ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ v Praze

Fakulta elektrotechnická
Katedra řídicí techniky

Simulace real-time komunikace pomocí OPNET Modeler

Diplomová práce

2004 Tomáš CIGÁNEK

ČVUT Praha

Abstrakt

Simulační program OPNET Modeler je vysoce efektivní a výkonný nástroj pro analýzu, návrh a modelování sítí. ORTE je projekt realizující tzv. middleware pro tvorbu real-time aplikace.

Cílem práce je propojit ORTE s OPNET Modelerem a pomocí speciálního rozhraní poskytovaného OPNET Modelerem, realizovat varianty ORTE funkcí pro přístup k síťové vrstvě tak, aby bylo možné namísto do síťové vrstvy systému posílat data přímo do OPNETu.

Dále je úkolem ověřit možnosti spolupráce obecné síťové aplikace a OPNET Modeleru pro případné další využití v projektech realizovaných na katedře řídící techniky ČVUT FEL.

Abstract

Simulation software OPNET Modeler is a high effective and efficient tool for analysis, design and modeling communication networks. ORTE is a project, which implements middleware for design and using real-time aplications.

Goal of this work is to interconnect ORTE with OPNET Modeler by using the special interface provided by OPNET Modeler and realize special versions of ORTE functions which accesses to network layer so they has variant for sending data to OPNET Modeler instead of common system network layer.

Other goal is to check posibility of cooperation of common network application and OPNET Modeler for their posible using in future on Departement of control engineering, CTU FEE in Prague.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne: ..... Podpis: .....

Poděkování

Tímto bych chtěl poděkovat všem, kteří mi při vzniku této práce pomáhali, ať už přímo, nebo nepřímo, zvláště pak Ing. Ondřeji Dolejšovi, vedoucímu mé diplomové práce za podporu a motivaci, dále Ing. Petru Smolíkovi, tvůrci projektu ORTE za ochotu a pomoc při pochopení principu fungování ORTE a práce s vlákny.

A samozřejmě i mojí rodině, která mě materiálně i psychicky podporovala, a v neposlední řadě také moc děkuji své snoubence za trpělivost a podporu, zvláště v závěrečné fázi mé práce.

Obsah

Seznam obrázků

1 Úvod

V dnešní době se s pojmem "síť" setkáváme na každém kroku a to nejen se sítí s velkým "S" - Internetem. Pojem sítě je dnes již široce známý nejen v oblasti průmyslu, kde se setkáváme s pojmy jako *průmyslové sítě, sběrnice, dálkové řízeni* apod., ale začíná pronikat i do domácností.

Velké firmy zabývající se informačními technologiemi představují vize domácností, kde jsou běžné technické prostředky jako televize, telefon, ale i např. lednice nebo mikrovlnná trouba, vzájemně propojeny a sdílejí mezi sebou informace a vzájemně spolupracují.

Je samozřejmé, že vývoj takovýchto, ale i běžných počítačových sítí, je velice nákladný a náročný, a proto byly vyvinuty prostředky pro jejich efektivní návrh, simulaci a analýzu. Jedním z nich je i OPNET Modeler vyvinutý firmou OPNET Technologies Inc.

OPNET Modeler je rozsáhlý a komplexní simulační nástroj s širokými možnostmi návrhu, simulace a analýzy sítí, jimž také odpovídá jeho cena. Proto jsme se po konzultaci s vedoucím diplomové práce a tvůrcem prostředí ORTE[1] dohodli, že by bylo vhodné rozšířit využití tohoto simulačního nástroje na projekty katedry řídící techniky ČVUT FEL, které se zabývají komunikací a kde by tento nástroj našel široké uplatnění. Začneme s využitím OPNET Modeleru pro program postavený na ORTE, s pozdější možností dalšího rozšíření na ostatní projekty katedry.

V kapitole 2 je představena tzv. *real-time* komunikace, její základní vlastnosti a požadavky na prostředí. Na tento úvod navazuje představení protokolu pro implementaci *real-time* aplikací a jeho praktické využití v projektu ORTE (kapitola 3).

Dále je v kapitole 4 představen simulační nástroj OPNET Modeler. Informace vychází z mých praktických zkušeností a poznatků a týkají se převážně oblastí, které byly v této práci využity (celá dokumentace k OPNET Modeleru verze 10.0 má 113MB).

V kapitole 5, je podrobně popsán návrh simulačního modelu v OPNET Modeleru a implementace rozhraní programu založeném na ORTE pro propojení s OPNETem.

V předposlední části (kapitola 6), je shrnut celý způsob vzájemné komunikace mezi OPNET Modelerem a externí aplikací s ohledem na různé možnosti této komunikace.

Na úplný závěr je pak v kapitole 7 implementován jednoduchý model s ukázkou funkčnosti a vytvořenými statistikami.

[1] OCERA Real-Time Ethernet (viz. kapitola 3 a [7])

2 Real-Time

2.1 Real-Time řízení a komunikace

Pro vysvětlení významu pojmu Real-Time[2] komunikace použiji příklad z běžného života. Pokud máme zájem s někým komunikovat, někomu něco sdělit, nebo získat požadovanou informaci, můžeme dnes použít jeden z mnoha dostupných komunikačních prostředků. Pošlu dopis, email, SMS[3] zprávu, případně informaci předám někomu dalšímu k vyřízení.

Toto jsou ovšem příklady tzv. Non-Real-Time[4] komunikace. Zpráva je odeslána, a já nevím kdy, a zda vůbec někdy, dorazí na místo určení. Samozřejmě většinou máme možnosti a prostředky k tomu, abychom mohli doručení zprávy ověřit, případně sledovat její cestu, ale obecně nemáme nikdy 100% jistotu, že zpráva dojde do nějakého předem stanoveného času, např. do jedné hodiny, od odeslání.

Na druhou stranu, ale lze použít prostředků, které nám umožní předat zprávu prakticky okamžitě a zároveň můžeme kontrolovat její přijetí a správné zpracování. Příkladem je používání telefonu, různých počítačových programů pro komunikaci v reálném čase, či obyčejný rozhovor mezi dvěma lidmi. Takovéto komunikaci pak můžeme říkat Real-Time.

2.1.1 Real-Time v řídící technice

V řídící technice je otázka rychlosti a spolehlivosti komunikace velice důležitá, neboť při výpadku řízení může často dojít k velkým škodám na řízené technologii, výrobcích apod.

V zásadě můžeme rozdělit komunikaci v řízení na tři základní skupiny:

- komunikace *klasická* (Non-Real-Time)
- komunikace *soft real-time*
- komunikace *hard real-time*

Model *klasické* komunikace spočívá v principu popsaném na začátku kapitoly 2.1. Obě strany si vyměňují zprávy a nemají žádné speciální požadavky na rychlost komunikace, nanejvýš si ověřují zda byla zpráva přijata.

Model *soft real-time* již splňuje vlastnosti real-time komunikace, což znamená, že požaduje maximální zaručené zpoždění při doručení zprávy.

Model *hard real-time* je v principu shodný s modelem *soft real-time*, pouze maximální povolené zpoždění doručení bývá zhruba desetkrát až stokrát menší než u *soft real-time* modelu.

[2] Real-Time komunikace – komunikace v reálném čase

[3] Short Message - označení krátké zprávy

[4] Tímto pojmem je myšlen opak k Real-Time komunikaci

2.1.2 Real-Time a Ethernet

Lze využít běžného média, jako je Ethernet[5], k Real-Time řízení a pokud ano tak jak?

Ethernet je definován jako mezinárodní standard komunikace na úrovni fyzické a linkové vrstvy ISO/OSI modelu. (viz. obr.2.1). V dalším textu bude pod pojmem *ethernetovská síť* myšlena síť fungující na bázi Ethernetu.

Obrázek 2.1: ISO/OSI model síťových vrstev

V sítích založených na Ethernetu se pro přístup k přenosovému médiu používá metoda CSMA/CD[6]. Základní princip spočívá v tom, že zařízení, které chce vysílat data, "poslouchá" provoz na síti a pokud je "volno" začne odesílat, přičemž zpětným poslechem kontroluje, zda nedošlo ke kolizi, způsobené současným vysíláním více stanic v jednom okamžiku. (podrobněji v [3]).

