každého typu (tj. celkem 2 kapsule, 1x *TS* a 1x *NonTS*) na adresu druhého uzlu z vybrané dvojice.

Výstupem testovacího nástroje *TraceRoute* je seznam uzlů, přes které byla kapsule doručována.

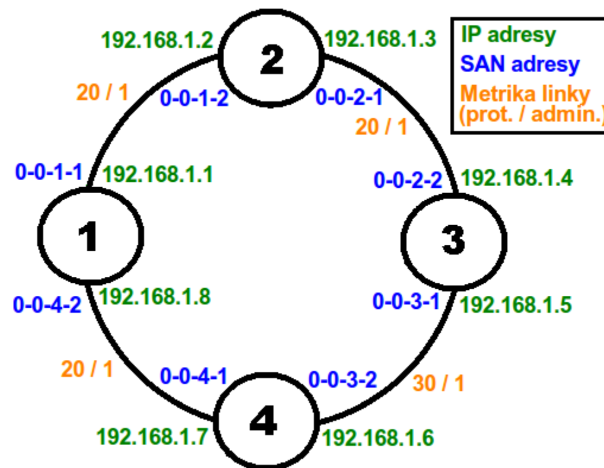
[3]TraceRoute - V kontextu této práce se jedná o aktivní aplikaci projektu SAN, která získává cestu doručovaných kapsulí. Slouží k testování konceptu směrování horší cestou.

Daný postup poté opakuji pro jiné náhodné dvojice uzlů i pro další testovací topologie.

4.3.1 Testovací topologie 1

Tato jednoduchá topologie skládající se pouze ze čtveřice uzlů propojená do kruhu byla ve své jednoduchosti navržena především pro základní ověření funkčnosti směrování *NonTS* kapsulí jinou než nejvýhodnější cestou.

Konkrétní nastavení metrik konkrétních linek, *SAN adres*[4] jednotlivých uzlů včetně odpovídajících IP adres[5] nutných pro spoustění v prostředí klasické IP sítě vystihuje následující schéma 4.8.



Obrázek 4.8: Schéma a nastavení testovací topologie 1

Testování na této topologii bylo prováděno následujícím způsobem. Po spuštění všech uzlů automaticky došlo k naplnění směrovacích tabulek implementovaným směrovacím algoritmem *AntNet*. Dalším krokem testu bylo pouze spuštění testovacího nástroje *TraceRoute* na uzlu 1, kterým byla na uzel 3 a zpět odeslána nejprve *TS* kapsule. Po návratu bylo z obsahu kapsule (tj. seznamu navštívených uzlů) vidět přes které uzly byla doručována. Následně byl tento postup zopakován i pro *NonTS* kapsuli a poté byly vzájemně porovnány výsledky (dosažené výsledky viz oddíl 7).

Přičemž správné výsledky by měli být následující. *TS* kapsule měla být doručena cestou přes uzel 2, jelikož cesta na uzel 3 před uzel 2 má kompletní metriku pouze $20/1 + 20/1 = 40/2$. Zatímco přes uzel 4 vychází metrika $20/1 + 30/1 = 50/2$ hůře (protože méně je lepé, viz oddíl 3.2.4), proto by měla být tato cesta použita pouze pro doručování *NonTS* kapsulí.

[4]SAN adresa - Jednoznačný identifikátor sít'ového rozhraní v lokálním úseku programovatelné sítě (např. 0-0-1-1) SANu.

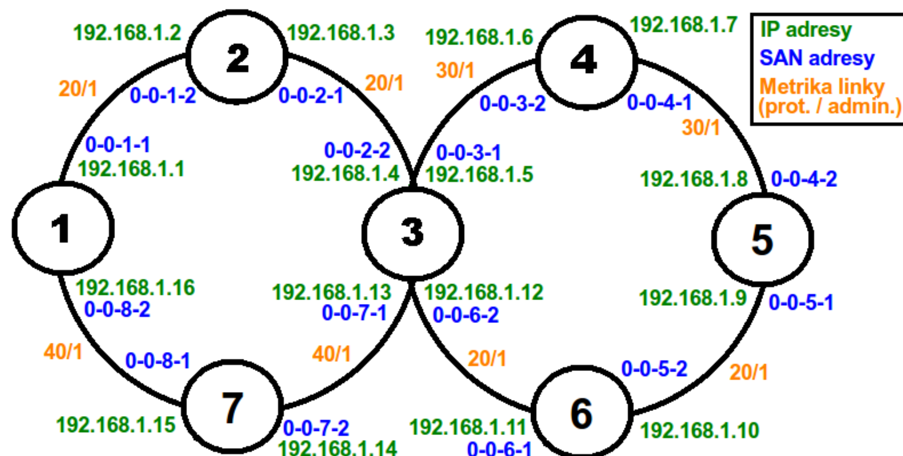
[5]IP adresa - Jednoznačný identifikátor sít'ového rozhraní v lokálním úseku standardní počítačové sítě (např. 192.168.1.1) používaný IP (internetový protokol) protokolem.

Pro ověření funkčnosti byl proveden na této topologii ještě další pokus, ale tentokrát mezi jinou dvojicí uzlů. Tentokrát mezi uzly 2,4 a výsledky jsou též v oddílu 7.

4.3.2 Testovací topologie 2

Po otestování základní funkčnosti směrovacího algoritmu včetně konceptu směrování horší cestou na testovací topologii 1 byla navržena složitější topologie 2, která by se měla více podobat reálným podmínkám. Skládá se ze 7 uzlů, konkrétní propojení uzlů, nastavení metrik sít'ových linek včetně adres jednotlivých uzlů ilustruje následující schéma 4.9.

Na této topologii byl proveden obdobný test, s tím rozdílem, že byly posílány kapsule mezi uzly 1 a 5. Přičemž opět bylo hlavním cílem zaznamenat skutečné cesty doručovaných kapsulí a poté je vzájemně porovnat s cestami teoretickými (tj. s těmi ručně vypočítanými na základě známých metrik).



Obrázek 4.9: Schéma a nastavení testovací topologie 2

Mezi uzly 1 a 5 existuje v této topologii hned čtveřice cest 1-2-3-4-5, 1-2-3-6-5, 1-7-3-4-5 a 1-7-3-6-5 (dále v textu označeny jako A, B, C, D). Přičemž metrika cesty A je 20/1+20/1+30/1+30/1=100/4, cesta B má výslednou metriku 20/1+20/1+20/1+20/1=80/4, metrika cesty C je 40/1+40/1+30/1 +30/1=140/4 a metrika poslední cesty D má hodnotu 40/1+40/1+20/1+20/1 =120/4.

Z těchto vypočítaných metrik je vidět, že pro *TS* kapsule by při správné funkčnosti měli být doručovány cestou B (tj. přes uzly 1-2-3-6-5) s nejvýhodnější metrikou 80/4. Cesta A (tj. uzly 1-2-3-4-5) by pak měla být využívána pro doručování *NonTS* kapsulí, jelikož má druhou nejvýhodnější metriku (viz mechanismus aktualizovaní směrovacích tabulek v oddílu 4.2.2).

Směrovací záznamy těchto cest by tedy včetně mnoha jiných měli být v příslušných směrovacích tabulkách uzlů 1 a 5 (skutečně dosažené výsledky viz oddíl 7).

Záznamy zbylé dvojice cest by měli zůstat uložené pouze ve směrovací tabulce aplikace, kde budou podle principu algoritmu představovat záložní cesty, které přijdou na řadu v výpadku některé používáné cesty a hlavně jejího záznamu z příslušné směrovací tabulky.

4.3.3 Testovací nástroj TraceRoute

Tento podprojekt představuje jednoduchou verzi testovacího nástroje, na ověření správně funkčnosti implementovaného směrovacího algoritmu *AntNet*.

Princip

Po spuštění na výchozím uzlu dojde k vytvoření datových kapsulí a jejich odeslání do sítě. Atributy jako cílová adresa, typ (tj. *TS* nebo *NonTS*) a počet odeslaných kapsulí jsou dány vstupními parametry příkazu.

Vytvořené kapsule jsou odeslány do sítě a jsou na cílový uzel směrovány pouze pomocí záznamů ve směrovacích tabulkách uzlů přes které jsou doručovány.

Zpracování kapsulí nástroje *TraceRoute* se rovněž spouští na každém uzlu, na který byla právě doručena. Během tohoto zpracovávání dochází k uložení adresy aktuálního uzlu a kapsule je odeslána dále. Obdobným způsobem jsou kapsule doručeny až na cílový uzel. Na tomto uzlu jsou kapsule otočeny a vráceny na původní uzel. Při zpáteční cestě probíhá doručování kapsulí obdobným způsobem, jen už do ní kvůli jednoduchosti nejsou ukládány adresy navštívených uzlů.

Po návratu kapsulí na výchozí uzel dojde k tisku seznamu navštívených uzlů. Z toho seznamu adres je vidět kudy jsou kapsule daného typu na dotyčný cílový uzel doručovány.

Spuštění

Testovací nástroj *TraceRoute* se spouští následujícím obdobným příkazem „***run TraceRoute 0-0-2-2 TS/NonTS 1***“. Prvním vstupním argumentem je adresa cílového uzlu (tj. adresa kam mají být vysílány kapsule), druhým argumentem udává typ kapsule a posledním argumentem je počet odeslaných kapsulí (tj. větší počet kapsulí může simulovat stálejší datový tok).

Implementace

Pro zjednodušení implemetace byly při implementaci tohoto jednoduchého testovacího nástroje použity stejné principy jako v implementaci směrovacího algoritmu *AntNet*.

K ukládání přenášených dat v kapsulích je rovněž používán mechanismus serializace implementovaný uvnitř SANu. Serializují se data jejichž strukturu popisuje třída *TraceRouteCapsule*. Dále je v adresáři zdrojových souborů

v separátní třídě *Utils* uložena množina pomocných sdílených procedur (sdílených mezi obě poloviny aktivní aplikace). Obě uvedené třídy jsou uloženy ve stejnojmenných zdrojových a hlavičkových souborech.

Vstupním bodem programu jak jsou soubory *TraceRoute.cpp* a *TraceRoute.h*. Poslední dvojice uložených souborů *DebugHelper.cpp* a *DebugHelper.h* obsahují procedury usnadňující odhalení chybějícího uvolňování alokované paměti.

