VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

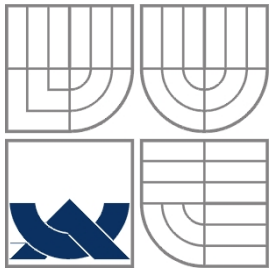
DEPARTMENT OF TELECOMMUNICATIONS

TVORBA MODELŮ PROTOKOLŮ V SIMULÁTORU OMNET++

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

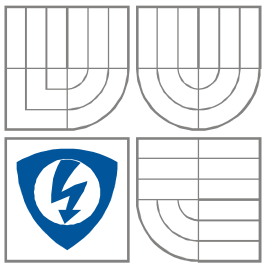
AUTOR PRÁCE Bc. Jan Vohralík
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

TVORBA MODELŮ PROTOKOLŮ V SIMULÁTORU OMNET++

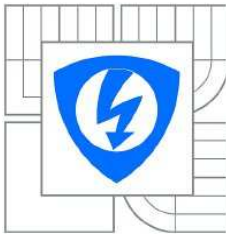
PROTOKOL MODEL DESIGN IN OMNET++ SIMULATOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Jan Vohralík
AUTHOR

VEDOUCÍ PRÁCE doc. Ing. Vít Novotný, Ph.D.
SUPERVISOR

BRNO, 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor

Telekomunikační a informační technika

Student: Bc. Jan Vohralík

ID: 77923

Ročník: 2

Akademický rok: 2009/2010

NÁZEV TÉMATU:

TVORBA MODELŮ PROTOKOLŮ V SIMULÁTORU OMNET++

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte problematiku modelování síťových komponentů a síťových protokolů. Seznamte se se simulačním prostředím OMNET++ a s modely síťových prvků a protokolů v tomto prostředí. Prostudujte způsob modifikace či tvorby modelů. Navrhněte a vytvořte či modifikujte modely různé úrovně abstrakce pro vybrané kabelové či bezdrátové prostředí. Navrhněte laboratorní úlohu.

DOPORUČENÁ LITERATURA:

[1] BANNISTER, Jeffrey, et al. Convergence Technologies for 3G Networks IP, UMTS, EGPRS and ATM. [s.l.] : John Wiley & Sons Ltd, 2004. 650 s. ISBN 0-470-86091-X.

[2] VARGA, András. OMNeT++ User Manual verze 4 [online]. 2009, URL: <http://omnetpp.org/doc/omnetpp40/manual/usman.html>.

Termín zadání: 29.1.2010 ***Termín odevzdání:*** 26.5.2010

Vedoucí projektu: doc. Ing. Vít Novotný Ph.D.

prof. Ing. Karel Vrba, CSc.

předseda oborové rady

ABSTRAKT

Cílem práce je seznámit se se simulačním nástrojem OMNeT++ a s jeho rošířením INET framework a prozkoumat možné řešení realizace rošíření balíku o dosud chybějící protokol RIP.

OMNeT++ je diskrétní simulační prostředí. Zdrojový kód je k dispozici zdarma pro výukové a výzkumné účely. Nejčastěji je používán pro simulaci komunikačních sítí, ale vzhledem k jeho obecné a flexibilní architektuře, může být úspěšně využíván i v jiných oblastech, jako je simulace komplexních IT systémů, v obslužných systémech nebo při simulacích hardware architektury.

Úvodní část práce se zabývá teoretickým popisem RIP protokolu a koncepcí simulačního prostředí. Návrh, implementace a diskuze jednotlivých vytvořených modulů a komponent je uvedeno v druhé části práce. Posledním úkolem bylo vytvořit laboratorní úlohu na základě získaných zkušeností. Implementační programovací jazyk je C++.

Klíčová slova: simulace, návr, C++, OMNeT++, INET framework, RIP

ABSTRACT

The aim of this Thesis is giving information about the network simulating system OMNeT++ and its extension INET framework and explore possible solution to implement missing protocol RIP.

OMNeT++ is a discrete-event simulation environment. Source code is available and free for teaching and research purposes. Its primary application area is the simulation of communication networks, but because of its generic and flexible architecture, is successfully used in other areas like the simulation of complex IT systems, queueing networks or hardware architectures as well.

The first part of Thesis decribes RIP protocol operations and conception of OMNeT++. At the second part of Thesis the implementation design and discussion of created modules and component is presented here. The final part of the Thesis was creating the laboratory problem on the basis experience gained. Implementation is realized in C++ programming language.

Keywords: simulation, design, C++, OMNeT++, INET framework, RIP

VOHRALÍK, J. *Tvorba modelů protokolů v simulátoru OMNeT++*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2010. 72 s. Vedoucí diplomové práce doc. Ing. Vít Novotný, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že jsem svou diplomovou práci na téma Tvorba modulů protokolů v simulátoru OMNET++ vypracoval samostatně pod vedením vedoucího diplomové práce, s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

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

V Brně dne 26. 5. 2010 Podpis autora ……

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce doc. Ing. Vítu Novotnému, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce. Dále bych chtěl poděkovat mé rodině za podporu a trpělivost, kterou se mnou měli v období psaní této práce.

V Brně dne 26.5.2010 Podpis autora ……

OBSAH

SEZNAM OBRÁZKŮ

Laboratorní úloha

SEZNAM TABULEK

ÚVOD

S trvalým rozvojem komunikačních sítí rostou nároky na jejich kvalitu, proto vznikají simulační programy pro testování různých přístupů a následné zvolení nejvhodnější technologie a parametrů telekomunikační sítě.

Mezi nejznámější patří komerční řešení OPNET modeler od společnosti OPNET Technologies. V nekomerční sféře existují například NS2 simulátor, Jasper, OMNeT++ a mnoho dalších, mnohdy již neaktualizovaných projektů. Mým úkolem je prozkoumat možnosti simulátoru OMNeT++.

OMNeT++ je diskrétní simulační systém, napsaný v C++. Prostředí je univerzální a může sloužit k simulaci rozličných problémů. Simulační modely jsou složeny z hierarchicky uspořádaných bloků. Komunikace je zajištěna pomocí přímých C++ funkcí a zpráv, které si mezi sebou moduly vyměňují. Struktura je popsána vlastním jazykem pro popis topologie funkce jednotlivých modulů potom v C++. Výhodou je licenční politika, pod kterou je OMNeT++ poskytován, jedná se o GNU. Díky tomu se simulátor těší oblibě, zejména ve výzkumném a akademickém prostředí.

Diplomová práce se zabývá studiem vlastností modelů simulačního prostředí OMNeT++ verze 4.0 a jeho síťové nadstavby INET framework. Cílem práce je implementace dosud chybějícího protokolu RIP a jeho nasazení v experimentální síti.

Práce je rozdělena na část teoretickou a praktickou. V úvodní části bude představena koncepce simulačního prostředí OMNeT++. Bude popsán způsob popisu struktury modelů a zařízení, k tomu je použit specifický jazyk Network description(NED). Poté budou podrobně popsány kroky nutné k implementaci vlastních modulů a zpráv. Druhá část se bude zabývat rozšiřujícím balíkem INET framework. Toto rozšíření přináší možnost simulovat celou škálu síťových prvků a protokolů. Ve třetí části bude popsán protokol RIP. Budou popsány základní vlastnosti, dostupné verze a formáty příslušných zpráv. Důraz bude kladen na prezentaci směrovacího algoritmu, jeho stabilitu a návaznost na směrovací tabulky. Implementace je rozdělena do několika fází. Nejprve bude vytvořen model směrovače, který bude složen z již existujících modulů nižších vrstev a uplně nového modulu, kde bude definováná funkcionalita RIP protokolu.

V závěru práce budou prezentovány výsledky formou laboratorní úlohy. Bude vytvořena experimentální datová síť, kde bude implementovaný směrovací protokol nasazen a otestován.

1 OMNeT++

OMNeT++ je diskrétní simulační prostředí pro modelování komunikačních sítí, multiprocesorových architektur a dalších distribuovaných, či paralelních systémů. Je naprogramován v jazyce C++ a distribuován pod volnou licencí pro akademické a nekomerční účely. Komerční varianta nese jméno OMNEST. Simulační prostředí je multiplatformní a běží tedy, jak na systémech unixového typu, tak pod Windows (verze 2000 a novější). Autorem projektu je András Varga, který na něm začal pracovat v roce 1992 na technické univerzitě v Budapešti.[10] Jeho cílem bylo vytvořit otevřený diskrétní událostní simulátor, který by mohl být alternativou pro již existující open - source nástroje (Ns simulátor), či komerční řešení (Opnet modeler). Díky modulární architektuře a rozsáhlé podpoře grafického rozhraní je možné systém snadno dále rozšiřovat o nové funkcionality nebo implementovat do dalších aplikací. Od svého založení v roce 1997 se komunita uživatelů a přispěvovatelů stále rozrůstá a je soustředěna kolem oficiálních webových stránek. [5]

Hlavní požadavky na systém:

- možnost rozsáhlých simulací,
- modulárnost, možnost použití vestavěných simulačních modelů,
- rozsáhlá podpora debug možností.

1.1 Diskrétní simulace

Simulátor pracuje v diskrétním čase, jednotlivé změny stavu jsou plánovány jako události a provádí se v nespojitých instancích času, doba vykonání události je nulová. Každá událost má přidělenu časovou značku, která odpovídá času výskytu události v systému. Předpokládá se, že mezi dvěma sousedními událostmi nedochází k žádným změnám v systému. Během simulace jsou jednotlivé události vybírány ze seznamu událostí a zpracovávány. Simulační doba je, na rozdíl od strojového času, který závisí na výkonu použitého hardwaru, nulová.

Jako důsledek zpracování se mění stav systému a jsou generovány nové události. Jádro prostředí je vytvořeno v jazyce C++, jehož architektura je modulární pro možnost snadného budoucího rozšiřování. Grafické uživatelské rozhraní je vytvořeno pomocí

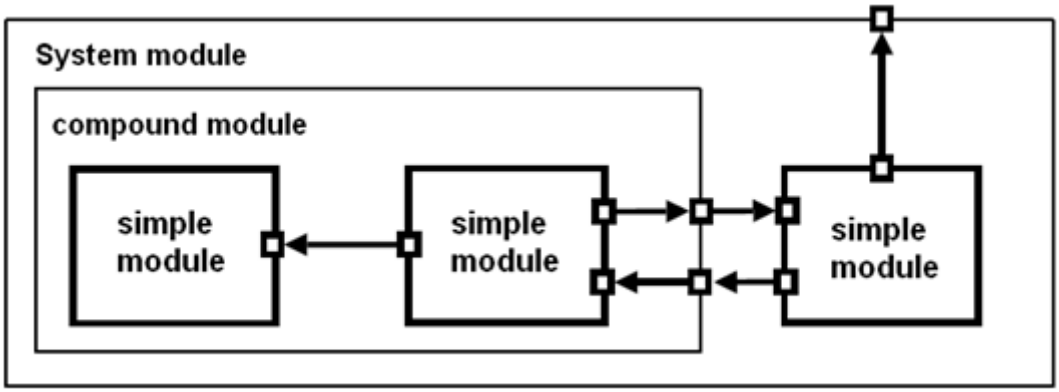
skriptovacího jazyka Tcl/Tk, který umožňuje snadné propojení s prostředím simulátoru napsanému v jazyce C++. Pro popis modulů a jejich vzájemného propojení je použit vlastní jazyk NED.

1.1.1 Struktura modelu

Simulační model se skládá z hierarchicky vnořených modulů, které spolu komunikují pomocí předávání zpráv. Struktura systému je sestavena z několika typů modulů a je popsána v jazyce NED. Moduly obsahující další podmoduly, které se nazývají složené moduly (Compound modules). Na nejnižší úrovni hiearchie se nachází jednoduché moduly (simple modules), tyto moduly obsahují algoritmy v C++ kódu popisující jejich chovaní. Hloubka vnoření není omezena.

1.1.2 Typy modulů

Oba výše zmíněné typy modulů jsou instancemi tzv. modulových typů (module types), což jsou uživatelem definované moduly popisující daný model. Dle [11] může instance těchto modulových typů uživatel dále skládat do složitějších struktur a vytvářet tak mnohem přesnější a komplexnější řešení. Na vrcholu hiearchie je systémový modul (system module) jako instance předchozího definovaného typu modulu. Všechny moduly daného modelu jsou instancovány jako submoduly a sub - submoduly systémového modulu, jak můžeme vidět na obrázku 1.



Obr. 1: Vztahy mezi moduly

Při implementaci vlastní simulace, kdy používáme moduly jako jednotlivé stavební bloky modelu, není již potřeba rozlišovat, zda se jedná o jednoduchý nebo složený modul.

Funkce jednoduchého modulu je možné rozdělit do několika složených modulů a naopak implementovat funkci složeného modulu v rámci modulu jednoduchého bez vlivu na ostatní uživatele.

1.1.3 Spojování modulů a zasílání zpráv

Každý modul může obsahovat libovolné množství vstupních a výstupních bran (gates), které jsou mezi sebou propojeny spoji (connections). Brány jsou vstupně/výstupní rozhraní modulů. Zprávy jsou odesílány z výstupních bran a jsou přijímány na vstupních branách na opačném konci spojení. Spojení může být vytvářeno pouze mezi branami na stejné úrovni hiearchického modelu. V rámci složeného modulu je možné sestavit spoj mezi jednotlivými jednoduchými moduly, nebo mezi branami jednoduchého modulu a složeného modulu (parent module), jak je názorně vidět na obrázku 1 dle [11].

Vzhledem k hiearchické struktuře modelů prochází obvykle zpráva přes nekolik spojů, které začínají i končí v jednoduchých modulech. Tuto skupinu spojení nazýváme cestou (route). Jednotlivým spojům v síti můžeme nadefinovat následující vlastnosti:

- zpoždění linky,
- chybovost linky,
- rychlost linky.

Zpoždění linky je doba, za kterou zpráva projde od výstupní brány prvního modulu ke vstupní bráně cílového modulu. Chybovost linky udává pravděpodobnost chybného přenosu zprávy a umožňuje jednoduché modelování rušení. Rychlost linky je udávána v bitech za sekundu a používá se pro výpočet doby přenosu paketu linkou.

Zasílání zpráv v simulačním prostředí OMNeT++ probíhá, buď přímo ze zdojového do cílového modulu, a nebo přes připravené cesty. Kromě základních informací nutných pro manipulaci s objektem zprávy může zpráva nést i libovolný typ datových struktur (například rámec, nebo paket).

1.2 Jazyk NED

Podle [11] jazyku NED umožňuje modulární popis sítě. Uživatel definuje strukturu sítě z jednotlivých součastí. Vkládá a zanořuje jednotlivé moduly, definuje brány a vytváří jednotlivá spojení. Jazyk NED je case sensitive, a proto je třeba dát pozor na velká a malá písmena identifikátorů. Vlastnosti popsané v jazyku NED se ukládají do souborů

s příponou .ned. S novou verzí OMNeTu 4.0 se mění i syntaxe jazyka NED a dochází k rozšíření o nové prvky. Příkladem mohou být dvoucestné brány.

1.2.1 Jednoduchý modul

Pro deklaraci jednoduchého modulu se použije klíčové slovo simple a složené závorky označující začátek a konec funkčního bloku. Mezi ně se umístí seznam jeho parametrů a bran. Příklad deklarace jednoduchého modulu může být následující:

```
simple Module_name      //jméno modulu
{    parameters:  //výčet parametru
        int capacity;
        @display("i=block/thing");
     gates:
        input in; //vstupní brány
        output out;     //výstupní brány
      inout gate; //vstupní/výstupní
}
```

Jak je vidět v předchozí deklaraci jednoduchého modulu naprosto chybí jakékoli definice operací. Toto OMNeT++ zajišťuje defaultním voláním C++ třídy stejného názvu jako je v NED definici modulu, zde Module_name.