Základním komunikačním standardem je v prostředí ethernetových sítí IP[7] protokol (blíže v [2]). Tento protokol se při svém vzniku ukázal jako velice výhodný a efektivní pro mnoho různých oblastí komunikace využívající ethernetovské sítě. Využití tohoto protokolu pro real-time se ale ukázalo nepříliš vhodné, protože tento protokol využívá HW a SW navržený pro nedeterministický Ethernet.

[5]původ slova Ethernet má kořeny ve slově éter (viz.[4])
[6]Carrier Sense Multiple Access/Collision Detection
[7]Internet Protocol

Na druhou stranu má IP protokol výhodu v tom, že ho lze využívat na mnoha různých médiích a dnes se již stává dostupným i pro rychlé a deterministické architektury jako je FireWire[8]. Stále je však většina HW a SW, které jsou dostupné pro IP, navržena pro nedeterministické ethernetovské sítě.

Jak již bylo zmíněno, pro přístup k přenosovému médiu se využívá metody CSMA/CD, což je metoda výhodná v tom, že zefektivňuje využití sběrnice a minimalizuje případné kolize vzniklé současným vysíláním více nódů[9], na druhou stranu ale tato vlastnost způsobuje, že komunikace není deterministická, protože i když víme, že data budou na síť odeslána v pořádku, tak nikdy nemůžeme přesně říci, za jak dlouho po našem požadavku na odeslání se tam ta data objeví (při velkém provozu na síti je více kolizí a tím se zvyšuje počet pokusů, které jsou nutné k úspěšnému odeslání dat).

Toto je závažný problém pro případné využití ethernetovských sítí pro real-time komunikaci, neboť při real-time komunikaci je jedním ze základních požadavků při přenosu zpráv přesné určení maximálního časového limitu, do kterého má zpráva dorazit k cílovému nódu - tzv. *delay* zprávy.

Jak je ale tedy možné, že na omezených sítích, založených na Ethernetu, jsou provozovány aplikace, které si vyměňují zprávy rychlostí řádově v kHz ?

Na základě znalosti metody CSMA/CD a znalostí z oblasti výpočtu pravděpodobnosti náhodných jevů, lze (dle [1]) za přijetí několika zjednodušujících podmínek, určit za jak dlouho po začátku vysílání nastane kolize

$$T_{err} = \frac{M}{\lambda} \tag{2.1}$$

kde T_{err} je čas od začátku vysílání do první kolize, M je počet paketů u nichž očekáváme, že budou přeneseny v čase menším než určený T_{max} a λ je počet odesílaných paketů za sekundu. Pro M platí rovnice

$$M = \frac{\ln(P_{OK})}{\ln(1 - P_{err})} \tag{2.2}$$

kde P_{OK} je pravděpodobnost, že paket bude odeslán se zpožděním menším, než je T_{max} , P_{err} je pravděpodobnost chyby (více informací viz. [1]).

Na základě těchto výpočtů lze získat následující tabulku, která popisuje vztahy mezi rychlostí zasílání zpráv, jejich velikostí, šířkou pásma, maximálního požadovaného časového zpoždění a dobou bezkolizního provozu.

[8]norma komunikačního protokolu dle IEEE 1394

[9]Nódem (ang. Node) budu označovat obecný prvek, který se účastní komunikace.

Bandwidth (Mbits/sec)	Packet Size (bytes)	Message Rate	Tmax (ms)	Time
100	128	1000	2	293K yrs
100	128	1000	1	1140 yrs
100	128	1000	0.5	9 yrs
100	1024	1000	2	604 yrs
100	1024	1000	1	2 yrs
100	128	5000	1.5	483 yrs
10	64	1000	7	9 yrs
10	64	1000	2	10 hrs
10	128	250	4	2 yrs
10	128	1000	7	1 yr
10	128	1000	2	1 hr
10	128	500	3.5	53 days
10	1024	100	8	23 yrs

Obrázek 2.2: Tabulka časů, pro které máme 99% jistotu, že síť bude fungovat bez překročení daného času zpoždění T_{max} (Zdroj: [1])

Např. třetí řádek vyjadřuje, že na sítí, s přenosovou kapacitou 100Mbit/sec, pakety o velikosti 128 bytů, posílané 1000x za vteřinu, nebudou mít po dobu devíti let na 99% větší zpoždění, než 0.5ms.

Vidíme tedy, že pokud nám stačí maximální zpoždění 7 ms, můžeme na 10MBitovém Ethernetu provozovat real-time aplikaci a máme 99% jistotu, že k chybě nedojde dříve, než za rok jejího provozu.

2.1.3 Middleware a komunikační modely

V této kapitole je definován pojem síťové vrstvy middleware a základní komunikační architektury na této vrstvě.

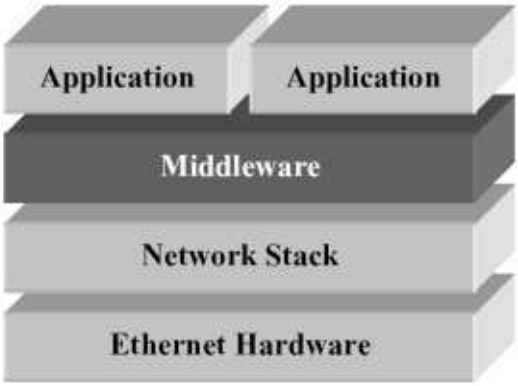
Network middleware[10] (dále už jen middleware), je mezivrstva mezi nižšími vrstvami síťového ISO/OSI modelu a vrstvou aplikační.(viz. [2]) (obrázek 2.3).

Jde o většinou o speciální software jehož základní funkce spočívá v tom, že poskytuje aplikacím, které chtějí komunikovat po síti, standardizované rozhraní, tzv. API[11], pomocí něhož lze jednoduše programovat síťové aplikace i bez hlubších znalostí principů fungování síťové komunikace na nižších úrovních.

Middleware obsahuje sadu funkcí, které lze rozdělit do několika základních kategorií:

[10] Můžeme volně přeložit jako " Síťový mezisoftware "

[11] APlication Interface



Obrázek 2.3: Middleware síťová vrstva je vložena mezi aplikace a nižší vrstvy síťového modelu.(Zdroj: [1])

- inicializační funkce (základní nastavení spojení, přidělení portů apod.).
- funkce pro odesílání (a případnou kontrolu a zpracování chyb).
- funkce pro přijímání (a opět pro možnou kontrolu chyb nebo předzpracování přijatých dat).
- funkce pro ukončování spojení (ukončuje spojení, uvolňuje porty apod.).
- ostatní funkce (různé pomocné funkce např. pro různé formátování a převody dat apod.).

Základní komunikační architektury na vrstvě middleware jsou:

- Point-to-Point
- Client-Server
- Publish-Subscribe

Point-to-Point: Architektura point-to-point je nejjednodušším způsobem síťového spojení. Klasický příklad je telefonování. Pro navázání spojení musíme znát adresu (telefonní číslo) stanice, s kterou chceme komunikovat. Jakmile je spojení navázáno, obě stanice jsou schopné oboustranné širokopásmové komunikace. (obr. 2.4).

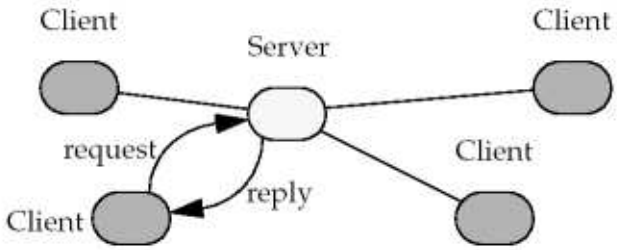
Obrázek 2.4: Spojení point-to-point.

Omezení tohoto způsobu je, že pokud začne komunikovat více stanic, celá síť se velmi rychle zahltí. Tato architektura byla použita při návrhu protokolu TCP[12] (viz.[2]) a zde se právě projevilo zmíněné omezení.