4.3.4 Přidané příkazy uzlu

getRouteTable

Tento příkaz slouží uživateli k jednorázovému přehledovému výpisu aktuální podoby směrovací tabulky uzlu (viz *Obrázek 4.10*). Konkrétní podoba zadání příkazu má tento tvar „***getRouteTable TS/NonTS***“. Kde parametr udává typ požadované směrovací tabulky.

```
> getRouteTable TS
        Server TS Route records: (count=4)
        Id              <0>
        Interface Id    <0>
        Target          <::1:1>
        NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
        Gateway         <::0>
        Admin-Metric    <0>
        Prot-Metric     <0>
        Is Active       <1>
        Hold Timer      <60>

        Id              <1>
        Interface Id    <1>
        Target          <::4:2>
```

Obrázek 4.10: Ukázka příkazu na výpis směrovací tabulky uzlu

ifconfig

Tento příkaz slouží v příkazové konzoli SANu k práci se sít'ovými rozhraní uzlu. Zadaní samotného příkazu „***ifconfig***“ vypíše do konzole seznam sít'ových rozhraní uzlu včetně jejich aktuálního nastavení (např. viz *Obrázek 4.11*).

```
> ifconfig 0 down

> ifconfig

        Interface0
                Status:         Down
                Address:        <::4:2>
                NetMask:        <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
                Network:        <::4:0>

        Interface1
                Status:         Up
                Address:        <::1:1>
                NetMask:        <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
                Network:        <::1:0>
```

Obrázek 4.11: Výpis seznamu a stavu sít'ových rozhraní

Dále tento příkaz umožňuje pomocí těchto parametrů ovládat stav sít'ových rozhraní uzlu. Zadáním „***ifconfig 0 down/up***“ je možno dané rozhraní

softwarově vypnout. První parametr udává číselný identifikátor rozhraní (lze zjistit z výpisu) a druhý parametr stav do kterého má být rozhraní přepnuto. Po jeho deaktivaci uzel přes toto rozhraní přestává komunikovat s okolní sítí a jsou deaktivovány příslušné záznamy ve všech směrovacích tabulkách (viz *Obrázek 4.12*).

```
> ifconfig 0 down

> getRouteTable TS
        Server TS Route records: <count=4>
        Id              <0>
        Interface Id    <0>
        Target          <::1:1>
        NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
        Gateway         <::0>
        Admin-Metric    <0>
        Prot-Metric     <0>
        Is Active       <0>
        Hold Timer      <60>

        Id              <1>
        Interface Id    <1>
```

Obrázek 4.12: Ukázka deaktivovaného směrovacího záznamu

KAPITOLA 5 Implementace

Před začátkem samotného programování jsem se musel důkladně seznámit s již hotovými zdrojovými kódy projektu a prostudovat prinpipy veškerých rozhranní, které C++ implementace SANu poskytuje aktivním aplikacím. Navíc bylo potřeba si důkladně rozmyslet, co vše bude budoucí implementace AntNetu ze SANu potřebovat a domluvit se s dlouhodobým správcem (*Ing. Vladimír Aubrecht* viz poděkování) zdrojových kódů projektu na případných úpravách.

V rámci této práce byl vytvořen následující balík zdrojových souborů popsaný v následujícím textu. Balík se jmenuje AntNet a obsahuje soubory implementace stejnojmenného směrovacího algoritmu. Tento balík je uložen stejně jako balíky zdrojových souborů ostatních aktivních aplikací napsaných pro SAN do projektové složky *ActiveApplication* umístěné v kořenovém adresáři projektu. Úspěšný překlad každé z těchto aktivních aplikací skončí vytvořením stejnojmenné dll knihovny, která navenek exportuje pouze základní dvojici přístupových funkcí (např. *AntNet()* a *RunCapsule()*).

Všechny napsané zdrojové soubory dodržují základní koncept programovacího jazyka C, že každý hlavičkový soubor (*.h) obsahuje pouze definice hlaviček procedur a funkcí, zatímco veškerý výkonný kód je uložen pouze ve zdrojových souborech (*.cpp). Hlavními soubory implementace směrovacího algoritmu, je dvojice *AntNet.cpp* a *AntNet.h*.

Hned na začátku je ještě zapotřebí objasnit rozdíl mezi směrovací tabulkou aplikace a tabulkou přímých sousedů. **Směrovací tabulka aplikace** obsahuje veškeré nasbírané směrovací záznamy, zatímco **tabulka přímých sousedních uzlů** obsahuje pouze jejich podmnožinu. Tato podmnožina je definována jako seznam přímo sousedících uzlů ve vzdálenosti jediného skoku kapsule, tj. uzlů ve stejné subsíti. Udržování této dvojice tabulek usnadňuje často prováděné operace výběru adresy pro předání kapsule, protože se nemusí procházet a třídit záznamy na uzlů ve stejné subsíti.

Všechny vytvořené třídy obsahují mimo níže popsaných procedur i standardní množinu základních procedur (tj. konstruktor, destruktor s potřeb-

nými getry a setry privátních atributů).

Nakonec je nutné poznamenat, že všechny níže uvedené cesty ke zdrojovým a hlavičkovým souborům jsou platné pouze pro současnou verzi projektu. V budoucnu je možné, že se uložená adresářová struktura projektu uloženého v online SVN uložišti [2] bude měnit.

5.1 AntNet.cpp a AntNet.h

Hlavičkový soubor obsahuje pouze definice dvojice již zmíněných exportovaných funkcí (*AntNet()* a *RunCapsule()*), představujících vstupní body aktivní aplikace. Jeden vstupní bod pro každou z polovin aktivní aplikace (první je pro lokální běh aplikace, zatímco druhý pro zpracování přijatých kapsulí).

Ve zdrojovém souboru je pak napsán výkonný kód, ale vzhledem k rozsáhlosti napsaných zdrojových kódů (přes 3500 řádek) je kód každé z funkcí uložen ve více samostatných souborech. Kód lokálního běhu obsahují zdrojový *MasterCode.cpp* a příslušný hlavičkový *MasterCode.h* soubor. Zatímco programové instrukce procesu zpracování přijatých kapsulí jsou uloženy ve zdrojovém souboru *SlaveCode.cpp* a příslušném hlavičkovém souboru *SlaveCode.h*.

5.2 MasterCode.cpp a MasterCode.h

5.2.1 Atributy třídy

mRouteService

Privátní atribut třídy *MasterCode*, obsahuje referenci na směrovací služby SANu. Jedná se především o rozhranní základních operací používaných při práci s jeho směrovacími tabulkami (např. operace pro vložení, aktualizování, získání směrovacích záznamů).

Konkrétní definice a implementace tohoto rozhraní obsahuje minimálně tato množina zdrojových souborů *IRouterService.h*, *RouteServiceServer.cpp*, *RouteServiceServer.h*, uložené v adresáři „*./SAN/SAN/SandBox/Server/Services/*“ dále pak *RouteServiceProxy.cpp*, *RouteServiceProxy.h* uložené v adresáři „*./SAN/SAN/SandBox/Client/RemoteServices/*“ a nakonec *RouteService.cpp*, *RouteService.h* uložené v adresáři „*./SAN/SAN/SandBox/API/*“.

mNetworkService

Privátní atribut třídy *MasterCode*, obsahuje referenci na sít'ové služby SANu. Jedná se především o rozhranní základních operací pro odesílání kapsulí.

Konkrétní definice a implementace tohoto rozhraní obsahuje minimálně tato množina zdrojových souborů *INetworkService.h*, *NetworkServiceServer.cpp*, *NetworkServiceServer.h*, uložené v adresáři „*./SAN/SAN/SandBox/Server/-*

Services/" dále pak *NetworkServiceProxy.cpp*, *NetworkServiceProxy.h* uložené v adresáři „*./SAN/SAN/SandBox/Client/RemoteServices/*" a nakonec *NetworkService.cpp*, *NetworkService.h* uložené v adresáři „*./SAN/SAN/SandBox/API/*".

mStorageService

Privátní atribut třídy *MasterCode*, obsahuje referenci na datové služby SANu. Jedná se především o rozhranní základních operací používaných při práci s datovým uložištěm pro data aktivních aplikací (např. operace pro zápis či odstranění dat).

Konkrétní definice a implementace tohoto rozhraní obsahuje minimálně tato množina zdrojových souborů *IStorageService.h*, *StorageServiceServer.cpp*, *StorageServiceServer.h*, uložené v adresáři „*./SAN/SAN/SandBox/Server/Services/*" dále pak *StorageServiceProxy.cpp*, *StorageServiceProxy.h* uložené v adresáři „*./SAN/SAN/SandBox/Client/RemoteServices/*" a nakonec *StorageService.cpp*, *StorageService.h* uložené v adresáři „*./SAN/SAN/SandBox/API/*".

mRun

Privátní atribut třídy *MasterCode*, představuje příznak nekonečného běhu první poloviny aktivní aplikace (tj. neustálé odesílání kapsulí) a nekonečného běhu kontrolního vlákna (tj. neustálé kontroly). Výchozí hodnota je *true* a k ukončení běhu aplikace slouží procedura *Stop()*.

mAntNetRoutingTable

Privátní atribut třídy *MasterCode*, obsahuje referenci na třídu *AntNetRoutingTable* reprezentující směrovací tabulku aplikace. Nese veškeré směrovací záznamy včetně tabulky přímých sousedních uzlů a obsahuje procedury a funkce pro jejich obsluhu (např. procedura na vložení záznamu).

Přesný obsah a implementace třídy viz oddíl 5.5 nebo přímo zdrojové soubory „*./SAN/AntNet/AntNetRoutingTable.cpp*" a „*./SAN/AntNet/AntNetRoutingTable.h*".

mLocalInterfaceList

Privátní atribut třídy *MasterCode*, jedná se o seznam sít'ových rozhraní instance SANu. Seznam je realizován pomocí kontejneru *std::vector<IInterface*>* viz oddíl 5.9.1 a soubor *typedefs.h*.

Přesný obsah a implementace používaného rozhraní viz zdrojové soubory *IInterface.cpp*, *IInterface.h*, uložené v adresáři „*./SAN/SAN/Network/*".

mHoldTimerThread

Privátní atribut třídy *MasterCode*, jedná se standardní samostatně činné kontrolní vlákno. Kód jeho běhu představují procedury *HoldTimerThreadLoop()*,

ProcessAppRouteTable() a *ProcessNodeRouteTable()*.

5.2.2 Start()

Veřejná procedura třídy *MasterCode* bez parametrů, sloužící ke spuštění odesílací části aktivní aplikace *AntNet*. Obsahuje logiku nekonečné vysílací smyčky, včetně vytvoření, nastavení adres, výchozího naplnění a samotného odeslání kapsule. Zároveň také odesílané kapsuli nastavuje příznak označující, že se má zpracovávat podle programového kódu aplikace *AntNet* na každém navštíveném uzlu. Jelikož při výchozím nastavení se kód aktivních aplikací spouští pouze na výchozím a konečném cílovém uzlu, což nedostačuje pro správný chod tohoto algoritmu.

5.2.3 Stop()

Veřejná procedura *MasterCode* bez parametrů, sloužící ke změně hodnoty řídící proměnné *mRun* a tím ke korektnímu ukončení běhu nekonečných smyček. První v odesílací části aplikace *AntNet* a druhé v kontrolní vláknu. Obě smyčky totiž používají stejnou řídící proměnnou.

5.2.4 Initialization()

Privátní procedura třídy *MasterCode* bez parametrů, řídí proces prvotní inicializace odesílací poloviny aktivní aplikace AntNet. Spouští plnění seznamu sít'ových rozhraní uzlu, inicializaci směrovací tabulky aplikace včetně tabulky přímých sousedních uzlů a z principu fungování algoritmu hned zapisuje jejich aktuální podobu do datové storage. Na konci této procedury dochází k vytvoření a spuštění běhu kontrolního vlákna.

5.2.5 InitRoutingAndNeighborTable()

Privátní procedura třídy *MasterCode* bez parametrů obsahující řídící logiku plnění směrovací tabulky aplikace včetně tabulky přímých sousedních uzlů. Na svém začátku si získá statické záznamy obou směrovacích tabulek uzlu načtené z konfiguračního souboru. Poté zavoláním procedury *FillApplicationRoutingTable()* naplní všemi originálními záznamy směrovací tabulku uzlu, ze které je nakonec procedurou *FillNeighborTable()* ještě naplněna tabulka přímých sousedních uzlů.

5.2.6 FillApplicationRoutingTable()

Privátní procedura třídy *MasterCode*, které je jako jediný parametr předávána reference na konkrétní směrovací tabulku uzlu. Procedura prochází záznamy z tabulky uzlu, kontroluje možnou přítomnost podobných záznamů v tabulce aplikace a nové originální záznamy do tabulky aplikace přidává.

Výstupem je globálně uložená (*mAntNetRoutingTable*) naplněná směrovací tabulka aplikace.