1.2.2 Složený modul

Složený modul je uvozen klíčovým slovem module a může být složen z více jednoduchých i složených modulů. Pomocí klíčového slova import lze vkládat celé .ned soubory a využít moduly v nich definované. Složené moduly mohou mít definovány brány a parametry jako moduly jednoduché, ale není jim přiřazeno žádné aktivní chování, žádný C++ kód. Pro deklaraci vnořených modulů se používá klíčové slovo submodules. Složené moduly mohou být dále rozšiřovány pomocí dědění vlastností z externích modulů. Pomocí dědění může složený modul získat nové submoduly, spojení, parametry a brány. Pro definici složených modulů se používá tato syntaxe:

```
module Host             //název modulu
{
     parameters:        //výčet parametru
        ...
   gates:               //výčet bran
        ...
   submodules:          //definice vnořených modulů
        ...
```

```
    connections:        //spojení v rámci modulu
        ...
}
module WirelessUser extends WirelessHostBase
{
    submodules:
        ...
    connections:
        ...
}
```

1.2.3 Komunikační kanál

Kanály definují parametry a chovaní spojení. Jsou to v podstatě jednoduché moduly, protože mají přiřazen C++. Komunikační kanály mohou mít přiřazeny následující parametry: zpoždění, chybovost a přenosová rychlost. Lze je přiřadit konkretnímu spoji, nebo vytvořit kanál a ten použít na více místech simulačního modelu. Syntaxe kanálu je následující:

```
channel C extends ned.DatarateChannel
{
    datarate = 100Mbps;
    delay = 100us;
    ber = 1e-10;
}
```

1.2.4 Simulační model

Simulační model (Network) tvoří vrchol hiearchie simulačního stromu. Klíčovým slovem pro deklaraci je network. Modul obsahuje definici uzlů a jejich vzájemného propojení pomocí bran a portu. Syntaxe je následující:

```
network Network         //název sítě
{
    submodules:         //uzly sítě
        node1: Node;
        node2: Node;
        node3: Node;
        ...
    connections:        //definice spojeni, zde duplexní
                        //linka s rychlostí 100Mbps
node1.port++ <--> {datarate=100Mbps;} <--> node2.port++;
node2.port++ <--> {datarate=100Mbps;} <--> node4.port++;
node4.port++ <--> {datarate=100Mbps;} <--> node6.port++;
        ...
}
```

1.3 Tvorba modulů a zpráv

1.3.1 Implementace jednoduchých modulů

Jednoduché moduly zajišťují veškerou aktivitu simulačního modelu. Jsou naprogramovány v jazyce C++ za pomoci simulační knihovny OMNeT++. Jednoduché moduly zapouzdřují C++ kód, který generuje a reaguje na události, jinými slovy implementuje chování modulu.

Nový jednoduchý modul vznikne jako potomek třídy cSimpleModule, která je stejně jako cCompoundModule odvozena od třídy cModule, [7]. Tato třída obsahuje virtuální metody, které může uživatel předefinovat či doplnit a přizpůsobit tak chování modulu svým potřebám. Základní metody jsou:

- `virtual void initialize()`,
- `virtual void handleMessage(cMessage *msg)`,
- `virtual void activity()`,
- `virtual void finish()`.

Metoda `void initialize()` vytváří objekty a propojuje je dle NED definic, je volána pro všechny objekty vytvářené objekty a zajišťuje tak načtení parametru a nastavení jednotlivých vnitřních promněnných.

`void handleMessage(cMessage *msg)` definuje chovaní modulu. Metoda je volána při přijetí nové zprávy a slouží ke zpacování událostí.

`virtual void activity()` také zpracová přijaté zprávy jako předešlá metoda, ale je založena na principu vzájemné interakce nepreemtivních vláken. V každém modulu může být provozována pouze jedna z těchto dvou metod.

Metoda `virtual void finish()`slouží k uložení dat získaných při simulaci a je volána po úspěšném dokončení simulace.

Každá nová třída musí být zaregistrována v simulačním jádru OMNeT++ pomocí makra `Define_Module()`. To musí být umístěno v souborech s koncovkou .cc, nebo .cpp, protože kompilátor z něj generuje kód. Pro korektní simulaci složitějších procesů je vhodné implementovat další metody, protože samotná třída cSimpleModule zajištuje jen nezbytné funkčí minimum.

1.3.2 Vytvoření zprávy

Nové zpravy je v OMNeT ++ možné vytvořit jako potomky základní třídy cMessage, nebo některé její podtřídy. Pomocí zpráv můžeme modelovat události, rámce, pakety, buňky včetně vysílaní, přijimání a vlastní cesty signálu sítí.

Při tvorbě nové zprávy musí být podle [12] nastaveny následující atributy centrální třídy cMessage:

- name,
- message kind,
- length,
- bit error flag,
- priority,
- time stamp,
- read only atributy.

Name (jméno) je atribut sloužící pro popis vytvářené zprávy, vyskytuje se především v grafickém prostředí, a proto je vhodné používat přesné vystihující jména. Atribut není přímo součastí cMessage, ale je poděděn z kořenové nadtřídy cOwnedObject. Messge kind je celočíslený atribut nesoucí informace o typu zprávy. Kladná čísla jsou volně dostupná programátorovi, záporná jsou vyhrazena pro vnitřní knihovnu OMNeT++. Atribut length se používá pro výpočet přenosového zpoždění při přenosu přes spojení s nastavenou hodnotou přenosové rychlosti. Bit error flag je nastaven simulačním jádrem na hodnotu *true* s pravděpodobností $1-(1-ber)^{length}$, při přenosu přes spojení s nastavenou hodnotou chybovosti. Atribut prioriry používá simulační jádro při řazení zpráv do front, za předpokladu stejného arival time(parametr výskytu události). Časový atribut time stamp nevyužívá jádro, ale slouží programátorovi pro individualní užití. Ostaní atributy třídy cMessage, které není možné modifikovat, slouží simulačnímu jádru k ukládání transportních informací.

Pro vytvoření nové zprávy použijeme konstruktor třídy cMessage:

```
cMessage *msg = new cMessage("MessageName", msgKind);
```

K takto vytvořené zprávě nyní můžeme přidat i ostatní implicitní atributy pomocí metody `set()`:

```
msg->setBitLength(length);
```

Obdobně i ostatní atributy je možné nastavit i přímo v konstruktoru, ale pro přehlednost se tato varianta nedoporučuje.

Pro přidání vlastních atribut musíme vytvořit vlastní třídu, kterou podědíme od cMessage. Pro každý nový atributu je potřeba vytvořit metody pro uložení a vyzvednutí hodnoty a privátní datové úložiště. Tento nový kód musíme následně integrovat do frameworku pomocí dalších nezbytných funkcí. Protože to může být dosti zdlouhavé, nabízí OMNeT++ mnohem příjemnější cestu, a to definice zpráv (Message definitions). Jedná se o velmi kompaktní syntaxi pro popis jednotlivých atributů. Překladač automaticky vygeneruje z definice zpráv požadovaný C++ kód.

Pro ilustraci uvedu jednoduchý příklad, zprávu nesoucí pouze zdrojovou a cílovou adresu.

```
message zprava
{
     int zdrojovaArdresa;
     int cilovaAdresa;
};
```

Po zpracování překladačem vzniknou dva soubory: zprava_m.h a zprava_m.cc.

Hlavičkový soubor zprava_m.h obsahuje deklaraci zprávy a musí být obsažen ve všech zdrojových souborech, kde má být naše zpráva použita. Obsah souboru je nasledující:

```
class zprava :public cMessage {

    virtual int getZdrojovaArdresa() const;

    virtual void setCilovaAdresa (int cilovaAdresa);

};
```

Soubor *zprava_m.cc* obsahuje generovaný kód pro přístup k našim datům z grafického prostředí Tkenv. Proto je potřeba ho kompilovat a připojit k naší simulaci. [12]

V případě, že potřebujeme implementovat nějakou funkci, či datový typ, který překladač přímo nepodporuje, můžeme generované soubory editovat a požadované metody doprogramovat.

2 INET Framework

INET Framework je sada síťových simulačních modelů pro simulační prostředí OMNeT++. Sada vychází z projektu IPSuite vytvořeného na Univerzitě v Karlsruhe během let 2000-2001. V roce 2003 převzal vývoj András Varga, pod jehož dohledem byly modely zrevidovány, zdokumentovány a rozšířeny o další. Sada obsahuje modely protokolů TCP/IP vrstvy, například IPv4, IPv6, SCTP, PPP, IEEE 802.11, TCP, UDP a několik aplikačních protokolů jako jsou Telnet nebo FTP. V roce 2005 byl framework rozšířen o modely pro simulaci mobilní a bezdrátové komunikace vycházející z Mobility Frameworku. Vzhledem k otevřenosti celého systému vznikají nové moduly prakticky neustále. Ke vzájemné komunikaci složí především komunitní web.[6]

Pro užívání modelů je nutné framework přilinkovat k simulátoru OMNeT++. Výsledkem kompilace frameworku je binární program, na kterém je založen celý princip používaní. Program vynikne kompilací všech jednoduchých modulů. Tento program při spustění načte konfigurační soubor omnetpp.ini z aktuálního adresáře. Obsahuje vlastní nastavení sítě a odkaz na .ned soubor, který obsahuje definici simulované sítě a ten zase odkazy na ned definice modulů, ze kterých je daná síť sestavena. Výsledný binární soubor muže být spustěn i na počítačích bez řádné instalace OMNeT++.

Simulaci lze spustit v grafickém prostředí, které se dle [6] skladá z hlavního okna, do kterého jsou postupně vypisovány informace o zpracování událostí a z aktivních inspektorů různých úrovní zapouzdření zobrazující animaci právě zpracovávané události. Simulace korektně končí vypršením nastaveného časového limitu nebo vykonáním všech naplánovaných událostí. Restart simulace je možný volbou „Rebuid network“ z menu „Simulation“.

2.1 Architektura INET Frameworku

Princip simulování provozu v sítí je obdobný jako u obecného OMNet++, moduly reprezentující síťová zařízení nebo jejich části si mezi sebou zasílají zprávy, které mohou simulovat rámce, pakety, datagramy etc., podle simulovaného protokolu. Protokoly jsou reprezentovány jednoduchými moduly, jejichž vnější rozhraní je definováno v jazyku NED a jejich funkční část je implementována v C++ třídě stejného jména. Tyto základní bloky lze libovolně kombinovat a skládat do větších celků, které reprezentují síťová zařízení (Router, Switch, StandardHost) pouze pomocí jazyka NED bez nutnosti dalšího překladu.

Hlavičky protokolů reprezentují zprávy, soubory .msg, ze kterých jsou automaticky generovány odpovídající C++ třídy, potomci třídz cMessage. Pokud protokol vyšší vrstvy chce odeslat PDU (protokol data unit), vygeneruje odpovídající zprávu, kterou zašle modulu nižší vrstvy daného protokolu. V této vrstvě se zpráva zapouzdří do jiné, nově vytvořené, která reprezentuje PDU této nižší vrstvy a je opět odeslána nižší vrstvě nebo partnerskému modulu. Při přijetí zprávy je proces opačný. V sadě INET existují mimo modulů protokolu také jednoúčelové moduly, zajišťující některé globální nastavení, například prvek pro automatické nastavení IP adres všem prvkům sítě a statických směrovacích tabulek (FlatNetworkConfigurator), nebo modul ChannelControl, který obsahuje informace o poloze bezdrátových prvků v síti.

2.2 Základní moduly pro routery a stanice

Součástí převážné většiny zařízení v síti jsou následující bloky:

- Notification Board,
- InterfaceTable,
- RoutingTable.

2.2.1 NotificationBoard

Jedná se o jednoduchý modul sloužící k výměně informací jako jsou změna routovací tabulky, změna handoveru, stav bezdrátového kanálu, nebo stavu rozhraní (up,down). Jednotlivé instance mezi sebou komunikují přímým volaním C++ zpráv, nikoli posílaním zpráv, jak je popsáno v [1]. V každém složeném modulu může být pouze jedna instance NotificationBoard.

2.2.2 Interface Table

Tento jednoduchý modul obsahuje tabulku síťových rozhraní (eth0, ppp0, wlan0 etc) ve stanicích a směrovačích. Modul smí mít nanejvýš jednu instanci v routeru nebo hostu. Protože neobsahuje žadné brány, je zapotřebí pro aktualizaci tabulky použít standardních volání C++ funkcí. Toto se provede standarně za pomoci knihovny OMNet++ (`cModule::submodule()`), která vrátí ukazatel na objekt jejího modulu (cModule*) a ten pak dynamicky přetypují na ukazatel na objekt tabulky (InterfaceTable*), což umožní volat veřejné funkce této třídy. Jednotlivé záznamy jsou objekty C++ třídy InterfaceEntry a jsou uloženy v STL kontejneru `std::vector < >`.

2.2.3 RoutingTable

V tomto modulu je uložena směrovací tabulka, která je obsažena ve všech IP modulech (hosts, routers). Jedná se o jednoduchý modul bez bran, a proto je potřeba knihovních funkcí pro změnu směrovacích záznamů. Obvykle je směrovací tabulka volána modulem IP, který používa funkce pro dotazovaní, přidávání a odebíraní směrovacích záznamů. Přistupovat sem mohou také směrovací protokoly (OSPF, RIP) a mohou tak dynamicky provádět nastavení směrovaní v síti.

2.3 Základní moduly síťové vrstvy

Pro síťovou vrstvu jsou typické tyto moduly a to pouze v jedné instanci:

- FlatNetworkConfigurator
- ScenarioManager
- ChanelControl

2.3.1 FlatNetworkConfigurator

Modul zajišťuje automatickou konfiguraci IP adres v simulační síti. Všechny zařízení v síti budou mít stejnou síťovou adresu a lišit se budou pouze v hostitelské části. Modul neobsahuje žádné brány a může být v modelu použit pouze jednou. Konfigurace probíha pouze jednou, a to na začátku simulace (stage) v těchto krocích[viz 1]:

1. Přidělení IP adres stanicím a směrovačům, pro jednoduchost mají všechny rozhraní stejnou adresu,
2. následuje rozpoznání topologie sítě za použití třídy cTopology a výpočet nejkratších cest ke všem cílovým stanicím. Pro výpočet je použit Dijkstrův algoritmus.
3. Posledním krokem je uložení veškerých informací do routovacích tabulek.

Aby nedocházelo ke zbytečným chybám v konfiguraci sítě, je doporučeno nepoužívat tento modul současně se statickou konfigurací pomocí externích směrovacích tabulek.

2.3.2 ScenarioManager

Tento modul se používá pro řízení dynamické změny v síti. Modul načítá skripty z externích XML souborů, ve kterých je možné specifikovat požadované změny parametrů, nebo celých prvků v síti.

2.3.3 ChannelControl

Tento modul se používá při simulování bezdrátových prvků v síti. Modul zajišťuje pohyb stanice v síti a případné přepínaní mezi přístupovými body.

3 Routing Information Protocol

Routing Information Protocol (RIP) je dynamický směrovací protokol typu distance vector, jehož metrika je založena na počtu skoků mezi počátkem a cílem. Protokol je velmi jednoduchý a je určen především pro malé sítě. Poprvé byl definován v RFC 1058[11].