Client-Server: Tato architektura se stala módním pojmem v osmdesátých letech dvacátého století. Sítě založené na architektuře client-server obsahují speciální nód, tzv. server, který je schopen současně komunikovat s více nódy - klienty. (obr. 2.5)

Pro tento komunikační model existuje několik implementací middleware, z těch novějších je to např. CORBA[13] a DCOM [14] (bližší informace viz. [5])

Tato architektura je výhodná v oblastech jako jsou databázové a transakční systémy nebo systémy pro centralizované shromažďování souborů. Pokud jsou ale informace (data) generovány ve velkém množství a velkým počtem nódu, tak vzniká problém, protože tato architektura vyžaduje, aby veškerá data byla poslána nejprve na server pro jejich další distribuci což není příliš efektivní. Také je zde neurčitost v podobě neznámé délky zpoždění pro doručení dat (nevíme jak je server zatížen a kdy se dostane k rozeslání našich dat ostatním nódům).



Obrázek 2.5: Spojení typu client-server.

Publish-Subscribe: Tato architektura se začala ve větší míře používat v devadesátých letech. V tomto komunikačním modelu rozlišujeme dva typy nódů:

- *Publisher*
- *Subscriber*

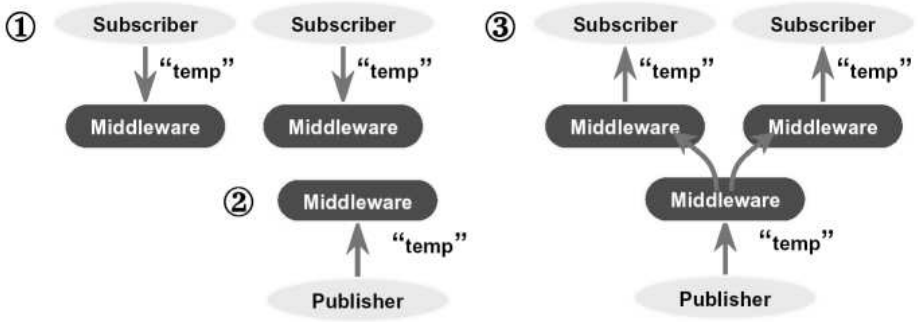
[12]Transmission Control Protocol
[13]Common Object Request Brooker Architecture
[14]Distributed extension of the Component Object Model

Publisher pomocí middleware přihlásí do sítě data která bude publikovat - poskytovat ostatním nódům a naopak *Subscriber* si pomocí middleware zaregistruje, která data chce přijímat. Příkladem v běžném životě je např. televize, noviny apod. Systémy založené na tomto komunikačním modelu jsou schopné doručovat velké množství dat s požadavkem na minimální časovou ztrátu při přenosu, dokonce i v případě, že samotný doručovací mechanismus zpráv není příliš spolehlivý.

Tento model poskytuje jednoduchý a efektivní přístup k datům. Jednotlivé nódy si pomocí middleware zaregistrují data, která chtějí přijímat, a systém jim je doručí. Nebo si naopak zaregistrují data, která chtějí publikovat ostatním nódům, a systém se postará o jejich distribuci k těm, kdo je vyžadují.

Model dokáže efektivně řídit a obsluhovat různorodé datové toky, jednoznačně určit jejich zdroje a cíle a poskytuje mechanismy pro eliminaci chyb komunikace a dalších nenadálých změn v sítí - např. je možné za provozu odebrat nód aniž by došlo k závažné chybě (to se týká samotné komunikace, samozřejmě pokud tento nód jako jediný publikoval důležitá data, pak to závažná chyba je) protože nód není jako v předchozích případech přímo spojen s jiným, ale pouze pomocí middleware publikuje svá data.

Funkci *Publish-Subscribe* komunikace ilustruje následující obrázek 2.6.



Obrázek 2.6: Publish-Subscribe model komunikace. (1) Nódy typu Subscriber se registrují pomocí middleware pro příjem publikace s názvem "temp". (2) Nód typu Publisher registruje pro poskytování publikaci s názvem "Temp". (3) Publikace "Temp" je doručena všem přihlášeným nódům. (Zdroj: [1])

Celkově lze tvrdit, že architektura *Publish-Subscribe* je velice dobrým řešením problému real-time komunikace. Nejsou zde žádné požadavky na data a datové toky mezi dvěma konkrétními nódy, které by zatěžovali síť jako v případě architektury *Client-Server*, což značně zvyšuje efektivitu tohoto modelu.

S určitým stupněm parametrizace je tento model schopen realizovat podporu pro různé typy datových požadavků real-time systémů, jako je kontinuální spojení, spolehlivé doručování stavových informací nebo skupinová komunikace.

2.2 Publish-Subscribe pro Real-Time komunikaci - RTPS

V předchozí kapitole byly ukázány výhody architektury *Publish-Subscribe* oproti ostatním (*Point-to-Point* a *Client-Server*). Avšak tak, jak byla architektura *Publish-Subscribe* popsána, požadavkům pro real-time komunikaci nepostačuje. Uvedeme několik dalších důležitých požadavků na tuto architekturu, aby ji bylo možno v real-time sítích použít.

Jednou ze základních vlastností real-time komunikace, je možnost kontroly vztahu mezi požadavkem na spolehlivost doručení zprávy a požadavkem na velikost zpoždění pří doručení, protože pokud požadujeme velkou spolehlivost doručení zprávy, potom je pravděpodobné, že se prodlouží doba pro její doručení - díky případnému opakování zprávy způsobeném chybami (ztráta paketů apod.). Je proto důležité mít možnost si tyto parametry nastavit pro každý typ zpráv.

Například pokud potřebujeme posílat aktuální hodnotu teploty v kotli, je třeba aby byla aktuální hodnota zaslána vždy co nejdříve a nemá smysl snažit se opakovaným posíláním doručit jednu hodnotu když aktuální stav už většinou bývá jiný. A naopak, pokud posíláme sekvenci řídících příkazů, je nanejvýš důležité, aby byly doručeny všechny části posloupnosti, byť s jistým zpožděním. Samozřejmě ve většině případů je nutné zvolit kompromis při definování maximálního počtu pokusů pro opakované vysílání dat při dodržení časového limitu pro doručení.

Bohužel většina současných síťových protokolů takovéto parametry u odesílaných zpráv nastavovat neumožňuje. Např. protokol TCP se pokouší doručit chybný paket velmi dlouho, řádově až minuty, a odmítá jakékoliv další pakety. Oproti tomu, např. protokol UDP[15] odesílá pakety, aniž by kontroloval, zda byly doručeny, což je z hlediska zpoždění výhodné, ale je nutné, aby se o kontrolu doručení dat starala aplikační vrstva.

Pro úspěšné využití *Publish-Subscribe* architektury je definován tzv. RTPS[16] komunikační protokol. Základní požadavky, které jsou na tento protokol kladeny jsou:

- Schopnost modelování času a časování každé události.
- Umožnění aplikaci definovat vlastní úroveň časování a spolehlivosti doručení pro odeslání každé zprávy.
- Kontrola a specifikace využívané paměti.
- Schopnost fungování v prostředí RTOS[17]

[15] User Datagram Protocol
[16] Real-Time Publish-Subscibe
[17] Real Time Operation Systems - prostředí Real-Time operačních systémů.

RTPS umožňuje aplikaci vyváženě nastavit všechny potřebné parametry jako je časování, doručitelnost dat i využití paměti. V RTPS rozlišujeme dva základní komunikační prvky pro vzájemnou výměnu dat (další komunikační prvky definované pro RTPS budou popsány v kapitole 3, která popisuje praktickou implementaci protokolu RTPS)

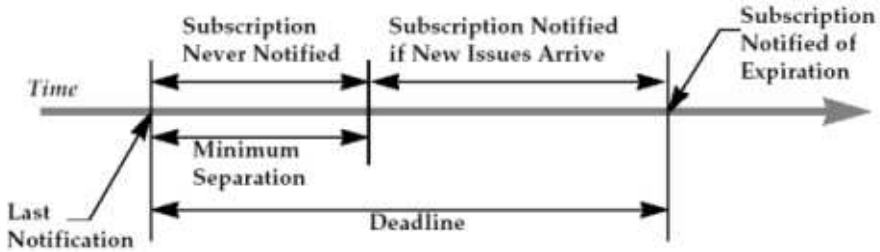
RTPS Subscriber: Subscriber se registruje pro odebírání vždy jedné publikace, což je specifická datová položka určitého typu např. hodnota čidla, alarm nebo řídící příkaz, jejíž hodnota se periodicky obnovuje.