5.2.7 FillNeighborTable()

Privátní procedura třídy *MasterCode*, které je jako jediný parametr předávána adresa sítě, do které je uzel konkrétním rozhraním připojen. Procedura filtruje záznamy směrovací tabulky aplikace na základě shodné adresy sítě a těmito záznamy plní tabulku přímých sousedních uzlů.

5.2.8 HoldTimerThreadLoop()

Privátní procedura třídy *MasterCode* bez parametrů. Obsahuje řídící kód kontrolního vlákna pracující s atributem *HoldTimer* ve směrovacích záznamech.

5.2.9 ProcessAppRouteTable()

Privátní procedura třídy *MasterCode*, které jsou jako parametry předávány ukazatel na konkrétní sít'ové rozhraní, jeho číselný identifikátor, ukazatel na směrovací tabulku aplikace včetně ukazatele na její konkrétní část (např. na tabulku přímých sousedních uzlů). Obsahuje výkonný kód kontrolního vlákna na zpracovávání směrovací tabulky aplikace (tj. snižování atributu *HoldTimer* a případné odebírání směrovacích záznamů).

5.2.10 ProcessNodeRouteTable()

Privátní procedura třídy *MasterCode*, které jsou jako parametry předávány ukazatel na konkrétní sít'ové rozhraní, jeho číselný identifikátor, typ směrovací tabulky a ukazatel na směrovací služby uzlu. Obsahuje výkonný kód kontrolního vlákna na zpracovávání konkrétní směrovací tabulky uzlu(tj. snižování atributu *HoldTimer* a případné odebírání směrovacích záznamů).

5.3 SlaveCode.cpp a SlaveCode.h

5.3.1 Atributy třídy

Každý privátní atribut třídy *SlaveCode* z následujícího výčtu **mRouteService**, **mNetworkService**, **mStorageService**, **mAntNetRoutingTable** a **mLocalInterfaceList** je svým názvem, datovým typem i použitím shodný se stejnojmenými atributem třídy *MasterCode* (popis těchto atributů viz oddíl 5.2.1).

mCapsule

Privátní atribut třídy *SlaveCode*, obsahuje referenci na instanci třídy *CCapsule*, která reprezentuje objekt konkrétní zpracovávané datové kapsule. Definici struktury této třídy obsahuje zdrojový a hlavičkový soubor „*Capsule.cpp*“ a „*Capsule.h*“ uložené v adresáři „*./SAN/SAN/Network/Services/*“.

5.3.2 ProcessCapsule()

Veřejná procedura třídy *SlaveCode* bez parametrů. Prvním příkazem procedury je deserializace bajtů do instance datového objektu (*AntNetCapsule*) aktivní aplikace *AntNet* z přijaté kapsule. Dále pak procedura obsahuje programovou logiku řídící zpracování všech přijatých kapsulí. Příkazy logiky tvoří větvení programu do čtyř větví podle směru doručované kapsule a toho zda je kapsule adresována danému uzlu nebo pouze prochází. V každé větvi se poté provádí příslušné kroky včetně volání odpovídajících metod.

Zpracování kapsule na mezilehlém uzlu se při doručování kapsule konečnému cílovému uzlu řídí procedurou *ForwardAntProcessing()*. Na konečném cílovém uzlu i na všech mezilehlých uzlech se zpracování kapsule během jejího návratu zpět na výchozí uzel řídí příkazy procedury *BackwardAntProcessing()*. Zatímco na výchozí uzlu proces zpracování představuje volaní procedury *ProcessReturnedCapsule()*, po jehož dokončení je ještě zapotřebí delegovat nové směrovací informace do směrovacích tabulek uzlu, což představuje volání další procedury *UpdateServerRoutingTables()*.

5.3.3 BackwardAntProcessing()

Privátní procedura třídy *SlaveCode*, které je jako jediný parametr předávána reference na datový objekt (konkrétně *AntNetCapsule*) aplikace *AntNet*. Procedura obsahuje řídící logiku procesu zpracování datové kapsule při jejím návratu zpět na výchozí uzel. Nejprve dochází k nastavení nové cílové adresy kapsule, poté ke zpracování přijatých a vložení nových směrovacích záznamů volaním procedury *MergeRoutingRecords()*. Následně je datový objekt zapsán zpět do kapsule a ta odeslána (viz popis algoritmu 3.2.3).

5.3.4 ForgetAddressFrom()

Privátní procedura třídy *SlaveCode*, které je jako první parametr předáván ukazatel na instanci datového objektu (*AntNetCapsule*) aktivní aplikace *AntNet* a jako druhý parametr index adresy daného uzlu v seznamu navštívených uzlů nalezený funkcí *InCycle()*.

Procedura je volána pouze když kapsule byla skutečně opakovaně doručena na daný uzel po nějaké smyčce (tj. index > 0) a mělo by podle kroků algoritmu dojít ke zkrácení seznamu navštívených uzlů na velikost danou indexem. Přičemž význam zkrácení seznamu spočívá v tom, že se pak kapsule může znovu zpracovat jako kdyby to bylo poprvé (viz oddíl 3.2.2).

5.3.5 ForwardAntProcessing()

Privátní procedura třídy *SlaveCode*, které je jako jediný parametr předávána reference na datový objekt (konkrétně *AntNetCapsule*) aplikace *AntNet*. Procedura obsahuje řídící logiku procesu zpracování datové kapsule při jejím doručování na konečný cílový uzel.

Procedura vybírá adresu náhodného přímého souseda pro další předání kapsule voláním sdílené procedury *SelectRandomAddress()*, ověřuje a řeší případné opakované doručování kapsule po smyčce a zaznamenává adresu daného uzlu do seznamu navštívených uzlů. Dále pak také procedura kontroluje aktuální velikost hodnoty parametru *HopCount* zpracovávané kapsule. Při jejím případném poklesu na polovinu počáteční velikosti otáčí kapsuli a spouští mechanismus shromažďování směrovacích záznamů jako kdyby kapsule byla právě doručena na konečný cílový uzel, což ještě umožňuje návrat kapsule zpět na výchozí uzel (viz popis algoritmu v oddílech 3.2.2 a 3.3).

5.3.6 InCycle()

Privátní funkce třídy *SlaveCode*, které je jako první parametr předáván ukazatel na začátek seznamu adres navštívených uzlů z datového objektu kapsule a jako druhý parametr velikost tohoto seznamu.

Funkce slouží ke kontrole zda kapsule nebyla doručena nejakou smyčkou opět na tento uzel (tj. je v seznamu sekvenčně vyhledávána adresa kteréhokoli rozhraní daného uzlu). Při prvním objeveném výskytu je prohledávání zastaveno a je vrácen index shodné adresy v seznamu. Tento index je dále použit pro zkrácování seznamu navštívených uzlů procedurou *ForgetAddressFrom()*. Pokud shodná adresa v seznamu nevyskytuje, znamená to, že kapsule byla na uzel doručena poprvé a v tom případě je vrácen záporný index, který není dále používán.

5.3.7 IsAddressOfThisServer()

Privátní funkce třídy *SlaveCode*, které je jako jediný parametr předávána cílová adresa kapsule. Funkce projde seznam síťový rozhranní uzlu a kontroluje, zda je cílová adresa kapsule shodná s některou adres uzlu. Kladný výsledek kontroly znamená, že kapsule je adresována tomuto uzlu a naopak. Funkce vrací výsledek této kontroly.

5.3.8 ProcessReturnedCapsule()

Privátní procedura třídy *SlaveCode* s jediným parametrem, kterým je ukazatel na datový objekt (*AntNetCapsule*) aktivní aplikace *AntNet*. Procedura obsahuje kroky procesu zpracování směrovacích záznamů z kapsule vrácené na výchozí uzel (viz popis algoritmu v oddílu 3.2.5) a příkazy pro jejich zápis do datové storage uzlu.

5.3.9 MergeRoutingRecords()

Privátní procedura třídy *SlaveCode*, které je jako jediný parametr předávána reference na datový objekt (konkrétně instance *AntNetCapsule*) aplikace *AntNet*.

K volání této procedury dochází na uzlech během návratu kapsule zpět na výchozí uzel. V proceduře se zpracovávají směrovací záznamy uložené v datovém objektu aplikace *AntNet* (instance *AntNetCapsule*). Poté se do objektu přidávají záznamy nové (viz popis algoritmu v oddílu 3.2.5).

Výstupem procedury je globálně uložená instance zpracovávané kapsule (*mCapsule*) naplněná směrovacími záznamy.

5.3.10 UpdateTablesInStorage()

Privátní procedura třídy *SlaveCode*, které je jako jediný parametr předávána reference na datový objekt (konkrétně instance *AntNetCapsule*) aplikace *AntNet*.

Ke spuštění procedury dochází na výchozím uzlu po dokončení zpracovaní vrácené kapsule. Procedura prochází zpracované směrovací záznamy, porovnává je se záznamy ve směrovací tabulce aplikace *AntNet*. Přičemž starší nebo méně výhodné záznamy aktulializuje a ostatní záznamy o zcela nových směrech do tabulky přidává. Nakonec procedura ukládá novou podobu směrovací tabulky aplikace do datové storage uzlu.

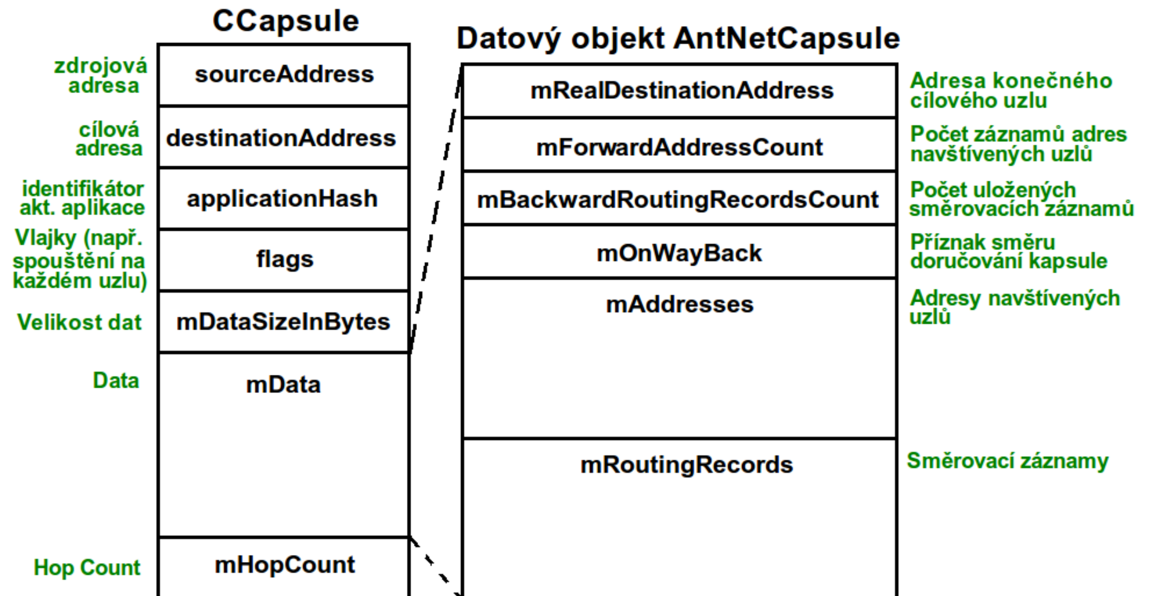
5.3.11 UpdateServerRoutingTables()

Privátní procedura třídy *SlaveCode* bez parametrů, která obsahuje první verzi programové logiky pro delegování nových směrovacích záznamů do správné směrovací tabulky uzlu. Základní koncepce spočívá v tom, že originální informace o úplně nejlepších směrech jsou propagovány do *TS* směrovací tabulky, zatímco se originální informace o druhých nejlepších směrech dostávají pouze do *NonTS* směrovací tabulky uzlu.