RIP verze 1 je „třídní“ protokol, to znamená, že nenese ve své Update zprávě informaci o podsíťové masce. Z tohoto důvodu protokol nepodporuje VLSM (Variable Length Subnet Masks) adresaci podsítí a stanic. VLSM je specifikované v RFC 1812[12]. RIP protokol přenáší celé směrovací tabulky mezi sousedními (přímo připojenými) směrovači. Aktualizované zprávy (Update message) o směrovacích tabulkách jsou zasílány v pravidelných intervalech, standardně 30 sekund, případně okamžitě při změně v topologii sítě.

3.1 Metrika a směrovací tabulka

Metrika RIP protokolu je založena na počtu skoků na cestě k cíli. Nejnižší metriku, nulovou, mají přímo připojené sítě ke směrovači a nejvyšší je hodnota 15. Metrika 16 označuje neplatnou cestu. Relativně nízká maximální hodnota 15 je použita z důvodu rychlejší konvergence směrovacích tabulek a zabránění vzniku směrovacích smyček.

Směrovací tabulka obsahuje informace o každé dosažitelné cílové stanici v síti. Tvoří ji tyto položky:

- Cílová IP adresa,
- Brána – adresa nejbližšího rozhraní sousedního směrovače na cestě k cíli,
- Síťová maska – adresa podsítě,
- Rozhraní,
- Metrika – počet skoků k cíli.

Základní směrovací tabulku pro přímo připojené sousedy, metrika 0, je nutno nakonfigurovat staticky. Budovaní a další aktualizace jsou již plně v kompetenci RIP protokolu, který si v pravidelných intervalech vyměňuje informace se všemi přímo

připojenými směrovači a ukládá informace o nejkratší cestě k cílově stanici. U více cest se stejnou metrikou se použije první v pořadí.

3.2 Směrovací algoritmus

Po každém přijetí aktualizované zprávy musí směrovač přičíst jedna k metrice každé nabízené cesty a porovnat svou stávající tabulku a aktuální informací. Neplatné cesty a nedostupné sítě jsou přenášeny s hodnotou 16.

Když směrovač přijme zprávu update, obsahující novou směrovací cestu od směrovače G s metrikou D, porovná ji s již existující. Pokud se přijatá cesta neshoduje s žádnou již existující cestou, tak ji uloží do své směrovací tabulky s metrikou min (16, D+1) přes směrovač G. Pokud tabulka již takovou cestu obsahuje, upraví její metriku na min (16, D+1). V případě, že je metrika již existující cesty do cílové sítě v tabulce větší (musí být větší a nikoliv rovná z důvodu zabránění oscilacím směrovacích cest) než D+1, upraví její metriku na D+1 přes G. [9] [11]

3.3 Časovače

RIP obsahuje několik časovačů pro řízení výkonnosti. Přehledně jsou uvedeny v Tabulka 1.

Tabulka 1: RIP časovače

Časovač	Interval	Popis
Update	30s	interval mezi periodickým odesíláním update zpráv
TimeOut	180s	pokud vyprší, je příslušná cesta označena jako neplatná
Flush	120s	pokud vyprší, je uchovávaná neplatná cesta vymazána

Zpráva update je obvykle odeslána každých 30 sekund plus malé náhodné zpoždění, které zabrání, aby všechny routery odeslaly Update současně. Vzhledem k pomalé konvergenci RIP protokolu u rozsáhlých sítí, kdy může trvat přenos informace z jednoho konce na druhý až za 15x30 sekund, se zavádí funkce Triggered update, kdy se informace o změně v síti odesílá okamžitě[viz 14].

Časovač TimeOut je spuštěn při uložení cesty do routovací tabulky a je restartován při každém přijetí Update pro danou cestu a metriku. Pokud vyprší, je daná cesta označena jako neplatná a spouští se mazací časovač Flush.

Do vypršení Flush časovače zůstává cesta součástí routovací tabulky a všech odesílaných Update zpráv. Metrika je nastavena na 16. Pokud během mazacího intervalu dorazí update s lepší metrikou, je stará cesta nahrazena aktuální cestou a časovač zrušen. Po vypršení je záznam o cestě smazán ze směrovací tabulky směrovače.

3.4 Formáty paketů

3.4.1 RIP verze 1

RIP k přenosu svých zpráv používá transportní protokol UDP. Pro vysílání a přijímáni datagram má přidělen port 520. Mimořádné požadavky na data ze směrovacích tabulek, často sloužící ke sledování topologie sítě, mohou přicházet z jiného portu, ale vždy směřují na port 520.

RIP datagram se skládá z oktetů. Jednotlivá pole jsou binární a nejvýznamnější bit je první. Maximální velikost je 512 oktetů, každý datagram tedy může nést informace o 25 cestách. Specifikace dle RFC1058[11] je uvedena v Tabulka 2

Tabulka 2: Formát zprávy RIPv1

0	8	16	31
příkaz	verze(1)	0	
identifikátor adresy		0	
IP Adresa			
0			
0			
Metrika			

Příkaz udává, zda se jedná o požadavek (=1) o RIP update zprávu, nebo odpověď (=2) na předchozí požadavek. Jako odpovědi jsou označeny i pravidelné update zprávy. Další oktet specifikuje verzi použitého RIP formátu. Zbývající dva oktety musí být nulové a zajišťují zpětnou kompatibilitu s nestandardními typy protokolu. Identifikátor adresy určuje použitou adresní rodinu, každý záznam má vlastní identifikátor adresy, AFI pro IP je 2. Další dva oktety jsou opět nulové. Následujících 14 oktetů je rezervováno pro síťovou adresu, v případě IP adresy zabírající pouze 4 oktety, je zbývajících 10 oktetů v RIP verze1 nevyužitých. Další 4 oktety jsou použity pro zaznamenání metriky dané cesty.

3.4.2 RIP verze 2

Nová verze protokolu RIP 2, podrobně popsána v RFC 2453 [12], obsahuje kromě standardních položek z RIPv1 několik vylepšení:

- Označení cesty
- Podsíťovou masku
- Adresu následujícího skoku
- Autentizaci
- Podporu pro multicast

Kompletní formát je uveden v **Tabulka 3.**

Tabulka 3: Formát zprávy RIPv2

0	8	16	31
Příkaz	verze(2)	zeroes	
identifikátor adresy		označení cesty	
IP Adresa			
podsíťová maska			
adresa sousedního směrovače			
Metrika			

Hlavička zprávy je shodná s předchozím typem a je tedy objasněna výše. Identifikátor adresy je také stejný jako u RIP verze1 s jednou výjimkou, a tou je hodnota 0xFFFF. V tomto případě zbytek záznamu obsahuje informace o autentifikaci (16 oktetů). Autentifikace má nyní jediný typ (=2) -heslo. Formát je uveden v **Tabulka 4**

Tabulka 4: Formát zprávy RIPv2 – Autentizace

0	8	16	31
Příkaz	verze(2)	0	
0xFFFF		typ autentizace	
Autentizace			
Autentizace			
Autentizace			
Autentizace			

Pole označení cesty obsahuje unikátní celočíselnou hodnotu, která se přenáší spolu s ostatními údaji o dané trase. Pole hraje důležitou roli při rozlišování tras naučených samotným RIP protokolem a trasou zjištěnou některým z externích protokolů, jako je OSPF nebo BGP. Podsíťová maska obsahuje adresu podsítě, kde se cílová stanice nachází. Pokud je nastavena nula, maska podsítě není definována. RIPv2 již také podporuje VLSM adresaci podsítí a stanic. Toto umožňuje protokolu pracovat s nesouvislými sítěmi. Další položkou je adresa dalšího skoku, která byla přidána pro zjednodušení přeposílání paketů a má obdobnou roli jako přeposílací adresa u OSPF protokolu. Metrika má stejné vlastnosti jako u protokolu verze1. Poslední ještě nezmíněnou novinkou u protokolu verze 2 je podpora multicastových adres. Pro snížení zátěže v síti nejsou zprávy Update posílány na všesměrovou adresu, ale na multikastovou adresu 224.0.0.9. Zařízení, která nepodporují RIPv2, zprávu nijak nezpracovávají a jednoduše ji smažou.

3.5 Stabilita protokolu

Pro zajištění stability protokolu a obraně proti směrovacím smyčkám využívá RIP následujících mechanizmů[dle 13]:

- Maximální počet přeskoků,
- Split Horizon,
- Poison reverse.

Maximální počet přeskoků je nastaven na 15, hodnota 16 již značí nedostupnou síť. Specifikace maxima je nutná pro zabránění tzv. počítaní do nekonečna (counting to infinity) v případě vzniku směrovací smyčky a přeposílaní chybných informací. Další často používanou metodou pro zabránění vzniku nežádoucích smyček je metoda Split horizont, kdy směrovač neposílá na dané rozhraní informace, které z tohoto směru získal. Technika poison reverse odesílá na dané rozhraní veškeré záznamy, avšak trasy z tohoto směru získané mají metriku nastavenou na nekonečno (16). A tím značí, že tyto cesty přes něj nejsou dostupné.

4 INSTALACE

V této kapitole bude popsána jak instalace simulačního nástroje OMNeT++, tak i jeho rozšíření INET frameworku. Zaměřím se na instalaci pod systémem Windows, kterou používám při své práci.

4.1 OMNeT++

Z domovských stránek [5] si stáhneme aktuální balíček omnetpp-xx-src.zip (xx značí číslo verze) se zdrojovými kódy. Archiv obsahuje kompletní programové vybavení včetně nástrojů třetích stran (perl, tcl/tk, libxmlě2 etc). Jediným externím nástrojem nutným k plnému využití OMNeT++ je nainstalované prostředí Java. Cesta k programu by měla být bez speciálních znaků a mezer například C:\OMNeTpp. Následně archiv rozbalíme.

Předpokladem pro korektní překlad je systém Windows 2000, nebo novější. V domovském adresáři spustíme soubor mingenv.cmg a otevře se příkazový interpret s již nastavenými systémovými proměnnými PATH do omnetpp-<version>/bin. Dalším krokem je kontrola nastavení souboru configure.user, kde je možné změnit některé parametry, například zvolit preferovaný typ xml parseru. Toto provedeme příkazem

```
./configure
```

Pokud vše proběhlo bez chyb, můžeme spustit překlad příkazem

```
make.
```

Úspěšnost instalace můžeme otestovat spuštěním ukázkových úloh

```
cd samples/dyna
```

```
dyna
```

poslední příkaz spustí úlohu v Tcl/Tk grafickém prostředí.Verze OMNeT++ 4.0 obsahuje již nové IDE prostředí založeném na prostředí Eclipse. [2] Pro jeho spuštění je potřeba mít nainstalovanou Java Runtime Environment 5.0 nebo novější. IDE spouštíme příkazem

```
omnetpp
```

Případná rekompilace prostředí se provádí posloupnosti příkazu

```
make clean
```

```
make
```

4.2 INET Framework

Pro korektní instalaci je třeba mít funkční verzi OMNeT++ a stažený balíček se zdrojovými kódy z domovských stránek[6]. Balíček rozbalíme do domovského adresáře (v našem případě C:\OMNeTpp) a spustíme interpret mingwenv.cmd.

4.2.1 Překlad pomocí příkazového řádku

Otevřeme adresář INET a zadáme příkaz pro automatické vytvoření makefiles souborů

```
make makefiles
```

Nyní již jen zkompilujeme celý framework obligátním příkazem

```
make,
```

pro debug verzi systému nebo příkazem

```
make MODE=release,
```

pro release verzi. Na závěr můžeme opět zkontrolovat úspěšnost instalace spuštěním vzorových úloh příkazy:

```
OMNeTpp/INET-xx> cd Examples
OMNeTpp /INET-xx/Examples> ./rundemo
```

4.2.2 Překlad z IDE prostředí

Z příkazového řádku spustíme IDE prostředí již známým příkazem

```
omnetpp
```

Jako workspace zvolíme adresář, kde jsme rozbalili INET framework (C:\OMNeTpp\INET-xx). Nyní musíme přilinkovat soubory INET pomocí příkazů

```
File -> Import -> General ->Existing Projects into Workspace
```

Jako root directory vybereme náš workspace adresář. Zatrhávací pole Copy projects into workspace musí zůstat nezvoleno. A klikneme na Finish. Nyní zkompilujeme celý projekt příkazem

```
Project | Build all.
```

Nebo stisknutím klávesové zkratky Ctrl+B. Úspěšnost překladu můžeme opět zkontrolovat spuštěním vzorových úloh ze složky

```
INET-xx\Examples.
```

5 Implementace

5.1 Použité programové vzbavení

Při implementaci jednotlivých komponent je využito následující programové vybavení:

- Operační systém: Windows XP service pack3,
- Kompilátor: MinGW C++ compiler,
- OMNeT: verze 4.0 src.windows,
- INET Framework: 20090325,
- Eclipse IDE 4.0,
- Java Runtime Environment 5.0.

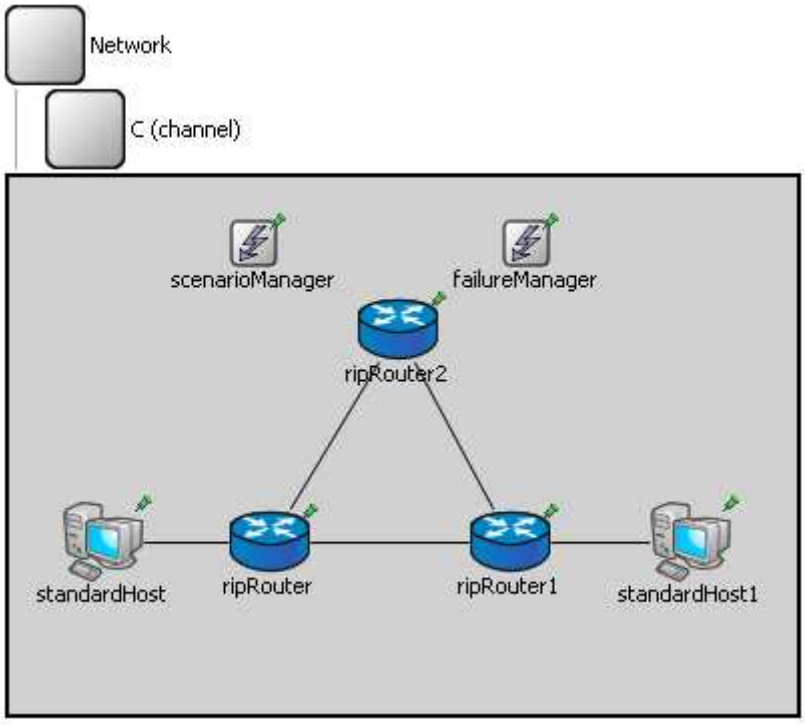
5.2 Model testovací sítě

Pro potřebu kontroly a testování nových modulů jsem vytvořil jednoduchý model sítě. Její topologie popsána jazykem NED je následující:

```
import ned.DatarateChannel;
import inet.nodes.inet.StandardHost;
import inet.world.ScenarioManager;
import inet.nodes.inet.FailedRouter;
import inet.networklayer.extras.FailureManager;
network Network
{
    @display("bgb=393,268");
    types:
        channel C extends DatarateChannel
        {delay = uniform(0.1ms, 1ms);}
    submodules:
        ripRouter: RipRouter
            @display("p=130,182");
        ripRouter1: RipRouter
            @display("p=250,182");
        ripRouter2: RipRouter {
            @display("p=194,76");
            gates:
                ethg[2];       }
        standardHost: StandardHost
            @display("p=47,182");
        standardHost1: StandardHost
            @display("p=340,182");
        scenarioManager: ScenarioManager
            @display("p=122,33");
        failureManager: FailureManager
            @display("p=258,33");
```

```
    connections allowunconnected:
        ripRouter1.ethg++ <--> C <--> standardHost1.ethg++;
        ripRouter.ethg++ <--> C <--> standardHost.ethg++;
        ripRouter2.ethg[0] <--> C <--> ripRouter.ethg++;
        ripRouter2.ethg[1] <--> C <--> ripRouter1.ethg++;
        ripRouter.ethg++ <--> C <--> ripRouter1.ethg++;
}
```

Při bližším prozkoumání předcházejícího kódu zjistíme, že obsahuje dva počítače, modul StandardHost, které budou zajišťovat datovou komunikaci v síti a tři směrovače, RipRouter, které budou podrobně popsány v další kapitole. Moduly ScenarioManager a Failure manager slouží k zajištění dynamických změn v síti, v našem případě způsobí výpadek routeru, který musí Rip protokol korektně vyřešit. Klíčové slovo display definuje umístění všech prvků v rámci pracovní plochy. V poslední části jsou nastaveny obousměrné spoje mezi příslušnými moduly. K nastavení přenosových vlastností jednotlivých spojů je vložen modul DatarateChannel, který byl již popsán dříve. V grafické podobě je celý model na Obr. 1.