Každý Subscriber si pro každou publikaci nastavuje dva základní parametry týkající se časování zpráv. Minimální vzdálenost (*minimum separation*) a vypršení limitu (*deadline*).

Minimální vzdálenost je minimální časový úsek mezi jednotlivými updaty dané publikace. Je ovlivněna jednak rychlosti, s jakou je Subscriber schopen zpracovávat nová data publikace, a jednak důležitostí publikace, méně důležité hodnoty není třeba aktualizovat tak často.

Naopak vypršení limitu označuje maximální čas od posledního přijetí zprávy, do kterého je očekávána aktualizace, než bude oznámena chyba. Každá publikace tak má zaručen časový úsek, ve kterém očekává aktualizovaná data.

Samozřejmě každý Subscriber, který přijímá jednu konkrétní publikaci, může mít tyto hodnoty nastavené jinak, například zobrazovací jednotce stačí údaj o teplotě aktualizovat každou vteřinu, zatímco řídící jednotka potřebuje aktuální údaj každých sto milisekund.



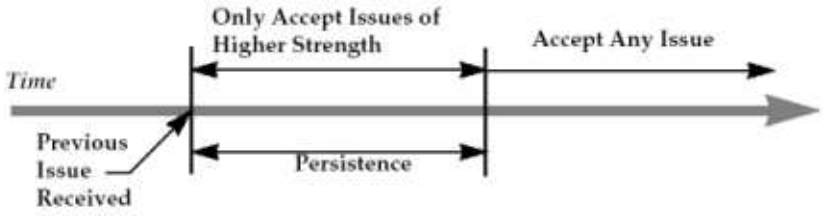
Obrázek 2.7: Obrázek ilustrující pojmy *Minimum Separation* a *Deadline* (Zdroj: [11])

RTPS Publisher: Publisher produkuje data pro jednotlivé publikace. Oproti Subscriberu, Publisher neovlivňuje rychlost publikování aktuálních dat a publikuje je tak, jak je má k dispozici (např. čidlo s převodníkem nemůže vydávat nová data rychleji než každých 10ms) a požadované časování zpráv si nastavují jednotliví Subscribeři.

V síti muže být samozřejmě více Publisherů, kteří poskytují shodná data (například záložní snímače apod.), takže je třeba u Publishera definovat ještě pojmy síla (*strenght*) a trvání (*persistence*).

Síla určuje prioritu publikace, tedy pokud jsou publikována stejná data od dvou různých Publisherů, pak v síti jsou platná pouze ta, která mají větší *strenght*. Trvání označuje dobu platnosti dat, tedy jak dlouho jsou konkrétní data platná.

Tento mechanismus lze využít pro efektivní zálohování, za normálního provozu publikuje data Publisher s vyšší hodnotou *strenght*, pokud dojde k výpadku, tak po vypršení času daného *persistence* začnou být platná data od Publishera s nižší *strenght*. Pokud dojde k zpětnému obnovení prvního Publishera, pak tento, díky vyšší *strenght*, převezme zasílání aktuálních dat. Tedy vhodným nastavením *persistence* můžeme zajistit plynulé zálohováni, bez přerušení provozu sítě.



Obrázek 2.8: Obrázek ilustrující pojmy *Persistence* a *Strenght* (Zdroj: [11])

V této části byly vedeny některé, ze základních vlastností komunikačního modelu *Publish-Subscribe*, včetně jeho možností v prostředí real-time. V následující kapitole jsou, představeny základní vlastnosti a funkce hotové implementace vrstvy *middleware* - ORTE, která vychází právě z definice RTPS modelu.

3 ORTE

V této kapitole jsou popsány základní vlastnosti protokolu *Publish-Subscribe* a dále základní prvky, vlastnosti a funkční vztahy implementace mezi-vrstvového rozhraní(*middleware*) ORTE patřící pod projekt OCERA.[18]

3.1 Protokol *Publish-Subscribe*

Základní vlastnosti komunikačního modelu *Publish-Subscribe* jsou naznačeny ve čtyřech bodech:

- Distribuce vydání pojmenovaných publikací.
- Jednoznačná deklarační a doručovací fáze.
- Přenos řízen událostmi (a nikoliv časem).
- Otevřená komunikace typu *one-to-many*[19].

Tyto vlastnosti jsou v následujících odstavcích popsány detailněji.

Publish-Subscribe model komunikuje pomocí vysílání a přijímání tzv. vydání pojmenovaných publikací (issues of named publications). Nejlépe je tento způsob srovnatelný s rozesíláním jednoho vydání (issue) daných novin (named publication) Každé vydání v sobě zahrnuje následující položky:

- *topic*: identifikátor publikace, které dané vydání patří.
- *type*: datový typ, který je vydáván.
- *issue identification*: jednoznačný identifikátor vydání.
- *user data*: uživatelská data.

Počátek komunikace se skládá ze třech fází:

- Deklarace zájmu *publishera* o publikování dat (přidáno nové čidlo teploty).
- Deklarace zájmu *subscribera* o odebírání dané publikace (panel zobrazování teploty).
- Začátek zasílání požadovaných dat.

[18] Open Components for Embedded Real-time Applications viz.[9]
[19] jeden zasílá data pro mnoho jiných.

Tyto tři fáze lze volat v libovolném pořadí a jsou mnohokrát opakovány, protože např. *subscriber* má zájem o určitá data ještě dříve než je nějaký *publisher* nabídne k odebírání a je třeba aby i později přidané nódy typu *subscriber* měly přehled o nabízených publikacích.

Přenos dat je tzv. *event driven*, tedy řízen událostmi. Subscriber, pro kterého jsou určena aktuálně vysílaná data, může přijetí dat provést dvěma způsoby.

První způsob je, že ihned po příchodu vydání publikace, je přijímací aplikace upozorněna pomocí speciální uživatelsky definované funkce a může data dále zpracovávat.

V druhém případě může middleware dočasně uložit došlá vydání a zpracovat data až později.

Tento komunikační model ze své podstaty nabízí distribuci dat způsobem *one-to-many*, tedy jeden publisher publikuje svá data a už nemusí vědět (a také ani neví), kolika subscriberům jsou tato data doručena. Stejné je to i z hlediska subscriberu. Ten neví, a ani nepotřebuje vědět, v které části sítě je zapojen publisher od něhož přijímá data. Toto je jedna z největších výhod tohoto modelu, můžeme za provozu a bez přerušení funkčnosti celé komunikace, přidávat a odebírat nódy.

3.2 ORTE jako implementace RTPS

V této části je rozšířen popis vlastností a možností protokolu RTPS, definovaných dříve, o možnosti, které přináší implementace ORTE a dále je upřesněno jaké druhy komunikace můžeme využívat.

K parametrům vydání konkrétní publikace definovaným v předcházející části přidáme:

- *Topic* - identifikátor vydání konkrétní publikace (např. číslo novin).
- *Type* - typ přenášených dat.

Druhy komunikace aneb, jak si lze předávat data.

Unidirectional (Send-Receive) Communication
Jde o nejběžnější způsob komunikace, pro všechny aplikace, využívající Publish-Subscribe model. Zdroj informací (Publisher) informace pouze publikuje, ale nepřijímá a naopak příjemce (Subscriber) data pouze přijímá.

Na straně Publishera rozlišujeme ještě jednu vlastnost publikace a to je **Time to keep**. Tato vlastnost určuje, jak dlouho je dané vydání publikace Publisherem udržováno pro případné nově připojené Subscribery. Ti totiž žádná předchozí data nedostali a tak nemají nastavenu ani minimální vzdálenost ani vypršení.

Bidirectional (Request-Reply) Communication
Zde se jedná o komunikaci podobnou modelu klient-server. Často je potřeba komunikovat tzv. se zpětnou vazbou tedy Publisher potřebuje vědět, zda a kdy požadovaná data dorazila. To je například pokud posíláme sadu hodnot, u které musí každá část dorazit k příjemci ve správném pořadí.U těchto zpráv tohoto typu komunikace je samozřejmě nutné definovat dodatečné vlastnosti jednotlivých publikací jako je např. maximální doba odezvy příjemce (a samozřejmě i odesílatele), typ požadavku a typ odezvy apod.