K této rozhodovací logiky je využíváno sdílených procedur pro vzájemné porovnávání směrovacích záznamů ze třídy *Utils*. Popis použitého algoritmu dále v textu viz oddíl 4.2.2.

5.4 AntNetCapsule.cpp a AntNetCapsule.h

Obsah této třídy představuje datovou strukturu veškerých dat zapisovaných aktivní aplikací *AntNet* do každé doručované kapsule (viz *Obrázek 5.1*). Pro zapisování tohoto objektu do kapsule je využíváno mechanismus serializace, který se používá a který byl prvoplánově implementován pouze pro vnitřní potřeby SANu pro práci s knihovnou *rpc.lib*. Nutnou podmínkou využívání tohoto mechanismu je implementování rozhraní *Serializable* (popis rozhraní viz oddíl 5.9.3).



Obrázek 5.1: Schéma struktury kapsule

5.4.1 Atributy třídy

mRealDestinationAddress

Privátní atribut třídy *AntNetCapsule* slouží pro ukládání adresy konečného cílového uzlu. Ukládání této adresy je provedeno takto, protože struktura datové kapsule používaná SANem obsahuje pouze jediný parametr cílové adresy (*destinationAddress*) a ten je v rámci implementace směrovacího algoritmu *AntNet* používán pro ukládání adresy přímého sousedního uzlu (tj. uzlu, kterému bude kapsule předána v následujícím kroku algoritmu).

Hodnotu parametru lze získat nebo nastavit pomocí getru *GetRealDestinationAddress()* a setru *SetRealDestinationAddress()*.

mForwardAddressCount

Privátní atribut třídy *AntNetCapsule* představuje čítač uložených záznamů adres navštívených uzlů během doručování kapsule konečnému cílovému uzlu.

Hodnotu parametru lze získat nebo nastavit pomocí getru *GetForwardAddressCount()* a setru *SetForwardAddressCount()*.

mBackwardRoutingRecordsCount

Privátní atribut třídy *AntNetCapsule* představuje čítač uložených směrovacích záznamů shromážděných během návratu kapsule zpět na výchozí uzel.

Hodnotu parametru lze získat nebo nastavit pomocí getru *GetBackwardRoutingRecordsCount()* a setru *SetBackwardRoutingRecordsCount()*.

mOnWayBack

Privátní atribut třídy *AntNetCapsule* představuje příznak aktuálního směru doručování kapsule. Výchozí hodnota je *false*, která označuje směr doručování na konečný cílový uzel, zatímco hodnota *true* odpovídá směru opačnému (false = forward / true = backward).

Hodnotu parametru lze získat nebo nastavit pomocí getru *GetOnWayBack()* a setru *SetOnWayBack()*.

mAddresses

Privátní atribut třídy *AntNetCapsule* představuje ukazatel na začátek pole ukládaných záznamů adres navštívených uzlů během doručování kapsule na konečný cílový uzel. Velikost pole resp. počet uložených záznamů udává atribut *mForwardAddressCount*.

Hodnotu parametru lze získat nebo nastavit pomocí getru *GetAddresses()* a setru *SetAddresses()*. Přidání nového záznamu je realizováno procedurou *AddAddress()*.

mRoutingRecords

Privátní atribut třídy *AntNetCapsule* představuje ukazatel na začátek pole ukládaných směrovacích záznamů během návratu kapsule zpět na výchozí uzel. Velikost pole resp. počet uložených záznamů udává atribut *mBackwardRoutingRecordsCount*.

Hodnotu parametru lze získat nebo nastavit pomocí getru *GetRoutingRecords()* a přidání nového záznamu je realizováno procedurou *AddRouteRecord()*.

5.4.2 AddAddress()

Veřejná procedura třídy *AntNetCapsule*, které je jako jediný parametr předán ukazatel na nový přidávaný záznam adresy navštíveného uzlu. Příkazy procedury provádí realokaci vetšího bloku paměti, překopírování původních záznamů včetně přidání nového, navýšení příslušného čítače uložených záznamů (*mForwardAddressCount*) a nakonec smazání původních záznamů včetně uvolnění paměti.

5.4.3 AddRouteRecord()

Veřejná procedura třídy *AntNetCapsule*, které je jako jediný parametr předán ukazatel na nový přidávaný směrovací záznam. Příkazy procedury provádí realokaci vetšího bloku paměti, překopírování původních záznamů včetně přidání nového, navýšení příslušného čítače uložených záznamů (*mBackwardRoutingRecordsCount*) a nakonec smazání původních záznamů včetně uvolnění paměti.

5.4.4 Serialize() a DeSerialize()

Dvojice veřejných procedur třídy *AntNetCapsule*, které musí být implementovány pro správné fungování používaného mechanismu serializace a deserializace.

K volání procedury *Serialize()* dochází při zapisování (serializaci) a procedury *DeSerialize()* při čtení (deserializaci) datového objektu (instance této třídy) do případně z doručované kapsule. Bližší popis těchto metod a rozhraní *Serializable* je v oddílu 5.9.3.

5.5 AntNetRoutingTable.cpp a AntNetRoutingTable.h

Tato třída představuje objekt sdružující směrovací tabulku aplikace *AntNet* a zároveň tabulku přímých sousedních uzlů. Tabulky slouží pro dlouhodobé uchovávání veškerých směrovacích záznamů z vrácených kapsulí. Jenže kvůli odlišnému mechanismu spouštění jednotlivých částí aktivní aplikace (viz oddíl 2.3.4) je veškerá pamět instancí této tabulky na konci běhu programu uvolněna. Takže je dlouhodobé uchovávání dat vyřešeno pomocí zapisování (serializace) současné podoby a načítání (deserializace) posledního stavu instance tohoto objektu do případně z datové storage. Třída musí pro využívání mechanismu serializace a deserializace implementovat rozhraní *Serializable* (popis rozhraní viz oddíl 5.9.3).

5.5.1 Atributy třídy

mAllRouteRecords

Privátní atribut třídy *AntNetRoutingTable* obsahuje seznam všech směrovacích záznamů ukládaných z vrácených kapsulí. Tento seznam je řešen pomocí standardní systémové kolekce (std::vector<CRouteRecord*>) viz soubor *typedefs.h*.

Počet uložených záznamů je uchováván vnitřně v rámci kolekce (lze získat pomocí getru *GetAllRouteRecordsCount()*). Samotný seznam lze získat pomocí getru *GetAllRouteRecords()*. Přidání nového záznamu je realizováno procedurou *AddRouteRecord()*.

mDirectNeighborRouteRecords

Privátní atribut třídy *AntNetRoutingTable* obsahuje seznam směrovacích záznamů na přímé sousední uzly (tj. na uzly ve stejně subsíti). Tento seznam je řešen pomocí standardní systémové kolekce (std::vector<CRouteRecord*>) viz soubor *typedefs.h*.

Počet uložených záznamů je uchováván vnitřně v rámci kolekce (lze získat pomocí getru *GetDirectNeighborRouteRecordsCount()*). Samotný seznam lze

získat pomocí getru *GetDirectNeighborRouteRecords()*. Přidání nového záznamu je realizováno procedurou *AddDirectNeighborRouteRecord()*.

5.5.2 AddDirectNeighborRouteRecord()

Veřejná procedura třídy *AntNetRoutingTable*, které je jako jediný parametr předáván ukazatel na nový přidávaný směrovací záznam cílený pouze na přímé sousední uzly. Příkazy procedury vytvářejí novou kopii přidávaného směrovacího záznamu, kterou poté přidávají nakonec seznamu (*mDirectNeighborRouteRecords*) a počet uložených záznamů je řešen vnitřně v rámci systémové kolekce. Vytváření kopie záznamu je bezpečnostní opatření, které má zvýšit stabilitu a hlavně zamezit vzniku neplatnosti dat, po případném vymazání vkládaného směrovacího záznamu někde vně této implementace.

5.5.3 AddRouteRecord()

Veřejná procedura třídy *AntNetRoutingTable*, které je jako jediný parametr předáván ukazatel na nový přidávaný směrovací záznam. Příkazy procedury vytvářejí novou kopii přidávaného směrovacího záznamu, kterou poté přidávají nakonec seznamu (*mAllRouteRecords*) a počet uložených záznamů je řešen vnitřně v rámci systémové kolekce. Vytváření kopie záznamu je bezpečnostní opatření, které má zvýšit stabilitu a hlavně zamezit vzniku neplatnosti dat, po případném vymazání vkládaného směrovacího záznamu někde vně této implementace.

5.5.4 Serialize() a DeSerialize()

Dvojice veřejných procedur třídy *AntNetRoutingTable*, které musí být implementovány pro správné fungování používaného mechanismu serializace a deserializace.

K volání procedury *Serialize()* dochází při zapisování (serializaci) a procedury *DeSerialize()* při čtení (deserializaci) instance této třídy do případně z datové storage. Bližší popis těchto metod a rozhraní *Serializable* je v oddílu 5.9.3.

5.5.5 RemoveRouteRecord()

Veřejná procedura třídy *AntNetRoutingTable*, které je jako první parametr předáván ukazatel směrovací tabulku, ze které má být odebrán směrovací záznam a jako druhý parametr je číselný identifikátor odebíraného záznamu. Procedura slouží na odebírání konkrétního směrovacího záznamu ze seznamu *mAllRouteRecords* nebo *mDirectNeighborRouteRecords*.

5.6 Utils.cpp a Utils.h

Tato třída je bez atributů. Obsahem této třídy jsou veřejné statické výkonné procedury a funkce sdílené oběma částem aktivní aplikace. Nachází se zde funkce pro realizaci výběru náhodné adresy, funkce na vzájemné porovnávání směrovacích záznamů, procedury abstrahující I/O operace datové storage a funkce k určení adresy, ze které bude kapsule příště odeslána.

5.6.1 ConvertVectorRouteRecord() a ConvertVectorRouteRecord()

Veřejné statické funkce třídy *Utils*, kterým jsou jako parametry předávány instance třídy *IRouteRecord*. Funkce slouží na hromadné přetypování instancí na třídu *CRouteRecord* kvůli objeveným nekompatibilitám.

5.6.2 CompareRouteRecords()

Veřejná statická funkce třídy *Utils*, které je předávána dvojice parametrů. První parametr je reference na původní a druhý parametr na nový směrovací směrovací záznam.

Procedura slouží ke vzájemnému porovnávání dvou směrovacích záznamů podle všech jejich atributů. Výsledkem porovnání je logická hodnota *true* nebo *false*, která udává zda jsou záznamy totožné či nikoliv.

Při volání této procedury je kvůli porovnávání metrik důležité dodržet správné pořadí parametrů (tj. porovnávaných směrovací záznamů), aby nedošlo k chybné interpretaci výsledků.

5.6.3 CompareRouteRecordsByMetrics()

Veřejná statická funkce třídy *Utils*, které je předávána dvojice parametrů. První parametr je reference na původní a druhý parametr na nový směrovací směrovací záznam.

Procedura slouží ke vzájemnému porovnávání dvou směrovacích záznamů podle jejich metrik. Výsledkem porovnání je logická hodnota *true* nebo *false*, která udává zda se záznamy v klíčovém parametru shodují či nikoliv.

Při volání této procedury je kvůli porovnávání metrik důležité dodržet správné pořadí parametrů (tj. porovnávaných směrovací záznamů), aby nedošlo k chybné interpretaci výsledků.

5.6.4 CompareRouteRecordsByNetwork()

Veřejná statická funkce třídy *Utils*, které je předávána dvojice parametrů. První parametr je reference na původní a druhý parametr na nový směrovací směrovací záznam.