Obr. 2 Topologie testovací sítě

Konfigurace modelu je uložena v souboru omnetpp.ini. Úplná definice pro náš model je v příloze B. Tento soubor obsahuje parametry určující chování jednotlivých modulů v rámci celého modelu sítě. Soubor může obsahovat odkazy na jiné .ini soubory za

pomoci direktivy include. Komentáře jsou označeny symbolem #. Povinná sekce společná pro všechny simulační běhy je nazvána [General]. Musí tu být definován NED model, který má být použit, volitelnou položkou je délka simulace definována proměnnými sim-time-limit a cpu-time-limit. Dále soubor obsahuje uživatelem definované hodnoty parametrů, které nejsou nastaveny v .ned souborech. Pro specifikaci jednotlivých parametrů musíme znát jeho úplné jméno, to je složeno z cesty v hierarchii modulů a názvu konkrétního parametru. Pro oddělení jmen odlišné úrovně hierarchie je použita tečková notace. Náš konfigurační soubor obsahuje parametry pro načtení směrovacích tabulek k jednotlivým uzlům sítě.

```
**.RipRouter1.routingFile = "R1.irt",
```

Provoz v simulační síti tvoří zprávy typu:

- `UDPBasicApp`, která v zadaném intervalu posílá UDP pakety na cílovou IP adresu. Nastavení intervalu provedeme tímto příkazem:

  ```
  **.udpApp[0].messageFreq = 0.1s,
  ```

  délku zprávy pomocí:

  ```
  **.udpApp[0].messageLength = 32 bytes
  ```

5.3 Příprava pro rozšíření

Směrovací protokol RIP musí úzce spolupracovat zejména se směrovací tabulkou a proto je nutné ještě před vlastní implementací prozkoumat metody pro její modifikaci.

5.3.1 Metody třídy RoutingTable

- K záznamům v tabulce mohou ostatní moduly přistupovat pomocí metod třídy RoutingTable, [1]. Záznamy nelze přímo modifikovat, pouze smazat a opětovně přidat pozměněný záznam.
- funkce pro přidání nového záznamu do tabulky

  ```
  void RoutingTable::addRoute(const IPRoute *entry)
  ```

Metoda `addRoute` má pouze jednu vstupní proměnou typu const IPRoute.

Třída IPRoute má tyto chráněné atributy:

- `IPAddress host;    // IP adresa cílové stanice`
- `IPAddress netmask; // IP adresa podsítě`
- `IPAddress gateway; // IP adresa brány`
- `InterfaceEntry *interfacePtr; // ukazatel na interface`
- `RouteType type;     // typ cesty`
- `RouteSource source; // specifikuje původce záznamu`
- `int metric;    // metrika cesty`

Proměnná `RouteType` je výčtový typ, který může nabývat pouze dvou hodnot:

1. Direct- zařízení je přímo připojené k routeru,
2. Remote- zařízení je připojené přes další router v síti.

Proměnná `RouteSource` je opět výčtový typ definující původ záznamu a nabývá následujících šesti hodnot:

1. Manual- manuálně přidaná statická cesta,
2. Ifacenetmask- cestu oznámil interface,
3. RIP- cesta přidána dynamicky RIP protokolem,
4. OSPF- cesta přidána dynamicky OSPF protokolem,
5. BGP- cesta přidána dynamicky BGP protokolem,
6. ZEBRA- cesta přidána na základě Quagga modelu.

Aby mohla být cesta uložena správně, obsahuje funkce rozhodovací algoritmus typu if-else. Pro uložení defaultní trasy musí byt pole Host a Netmask nulové. K ověření této skutečnosti, slouží boolovská funkce `bool IPAddress::isUnspecified,` která vrací True při nulových IP adresách. Existenci rozhraní pro ukládanou cestu testuje metoda:

```
if (!entry->getInterface())
  error("addRoute(): interface cannot be NULL");
```

posledním testem je ověření, zda neukládám novou defaultní cestu, namísto již existující. Pokud ano, musím původní záznam nejdříve smazat pomocí metody:

```
if (entry->getNetmask().isUnspecified() && getDefaultRoute()!=NULL)
 deleteRoute(getDefaultRoute());
```

Na závěr je potřeba rozlišit, zda je ukládaná cesta multicastová, nebo unicastová a podle výsledku ji uložit do správného datového kontejneru.

```
if (!entry->getHost().isMulticast())
    routes.push_back(const_cast<IPRoute*>(entry));
else
    multicastRoutes.push_back(const_cast<IPRoute*>(entry));
```

Posledním krokem je odeslání informace o nové cestě modulu NotificationBoard, který informaci zobrazí uživateli v grafickém prostředí.

```
nb->fireChangeNotification(NF_IPv4_ROUTE_ADDED, entry);
```

- Funkce pro smazání záznamu:

```
bool RoutingTable::deleteRoute(const IPRoute *entry)
```

Funkce vrací *true*, pokud byl záznam úspěšně smazán, *False* při nenalezení hledaného záznamu. Aby bylo možné záznam korektně smazat, musíme ho nejdříve vyhledat v tabulce. Protože je routovací tabulka implementována jako dynamický kontejner typu vector, použijeme pro prohledaní knihovní funkci:

```
RouteVector::iterator i = std::find(routes.begin(), routes.end(), entry);
```

Vstupní proměnné router.begin() a routers.end() vrátí iterátory na první a poslední prvek v tabulce a tím ohraničí oblast hledaní. Pokud metoda nevrátí iterátor na konec pole, byla trasa nalezena a můžeme ji následující skupinou příkazů smazat,

```
if (i!=routes.end())
{
    nb->fireChangeNotification(NF_IPv4_ROUTE_DELETED, entry);
    routes.erase(i);
    delete entry;
    invalidateCache();
    updateDisplayString();
    return true;
}
return false;
```

Příkaz `fireChangeNotification` slouží opět k informování uživatele o prováděném kroku. Následně je smazána samotná položka směrovací tabulky i proměnná vedoucí k jejímu nalezení. Na závěr je smazána směrovací cache, upraveny zobrazované informace o směrovací tabulce a odesláno *true*, deklarující úspěšné smazaní záznamu.

5.3.2 Načítaní statických záznamů

Protože RIP protokol nepracuje bez staticky nastavených adres přímých sousedů, musíme je načítat z externích souboru. OMNeT podporuje dva typy souboru s příponami .irt a .mrt. Načítaní se provádí behem inicializce modulu RoutedTable pomocí funkcí knihovní třídy RoutingTableParser.[1]

Pro bezchybné načtení požadovaných záznamů musí mít soubor přesný formát. V první části souboru jsou nakonfigurována jednotlivá rozhraní, na konci pak soubor obsahuje statické směrovací záznamy. Každý nový záznam začíná klíčovým slovem name a pokračuje výčtem dalších klíčových slov a jejich parametrů. Seznam klíčových slov:

```
name:          //přiřazuje název rozhraní (např. eth0, ppp0)
inet_addr:     //IP adresa
Mask:          //maska sítě
Groups:        //multicastové skupiny. 224.0.0.1 je přidána automaticky
MTU:           //MTU on the link (např. pro Ethernet 1500)
Metric:        //celočíselná hodnota metric
flags:         //příznaky BROADCAST, MULTICAST, POINTTOPOINT
```

Směrovací část, začíná klíčovým slovem Route, následují jednotlivé záznamy v tomto pořadí:

```
Destination  Gateway  Netmask  Flags  Metric Interface
```

Všechny proměnné mají standardní význam a byly již popsány na předchozích stranách.

5.4 Vlastní modely pro OMNeT

Tento oddíl obsahuje popis nových modelů, potřebných pro rozšíření INET frameworku o RIP protokol. Modely vychází z již existujících jednoduchých modulů a nově vytvořeného modulu RIP, jehož funkcionalita je podrobně popsána v **kapitole 3**.

5.4.1 RipRouter

Model RIP routeru je založen na již exitujícím modelu Router, který nabízí INET framework. Model jsem rozšířil o podporu RIP protokolu pracujícím na transportním

protokolu UDP. Abych nenarušil, již existují simulace, pracující s knihovní verzí modelu, vytvořil jsem kopii a doplnil ji o následující řádky:

```
udp: UDP
      @display("p=144,140; i=block/table2");
rip: Rip
      @display("p=42,158; i=block/network2");
```

Ještě musíme doplnit parametry přidaných modulů:

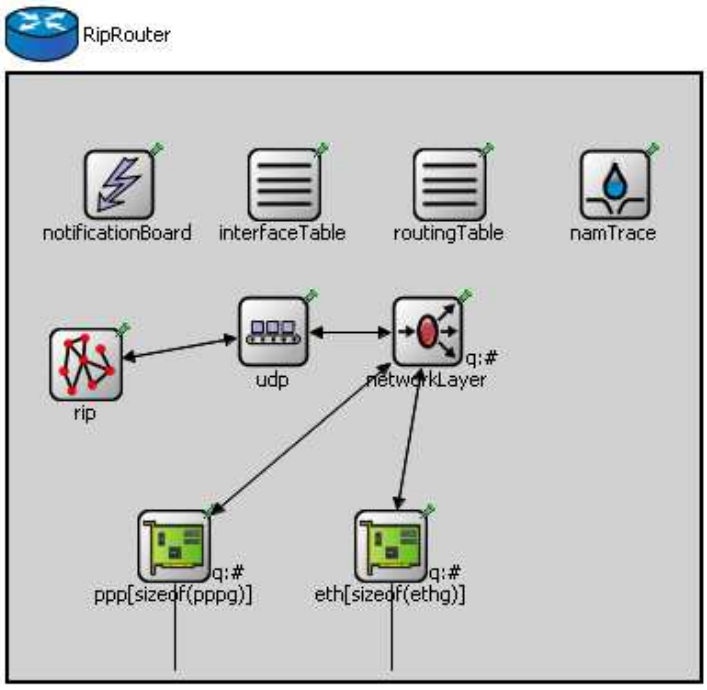
```
      int numUdpApps = 1; // router může obsahovat pouze jednu instanci
          // RIP modulu
      string udpAppType = default("n/a");
```

Posledním krokem je nastavení spojů pro komunikaci mezi novými a stávajícími moduly:

```
connections allowunconnected:
      udp.ipOut --> networkLayer.udpIn;
      udp.ipIn <-- networkLayer.udpOut;

      rip.udpOut --> udp.appIn++;
      udp.appOut++ --> rip.udpIn;
```

Jak je patrné z předchozího kódu, jedná se o jednosměrné spoje příslušných vstupních a výstupních bran modulů. Directiva allowunconnected umožňuje provozovat síť s nepřipojenými branami. Grafická podoba modelu je zobrazena na **Obr. 3**.Kompletní NED definice je součástí přílohy A1.



Obr. 3: Struktura modulu RipRouter

5.4.2 Jednoduchý modul Rip

Modul Rip, je aktivní modul, proto mu kromě NED definice parametru a bran musíme vytvořit i c++ soubor, ve kterém bude popsáno jeho chování. Modul má jeden vstup a jeden výstup napojený na transportní protokol UDP, a proto musí mít mezi parametry nadefinované čísla portů. Parametry jsem zvolil tyto:

- `localPort` - číslo portu, na kterém se vytvoří lokální spojení s UDP modulem,
- `destPort` - číslo portu, na který se budou zprávy odesílat,
- `messageLength @unit("B")` - základní velikost zprávy, udávaná v bytech,
- `destAddresses = default("")` - cílové adresy pro zasílání zpráv,
- `UpdateTime @unit("s") = default(30s)` - interval odesílání Update zpráv,
- `TimeOut @unit("s") = default(180s)` - interval platné trasy,
- `Flush @unit("s") = default(120s)` - interval po kterém je neplatná cesta vymazána.

Poslední tři parametry definují intervaly RIP časovačů, mají nastaveny defaultní hodnoty dle RFC1058[11], ale je možné je libovolně změnit v konfiguračním souboru omnet.ini. Celá NED definice je v příloze A2.

5.4.3 Class Rip

Třída Rip definuje chování RIP protokolu. Je odvozena od třídy UDPAppBase, která je součástí INET frameworku. Jelikož se jedná jednoduchý modul, musí obsahovat metody nutné pro začlenění do simulačního prostředí:

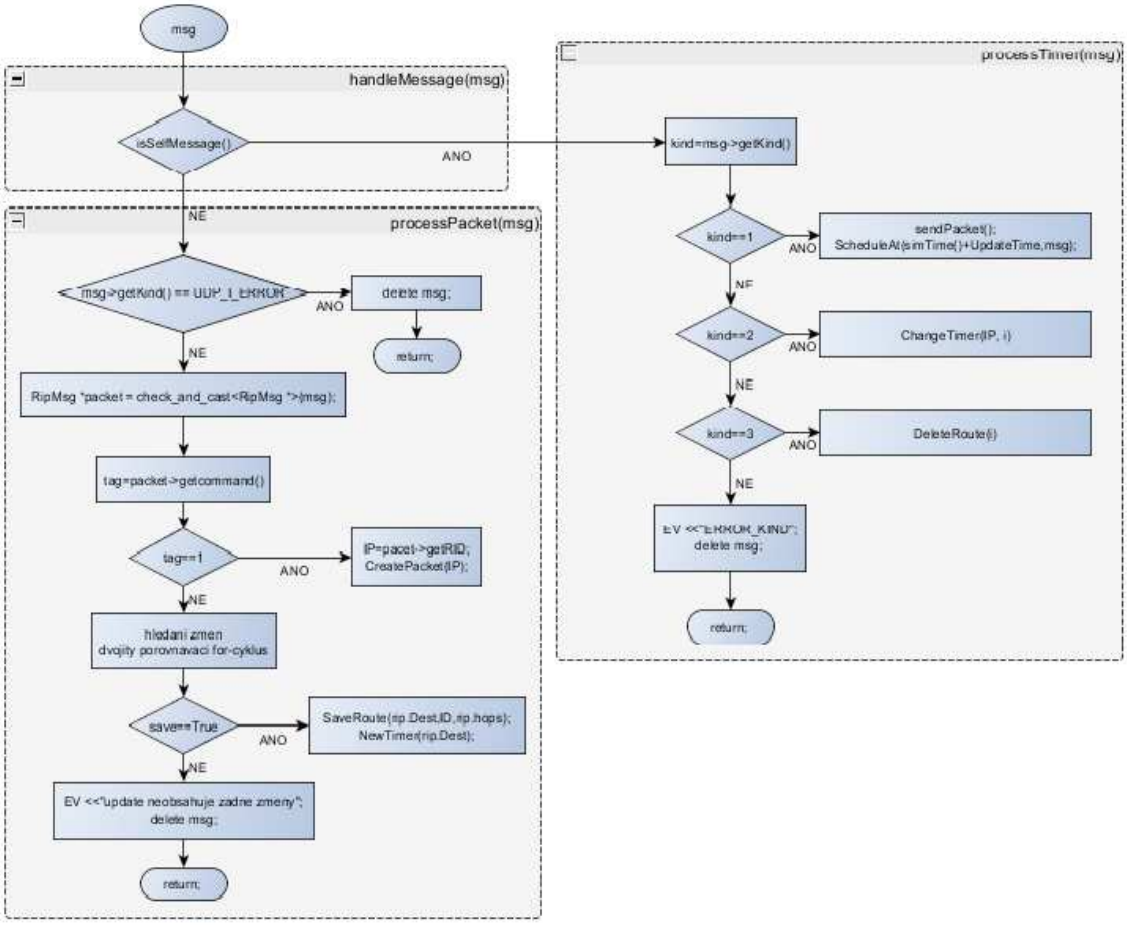
```
virtual void initialize(int stage);
virtual void handleMessage(cMessage *msg);
virtual void finish();
```

Dále jsou definovány metody pro zpracování RIP zpráv a vypršení časovačů:

```
virtual cPacket *createPacket(IPAddress dest);
virtual void sendPacket();
virtual void processPacket(cPacket *msg);
virtual void processTimer(cMessage *msg);
virtual void saveRoute(cPacket *msg);
virtual void NewTimer (IPAddress routeIP);
virtual void ChangeTimer(IPAddress IP, int i);
```

Celý výpis deklarace třídy je v příloze B1.