Z hlediska způsobu komunikace můžeme ještě u Publishera definovat módy:

- *Synchronní.* Informace by měla být poslána okamžitě a v rámci kontextu dané aplikace.
- *Signálový.* Požaduje aby data byla poslána ihned, ale skrz jiné vlákno middleware vrstvy. Tento mód se používá v případě že aplikace provádí časově kritické operace a nemůže (nechce) být zpožďována odesíláním dat.
- *Asynchronní.* Definuje, že by data neměla být odesílána okamžitě, ale dočasně uložena. O odeslání se potom postará definované vlákno middleware. Tento způsob je užitečný pokud chceme posílat větší množství dat pomocí několika síťových transakcí.

U Subscribera rozlišujeme dva módy:

- *Okamžité přijetí.* Vydání je aplikací na straně Subscribera přijato okamžitě bez prodlev.
- *Vyzvedávání.* Aplikace si vyzvedne přijatá vydání z dočasného uložení až po zavolání specifické funkce. Tento mód je výhodný pro aplikace, které nemají dobrou mezi vláknovou bezpečnost nebo nemohou jednoduše zpracovávat událost vyvolanou příchodem dat.

3.3 Základní objekty ORTE

Jelikož, je celý protokol RTPS založen na objektech, musíme nyní nadefinovat objekty a jejich vlastnosti, které budeme dále používat. Tyto objekty popisují typ zasílaných dat, způsob jakým mají být zasílány apod. Existují čtyři základní typy objektů:

- *Manager (M)* Speciální objekt, jehož důležitým úkolem je vyhledávat další managery v sítí a komunikovat s nimi.
- *ManagedApplication (MA)* Aplikace, jejíž činnost je řízena jedním nebo i více Managery.

- *Writers (Publication, CSTWriter)*[20] Tento typ aplikace poskytuje základní přenášená data.
- *Readers (Subscription, CSTReader)* Získávají ze sítě informace poskytované aplikacemi typu Writers.

Manager je nezávislý proces, vytvořený při startu celého programu. Jde o speciální aplikaci, která napomáhá ostatním aplikacím vyhledávat se navzájem, manager[21] si udržuje záznamy o všech dalších aplikací, které ovládá, a o jejich atributech. Pro tuto komunikaci má manager vždy speciální aplikaci typu CSTWriter.Kdykoliv se tedy k manageru přihlásí nová aplikace, manager mj. zjistí její atributy a parametry a rozešle je všem známým nódům v síti.

Typ Publication je využívaná k rozesílání dat odpovídajícím aplikacím typu Subscription. CSTWriter a CSTSubscriber jsou ekvivalenty aplikací typu Publication a Subscription, ale jsou využívány pro distribuci systémových zpráv pomocí stavově synchronizovaného protokolu.

Každý konkrétní manager se skládá z pěti objektů:

- *WriterApplicationSelf* Objekt typu CSTWriter, skrze nějž Manager poskytuje informace sám o sobě ostatním nódům v síti.
- *ReaderManagers* Typ CSTReader pomocí kterého Manager získává informace o ostatních Managerech v síti.
- *WriterManagers* Typ CSTWriter, který poskytuje všem aplikacím přihlášeným k tomuto Manageru informace o všech dostupných Managerech v síti.
- *ReaderApplications* CSTReader používaný pro registraci lokální nebo vzdálené aplikace k tomuto Manageru.
- *WriterApplications* CSTWriter kterým Manager posílá ostatním Manageům v síti informace o aplikacích, které má zaregistrovány.

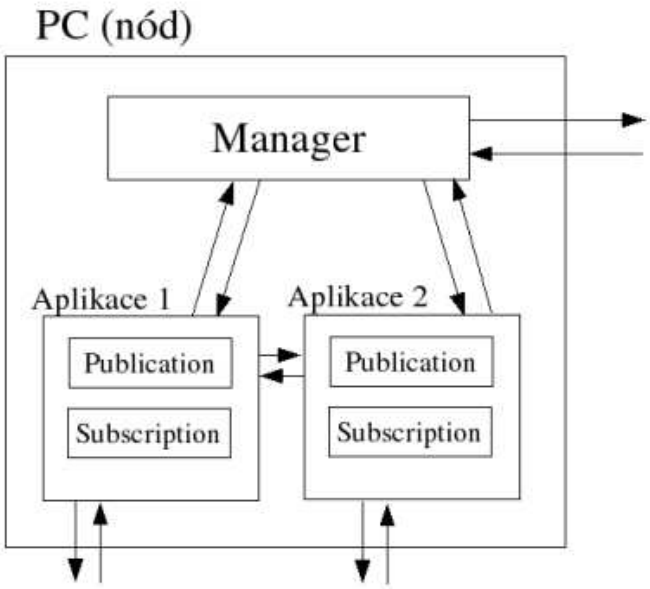
Každá aplikace je vždy řízena jedním nebo více managery. Pro zveřejňování svých atributů používá aplikace speciální objekt typu CSTWriter, který obsahuje základní parametry aplikace a také seznam managerů, ke kterým je aplikace přihlášena. Tento objekt je periodicky vysílán (bez nutnosti potvrzovat doručení) a přijat každým managerem, který je v seznamu. Ostatní manageři tuto zprávu ignorují.

[20] písmena CST: composit state type - jde o typ objektu pracující s popisnými informacemi o některém z nadřízených objektů

[21] managerem s malým "m" je dále myšlen objekt Manager z předchozího seznamu a stejně tak aplikací je myšlen objekt typu ManagedApplication

Na konci celého procesu mají všichni manageři prozkoumány své aplikace a každá aplikace zná své managery.

Zjednodušený model jednoho síťového nódu je na obrázku 3.9.



Obrázek 3.9: Zjednodušený model nódu.

Každá aplikace se skládá z celkem sedmi objektů:

- *WriterApplicationSelf* Objekt typu CSTWriter, skrze nějž se aplikace registruje k manageru.
- *ReaderApplications* Typ CSTReader pomocí kterého aplikace získává informace o ostatních aplikacích na síti.
- *ReaderManagers* Typ CSTReader, kterým aplikace získává informace o managerech.
- *WriterPublications* Typ Writer, kterým aplikace poskytuje jednotlivá vydání jedné nebo několika instancím objektu Subscription, a využívá právě *publish-subscribe* protokolu.
- *ReaderPublications* Typ Reader, kterým aplikace přijímá informace o objektech typu Subscription.

- *WriterSubscriptions* Typ Writer, který poskytuje informace o objektu Subscription objektům typu Publications.
- *ReaderSubscriptions* Typ Reader, který přijímá jednotlivá vydání od jedné nebo více instancí objektu Publication, opět za použití protokolu *publish-subscribe*.

Zde je praktický příklad komunikace. Máme síť tvořenou dvěma nódy, každý nód má svého managera a každý manager řídí dvě aplikace. (obr. 3.10)

1 Aplikace MA1.1 zašle informace o sobě lokálnímu manageru M1.

2 Manažer M1 pošle seznam vzdálených managerů Mx a seznam ostatních lokálních aplikací MA1.x.

3 Aplikace MA1.1 je managerem M1 představena ostatním managerům Mx

4 Parametry všech vzdálených aplikací jsou zaslány aplikaci MA1.1.

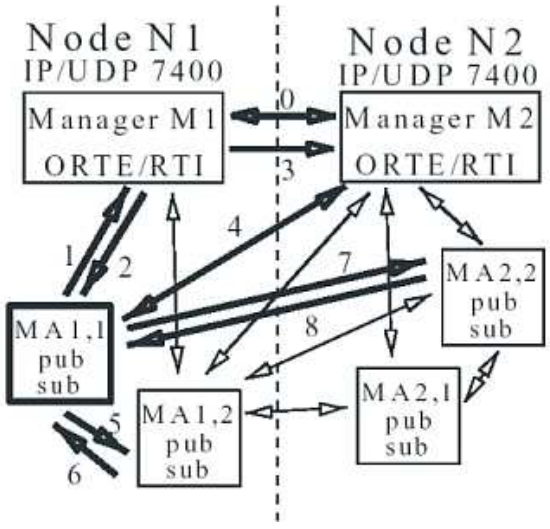
5 Aplikace se dotazuje lokálních aplikací MA na jejich CS[22]

6 Všechny lokální aplikace posílají jejich CS.

7 Aplikace se dotazuje vzdálených aplikací na jejich CS.

8 Vzdálené aplikace posílají své CS.

Ty objekty typu *Publisher* a *Subscriber*, u kterých souhlasí parametry *Topic* a *Type* spolu potom začínají datovou komunikaci.