Procedura slouží ke vzájemnému porovnávání dvou směrovacích záznamů podle jejich cílových sítí (tj. dvojice adresy a sít'ové masky). Výsledkem porovnání je logická hodnota *true* nebo *false*, která udává zda se záznamy v klíčovém parametru shodují či nikoliv.

5.6.5 CompareRouteRecordsForUpdate()

Veřejná statická funkce třídy *Utils*, které je předávána dvojice parametrů. První parametr je reference na původní a druhý parametr na nový směrovací směrovací záznam.

Procedura slouží ke vzájemnému porovnávání dvou směrovacích záznamů. Vysledkem porovnání je logická hodnota *true* nebo *false*, která udává zda má být provedena aktualizace starého směrovacího záznamu tím novým či nikoliv. Porovnávání probíhá v následujících krocích. Nejprve se záznamy testují na shodu cílové sítě (tj. dvojice adresy a sít'ové masky). Při jejich shodě se dále voláním procedury *CompareRouteRecordsByMetrics()* vzájemně porovnávají jejich metriky. Pokud jsou nové výhodnější vrací se *true*, jinak *False*.

Při volání této procedury je kvůli porovnávání metrik důležité dodržet správné pořadí parametrů (tj. porovnávaných směrovací záznamů), aby nedošlo k chybné interpretaci výsledků.

5.6.6 GetNextInputAddress()

Veřejná statická funkce třídy *Utils*, které jsou předávány čtyři parametry. První parametr je ukazatel na pamět, do které má být uložena nalezená adresa, tj. výstupní hodnota. Druhý parametr je reference na záznam adresy příštího cílového uzlu, kterému bude kapsule předána. Třetí parametr je ukazatel na tabulku přímých sousedů daného uzlu a poslední parametr je ukazatel na seznam sít'ových rozhraní daného uzlu.

Procedura slouží k nalezení záznamu adresy lokálního sít'ového rozhraní podle záznamu adresy příštího cílového uzlu, kterému bude v dalším kroku algoritmu kapsule předána (tj. některému z přímých sousedních uzlů).

Nalezená adresa je ukládána do kapsule do seznamu navštívených uzlů.

5.6.7 GetRouteRecordByAddress()

Privátní statická funkce třídy *Utils*, které je předávána dvojice parametrů. První parametr je ukazatel na směrovací tabulku, ve které má být hledán výsledný záznam. Druhým parametrem je ukazatel na záznam adresy, podle které má být výsledný záznam nalezen.

Procedura slouží k prohledání předaného seznamu a nalezení konkrétního směrovacího záznamu podle jeho cílové adresy.

Návratovou hodnotou funkce je ukazatel na nalezený směrovací záznam. Pokud záznam nebude nalezen vrací se hodnota *NULL*, což ale znaména chybu, protože uzel v tomto okamžiku neví kam má kapsuli odeslat. Do tohoto stavu, by se spravně nakonfigurovaný uzel neměl dostat.

// procedura na slouceni dvojice smerovacich zaznamu do jedineho (pouziva se pro sdruzovani zaznamu na jednotlive uzly do jedineho zaznamu na sit)

5.6.8 MergeRouteRecordsToNetwork()

Veřejná statická funkce třídy *Utils*, které je předávána trojice parametrů. První dvojice parametrů jsou ukazatele na směrovací záznamy, které mají být sloučeny a třetím je ukazatel na pamět, do které májí být uložen výsledný záznam. Funkce slouží na sloučení dvojice směrovacích záznamů do jediného výstupního záznamu, který bude ukazovat na cílovou sít'.

5.6.9 ReadTablesFromStorage()

Veřejná statická funkce třídy *Utils*, které je předávána dvojice parametrů. První parametr je ukazatel na pamět, do které májí být uložena načtená data (tj. instance třídy *AntNetRoutingTable*). Druhý parametr je ukazatel na datovou storage, ze které májí být data čtena.

Procedura slouží k načtení (deserializaci) aktuální podoby směrovací tabulky aplikace včetně tabulky přímých sousedních uzlů z datové storage.

5.6.10 SelectRandomAddress()

Veřejná statická funkce třídy *Utils*, které je jako první parametr předáván ukazatel na pamět, do které má být uložena náhodně vybraná adresa. Druhým parametrem je reference na směrovací tabulku, ze které se má náhodná adresa vybírat (tabulka označuje množinu záznamů, tj. ze všech nebo pouze z přímých sousedních uzlů). Poslední třetí parametr je ukazatel na seznam sít'ových rozhraní daného uzlu. Ten zde slouží ke zvýšení efektivity algoritmu, protože je pomocí něj kontrolováno, zda nebyla jako náhodná adresa vybrána některá z lokálních adres. Výběr lokální adresy by totiž měl za následek opakované zpracovávání kapsule na témže uzlu namísto jejího odeslání na jiný uzel.

Zvolená výstupní adresa je cílovou adresou náhodně vybraného směrovacího záznamu. Přičemž náhodný výběr směrovacího záznamu je realizován pomocí základního generátoru náhodných čísel, tj. prostřednictvím funkce *rand()*, ze standardní knihovny *stdlib.h.*

5.6.11 WriteTablesToStorage()

Veřejná statická procedura třídy *Utils*, které je předávána dvojice parametrů. První parametr je reference na objekt sdružující směrovací tabulku aplikace a tabulku přímých sousedních uzlů (tj. instance třídy *AntNetRoutingTable*). Druhý parametr je ukazatel na datovou storage, do které má být objekt uložen.

Procedura slouží k zápisu (serializaci) aktuální podoby směrovací tabulky aplikace včetně tabulky přímých sousedních uzlů do datové storage.

5.7 DebugUtils.cpp a DebugUtils.h

Tato třída je bez atributů. Obsahem této třídy jsou veřejné statické pomocné procedury především na tisk ladících informací a logování.

5.7.1 PrintAddressTable()

Veřejná statická procedura třídy *DebugUtils*, které je jako první parametr předáván ukazatel na začátek pole záznamů adres navštívených uzlů a jako druhý počet uložených záznamů.

Procedura slouží ke kontrolnímu tisku seznamu adres navštívených uzlů ukládaného do kapsule během jejího doručování konečnému cílovému uzlu.

5.7.2 PrintCapsuleData()

Veřejná statická procedura třídy *DebugUtils*, které je jako jediný parametr předáván ukazatel na datový objekt (instanci třídy *AntNetCapsule*) aktivní aplikace *AntNet*.

Procedura slouží ke kontrolnímu výpisu obsahu tohoto objektu (tj. veškerá data posílaná aplikací *AntNet*). Struktura datového objektu viz oddíl 5.4.

5.7.3 PrintInterfaces()

Veřejná statická procedura třídy *DebugUtils*, které je jako jediný parametr předáván na seznam síťových rozhraní uzlu. Tento seznam představuje systémová kolekce *std::vector<IInterface *>* (viz soubor *defines.h*). Procedura slouží ke kontrolnímu tisku seznamu síťových rozhraní daného uzlu včetně jejich aktuálního nastavení.

5.7.4 PrintRoutingTable()

Dvojice statických veřejných procedur třídy *DebugUtils*, které jsou rozlišené typem a počtem parametrů. Principiálně je oběma předávána směrovací tabulka, která má být procedurou vytisknuta.

První verzi je jediným parametrem předávána reference na kolekci *std::vector<IRouteRecord*>* (viz soubor *defines.h*), zatímco druhá verze dostává prvním parametrem ukazatel na začátek pole směrovacích záznamů a druhým počet těchto záznamů.

5.8 Defines.h

Tento hlavičkový soubor obsahuje nasledující preprocesorová makra pro aktivaci příslušných ladících a kontrolních výpisů a jejich logování (např. tisknutí obsahu posílaných kapsulí). V odladěné verzi mohou být tato makra zakomentována, tak aby se algoritmus tisknutím informací zbytečně nezdržoval. Ovšem pro ověření správné funkčnosti nebo pro usnadnění dalších programátorských zásahů lze odkomentováním příslušná makra a výpisy opět aktivovat.

Implementace *AntNetu* má totiž v současné verzi vlastní logovací machanismus, jelikož se z časových důvodů podařilo zpřístupnit logovací mechanismus aktivním aplikacím.

Dále se pak v tomto souboru nachází definice směrovacím algoritmem používaných konfiguračních konstant a klíčů (např. délka uspání procesu mezi odesíláním jednotlivých kapsulí).

Po nich následuje seznam vnitřních návratových chybových kódů pro případy externích chyb, které algoritmus sám o sobě není schopen vyřešit (např. při pokusu o čtení z prázdné datové Storage, když na daný uzel přijdě kapsule *AntNetu* a zároveň když na daném uzlu není *AntNet* spuštěn). Výsledkem těchto chyb je ukončení běhu algoritmu.

5.8.1 Ladící makra

LOCAL_INTERFACE_LIST

Po aktivaci tohoto makra bude dotyčný bude směrovací algoritmus na daném uzlu tisknout seznam nalezených sít'ových rozhraní s jejich aktuálním nastavením (tj. adresy a sít'ové masky).

SERVER_NONTS_ROUTING_TABLE

Toto makro aktivuje kontrolní vypisování výchozí podoby *NonTS* směrovací tabulky uzlu (tj. záznamy statických cest načtené z konfiguračních souborů).

SERVER_TS_ROUTING_TABLE

Toto makro aktivuje kontrolní vypisování výchozí podoby *TS* směrovací tabulky uzlu (tj. záznamy statických cest načtené z konfiguračních souborů).

APPLICATION_ROUTING_TABLE

Povolením makra se aktivuje kontrolní vypisování výchozí podoby směrovací tabulky aplikace (tj. implementace směrovacího algoritmu). Tato tabulka by měla obsahovat množinu originálních záznamů algoritmem sloučených *NonTS* a *TS* směrovacích tabulek uzlu).

NEIGHBOR_TABLE

Po aktivaci tohoto makra bude na daném uzlu docházet k tisknutí obsahu výchozí podoby tabulky přímých sousedních uzlů, tj. uzlů ve stejné subsíti. Tato tabulka by měla být naplněna množinou originálních záznamů získaných sloučením *NonTS* a *TS* směrovacích tabulek uzlu).

RANDOM_REAL_DESTINATION_ADDRESS

Toto makro spouští na daném uzlu tisk náhodně vybrané adresy konečného cílového uzlu (tj. adresy uzlu, kterému bude kapsule adresována). Adresa je vybírána z adres všech známých uzlů, tj. ze směrovací tabulky aplikace.

RANDOM_ONE_HOP_ADDRESS

Toto makro spouští na daném uzlu tisk náhodně vybrané adresy přímého sousedního uzlu (tj. adresy uzlu, kterému bude kapsule předána). Adresa je vybírána pouze z adres známých přímých sousedních uzlů, tj. z tabulky přímých sousedů.

CAPSULE_SOURCE_ADDRESS

Toto makro spouští na daném uzlu tisk adresy lokálního rozhraní, kterým bude kapsule odeslána do sítě a kterou bude mít kapsule nastavenu jako zdrojovou.

SENDING_DATA

Směrovací algoritmus bude, na daném uzlu, po aktivaci tohoto makra, před každým odesláním kapsule aktivní aplikace AntNet tisknout její obsah. Hlavně tedy seznamy navštívených uzlů a nasbíraných směrovacích záznamů (přesný obsah viz oddíl 5.4 popisující obsah kapsule).

RECEIVED_DATA

Směrovací algoritmus bude, na daném uzlu, po aktivaci tohoto makra, po každém přijmutí kapsule aktivní aplikace AntNet tisknout její obsah. Hlavně tedy seznamy navštívených uzlů a nasbíraných směrovacích záznamů (přesný obsah viz oddíl 5.4 popisující obsah kapsule).