Chování Rip modulu a volání jednotlivých metod znázorňuje vývojový diagram na **Obr. 4**. Po spuštění aplikace dojde k inicializaci, ve které jsou nastaveny parametry modulu. Veškerá komunikace probíhá pomocí zpráv a řídí ji metoda `handleMessage()`, která na základě typu příchozí zprávy určí, kterou metodou bude daná zpráva zpracována. Pokud se jedná o vlastní zprávu (self message), generovanou některým z časovačů, tak se zavolá metoda `processTimer()`, ta zjistí, o jaký časovač se jedná (Update, TimeOut, Flush) a zprávu zpracuje. Pokud se nejednalo o vlastní zprávu, jde o příchozí paket z nižší vrstvy, je zavolána metoda `processPacket()`, ta zjistí, zda se jedná o zprávu typu regest, nebo response a zavolá přidělenou funkci pro zpracování.



Obr. 4: Vývojový diagram volání metod RIP modulu po přijetí zprávy

5.4.3.1 Initialize

Funkce `virtual void initialize(int stage)` má jednu vstupní proměnou, která zajišťuje zpracování příkazů ve správné inicializační vlně. Protože náš modul pracuje se směrovací tabulkou a tabulkou rozhraní, musíme počkat, než budou sami inicializovány. Protože směrovací tabulka načítá statické záznamy ze souboru během první vlny a k přiřazení rozhraní dochází ve druhé vlně, může naše inicializace proběhnout nejdříve ve vlně třetí. Součástí inicializace je přiřazení parametru ze souboru omnet.ini do příslušných proměnných. Dále jsou nastaveny proměnné pro tvorbu statistik, definuji tři:

- počet odeslaných zpráv,
- počet přijatých zpráv,
- počet změn ve směrovací tabulce.

Dalším důležitým krokem je navázání modulu na transportní protokol, použijeme funkci `void UDPAppBase::bindToPort(int port)`, která odešle do modulu UDP zprávu s požadavkem na přesměrování komunikace. Zpráva vytvoří proměnnou typu UDPControlInfo, kam uloží číslo našeho lokálního portu a vygenerované socketID.

```
UDPControlInfo *ctrl = new UDPControlInfo();
ctrl->setSrcPort(port);
ctrl->setSockId(UDPSocket::generateSocketId());
msg->setControlInfo(ctrl);
send(msg, "udpOut");
```

Po zpracování v UDP přesměruje veškerou komunikací s naším `local port` do tohoto modulu.

Na závěr inicializace vytvoříme zprávu Update a pomocí vnitřního časovače ji odešleme do systému.

```
cMessage *msg = new cMessage("sendUpdate",1);
scheduleAt(simTime(), Update[0]);
```

Odezvou systému na přijetí vnitrní zprávy sendupdate je odeslání zprávy RIPUpdate na všechny přímo připojené směrovače.

5.4.3.2 SendPacket()

Tato funkce je velice jednoduchá a slouží k odeslání RipUpdate zpráv na všechny sousední směrovače. K tomu využívá dvou funkcí, první je `createpacket(IPAddress dest)`, která je popsána v následující části a druhou pak funkce `void UDPAppBase::sendToUDP(cPacket *msg, int srcPort, const IPvXAddress& destAddr, int destPort)`, která při odesílání opatří naší zprávě patřičné `UDPControllInfo`, které je nutné pro přenos UDP datagramu v síti.

5.4.3.3 CreatePacket()

Funkce `cPacket *Rip::createPacket(IPAddress dest)` vytváří RipUpdate zprávu, kterou odesílá na cílovou adresu. Formát zprávy jsem vytvářel podle RFC pro RIP verze 1[11].

```
packet RipMsg
{
    int command = 2;    // zpráva typu odpověď
    Ripr Rip[]; // pole struktur
}
```

Protože zpráva musí být univerzální a přenášet libovolný počet položek z intervalu 0-25, používám dynamické pole struktur. Každá struktura nese informace o jedné stanici, tedy její IP adresu a metriku.

```
struct Ripr
{
    int Afi=2; // posílam IP adresu
    IPAddress Address; // IP adresa
    int Metric;         //metrika cesty
}
```

Pro uložení dat ze směrovací tabulky do vytvořené zprávy se používá funkce `const IPRoute *RoutingTable::getRoute(int i)` třídy routingTable, která načítá i-tý záznam tabulky do proměnné IPRoute. V tuto chvíli už jednoduše přiřadím odpovídající proměnné IPRoute proměnným vytvořené RIP zprávy. Tento postup aplikuji na všechny záznamy v tabulce. Kompletní zprávu vrátím funkci `SendPacket()` direktivou return k odeslání.

5.4.3.4 ProcessMessage()

Tato funkce je volána po přijetí RipUpdate zprávy a slouží k uložení informací o nových, či lepších cestách k cílovým stanicím. Pomocí dvou vnořených for - cyklů funkce porovnává postupně každou položku z přijaté zprávy se všemi záznamy ve směrovací tabulce. Pokud už trasa v tabulce existuje, je pouze resetován časovač TimeOut. Pokud tabulka cílovou stanici ještě neobsahuje, tak ji uloží zavoláním funkce `void Rip::SaveRoute(rip.Dest,ID,rip.hops).` Zároveň je zavolána funkce `void Rip::NewTimer (IPAddress routeIP)` pro nastavení TimeOut časovače pro novou trasu. Třetí možností je, že porovnávaná cesta sice již v tabulce je zanesena, ale má vyšší metriku. V tomto případě musíme záznam v tabulce smazat a posléze opět uložit s nižší metrikou. Tento postup je nutný, protože přístupové funkce směrovací tabulky neumožňují modifikaci uložených hodnot. Časovač příslušný dané položce se mazat nemusí a je pouze restartován.

5.4.3.5 NewTimer()

Funkce pracuje s globálními datovými kontejnery a zajišťuje správu TimeOut časovače. Pro ukládání časovače je použit datový typ `vector.` Jedná se o datový kontejner z knihovny STL jazyka C++. Vektor je dynamická šablona jednorozměrného pole. Jednotlivé prvky nemusí být v paměti uloženy za sebou, a proto můžeme vkládat a rušit prvky zcela libovolně. Ve funkci jsou použity metody `push_back`, která vkládá nový prvek na konec pole. Velikost vektoru se tak zvýší o jeden prvek. A metodu `erase`, pro mazání libovolného prvku z kontejneru. Tato metoda je o něco složitější, protože je nutné pracovat s iterátory, což je v podstatě obdoba ukazatelů pro kontejnery. Více o iterátorech je možné nalézt v literatuře[8].

Jednotlivé časovače od sebe odlišuji dle IP adresy trasy. Vstupní proměnou proto musí být příslušná IP adresa. Po přijetí nejprve zjistím, zda je již časovač s touto adresou uložen v kontejneru, či nikoliv. Používám for cyklus a jednoduchou podmínku `if.`

```
if (pointer->getID()==routeIP)
{
      cancelEvent(TimeOut[w]);
      EV << "Restart casovace "<< routeIP.str() <<" \n";
      scheduleAt(simTime()+par("TimeOut"), TimeOut[w]);
}
```

Porovnávám IP adresu trasy s IP adresou položek v kontejneru. Při shodě restartuji časovač postupným zavoláním funkcí cancelEvent(), která zruší nedokončený odpočet a scheduleAt, která nastaví odpočet nový. Příkaz EV slouží pro výpis v GUI prostředí.

Pokud se IP adresy neshodují, vytvořím novou zprávu typu časovač a uložím jí do kontejneru metodou TimeOut.push_back(ti). Zpráva typu Timer je velice jednoduchá a obsahuje pouze IP adresu a celočíselný příznak udávající typ časovače, který ji vytvořil.

Při každém ukládání je nutné kontrolovat, zda pro danou trasu neběží časovač pro mazání trasy ze směrovací tabulky. Tato situace nastává při výpadku v síti. V případě, že běží, musíme ho bezpodmínečně zrušit, abychom záznam neztratili.

5.4.3.6 ProcessTimer()

Funkce zpracovává příchozí vnitřní zprávy od časovačů. Na základě typu časovače, který zprávu poslal, zavolá odpovídající obslužnou rutinu.

```
void Rip::processTimer(cMessage *msg)
{
if (msg->getKind()==1)
        {       // časovač Update
          sendPacket();
          scheduleAt(simTime()+(double)par("messageFreq"),msg);
        }
else if (msg->getKind()==2)
        {       // časovač TimeOut
          processTimeOUT();
        }
else if(msg->getKind()==3)
        {       // časovač Flush
          delRoute();
        }
else {delete msg;
        }
}
```

Funkce je složena z řetězce podmínek if-else. Rozhodujícím faktorem je proměnná *kind*, která je součástí příchozí zprávy. Protože se jedná o chráněnou proměnou, používá se pro přístup metoda getKind(). Časovač pro update je označen číslem jedna a jeho přijetí vyvolá funkci sendPacket(), která odešle všem svým sousedům RIPupdate zprávu a spustí nové odpočítávání. Příznak Kind rovný dvěma značí vypršení časovače

TimeOut, tedy neplatnou cestu. Funkce `processTimeOut()` přiřadí dané cestě metriku 16(neplatná cesta) a spustí časovač pro úplné smazání záznamu ze směrovací tabulky. Kind rovný třem řeší pravě tuto variantu, kdy po vypršení časovače Flush je záznam nevratně smazán.

5.4.3.7 ChangeTimer()

Funkce řeší situaci po vypršení časovače TimeOut. Vstupní proměnou je číslo(pozice) záznamu nové neplatné cesty. Protože používáme pro zabránění směrovacích smyček metodu Poison Reverse, musíme nastavit metriku cesty na 16. Jak už bylo zmíněno v teoretické části, třída RoutingTable nemá metodu pro modifikaci pouze jednoho záznamu v tabulce. Změna se řeší vytvořením proměnné typu IPRoute, do kterého jsou přiřazeny všechny informace o dané cestě. Poté změníme požadovanou dílčí proměnnou pomocí metod třídy IPRoute, původní záznam v tabulce smažeme a vložíme záznam nový, modifikovaný. Dalším krokem je spuštění Flush časovače pro zneplatněný záznam. Nejprve musíme vytvořit samotný časovač pomocí metody `push_back(i)` třídy `std::vector`. Tato metoda vytvoří nový prvek v kontejneru Flush časovače a zařadí ho na konec pole. Nyní mu ještě přiřadíme příznak pro označení této neplatné cesty a pomocí knihovní funkce `scheduleAt` ho odešleme do systému.

5.4.4 FailedRouter

Složený model Failed router vychází z implementovaného RIPRouteru. Představuje vadný prvek v síti a slouží k testovaní dynamických vlastností RIP protokolu. Chybí mu směrovací tabulka a místo modulu RIP obsahuje modul Fail.

Třída Fail popisuje chování jednoduchého modulu Fail. Třída obsahuje pouze základní metody `initialize()` a `handleMessage(msg)`. Jakákoliv přijatá zpráva je ihned smazána.

5.4.5 ScenarioManager

Tento modul zajišťuje dynamické prvky chování simulační sítě. Změny nelze provádět přímo za běhu simulace, ale musí se nastavit již před jejím přeložením a spuštěním. K popisu chování se používá **.xml skript**. V našem případě skript spouští modul FailureManager, který implementuje poruchy uzlů v síti a vypadá následovně:

```
<scenario>
        <at t="3.1s">
                <shutdown module="failureManager" target="ripRouter"/>
        </at>
        <at t="8.0s">
                <startup module="failureManager" target="ripRouter"/>
        </at>
</scenario>
```

Skript začíná klíčovým slovem *scenario*, které značí jeho počátek. Příkaz *at* slouží k řetězení požadavků na změnu v síti a jeho atribut *t* udává čas, kdy má k provedení příkazu dojít. Příkaz *shutdown* provádí záměnu modulu definovaným za klíčovým slovem *target* za vadný modul. Odpovídající dvojice jsou nastaveny v modulu FailureManager. Inverzní příkaz *startup* nahrazuje vadný modul zpět modulem fungujícím. V našem případě je modul ripRouter nahrazen v čase t = 3.1s modulem FailedRouter. A v čase t = 8s je opět vracen zpět.

5.4.5.1 Class ScenarioManager

Třída popisující chování jednoduchého modulu ScenarioManager je stále ještě ve stádiu vývoje, a proto nemá implementovánu celou řadu užitečných metod. Naštěstí námi využívaná metoda pro volání FailureManageru je již funkční a můžeme ji s drobnými úpravami použít.

Během inicializační fáze dochází k načtení .xml souboru a nastaví se časovače pro všechny nalezené časové údaje (parametr *t*). Při vypršení časovače je volána metoda `handleMessage()`, která zpracuje příchozí vnitřní zprávu a zavolá metodu `processcommand(node)`, kde node značí modul, kterého se změna týká. V našem případě se jedná o Failure manager, o kterém pojednává další kapitola.

5.4.6 FailureManager

Tento jednoduchý modul implementuje v návaznosti na scenarioManager rušení a spouštění modulů za běžící simulace. Neobsahuje žádné parametry a nevysílá ani nepřijímá žádné zprávy. Metody `handleMessage(msg)` a `initialize()` neobsahují žádný kód.

Modul pracuje na základě přímých volání funkce:

```
void FailureManager::processCommand(const cXMLElement& node),
```

ta voláním `fce cModule *target=getTargetNode(node.getAttribute("target"))` načte cílový modul a pomocí řetězce podmínek if-else if hledá k přijatému modulu modul chybný. Pro náš ripRouter a příkaz `shutdown` platí podmínka:

```
if(!strcmp(target->getModuleType()->getName(), "RipRouter"))
   replaceNode(target, "FailedRipRouter")
```

a naopak při inverzním volání `startup podminka:`

```
if(!strcmp(target->getModuleType()->getName(), "FailedRipRouter"))
   replaceNode(target, "RipRouter"),
```

která vrací do sítě zpět původní router. Funkce replaceNode() vyhledává v knihovně celého simulačního prostředí cílový modul a při nalezení na něj přepojí všechny existující linky voláním funkce `reconnectGate(old->gate,new->gate)`. Posledním krokem je aktivace modulu metodou `module->scheduleStart(simTime())` a inicializace jeho parametrů.