Obrázek 3.10: Komunikace mezi dvěma síťovými nódy.

newpage

[22]CS - composit state - seznam všech atributů daného objektu

4 OPNET Modeler

V této části představím alespoň základní vlastnosti rozsáhlého a komplexního simulační software OPNET Modeler a shrnu své praktické zkušenosti s tímto software.

OPNET Modeler je jedním z několika produktů firmy OPNET Technologies Inc. Umožňuje modelovat a simulovat funkčnost prakticky jakékoliv možné architektury sítě, za použití rozsáhlých knihoven již hotových modelů, s možnostmi vytvořit si prakticky jakýkoliv komunikační model. Je velice efektivním nástrojem pro vývojáře z oblasti sítí, protože umožňuje vytvořit nový síťový prvek (komunikační model, formát paketu, procesní model chování serveru a další) a nasimulovat jeho chování aniž by bylo nutno něco fyzicky vytvářet.

OPNET Modeler disponuje hierarchicky uspořádaným grafickým rozhraním (viz. kapitola 4.1), které umožňuje umožňuje velice efektivní práci s jednotlivými částmi daného projektu, vlastním simulačním jádrem a je postaven na jazyku C. Veškeré modely jsou plně přenositelné mezi systémy Windows a UNIX. Umožňuje pracovat s externími moduly, má vlastní debugger, pomocí kterého lze celou simulaci krokovat a zjišťovat parametry simulace při jejím běhu. Simulace má možnost nastavení optimalizace výsledného kódu pro sekvenční nebo plně paralelní simulaci založenou na diskrétních událostech, hybridní a analytickou simulaci a také, v mé práci využitou, podporu pro tzv. kosimulaci, tedy simulaci propojenou buď s další simulací OPNET Modeleru, nebo s úplně nezávislou externí aplikací.

Umožňuje generovat široké spektrům grafických znázornění statistik simulace, generovat výsledné zprávy ve formátu XML[23] nebo HTML[24], případně předávat data do tabulek, stejně tak umí zpracovat vstupní data z těchto formátů a dalších, které jsou generovány různými síťovými nástroji. Dále obsahuje prohlížeč animací, pomocí kterého je možné si proběhlé simulace přehrát.

Celá simulace navíc probíhá s velmi efektivním zrychlením, takže je možné např. nasimulovat chování rozsáhlé a komplikované sítě po dobu několika měsíců za několik hodin, navíc je možné nastavit parametry simulace v závislosti na čase např.v nočních hodinách je zatížení sítě 3x menší apod.

Obrovská výhoda OPNET Modeleru je, že všechny dostupné knihovny (síťových prvků, definic paketů a linkových vrstev, serveru apod.) mají dostupný zdrojový kód, takže je lze upravovat, což je velice výhodné potřebujeme-li např. vyzkoušet nové možnosti nějakého standardního prvku.

Vzhledem k rozsáhlým možnostem OPNETu předesílám, že se budu v popisu zaměřovat převážně na oblasti, které alespoň částečně souvisejí s mojí diplomovou prací v tomto simulačním prostředí.

[23]Extensible Markup Language
[24]HyperText Markup Language

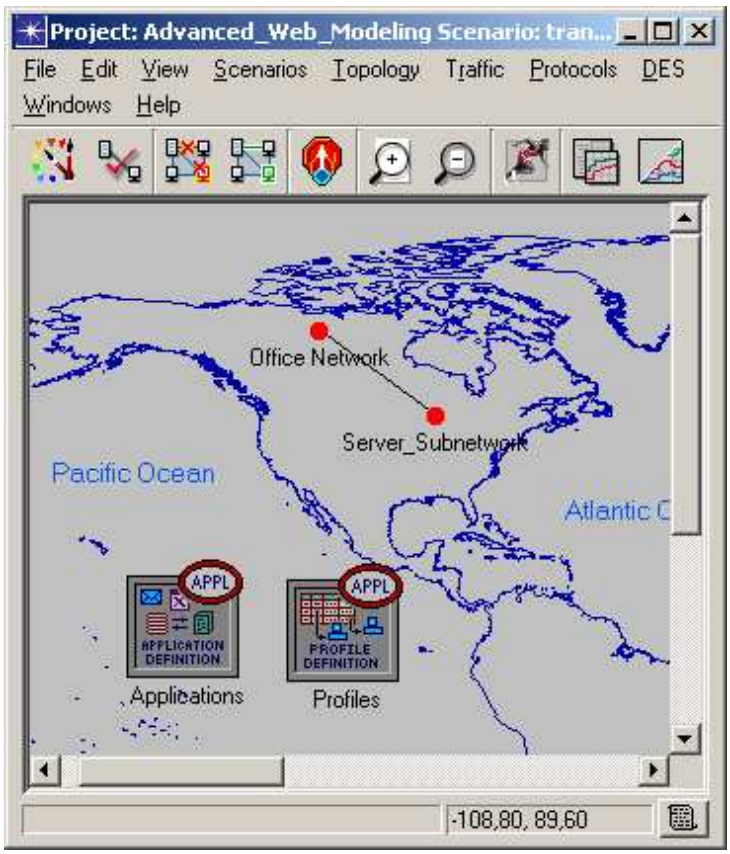
4.1 Základní popis prostředí OPNET Modeler

Jak bylo již zmíněno, celý simulační model v OPNET Modeleru se skládá z jednotlivých grafických rozhraní popisující jednotlivé komponenty a tyto rozhraní jsou vnořeny do několika úrovní. V následujících odstavcích jsou popsány jednotlivé úrovně a dále několik dalších důležitých vlastností prostředí, přímo související s touto prací.

Důležitou vlastností OPNETu, která je společná všem komponentům daného modelu a kterou je vhodné zmínit už zde na začátku je objektové chování všech částí.

Každý prvek simulačního modelu je objekt s vlastním ID[25] a parametry které jsou pro tento objekt typické. Pomocí nástrojů OPNETu tak můžeme během simulace pracovat odkudkoliv s jakýmkoliv objektem daného modelu, například poslat přerušení libovolnému IP procesu v kterémkoliv nódu kterékoliv sítě.

4.1.1 Globální síťový model



Obrázek 4.11: Globální model sítě - umožňuje zahrnout do simulace i geografickou polohu jednotlivých prvků.