UPDATED_APPLICATION_ROUTING_TABLE

Následkem povolení tohoto makra bude na uzlu po návratu každé kapsule (tj. návrat na výchozí uzel) a jejím zpracování tisknuta aktualizovaná nová podoba směrovací tabulky aplikace.

UPDATED_DIRECT_NEIGHBOR_TABLE

Následkem povolení tohoto makra bude na uzlu po návratu každé kapsule (tj. návrat na výchozí uzel) a jejím zpracování tisknuta aktualizovaná nová podoba tabulky přímých sousedů daného uzlu.

UPDATED_SERVER_NONTS_ROUTING_TABLE

Toto makro na daném uzlu aktivuje po návratu každé kapsule (tj. návrat na výchozí uzel) a jejím zpracování tisk aktualizované nová podoby směrovací NonTS tabulky uzlu.

UPDATED_SERVER_TS_ROUTING_TABLE

Toto makro na daném uzlu aktivuje po návratu každé kapsule (tj. návrat na výchozí uzel) a jejím zpracování tisk aktualizované nová podoby směrovací TS tabulky uzlu.

FORWARD_ANT_PROCESSING_CYCLE_DETECT

Aktivací tohoto makra algoritmus dostane pokyn k tisknutí výsledků kontroly zda kapsule není doručována po nějaké smyčce. Princip a důvod tohoto kontrolování je popsán v oddílu 3.2.2.

RESULT_OF_CONTROL_HOPCOUNT

Aktivací tohoto makra algoritmus dostane pokyn k tisknutí výsledků kontroly aktuální velikosti atributu *HopCount* u kapsulí během procesu doručování konečnému cílovému uzlu.

HOLD_TIMER_THREAD_UPDATES

Toto makro na daném uzlu aktivuje tisknutí výsledků jednotlivých kontrol hodnoty *HopCount* a aktuálních podob směrovacích tabulek.

MUTEX_ACTION

Toto makro povoluje kontrolní výpisy synchronizačních akcí (označení míst zamykání a odemykání sychronizačního mutexu).

HOLD_TIMER_THREAD

Toto makro zakázáním tohoto makra dojde k zakomentování instrukcí na vytvoření a spuštění kontrolního vlákna s nebude tak docházet ke kontrolám a snižování atributu *HoldTimer*. Používáno především pro ladění. Standardní stav je povolené makro.

OSTATNI_LADICI_VYPISY

Toto makro aktivuje vypisování kontrolních výpisů na začátcích a koncích jednotlivých procedur a funkcí, ke snadnějšímu odhalení místa havárie programového kódu. Používáno především při prvotní tvorbě.

5.8.2 Programové konstanty

NEW_HOLD_TIMER

Hodnota této konstanty udává kolik času ve vteřinách se má jako nový parametr platnosti (tzv. hold timer viz oddíl 5.9.2) nastavit novým, případně aktualizovaných směrovacím záznamům. Výchozí nastavení je 60s.

SLEEP_TIME

Hodnota této konstanty v mikrosekundách udává délku uspání odesílacího procesu mezi odesláním dvou po sobě následujících kapsulí. Výchozí nastavení je 5000000microsec.

HOLD_TIMER_THREAD_SLEEP

Hodnota této konstanty v mikrosekundách udává délku uspání kontrolního vlákna mezi jednotlivými kontrolami. Výchozí nastavení je 1000000microsec.

DECREASING_HOLDTIMER_SPEED

Hodnota této konstanty udává o jakou hodnotu se má při každé kontrole snižovat hodnota atribut *HoldTimer* u směrovacích záznamů. Výchozí nastavení je 1.

5.8.3 Návratové chybové kódy

NO_SAVED_DATA

Ke vzniku této chyby může na uzlu dochází v okamžiku, když na daném uzlu neběží žádná instance AntNetu a přijde na tento uzel kapsule směrovacího algoritmu od jiného uzlu. Protože po přijmutí kapsule se na uzlu spustí druhá polovina aktivní aplikace (podle funkce *RunCapsule()*) a snaží se o aktualizování směrovacích dat z datové storage. Jenže data v této storage inicializuje a poprvé zapisuje pouze lokálně běžící instance směrovacího algoritmu (tj. první polovina aktivní aplikace běžící podle funkce *AntNet()*).

Výchozí návratovou hodnotou je 1 a řešení po vzniku této chyby spočívá ve spuštění instance směrovacího algoritmu na daném uzlu.

5.9 Další důležité soubory

Toto jsou soubory mimo balík vytvořených zdrojových souborů, které jsou uloženy v různých částech projektu SAN, ale zároveň jsou pro vytvořenou implementaci nějak důležité.

5.9.1 Typedefs.h

Kvůli zamezení problémům vznikajících používáním různě se překrývajících jmených prostorů (tj. konstrukcí *using namespace X::Y::Z*) a jelikož nebylo vyjímkou, že je v souboru používáno několik těchto prostorů najednou, tak bylo v tomto projektu používání zmíněné konstrukce zákázáno.

Tento zákaz na druhou stranu vedl ke snížení přehlednosti a čitelnosti všech zdrových kódů projektu SAN, protože se u každého používaného typu (např. *CCapsule*) musí napsat jeho jméno včetně celé cesty přes jmené prostory (např. *SAN::Network::Services::CCapsule*). Takže byl pro její opětovné navýšení vytvořen tento hlavičkový soubor. Jeho obsahem jsou redefinice všech používaných datových typů celého projektu, včetně implemetace AntNetu.

Toto řešení přináší za cenu mírné nepraktičnosti zvýšení stability, omezení vzniku chyb tímto zakázaným používáním jmenných prostorů a možnost jejich snadnějšího debugování.

5.9.2 IRouteRecord.h a RouteRecord.h

Kvůli jednoduchosti, praktičnosti a omezení vzniku případných chyb tato implementace směrovacího algoritmu *AntNet* narozdíl od všech předchozích nedefinuje žádné vlastní datové typy (např. vlastní podobu směrovacího záznamu). Namísto toho rovnou využívá datové typy SANu (např. třídu směrovacího záznamu *CRouteRecord*, která je včetně příslušného rozhraní definováná v této dvojici hlavičkových souborů uložených v adresáři „*./SAN/-SAN/Network/Routing/*“).

5.9.3 ISerializable.h, Serializable.cpp a Serializable.h

Tyto soubory se nachází v adresáři „*./SAN/rpc/San/Rpc/Communication/*“ a nachází se v nich definice včetně částí implementace rozhraní *Serializable*.

Toto rozhraní implementují třídy datových objektů *AntNetCapsule* a *AntNetRoutingTable*, které se serializují do souvislého pole bajtů, jenž je poté zápisováno do doručované kapsule případně do datové storage.

Při používání tohoto rozhraní je potřeba před každým zavoláním procedury *serialize()* zavolat proceduru *resetCompleteSize()* na resetování vnitřních čítačů serializace. Jinak zvláště při opakovaném volání hrozí vznik špatně debugovatelné chyby dané špatnými hodnotami čítačů velikosti serializovaných dat.

5.9.4 CCapsule.cpp a CCapsule.h

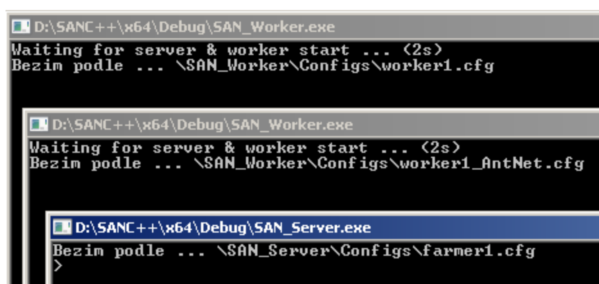
V těchto souborech se nachází definice a implementace struktury doručovaných kapsulí (tj. třída *CCapsule*). Soubory jsou uloženy v adresáři „*./SAN/SAN/Network/Services/*“.

KAPITOLA 6

Uživatelský manuál

6.1 Struktura konfiguračních souborů

Běh jedné instance SANu v současnosti představuje spouštění trojice základních procesů jednoho *SAN_Farmer* a dvou *SAN_Worker* (viz *Obrázek 6.1*). Podrobný popis jejich běhu by tvořil dokumentaci celého projektu a rozsahem by určitě předčil i rozsah tohoto dokumentu. Ale ve zkratce lze považovat *SAN_Farmer* za proces řídící chod celého serveru, který má na starosti vše důležité. Oproti tomu první proces *SAN_Worker* pouze dostává úkoly, především v něm běží zpracovávání veškerých přijatých kapsulí. A ve druhé instanci *SAN_Worker* v současnosti neustále běží směrovací algoritmus *AntNet*.



Obrázek 6.1: Ukázka spuštěného uzlu

Každý z těchto procesů má vlastní konfigurační soubor s odpovídajícím názvem a jejich obsah popisují následující oddíly. Mezi zrojovými soubory se nachází i množina těchto konfiguračních souborů obsahující nastavení několika testovacích topologií (jejich popis viz oddíl 4.3) a jejich funkční ukázka je k nahlédnutí v přílohách A a B tohoto dokumentu.

6.1.1 farmer.cfg

Tento konfigurační soubor obsahuje nastavení procesu *SAN_Farmer* a je rozčleněn do různých oddílů či sekcí. První oddíl ***[definition]*** je ve struktuře konfiguračního souboru prozatimní, shodná ve všech konfiguračních souborech dané topologie a obsahuje definice promenných (substitucí). Jsou zde definovány především IP adresy, které se tímto snadněji mění v rámci celého souboru. Množství těchto adres odpovídá množství používaných sít'ových rozhraní v dané topologii, tj. každé rozhraní zatím musí mít kvůli funkčnosti RPC vlastní IP adresu.

Druhá sekce ***[paths]*** obsahuje nastavení cest k adresářům, které používá tento proces. Následující oddíl ***[init]*** udává seznam automaticky spouštěných služeb (resp. procesů) hned po startu SANu a jejich logování na úrovni uzlu.

Část ***[rpc]*** obsahuje klíčové parametry RPC mechanismu, který používají procesy *SAN_Farmer* a *SAN_Worker* pro vzájemnou komunikaci. Zde je především důležité mít nastavenu správnou IP adresu včetně komunikačního port odpovídající nastavení v konfiguračních souborech *worker.cfg* a *worker_AntNet.cfg* .

Poté následuje jedna či více sekcí ***[network/interface[]]***, kde každá tato sekce definuje jedno sít'ové rozhraní uzlu. Rozhraní je indentifikováno na základě automaticky přidělovaného identifikátoru *interface_id*, které je klíčové při zapisování statických směrovacích záznamů na loopback a přímé sousední uzly.

Následující množina oddílů ***[network/resolver[]]*** je v konfiguračním souboru také prozatimní. Tyto oddíly totiž udržují informace pro převod IP adresy na SAN adresu a zpět.

Na předposledním místě se v nachází množina sekcí ***[network/route[]]***, kde každá představuje samostatný směrovací záznam. Nejprve jsou definovány loopback-ové [1] směrovací záznamy na lokální rozhraní a poté skupina směrovacích záznamů na přímé sousední uzly. Ruční definice těchto záznamů je zde z principu směrovacího algoritmu *AntNet* důležitá, jelikož si je uzel zatím není schopen zjišt'ovat automaticky, např. pomocí broadcastu na lokálním segmentu sítě. Směrovací záznamy na ostatní uzly již budou generovány směrovacím algoritmem.

Na konci souboru je dvojice sekcí ***[network/defaultGateway[]]*** představující definici výchozích bran doručovaných kapsulí. Brány jsou 2, pro každý typ kapsule (*TS/NonTS*) maximálně jedna. Přičemž obě mohou ukazovat stejným směrem, ale brána pro *TS* kapsule má principiálně vyšší prioritu.