6 Laboratorní úloha

Název: **Simulace RIP protokolu v prostředí OMNeT++**

6.1 Cíl

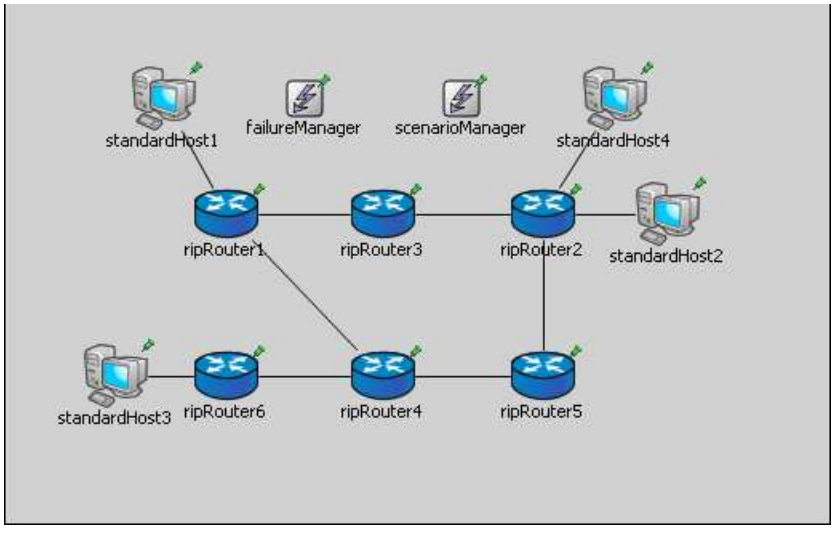
Cílem laboratorní úlohy je demonstrace nasazení směrovacího protokolu RIP v datové síti. Vytvoření jednoduché sítě obsahující základní síťové prvky v simulačním prostředí OMNet++ 4.0. V simulaci je kladen důraz na dynamické vlastnosti RIP protokolu.

6.2 Požadavky na pracoviště

- Hardwarové požadavky: PC s RAM minimálně 256MB, 2GB volného místa na disku
- Softwarové požadavky: OMNeT 4.0, INET Framework, Java

6.3 Zadání

- Seznamte se s problematikou dynamického směrovaní v datových sítích. Nastudujte jednotlivé směrovací protokoly, především protokol RIP a parametry ovlivňující jeho chování.
- Na základě vytvořeného modelu datové sítě, který je uveden na **Obrázek 1**, navrhněte na volný list papíru směrovací tabulky jednotlivých prvku v síti.
- Uvažujte výpadek směrovače RipRouter2 a zaznamenejte případné změny ve směrovacích tabulkách.
- V programu OMNeT++ proveďte simulaci RIP protokolu v modelové síti a vysledné statistiky porovnejte se svými teoretickými předpoklady.



Obrázek 1: Schéma simulované datové sítě

Směrovací tabulky vytvářejte v tomto formátu:

```
Destination  Gateway  Netmask  Flags  Metric Interface
```

Detailní informace lze nalézt v literatuře [1]

6.4 Teoretický úvod

6.4.1 Úvod do RIP

Protokol RIP je dynamický směrovací protokol. Pro určení nejlepší cesty paketu sítí používá vzdálenostní metriku mezi odesilatelem a příjemcem. Metrika je dána počtem přeskoků(směrovačů) v cestě. Ostatní parametry linek nebere v úvahu. Maximální počet přeskoků je omezen na 15, aby byla zajištěna rychlá konvergence směrovacích tabulek.

Každý směrovač posílá svým sousedům v pravidelných intervalech pakety obsahující celou svoji směrovací tabulku. Z přijatých informací si každý směrovač vypočítá optimální směry pro vlastní směrovací tabulku. Protokol RIP je přenašeč pakety UDP na aplikačním portu 520.

Existují dvě verze tohoto směrovacího protokolu. RIPv1 je jednoduší a je definován v RFC 1058, [2]. Simulátor OMNeT má momentálně implementovanou pouze tuto verzi.

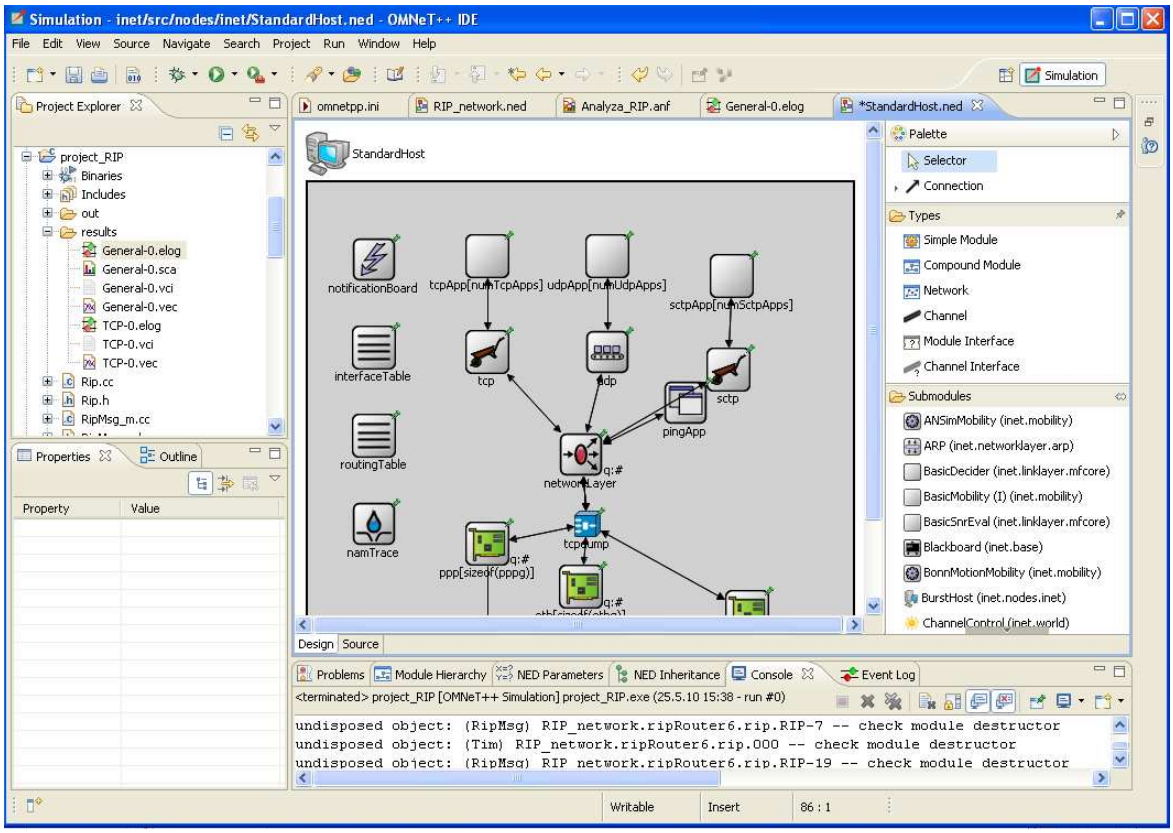
6.4.2 Simulační prostředí OMNeT

OMNeT++ je diskrétní simulační prostředí pro modelování komunikačních sítí. Je naprogramován v jazyce C++ a distribuován pod volnou licencí pro nekomerční účely.

V OMNeTu verze 4.0 je implementováno grafické prostředí, které je založené na platformě Eclipse. Vytváření modelů je tak snazší, rychlejší a přístupnější většímu počtu uživatelů.

Prostředí se skládá z různých editorů upravených pro potřeby OMNeT++. V horní části je klasické roletové menu. V levé horní části je souborový manažer (Project explorer), zobrazující všechny soubory vytvořené simulace. Vlevo dole se nachází okno s vlastnostmi zvoleného modulu. V pravé častí je umístěna paleta nástrojů. Ta má tři části, v první (horní) je možno zvolit mezi selektorem, slouží k výběru modulu a spojením, vytváří vazbu mezi moduly. Ve druhé se nachází výčet typů modulů simulátoru. Třetí část okna obsahuje jednotlivé submoduly, jak importované z knihovny INET, tak i moduly vytvořené uživatelem. Přiklad pracovní plochy s otevřeným NED editorem složeného modulu standardHost je na **Obrázek 2**.

Simulátor má podrobnou dokumentaci přistupnou na webových stránkach[4], kde mohou zájemci nalézt detailní informace o jednotlivých modulech a funkcích.



Obrázek 2: Grafické prostředí OMNeT IDE

6.5 Vytvoření simulační sítě

1) Spustíme simulační prostředí OMNeT++.
2) Zvolíme položku **File > New**… vybereme, že chceme vybrat nový projekt. Zadáme název projektu (třeba **project_RIP**) a klikneme na **Next**.
3) V dalším okně vybereme ze složky simpleproject položku **Empty project** a opět klikneme **Next.**
4) V dalším okně necháme zaškrtnuté oba debugery a vybereme položku **Advanced Settings..**, kde v menu project references zvolíme **INET** a potvrdíme stiskem **OK**, tím do palety nástrojů přilinkujeme moduly z INET frameworku.
5) Přípravu ukončíme tlačítkem **Finish**.

6) Nyní musíme vybudovat síť. Vybereme **File > New**... vybereme položku network description file, pojmenujeme ho (třeba **RIP_network**) a klikneme na **Finish**.
7) V otevřeném NED editoru umístíme na plochu z palety objektů šest směrovačů **RipRouter**, čtyři stanice **StandardHost** a objekty **ScenarioManager**, **FailureManager** a **Channel**.
8) Prvky pospojujeme pomocí technologie **ethernet** (gate eth) a rozmístíme podle Obrázek 1.
9) Projekt uložíme.

6.6 Nastavení parametrů simulace

10) Protože RIP protokol nepracuje bez znalosti IP adres přímých sousedů, musíme staticky nakonfigurovat směrovací tabulky pro všechny prvky v síti. Vybereme položku **File > New**... a zvolíme, že chceme vybrat nový textový soubor.
11) Do souboru musíme nadefinovat rozhraní a cesty k sousedům. Příklad souboru pro stanici **standardHost** je na na **Obrázek 3.**

standardHost.irt

```
1 ifconfig:
2 #interface 0 to router
3 name:   eth0  inet_addr: 10.1.7.0   MTU: 1500
4 ifconfigend.
5 #Routing table
6 route:
7 default:    10.1.1.0      0.0.0.0    G   0   eth0
8 routeend.
9
```

Obrázek 3: Formát směrovací tabulky

12) Klikneme na symbol pro uložení(disketa), v dialogovém okně zadáme název souboru (třeba **standardHost.irt**) a klikneme na **Finish.**
13) Předchozí dva body opakujeme pro **všechny aktivní prvky** v síti.
14) Vytvoříme další textový soubor, ale tentokrát ho uložíme jako **scenario.xml**. V tomto souboru nakonfigurujeme scénář výpadku směrovače. Kód musí začínat a končit klíčovým slovem <*scenario*> a následuje definice času a cílového směrovače. Jednu variantu nastavení je možné vidět na **Obrázek 4**.

scenario.xml

```
<?xml version="1.0"?>
<scenario>
    <at t="500s">
        <shutdown module="failureManager" target="ripRouter2"/>
    </at>
    <at t="1200s">
        <startup module="failureManager" target="ripRouter2"/>
    </at>
</scenario>
```

Obrázek 4: Scénář změn v síti během simulace

15) Posledním krokem je vytvoření konfiguračního souboru, kde všechny předcházející soubory napojíme na naši simulovanou síť. Soubor vytvoříme opět kombinací **File** > **New**... a zvolíme, že chceme nový **Initialization file**. Název ponecháme **omnetpp.ini**.
16) Nejprve přiřadíme konfigurační soubor k naší simulační sítí. V menu vybereme položku **General** a do příslušného pole zadáme název naší sítě RIP_network.
17) Vrátíme se do menu **Parameters** a připojíme vytvořené směrovací tabulky. Nejjednodušší způsob je přes tlačítko **Add**. , kde pomocí filtru omezíme nabídku parametru na směrovací tabulky (**routingFile**) a stiskneme **Ok**.
18) Všem záznamům přiřadíme odpovídající vytvořené soubory.
19) Nyní musíme nadefinovat RIP protokol. Většina parametrů je již předdefinována a obsahuje defaultní hodnoty dle RFC1058, [2] ale pokud chceme, můžeme je na tomto místě změnit. Opět klikneme na **Add** a omezíme filtr na parametry modulu rip a potvrdíme **Ok**. Nejdůležitějším parametrem jsou **cílové adresy**(destAddresses), kam přiřadíme adresy **sousedních** uzlů každého směrovače. Nastavení pro naší simulační síť je na Obrázek 5.

omnetpp.ini

```
**.ripRouter.rip.destAddresses = "ripRouter2 ripRouter3"
**.ripRouter1.rip.destAddresses = "ripRouter2 ripRouter4 "
**.ripRouter2.rip.destAddresses = "ripRouter ripRouter1"
**.ripRouter3.rip.destAddresses = "ripRouter ripRouter5 ripRouter4"
**.ripRouter4.rip.destAddresses = "ripRouter3 ripRouter1"
**.ripRouter5.rip.destAddresses = "ripRouter3"
```

Obrázek 5: Nastavení sousedních směrovačů

20) Nyní připojíme vytvořený scénář pro chybu v síti. Vybereme parametr **scenarioManager.scrip** a přiřadíme mu náš .xml soubor.

21) Zbývá nadefinovat provoz v síti. Můžeme vybírat ze široké škály aplikačních protokolů, nebo vytvořit protokol vlastní. V naší úloze použijeme jeden TCP a jeden UDP provoz. Nejprve definujeme UDP. Klikneme na **Add** a vybereme položky **numUDPApps** a **udpAppType**.
22) Zvolíme, že chceme jeden provoz typu **UDPEchoApp**. Přiřadíme mu číslo portů a frekvenci zasílaní.
23) Poté zdrojovému uzlu určíme cílovou IP adresu cílové stanice příkazem **`udpApp[0].destAddresses="stanice"`**.
24) Obdobně nastavíme TCP aplikaci, zvolíme typ **TCPSessionApp**, která komunikuje s cílovou stanicí(serverem) v blocích dat a očekává odpověď o přijetí.
25) Zvolíme čas spuštění protokolu příkazem `tOpen`. Nastavíme ještě IP adresu stanice, na kterou se budeme připojovat. Příklad konfigurace je na obr.
26) **Projekt uložíme**.