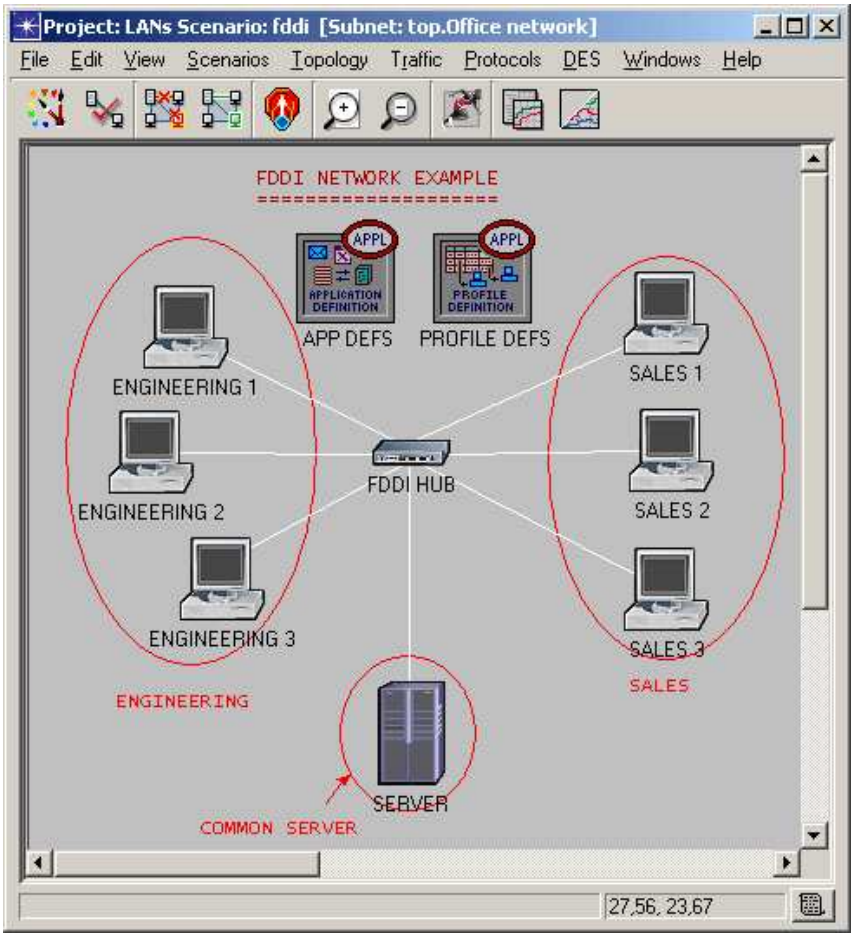
[25] Jednoznačný identifikátor

Na nejvyšší úrovni je rozhraní pro tzv. globální síťový model jak ukazuje předchozí obr. 4.11.

Toto rozhraní umožňuje zahrnout do simulace různé vzdálenosti mezi komunikačními prvky a parametry sítě z toho vyplývající (zpoždění, útlum apod.).

4.1.2 Síťový model

Nižší úroveň je rozhraní pro samostatný síťový model, které popisuje jednotlivé lokální sítě. Z obrázku 4.12 lze vidět, že za použití palety s předdefinovanými komponentami, je velice jednoduché vytvořit model i komplikovanější a rozsáhlejší sítě. V každém rozhraní si lze doplňovat k modelu vlastní poznámky a vysvětlivky, využívat rozsáhlou knihovnu grafických symbolů pro jednotlivé prvky. Je zde (a také na vyšší úrovni) možnost vložit prvky pro konfiguraci aplikací, profilů apod. (popsáno v dalších částech).



Obrázek 4.12: Ukázka lokálního modelu sítě s vlastními popisky a vysvětlivkami

Dále je možné si ke každému prvku přiřadit některou z množství předdefinovaných statistik a tak je možné sledovat na síti průběh mnoha veličin, které bychom v reálné aplikaci jen velice těžko zjišťovali, případně aktivovat modul pro záznam animací a po proběhnutí simulace si spustit animaci jejího průběhu, opět s mnoha možnostmi konfigurace.

Velice užitečnou vlastností je možnost definice takzvaných "Scénářů" (*Scenarios*), které umožňují v rámci jednoho projektu, nadefinovat více různých sítí s různými prvky a různými parametry simulace.

Je to výhodné zvláště v případě, že simulujeme určitý druh sítě a potřebujeme jednoduše přepínat mezi konfiguracemi. Například simulujeme-li chování FTP[26] serveru, můžeme mít jeden scénář pro 10 klientů, druhý pro 10 000 klientů, třetí pro zkoumání vlivu dalších provozovaných služeb na výkon serveru a podobně. Pokud bychom ale chtěli simulovat zcela odlišné sítě, pak se spíše vyplatí založit nový projekt.

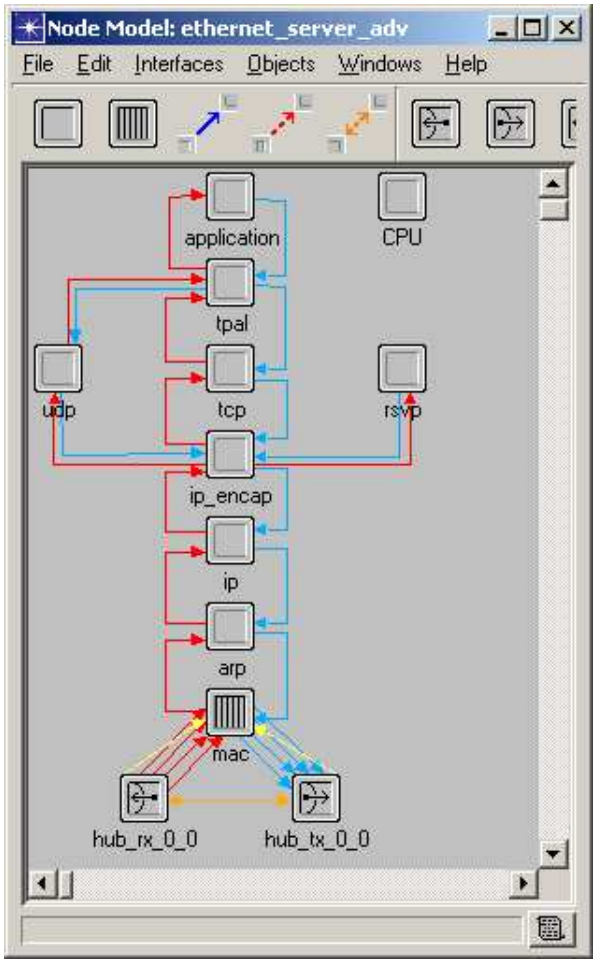
4.1.3 Node model

Další nižší úroveň rozhraní již definuje tzv. *Node model* jednotlivých komunikačních prvků. Kde každý "nód" tvoří několik procesních modelů vzájemně propojených a spojení mezi nimi představují datové (případně statistické nebo logické) kanály (obr. 4.13).

Samozřejmě je možné komunikovat mezi nódy i jinak než pomocí znázorněných vazeb, a to pomocí signálů zasílaných přímo konkrétnímu nódu (procesu). To je jedna z dalších vlastnostech OPNET Modeleru, že jsou prvky všech úrovní řazeny do stromové struktury celého modelu, takže pokud chci poslat zprávu nějakému procesu s kterým nejsem přímo propojen a znám jeho jméno (definované uživatelem v atributech), zjistím si ID rodičovského prvku a díky němu pak zjistím ID prvku, kterému pak zprávu pošlu.

Další zajímavou a důležitou vlastností objektového modelu prvků je tzv. předávání hodnot atributů do vyšších úrovní. Tímto způsobem si mohu klíčové parametry simulace, zobrazit až u objektů nejvyšší úrovně (např. IP adresy primárně definované na procesním modelu popisujícím proces IP vrstvy) a nemusím se složitě "proklikávat" až k objektu, kterého se daný parametr konkrétně týká.

[26]FTP - File Transfer Protocol



Obrázek 4.13: Node model - model hierarchie jednotlivých procesů a jejich propojení (zde je konkrétně o model stanice připojené k Ethernetu)