6.1.2 worker.cfg

V oddílu ***[paths]*** jsou konfigurovány cesty k adresářům, které používá proces *SAN_Worker*. Sekce ***[init]*** udává seznam automaticky spouštěných služeb

[1] Loopback - Virtuální lokální sít'ové rozhraní.

(resp. procesů) hned po startu SANu, včetně seznamu logovaných služeb na úrovni uzlu. Poslední část ***[rpc]*** obsahuje klíčové parametry RPC mechanismu, které používají procesy *SAN_Farmer* a *SAN_Worker* pro vzájemnou komunikaci. Zde je především důležité mít nastavenu správnou IP adresu včetně dvojice komunikačních portů odpovídající nastavení v konfiguračním souboru procesu *SAN_Server* (tj. zeleně označené sekce si musí odpovídat).

6.1.3 worker_AntNet.cfg

Obsah tohoto konfiguračního souboru je v současnosti téměř totožný s obsahem konfiguračního souboru *worker.cfg*. Jediné v čem se nastavení těchto procesů liší je rozdílný komunikační port v oddílu ***[rpc]*** a v sekci ***[init]***, kde je nastaveno automatické spuštění směrovacího algoritmu.

6.2 Spuštění

6.2.1 SAN

Pro usnadnění realizace experimentu ověřující funkčnost konceptu směrování horší cestou viz oddíl 4 byly vytvořeny tyto sady spouštěcích dávkových souborů usnadňující spouštění vícera instancí. Ke spuštění konkrétního instance slouží dávkový soubor ***runX.bat***, kde X udává číslo uzlu v dané topologii. Schémata topologií včetně nastavení jednotlivých instancí obsahují příslušné konfigurační soubory (viz oddíl 4.3). Soubor ***runAll.bat*** pak pouze shnuje obsah všech spouštěcích dávkových souborů dané topologie a spouští všechny instance najednou. V tomto adresáři je vytvořen ještě poslední dávkový soubor ***killAll.bat***, který je možno použít pro hromadné ukončení běhu všech běžících instancí.

6.2.2 AntNet

Spuštění samotné implementace směrovacího algoritmu *AntNet* bez uzlu (tj. procesů *SAN_Farmer* a *SAN_Worker* nemá význam ani není dost dobře možné, protože je se pomocí vývojového prostředí MSVS 11 [2] zdrojový kód překádá do *.dll knihovny *AntNet.dll*.

Směrovací algoritmus v součastnosti nelze ani spustit příkazem z příkazové konzole procesu *SAN_Farmer*, jelikož v projektu SAN zatím chybí podpora pro zprávu, plánování a spouštění procesů. Spuštěním směrovacího algoritmu příkazem z konzole dojde sice k jeho startu v procesu *SAN_Worker*, jenže tento proces již nebude moci obsluhovat přijímané kapsule a uzel tak nebude fungovat plnohodnotně.

Jediný možný způsob úspěšného spuštění uzlu včetně běžícího směrovacího algoritmu je pomocí vytvořených dávkových souborů ***runX.bat***. Dávkový soubor postupně spustí trojici samostatných procesů (1x *SAN_Farmer*

[2]MSVS 11 - Microsoft Visual Studio 11.

a 2x *SAN_Worker*). Kde v jednom *SAN_Workeru* poběží samostatně směrovací algoritmus, zatímco druhý zůstane v pohotovosti a může zpracovávat přijímané kapsule.

Takže algoritmus startuje pomocí dávkových souborů automaticky a jeho činnost lze ověřit kontrolním výpisem směrovací tabulky uzlu (viz oddíl 4.3.4).

KAPITOLA 7

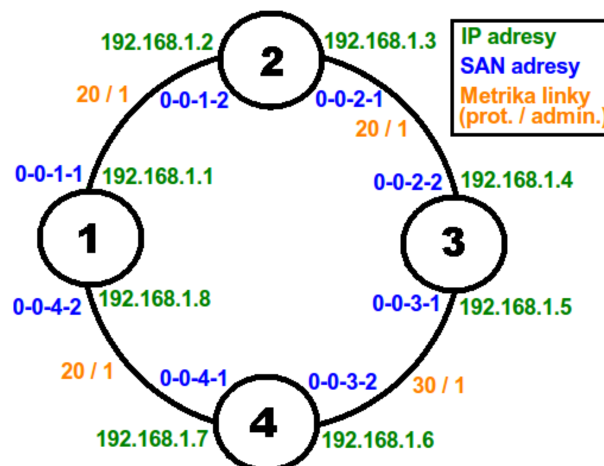
Dosažené výsledky

7.1 *AntNet*

7.1.1 Testovací topologie 1

Na sérii obrázků *7.2, 7.2, 7.4, 7.5, 7.6, 7.7, 7.8 a 7.9* se nachází výpisy obsahu směrovacích tabulek jednotlivých uzlů v testovací *topologii1* (viz schéma 7.1).

Testovací topologie 1



Obrázek 7.1: Schéma a nastavení testovací topologie 1

Uzel 1 - TS směrovací tabulka

```
Id              <0>
Interface Id    <1>
Target          <0::1:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <0>
Target          <0::4:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <1>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <0>
Target          <0::4:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <1>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <5>
Interface Id    <1>
Target          <0::3:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:2>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.2: TS směrovací tabulka uzlu 1

Uzel 1 - NonTS směrovací tabulka

```
Id              <0>
Interface Id    <1>
Target          <0::1:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <0>
Target          <0::4:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <1>
Target          <0::1:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <0>
Target          <0::4:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <1>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:2>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.3: NonTS směrovací tabulky uzlu 1

Uzel 2 - TS směrovací tabulka

```
Id              <0>
Interface Id    <0>
Target          <0::1:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <1>
Target          <0::2:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <0>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <1>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <0>
Target          <0::4:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <5>
Interface Id    <0>
Target          <0::3:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:1>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.4: TS směrovací tabulka uzlu 2

Uzel 2 - NonTS směrovací tabulka

```
Id              <0>
Interface Id    <0>
Target          <0::1:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <1>
Target          <0::2:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <0>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <1>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <0>
Target          <0::4:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:1>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>

Id              <6>
Interface Id    <0>
Target          <0::3:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::1:1>
Admin-Metric    <3>
Prot-Metric     <70>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.5: NonTS směrovací tabulky uzlu 2

Uzel 3 - TS směrovací tabulka

```
Id              <0>
Interface Id    <0>
Target          <0::2:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <1>
Target          <0::3:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <0>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <1>
Target          <0::3:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::3:2>
Admin-Metric    <1>
Prot-Metric     <30>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <0>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <5>
Interface Id    <1>
Target          <0::4:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::3:2>
Admin-Metric    <1>
Prot-Metric     <30>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.6: TS směrovací tabulky uzlu 3

Uzel 3 - NonTS směrovací tabulka

```
Id              <0>
Interface Id    <0>
Target          <0::2:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <1>
Target          <0::3:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <0>
Target          <0::2:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:1>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <1>
Target          <0::3:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::3:2>
Admin-Metric    <1>
Prot-Metric     <30>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <0>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:1>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>

Id              <5>
Interface Id    <1>
Target          <0::4:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::2:1>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.7: NonTS směrovací tabulky uzlu 3

Uzel 4 - TS směrovací tabulka

```
Id              <0>
Interface Id    <0>
Target          <0::3:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <1>
Target          <0::4:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <0>
Target          <0::3:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::3:1>
Admin-Metric    <1>
Prot-Metric     <30>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <1>
Target          <0::4:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <0>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::3:1>
Admin-Metric    <1>
Prot-Metric     <30>
Is Active       <1>
Hold Timer      <60>

Id              <5>
Interface Id    <1>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.8: TS směrovací tabulky uzlu 4

Uzel 4 - NonTS směrovací tabulka

```
Id              <0>
Interface Id    <0>
Target          <0::3:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <1>
Interface Id    <1>
Target          <0::4:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF>
Gateway         <0::0>
Admin-Metric    <0>
Prot-Metric     <0>
Is Active       <1>
Hold Timer      <60>

Id              <2>
Interface Id    <0>
Target          <0::3:1>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::3:1>
Admin-Metric    <1>
Prot-Metric     <30>
Is Active       <1>
Hold Timer      <60>

Id              <3>
Interface Id    <1>
Target          <0::4:2>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:2>
Admin-Metric    <1>
Prot-Metric     <20>
Is Active       <1>
Hold Timer      <60>

Id              <4>
Interface Id    <0>
Target          <0::2:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:2>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>