6.7 Spuštění simulace

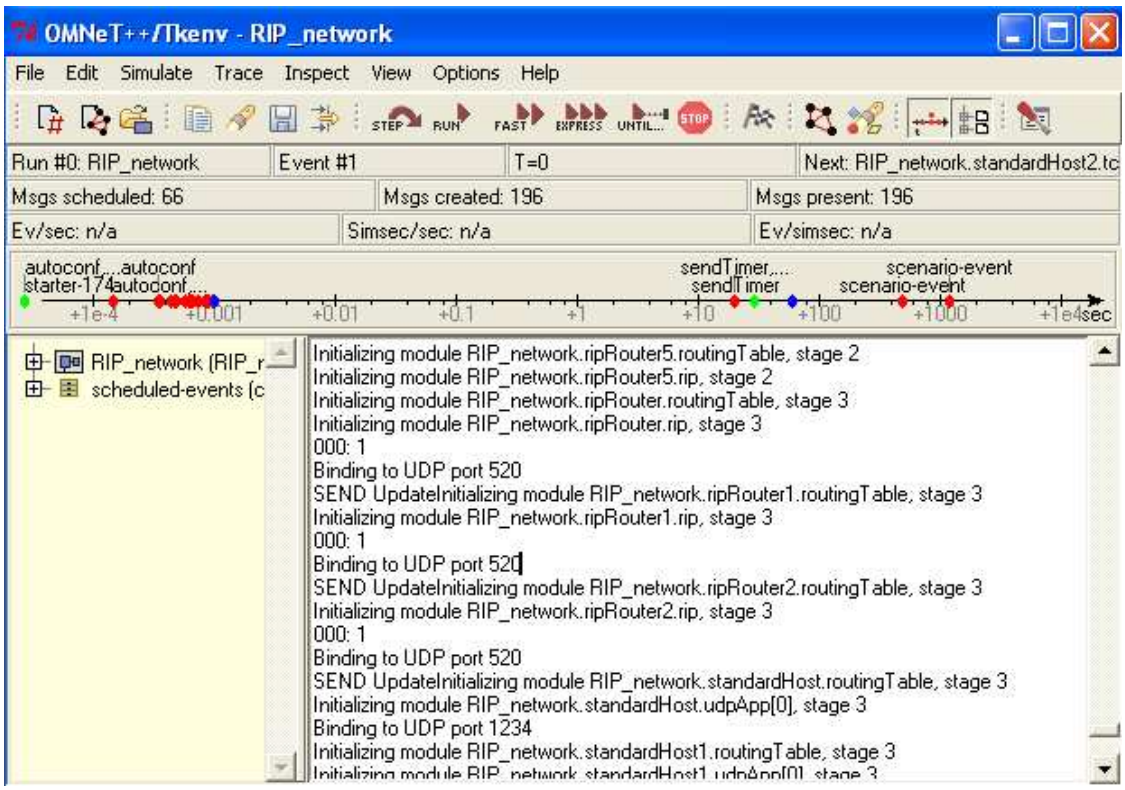
27) Před samotným spuštěním musíme projekt zkompilovat, vybereme **Project > Build Project**. Průběh kompilace můžeme sledovat v okně Console. V případě chyb v kódu je výhodnější přepnout se do okna **Problems**, kde nalezneme detailnější popis vzniklých problémů.
28) Po úspěšné kompilaci můžeme spustit simulaci. Z menu vybereme **Run > Run configuration...**, v otevřeném dialogovém okně zkontrolujeme automaticky nastavené jméno projektu, pracovní složku a konfigurační soubor. V položce default interface zvolíme **Tcl/Tk** pro spuštění simulace v grafickém režimu. Druhou možností je volba **Commnad line,** kdy se projekt spustí v příkazovém řádku a průběh simulace je možné sledovat v okně **Console**.
29) Pro zobrazení sekvenčního grafu průběhu komunikace mezi jednotlivými uzly musí povolit proměnnou **Record EventLog**.
30) Proměnná **Debug on Error** slouží k zastavení simulace v místě chyby(například při volaní časovače do minulosti), a otevření debuggeru pro vyhledání příkazu, který chyby vyvolal.
31) Klikneme na tlačítko **Run** a spustíme simulaci.

6.8 Průběh simulace

Grafické prostředí Tcl/Tk se skládá z těchto částí:

- Panel nástrojů, který se nachází přímo pod hlavním menu okna. Obsahuje tlačítka pro nastavení rychlosti simulace.
- Status labely, které zobrazují aktuální stav simulala. Na horním řádku je název sítě, číslo aktuální události, simulační čas a název modulu, kde se uskuteční příští událost. Další řádek zobrazuje počet zpráv v FES (Future Event List), celkový počet zpráv v průběhu celé simulace a aktuální počet aktivních zpráv.
- Časová osa, která zobrazuje obsah FES. Obsah je možné filtrovat a po dvojitém kliknuti můžeme otevřít inspektor zpráv.
- Panel objektů, kde jsou zobrazeny všechny použité moduly v paměti. Po dvojitém kliku se otevře detail objektu v samostatném okně.
- Log, obsahuje výstup simulace. Obsah je možné filtrovat, aby zahrnoval pouze zprávy konkrétních modulů.

Na rozdíl od Opnetu, kde simulace probíhá převážně na pozadí a my můžeme pracovat až s výstupy, je v OMNeTu možné sledovat simulaci v přímém přenosu a řídit její rychlost. Grafické rozhraní je znázorněno na **Obrázek 6.**



Obrázek 6: Grafické rozhraní Tkenv

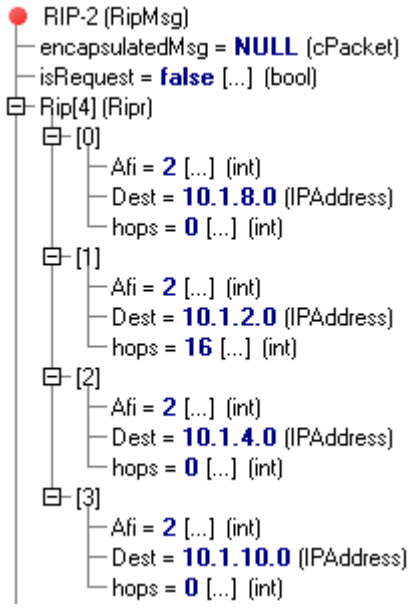
32) Začneme otevřením inspektoru sítě, z panelu nástrojů zvolíme tlačítko **Inspekt Network** a nově otevřené okno obsahuje strukturu sítě. Dvojitým klikem na jednotlivé bloky se můžeme ponořit do hierarchické struktury modulů a sledovat parametry jednotlivých bloků.

33) Nyní spustíme samotnou simulaci kliknutím na tlačítko **Run** z panelu nástrojů.

34) V inspektoru sítě můžeme sledovat výměnu zpráv mezi bloky.

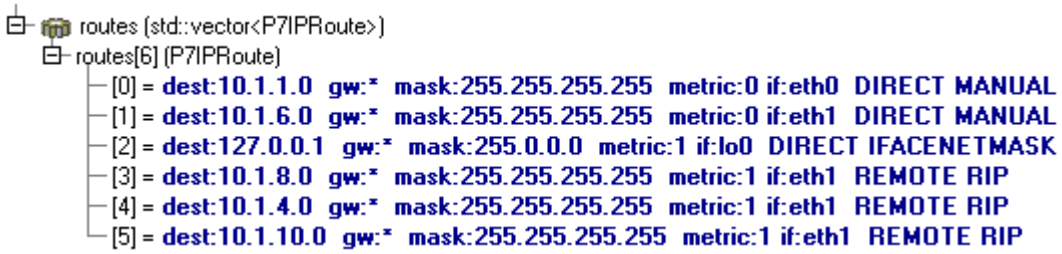
35) Pro nás nejzajímavější jsou **RIPupdate** zprávy a jejich vliv na směrovací tabulky. Dvojitým kliknutím na modul RipRouter2 otevřeme objektový inspektor s jeho vnitřní strukturou, pravým tlačítkem klikneme na jednoduchý modul rip a zvolíme možnost **Fast run until next even in this module**, tím se posuneme v čase až do vygenerování **RipUpdate** paketu.

36) Dvojitým klikem otevřeme zprávu a zobrazíme její parametry. Porovnáme s hodnotami ve směrovací tabulce. Obsah prvního Ripupdate paketu od RipRouteru2 pro RipRouter3 je na Obrázek 7.



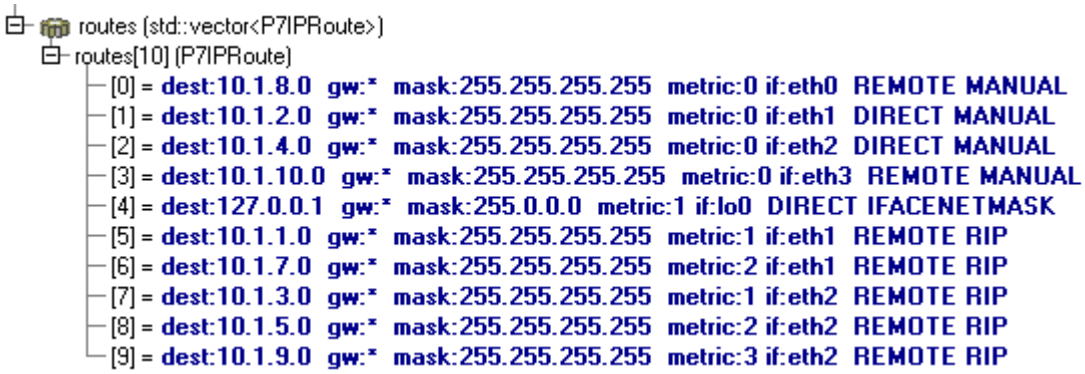
Obrázek 7: RipRouter2-Update

37) Nyní můžeme sledovat cestu paketu síti nebo otevřeme sousední RipRouter3 a opět skočíme v čase až do přijetí našeho paketu. Stiskem klávesy **F4** posuneme simulaci o jeden krok a necháme modul rip zprávu zpracovat. Dochází k zaznamenání třech neznámých adres do směrovací tabulky. Celou směrovací tabulku ukazuje **Obrázek 8**



Obrázek 8: RipRouter3-Směrovací tabulka po přijeti Update

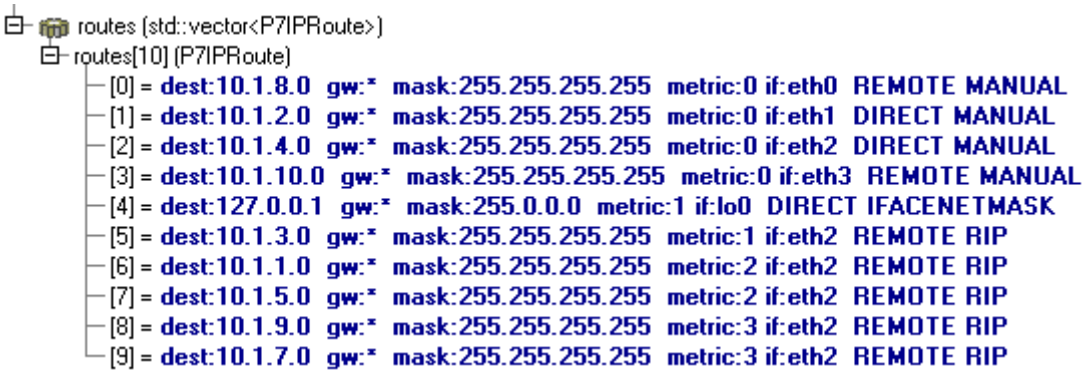
38) Opětovným spuštěním simulace můžeme sledovat výměny zpráv v síti a postupnou konvergenci směrovacích tabulek. Výsledná směrovací tabulka pro směrovač ripRouter3 je na **Obrázek 9**.



Obrázek 9: RipRouter3-úplná směrovací tabulka

39) V čase t = 500s dojde k chybě na směrovači a tento již dál, neodesílá žádné zprávy.

40) V důsledku toho vyprší časovače TimeOut a metriky cesty přes tento směrovač jsou nastaveny na 16 (nedostupné). V tomto okamžiku se začnou v postižených tabulkách uplatňovat informace o původně delší trase, cíl má vyšší metriku. Směrovací tabulka pro směrovač ripRouter1 je na **Obrázek 10**.



Obrázek 10: RipRouter3-směrovací tabulka v době výpadku v síti

41) V čase t =1200s je chyba odstraněna a směrovače po výměně RIP update zpráv, přepočítají trasu na původní (kratší) verzi zobrazenou na **Obrázek 9**.
42) Do vypršení času simulace se již trasa nemění. Z menu vybereme **File** > **Exit** a simulaci ukončíme.

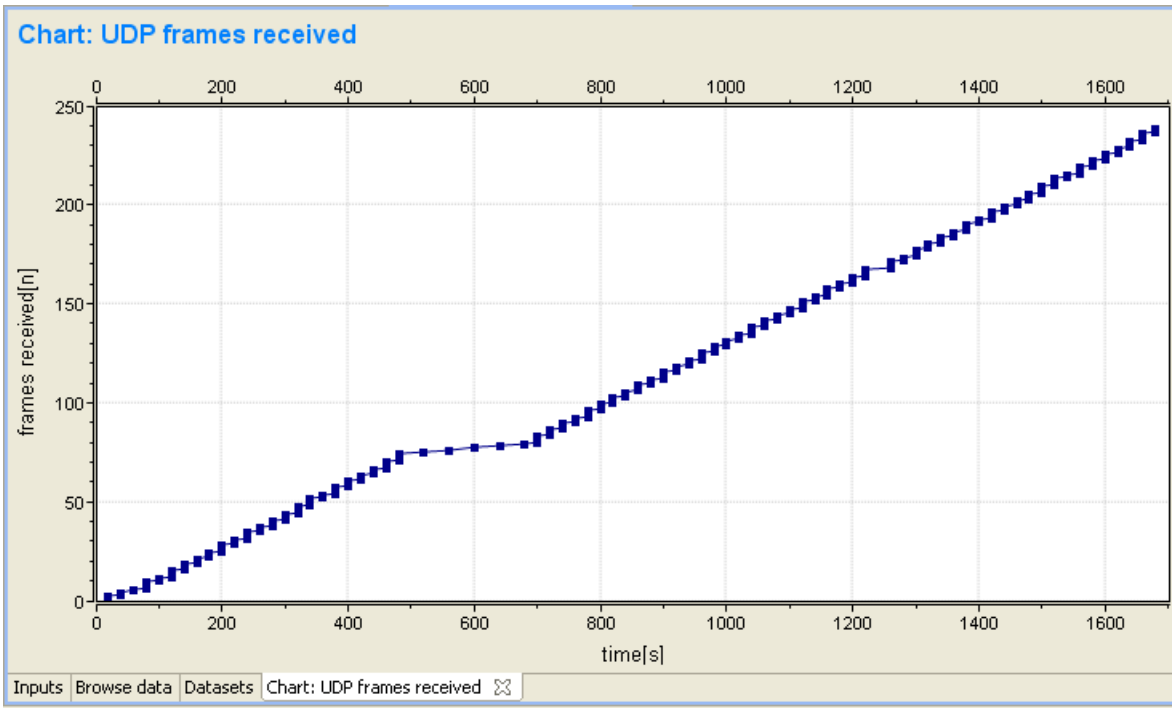
6.9 Výsledky simulace

Přesto, že princip směrovacího protokolu RIP je zřejmý již z průběhu simulace, provedeme pro kontrolu ještě statistickou analýzu simulace. Výsledky simulace jsou zaznamenány jako skalární hodnoty, vektorové hodnoty a histogramy. V OMNeT++ 4.0 je nástroj pro analýzu integrován do prostředí Eclipse.