Id              <5>
Interface Id    <1>
Target          <0::1:0>
NetMask         <FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFF:0>
Gateway         <0::4:2>
Admin-Metric    <2>
Prot-Metric     <40>
Is Active       <1>
Hold Timer      <60>
```

Obrázek 7.9: NonTS směrovací tabulky uzlu 4

7.1.2 Testovací topologie 2

Během testování se bohužel objevily nové technické potíže, které neumožňily spuštění vícera uzlů na jediném testovacím počítači. Tyto potíže se týkají klíčové části projektu, serializace dat, a tím pádem ovlivňují chod celého projektu. Problém je zapříčiněn současným neefektivním návrhem a implementací serializace dat, takže při spuštění vícera instancí dochází k přílišnému vytěžování procesoru testovacího počítače. Což způsobuje postupné zpomalování výměny datových kapsulí a ve finále vede až k havárii spuštěných uzlů.

Přitom testování probíhalo na dvojici rozdílných konfigurací počítačů. Oba testovací počítače využívají pro práci moderní 64-bitové čtyř jádrové procesory od různých výrobců, tj. AMD a Intel, a oba disponovali operační pamětí o velikosti 4GB.

V omezeném čase mi bohužel nedostatečná podpora sít'ové komunikace současné verze projektu nedovolila ani spustění testovací topologie přes vícero testovacích počítačů. Z těchto důvodu nejsou pro testovací *topologii2* (viz schéma 4.9) k dispozici žádné konkrétní výsledky.

Výskyt těchto problémů samozřejmě znemožnil i přímé otestování funkčnosti konceptu směrování cestou pomocí vytvořeného nástroje *TraceRoute*. Jediné z čeho je tedy v současnosti patrná veškerá funkčnost vytvořené implementace samotného směrovacího algoritmu *AntNet* včetně konceptu směrování horší cestou je výše uvedený obsah směrovacích tabulek dosažený na testovací *topologii1*.

KAPITOLA 8

Závěr

Detaily dosažených výsledků uvedených v kapitole 7 ukazují, i přes uvedené problémy během testování, úspěšné plnění příslušných směrovacích tabulek a prezentují tak úspěšnou implementaci modifikovaného algoritmu *AntNet*. Z jejich obsahu je již nyní patrné, že po odstranění stávajících problémů se serializací dat může uzel získanými směrovacími záznamy bez problémů směrovat datové toky citlivé na čas doručení lepší cestou a ostatní datové toky přesměrovávat na horší linky.

K opravě zmiňovaných problémů nemohlo dojít v důsledku nedostatku časových prostředků, jelikož lokalizace a odstranění jakéhokoli problému u takto rozsáhlého projektu jakým již v současnosti *SAN* je, je během na dlouhou trat'. Přesto se nemalé množství, v této práci neprezentovaných, prací odvedených na tomto projektu výrazně podepsalo na jeho celkové stabilizaci.

Samotný výsledek této práce pak do budoucna otevírá další možnosti k provádění nových experimentů v prostředí aktivních počítačových sítí. Což může v budoucnu vést ještě k případnému vylepšení samotného algoritmu, které bude spočívat v navržení nových metrik, které lépe popíší problematiku prioritizace kapsulí s ohledem na snížení pravděpodobnosti zahazování kapsulí na jinak plně využívanách linkách.

Cílem této práce tedy hlavně bylo implementovat směrovací algoritmus *AntNet* včetně jeho rozšíření pro směrování horší cestou do projektu *Smart Active Node*. K čemuž bylo nutné se seznámit jak s vlastním směrovacím algoritmem, tak i s principy aktivních sítí, konkrétné pak s projekty Smart Active Node a Grade32. Čímž jsem splnil všechny body zadání.

Přehled zkratek

SAN - Smart Active Node

SVN - Subversion, systém pro správu a verzování zdrojových kódů.

RPC - Remote Procedur Call, mechanismus volání vzdálených procedur.

TTL - Time To Live

RFC - Request for Comment, což je IETF dokument pro publikaci nových standardů

IETF - Internet Engineering Task Force

TS - Time Sensitive

NonTS - Non Time Sensitive

MSVC - Microsoft Visual Studio

Použitá terminologie

Smart Active Node - Název celého projektu, který lze pod tímto výrazem nebo zkratkou SAN dohledat na internetu.

AntNet - Název směrovacího algoritmu.

Subsít' - Podmnožina uzlů celé sítě, kterou lze z vnějšího pohledu adresovat stejnou sít'ovou adresou.

SAN adresa - Datový typ, skládající se ze čtveřice 64-bitových čísel. Používaný k identifikaci sít'ových uzlů v projektu SAN a jako součást směrovacích záznamů.

Kapsule - Zpráva přenášená aktivní sítí, obdoba paketu z pasivních sítí.

Time Sensitive - Operace doručení zprávy je citlivá na doručovací čas. Musí být doručena co nejrychleji, proto je pro ní vybírána nejlepší cesta.

Non Time Sensitive - Operace doručení zprávy není citlivá na doručovací čas. Může doručena se zpožděním a horší cestou.

Default Gateway - Výchozí brána, představující výchozí směr k dalšímu uzlu, kterým se má předat zpráva, pro jejíž cílový uzel není ve směrovací tabulce specifický záznam.

HopCount - Obdoba parametru TTL (Time To Live), používaného v klasických sítích. Udává délku života kapsule, resp. maximální počet přeskoků mezi uzly, jenž může kapsule absolvovat. Je to jedno z bezpečnostních opatření sítě proti jejímu zahlcení.

Stogare - Datové uložiště SAN Serveru pro data aktivních aplikací. Je uloženo lokálně pro každou instanci.

Programovatelná sít' - Počítačová sít' řízená programovým kódem, jehož úpravami lze modifikovat její chování.

Výchozí uzel - Uzel na němž běží proces směrovacího algoritmu a ze kterého byla konkrétní kapsule odeslána.

Přímý sousední uzel - Uzel ve vzdálenosti $HopCount = 1$ od původního uzlu.

Mezilehlý uzel - Každý uzel nacházející se mezi zdrojovým a cílovým uzlem.

Konečný cílový uzel - Uzel jemuž je adresována kapsule.

Seznam obrázků

2.1 Schéma běhu aktivní aplikace (Java verze) 6
2.2 Schéma běhu aktivní aplikace (C++ verze) 7

3.1 Běžící proces směrovacího algoritmu 11
3.2 Modelový příklad vyslané kapsule z uzlu 1 na uzel 2 12
3.3 Příklad možného průchodu kapsule sítí 14
3.4 Příklad průchodu smyčkou při doručování kapsule 15
3.5 Ukázka nastavení kapsule při návratu 16
3.6 Zbytečný směrovací záznam pro uzel 1. 17
3.7 Zápis směrovacích záznamů na konečném cílovém uzlu. 18
3.8 Zápis směrovacích záznamů na mezilehlých uzlech. 19

4.1 Ukázka různých cest mezi uzly 1 a 4 23
4.2 Pouze nejvýhodnější cesta mezi uzly 1 a 3 23
4.3 Ukázka datových proudů mezi uzly 1 a 3 24
4.4 Ukázka dvojice dat. proudů A a B u základní verze směrování 24
4.5 Ukázka dvojice dat. proudů A a B u směrování horší cestou . 25
4.6 Ukázka smyčky výchozích bran 27
4.7 Změna doručovacího směru kapsule 27
4.8 Schéma a nastavení testovací topologie 1 29
4.9 Schéma a nastavení testovací topologie 2 30
4.10 Ukázka příkazu na výpis směrovací tabulky uzlu 32
4.11 Výpis seznamu a stavu sít'ových rozhraní 32
4.12 Ukázka deaktivovaného směrovacího záznamu 33

5.1 Schéma struktury kapsule 42

6.1 Ukázka spuštěného uzlu 56

7.1 Schéma a nastavení testovací topologie 1 60
7.2 TS směrovací tabulka uzlu 1 61
7.3 NonTS směrovací tabulky uzlu 1 62

7.4 TS směrovací tabulka uzlu 2 63
7.5 NonTS směrovací tabulky uzlu 2 64
7.6 TS směrovací tabulky uzlu 3 65
7.7 NonTS směrovací tabulky uzlu 3 66
7.8 TS směrovací tabulky uzlu 4 67
7.9 NonTS směrovací tabulky uzlu 4 68

Literatura

[**1**] : KOUTNÝ, Tomáš. Smart Active Node [online]. 15. Května 2012. Dostupné z: <*http://www.san.zcu.cz*>

[**2**] : KOUTNÝ, Tomáš. SVN uložiště zdrojových kódů projektu. 15.Května 2012. Dostupné z: <*https://smartactivenode.svn.sourceforge.net/svnroot/smartactivenode*>

[**3**] : MALKIN, G. RIP Version 2. RFC-2453, [online]. 15. Května 2012. Dostupné z: <*http://tools.ietf.org/rfc/rfc2453.txt*>

[**4**] : PEPELNJAK, Ivan. EIGRP Network Design Solutions, vydáno 25. Října 1999, Cisco Press. ISBN-10: 1-58705-895-2.

[**5**] : MOY, J. Ospf version 2. RFC-2328, [online]. 15. Května 2012. Dostupné z: <*http://www.ietf.org/rfc/rfc2328.txt*>

[**6**] : ORAN, D. OSI IS-IS Intra-domain Routing Protocol. RFC-1142, [online]. 15. Května 2012. Dostupné z: <*http://tools.ietf.org/html/rfc1142*>

[**7**] : REKHTER, Y. A Border Gateway Protocol 4 (BGP-4). RFC-1771, [online]. 15. Května 2012. Dostupné z: <*http://www.ietf.org/rfc/rfc1771.txt*>

[**8**] : Vacek Zdeněk. Nalezení topologie aktivní počítačové sítě. Plzeň 2010. Bakalářská práce. Západočeská univerzita v Plzni. Vedoucí práce Ing. Tomáš Koutný, Ph.D.

[**9**] : KOUTNÝ Tomáš, SÝKORA Jakub, "Lessons Learned on Enhancing Performance of Networking Applications by IP Tunneling through Active Networks", International Journal on Advances in Internet Technology, Volume 3, Numbers 3 4, 2010, pp. 233-244, ISSN: 1942-2652

Přílohy

A) Příklad konfigurace SAN_Farmer

Legenda:
1) Meziprocesová komunikace RPC (nutno nastavit i v worker.cfg a worker_AntNet.cfg)
2) Sít'ová rozhraní uzlu (může zde být i více záznamů)
3) Definice výchozích bran (pro každý typ kapsule může být pouze jedna)
4) Směrovací záznamy na loopback a přímé sousední uzly

- [**definition**]
- #IP_adresa_pro_RPC,_muze_byt_stejne_s_libovolnou_IP_uzlu
- IP = 10.10.115.33
- #IP uzlu 1
- IP1 = 192.168.1.1
- #IP uzlu 2
- IP2 = 192.168.1.2
- IP3 = 192.168.1.3
- #IP uzlu 3
- IP4 = 192.168.1.4
-
- [**paths**]
- log = log$_f$$armer1.txt$
− *java_sdk = jdk/*
− *compile_temp = temp/*
− *make_path =*
− *repository = repository/*
−
− [**init**]
− *active_codes[] = route*

– **[rpc]**
– san_protocol = tcp
– san_description = rpc
– tcp_address = $IP
– tcp_incoming_port = 64001
–
– #interface_k_uzlu_2_(interface_id = 0)
– **[network/interface[]]**
– san_address = 0000000000000000 – 0000000000000000 –
– 0000000000000001 – 0000000000000001
– san_network_mask = FFFFFFFFFFFFFFFF –
– FFFFFFFFFFFFFFFF –
– FFFFFFFFFFFFFFFF –
– 0000000000000000
– san_protocol = tcp
– san_description = network
– tcp_address = $IP1
– tcp_incoming_port = 50000
– tcp_outgoing_port = 50000
–
– #IP_resolver_(prozatimni_reseni)
– **[network/resolver[]]**
– san_address = 0000000000000000 – 0000000000000000 –
– 0000000000000001 – 0000000000000001
– tcp_address = $IP1
–
– **[network/resolver[]]**
– san_address = 0000000000000000 – 0000000000000000 –
– 0000000000000001 – 0000000000000002
– tcp_address = $IP2
–
– **[network/resolver[]]**
– san_address = 0000000000000000 – 0000000000000000 –
– 0000000000000002 – 0000000000000001
– tcp_address = $IP3
–
– **[network/resolver[]]**
– san_address = 0000000000000000 – 0000000000000000 –
– 0000000000000002 – 0000000000000002
– tcp_address = $IP4
–
– #loopback_interface_0
– **[network/route[]]**
– interface_id = 0
– gateway = 0000000000000000 – 0000000000000000 –

– $0000000000000000 - 0000000000000000$
– $address = 0000000000000000 - 0000000000000000 -$
– $0000000000000001 - 0000000000000001$
– $network_mask = FFFFFFFFFFFFFFFF -$
– $- FFFFFFFFFFFFFFFF -$
– $- FFFFFFFFFFFFFFFF -$
– $- FFFFFFFFFFFFFFFF$
– $administrative_metric = 20$
– $protocol_metric = 20$
– $active = true$
– $\#hold_timerjevsekundach$
– $hold_timer = 60$
–
– $\#primy_sousedni_uzel_2$
– [**network/route**[]]
– $interface_id = 0$
– $gateway = 0000000000000000 - 0000000000000000 -$
– $0000000000000001 - 0000000000000002$
– $address = 0000000000000000 - 0000000000000000 -$
– $0000000000000001 - 0000000000000002$
– $network_mask = FFFFFFFFFFFFFFFF -$
– $- FFFFFFFFFFFFFFFF -$
– $- FFFFFFFFFFFFFFFF -$
– $- 0000000000000000$
– $administrative_metric = 20$
– $protocol_metric = 20$
– $active = true$
– $\#hold_timerjevsekundach$
– $hold_timer = 60$
–
– $\#default_gateway_(definice_vychozi_brany_pro_TS_kapsule)$
– [**network/defaultGateway**[]]
– $interface_id = 0$
– $capsule_type = TS$
– $\#default_gateway_(definice_vychozi_brany_pro_NonTS_kapsule)$
– [**network/defaultGateway**[]]
– $interface_id = 0$
– $capsule_type = NonTS$

B) Příklad konfigurace SAN_Worker

- [**paths**]
- log = $\log_c lient1_worker.txt$
$- repository = repository/$

$-$ [**init**]
$- active_codes[] = route$
$- log[] = AntNet$

$-$ [**rpc**]
$- san_username = unknown$
$- san_password = unknown$
$- san_protocol = tcp$
$- san_description = rpc$
$- tcp_address = 10.10.115.33$
$- tcp_outgoing_port = 64001$
$- tcp_incoming_port = 63001$