6.9.1 Vektorová data

Vektorová data obsahuje soubor s příponou .vec. Postup pro tvorbu požadovaného grafu je následující:

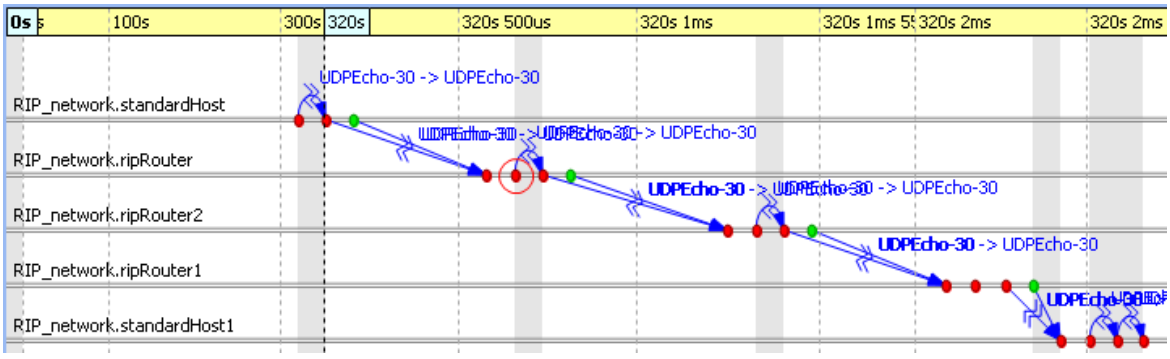
1. Vytvoříme nový soubor pro analýzu, z menu **File** > **New** > **Analysis File**. Zvolíme jméno, umístění souboru a stiskneme **Finish**.
2. Stiskneme tlačítko **Add** a z nabídky vybereme naše vektorová data (Projekt_RIP/Results/**General-0.vec**).
3. Přepneme se do záložky Browse data a vybereme z nabídky modul **standardHost2** a parametr **FramesReceived**. Jedná se o počet přijatých rámců během celé simulace.
4. Pravým tlačítkem klikneme na položku a z nabídky vybereme příkaz **Plot**, který vykreslí charakteristiku do nového okna. Výsledek je možné vidět na
5. **Obrázek 11**.



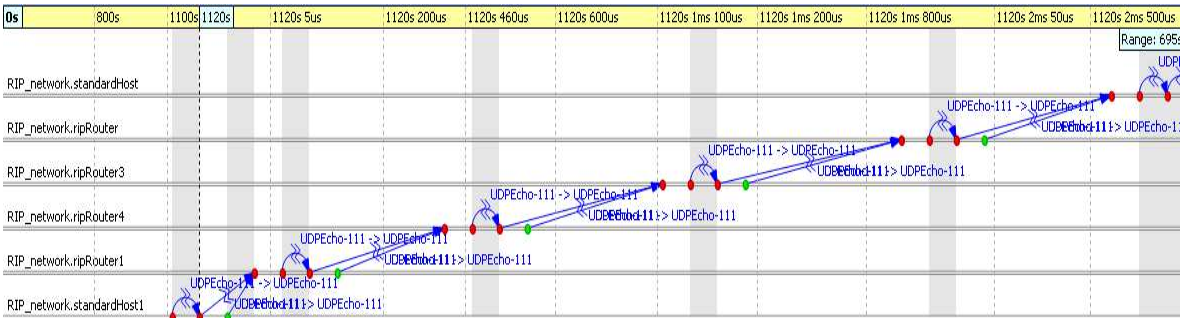
Obrázek 11: standardHost2-Počet přijatých paketů

6.9.2 Eventlog

V souboru s příponou .elog lze najít soupis všech zpráv odeslaných v průběhu simulace s informacemi o událostech, které je vyvolaly. Soubor dále obsahuje informace o topologii modelu a propojeních jednotlivých modulů. Sekvenční graf zobrazuje tyto informace v grafické podobě a pomáhá k pochopení komplexních simulačních modelů a procesů. Soubor hodnot lze filtrovat a zobrazit jen dílčí části. Prostředí je velice intuitivní a proto ho nebudu blíže popisovat. Na **Obrázek 12** je znázorněna cesta UDP paketu sítí v čase t = 320s. Na **Obrázek 13** je pro porovnání znázorněn přenos stejného paketu po rekonfiguraci směrovacích tabulek v důsledku výpadku směrovače RipRouter3.



Obrázek 12: cesta UDP rámce sítí



Obrázek 13: cesta UDP rámce síti po rekonfiguraci tabulek

6.10 Kontrolní otázky a úkoly

1. Analyzujte vliv výpadku linky na provoz aplikačních protokolů. Zdůvodněte vykreslený graf na **Obrázek 11**
2. Popište a vysvětlete, jaký má vliv výpadek směrovače RipRouter3 na směrovací tabulky ostatních směrovačů.
3. Přidejte do simulace výpadek směrovače RipRouter3 v čase 1000 sekund až 1800 sekund od začátku simulace. Analyzujte vliv na směrovací tabulky a aplikační protokoly v síti.

LITERATURA

[1] DOYLE, J., CARROLL, J. *CCIE Professional Development Routing TCP/IP*. Volume I. Indianapolis: Cisco Press, 2005. 936 s. ISBN: 1-58705-202-4

[2] HENDRICK, C. *Routing Information Protocol 1058* [online]. 1988. [cit. 2010-04-06]. Dostupné z URL: <http://tools.ietf.org/html/rfc1058>

[3] MALKIN, G. *Routing Information Protocol 2453 Version 2* [online]. 1998. [cit. 2010-04-08]. Dostupné z URL: < http://tools.ietf.org/html/rfc2453>

[4] *OMNeT++ Discrete Event Simulation Systém - Comunity Site* [online]. [cit. 2010-04-03]. Dostupné z URL: <http://www.omnetpp.org/>

ZÁVĚR

Hlavním cílem této práce bylo zdokumentovat problematiku tvorby nových modelů do simulačního prostředí OMNeT++, konkrétně do jeho rozšíření pro simulaci počítačových sítí INET framework. Práce popisuje vlastní realizaci řešení pro implementaci RIP protokolu. Text je rozdělen do několika částí, které na sebe logicky navazují. V teoretické části bylo nutné se podrobně seznámit s problematikou simulací v prostředí OMNeT++, které představuje diskrétní simulační systém. Skládá se z hierarchicky vnořených modulů, které spolu komunikují pomocí předávání zpráv. Další část se zabývá popisem dynamického směrovacího protokolu, který je v praktické části práce integrován do rozšíření simulačního prostředí INET frameworku.

Autorem projektu je András Varga, který se vývojem tohoto programu zabývá od roku 1992. Jádro systému je naprogramováno v jazyce C++ a je distribuováno pod volnou licencí pro akademické a nekomerční účely. Modulárnost a otevřenost zajistily celému systému značnou oblibu mezi ostatními open-source nástroji pro simulaci datových sítí.

Routing Information Protocol (RIP) je směrovací protokol typu distance vector, jehož metrika je založena na počtu skoků mezi počátkem a cílem.V diplomové práci je řešení realizováno dle starší verze RIP, popsané ve standardu RFC 1058.

V praktické části je definován model RIP routeru, který je založen na již existujícím modelu Router, který nabízí INET framework. Model jsem rozšířil o podporu RIP protokolu, který pracuje na transportním protokolu UDP, port 520. Jedná se o jednoduchý modul, jehož chování popisuje třída Rip.cc. Třída obsahuje funkce pro vysílání a příjem RIP Update zpráv, které si sousední směrovače vyměňují v pravidelných intervalech. Na základě z nich získaných informací počítají nejkratší cestu ke všem ostatním stanicím v síti. Každý záznam v tabulce má přiřazen časovač typu TimeOut, po jehož vypršení se cesta stává neplatnou a je spuštěn proces pro smazání záznamu ze směrovací tabulky. Pro zajištění stability protokolu proti směrovacím smyčkám je použita metoda Poison reverse.

Pro ověření funkčnosti směrovacího algoritmu byla vytvořena experimentální síť, která obsahuje moduly zajišťující dynamické změny v simulační síti-FailureManager a ScenarioManager.

Kromě rozšíření stavající implementace RIP protokolu o další funkce bych další možné pokračování viděl v práci na modelech podporující dynamické chování simulační sítě. Jedná se především o modul FailureManager, kde dosud chybí jakákoliv realizace výpadku konkrétní linky v siti.

Součástí diplomové práce je laboratorní úloha, která demonstruje dosažené výsledky a může studentům posloužit k výuce a procvičení znalostí v oblasti směrovaní v komunikačních sítích.

LITERATURA

[1] *INET Framework for OMNeT++/OMNEST* [online]. [cit. 2009-10-11]. Dostupné z URL: <http://inet.omnetpp.org/doc/INET/neddoc/index.html>

[2] The Eclipse Foundation. *Webové stránky programu* [online]. [cit. 2009-10-11]. Dostupné z URL: <http://www.eclipse.org/>

[3] MOY, J. *OSPF - Anatomy of an Internet Routing Protocol*. New Jersey: Pearson Education Corporate, 2000. 348 s. ISBN 0-201-63472-4.

[4] NOVOTNÝ, V. *Architektura sítí*. Vysoké učení technické v Brně. Fakulta elektrotechnická. Elektronické skriptum, Brno, ČR 2002.

[5] *OMNeT++ Discrete Event Simulation Systém - Comunity Site* [online]. [cit. 2010-04-03]. Dostupné z URL: <http://www.omnetpp.org/>

[6] *INET Framework - Comunity Site* [online]. [cit. 2010-04-05]. Dostupné z URL: <http://inet.omnetpp.org/>

[7] *OMNeT++ API Reference 4.0* [online]. [cit. 2010-04-05]. Dostupné z URL: <http://www.omnetpp.org/doc/api/index.html>

[8] PRATA, S. *Mistrovství v C++*. Praha: Computer Press, 2007. 1120s. ISBN 978-80-251-1749-1

[9] PUŽMANOVÁ, R. *TCP/IP v kostce.* České Budějovice: Kopp, 2000. 620s. ISBN 978-80-7232-388-3

[10] VARGA, A. *OMNeT++ User Manual verze 4.0* [online]. [2009-10-11]. Dostupné z URL: <http://www.omnetpp.org/doc/manual/usman.html>

[11] HENDRICK, C. *Routing Information Protocol 1058* [online]. 1988. [cit. 2010-04-06]. Dostupné z URL: <http://tools.ietf.org/html/rfc1058>

[12] MALKIN, G. *Routing Information Protocol 2453 Version 2* [online]. 1998. [cit. 2010-04-08]. Dostupné z URL: < http://tools.ietf.org/html/rfc2453>

[13] SHAMIN, F., AZIZ, Z., LIU, J., MARTEY, A. *Troubleshooting IP Routing Protocols*. Indianapolis: Cisco Press, 2002. 912 s. ISBN: 1-58705-019-6

[14] DOYLE, J., CARROLL, J. *CCIE Professional Development Routing TCP/IP*. Volume I. Indianapolis: Cisco Press, 2005. 936 s. ISBN: 1-58705-202-4

ZKRATKY

AFI	Authority and Format Identifier
BGP	Border Gateway Protocol
FTP	File Transfer Protocol
GNU	General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
NED	Network Description
NS2	Network Simulator
OSPF	Open Shortest Path First
PDU	Protocol Data Unit
RFC	Request for Comments
RIP	Routing Information Protocol
SCTP	Stream Control Transmission Protocol
STL	Standard Template Library
Tcl/Tk	Tool Command Language/Toolkit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLSM	Variable Length Subnet Masks
XML	Extensible Markup Language

PŘÍLOHA A

Definice modelů

A1. RipRouter.ned

```
import inet.base.NotificationBoard;
import inet.linklayer.ethernet.EthernetInterface;
import inet.linklayer.ppp.PPPInterface;
import inet.networklayer.common.InterfaceTable;
import inet.networklayer.ipv4.RoutingTable;
import inet.util.NAMTraceWriter;
import inet.nodes.inet.NetworkLayer;
import inet.transport.udp.UDP;
import inet.applications.udpapp.UDPEchoApp;
module RipRouter
{
    parameters:
        @node();
        int numUdpApps = default(0);
        string udpAppType = default("n/a");
        string routingFile = default("");
        @display("i=abstract/router");

    gates:
        inout pppg[];
        inout ethg[];
    submodules:
        namTrace: NAMTraceWriter {
            parameters:
                namid = -1; // auto
                @display("p=330,60");
        }
        notificationBoard: NotificationBoard {
            parameters:
                @display("p=60,60");
        }
        interfaceTable: InterfaceTable {
            parameters:
                @display("p=150,60");
        }
        routingTable: RoutingTable {
            parameters:
                IPForward = true;
                routerId = "auto";
                routingFile = routingFile;
                @display("p=240,60");
        }
        networkLayer: NetworkLayer {
            parameters:
                @display("p=228,140;q=queue");
            gates:
                ifIn[sizeof(pppg)+sizeof(ethg)];
                ifOut[sizeof(pppg)+sizeof(ethg)];
        }
```

```
        ppp[sizeof(pppg)]: PPPInterface {
            parameters:
                @display("p=90,257,row,110;q=l2queue");
        }
        eth[sizeof(ethg)]: EthernetInterface {
            parameters:
                @display("p=208,257,row,110;q=l2queue");
        }

        rip: Rip {
            @display("p=42,158");
        }
        udp: UDP {
            @display("p=144,140");
        }


    connections allowunconnected:
        // connections to network outside
        for i=0..sizeof(pppg)-1 {
            pppg[i] <--> ppp[i].phys;
            ppp[i].netwOut --> networkLayer.ifIn[i];
            ppp[i].netwIn <-- networkLayer.ifOut[i];
        }

        for i=0..sizeof(ethg)-1 {
            ethg[i] <--> eth[i].phys;
            eth[i].netwOut --> networkLayer.ifIn[sizeof(pppg)+i];
            eth[i].netwIn <-- networkLayer.ifOut[sizeof(pppg)+i];
        }
        udp.ipOut --> networkLayer.udpIn;
        udp.ipIn <-- networkLayer.udpOut;

        rip.udpOut --> udp.appIn++;
        udp.appOut++ --> rip.udpIn;
}
```

A2. RIP.ned

```
simple Rip
{
    parameters:
        int localPort=default(520);
        int destPort=default(520);
        volatile int messageLength @unit("B")=default(32B); // length of
messages to generate, int bytes
        string destAddresses = default(""); // list of \IP addresses, separated
by spaces
        int UpdateTime @unit("s") = default(30s);
        int TimeOut @unit("s") = default(180s);
        int Flush @unit("s") = default(120s);
        @display("i=block/network2");
    gates:
        input udpIn;
        output udpOut;
}
```

PŘÍLOHA B

Třída Rip

B1 class Rip

```
class Rip : public UDPAppBase
{
  protected:
    std::string nodeName;
    int localPort, destPort;
    int UpdateTime, TimeOutTime , FlushTime;
    std::vector<IPvXAddress> destAddresses;

    static int counter;
    cMessage *R1();
    int numSent;
    int numReceived;
    IRoutingTable *rt;
    IInterfaceTable  *ift;
    std::vector<cMessage*> Flush;
    std::vector<cMessage*> TimeOut;
    std::vector<cMessage*> Update;

  protected:
    // chooses random destination address
    virtual IPvXAddress chooseDestAddr();
    virtual cPacket *createPacket(IPAddress dest);
    virtual void sendPacket();
    virtual void processPacket(cPacket *msg);
    virtual void processTimer(cMessage *msg);
    virtual void saveRoute(cPacket *msg);
    virtual void NewTimer (IPAddress routeIP);
    virtual void ChangeTimer(IPAddress IP, int i);
    virtual int numInitStages() const {return 4;}
    virtual void initialize(int stage);
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};
```

PŘÍLOHA C

C1. Obsah DVD

Součástí práce je DVD s elektronickou verzí tohoto dokumentu ve formátu PDF a zdrojové kódy použitých verzí OMNeTu 4.0 a INET frameworku. DVD obsahuje také zdrojové kody a text Laboratorní úlohy.

Tabulka 5: Struktura DVD

Adresář	Obsah
src/orig-src/	Zdrojové kódy použitých verzí balíčku OMNeT++ a INET framework
src/Rip	Zdrojové kody RIP protokolu
src/Lab/	Zdrojové kódy a text laboratorní úlohy v pdf
/text/	Elektronická verze diplomové práce v pdf.