4.1.4 Process model

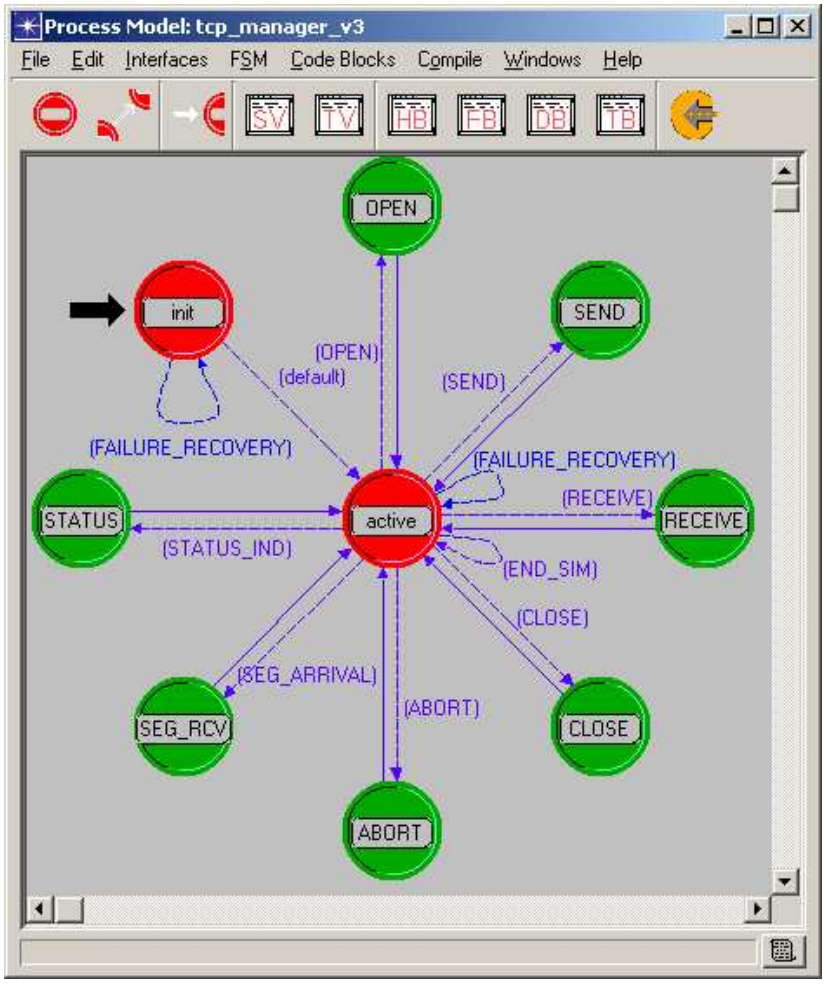
Nejnižší úrovní je tzv. *Process model* (viz. obr. 4.14). Tento model je reprezentován pomocí konečného stavového automatu a popisuje jednotlivé stavy procesu a přechody mezi nimi. Vlastní kód procesu je napsán v jazyce C. Stav procesu může být dvojího typu:

- *Forced* - při přechodu procesu do tohoto stavu se vykoná kód, který tento stav obsahuje, a automaticky dojde k přechodu do stavu dalšího (pokud je splněna případná podmínka přechodu).
- *Unforced* - po přechodu do tohoto stavu v něm proces zůstává tak dlouho dokud nedojde k dalšímu přerušení procesu.

Každý stav je dále dělen na dvě části. Tzv. *Enter Execs* - vstupní stav, obsahuje kód který se vykoná když proces přejde do daného stavu. *Exit Execs* - kód

toho stavu je vykonán, v případě *Unforced* stavu, až při opuštění toho stavu díky přerušení, a v případě *Forced* stavu se vykoná ihned po kódu z *Enter state*.

Tedy v případě *forced* stavu je jedno, zda je kód umístěn v *Enter Execs* nebo v *Exit Execs*. U *unforced* stavů se většinou využívá k vykonání akcí závislých na opouštění tohoto stavu (uvolnění alokované paměti, získání parametrů přerušení apod.).



Obrázek 4.14: Process model - každý proces je popsán pomocí stavů a přechodů mezi nimi.

Přechody mezi jednotlivými stavy rozdělujeme také na dva druhy:

- *Podmíněné* - pro přechod do dalšího stavu je nutné nejen aby proces dostal signál přerušení, ale musí být splněna další, uživatelem definovaná podmínka.
- *Nepodmíněné* - pro přechod do dalšího není nutná žádná další podmínka.

K přechodům mezi jednotlivými stavy, lze, kromě podmínky přechodu, přiřadit ještě tzv. přechodovou funkci. Jde o funkci, která se zavolá vždy, když dojde k aktivaci tohoto přechodu. Toto je velice výhodné, zvláště tehdy, potřebujeme-li, před tím než se dostaneme do druhého procesu, zpracovat nějaké informace, které jsme získali právě přerušením, které vyvolalo přechod.

4.1.5 Přerušení

Přerušení (*Interrupts*) jsou jedním ze základních nástrojů celé simulace. Můžeme říci, že celá simulace se odehrává pomocí přerušení. Druhů přerušení je víc, pro ilustraci uvádím ty, které ve své práci využívám:

- *Self interrupt* - Vlastní přerušení. Používané např. je-li třeba z některého stavu procesu (inicializace) přejít dále a pro přechod není třeba žádných dalších podmínek z vnějšku.
- *Remote interrupt* - Vzdálené přerušení. Přerušení od jakéhokoliv jiného procesu v síti.
- *Stream interrupt* - Přerušení způsobené příchodem tzv. *streamu*, což je vlastně datový proud od jiného procesu, může přijít jen od procesu s kterým je daný proces propojen (obr. 4.13).
- *External system interrupt* - Přerušení způsobené externí aplikací (blíže v kapitole 5).
- *Begin sim interrupt* - První přerušení inicializující začátek simulace.
- *End sim interrupt* - Přerušení indikující konec simulace.

Každé přerušení má, kromě hodnoty určující typ přerušení, ještě tzv. hodnotu přiřazenou k danému přerušení, která může být definovaná uživatelsky nebo je přiřazena automaticky jako v případě *Stream interruptu*, kde hodnota udává, z kterého procesu přišla data.

4.1.6 ICI

Dalším důležitým prvkem, které je také dosti využíván v této práci je tzv. ICI[27]. Díky němu, lze ke každému informačnímu toku v procesu, ať je to datový proud nebo jakékoliv jiné přerušení, přídavnou datovou strukturu libovolného rozsahu. Struktura vždy obsahuje názvy položek a jejich hodnotu.

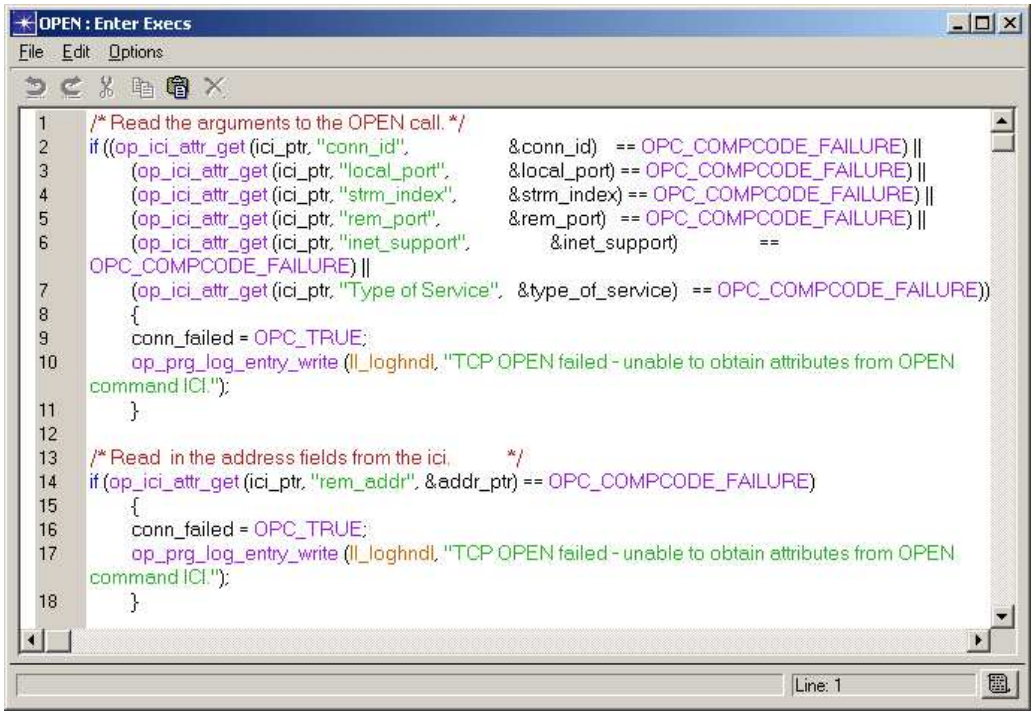
[27]Interface Control Information

Tato struktura je naplněna daty a před odesláním dat nebo vyvoláním jiného přerušení je k tomuto připojena a druhá strana ji může opět získat z tohoto přerušení.

Tento způsob předávání informací je velice efektivní, protože jím lze velice jednoduše zpřehlednit a zefektivnit komunikaci mezi procesy. Jak je zmíněno dříve, každé přerušení má kromě typu jen jeden parametr a ten ne vždy lze uživatelsky nastavit, tímto způsobem lze např. mezi dvěma procesy komunikovat pomocí stále stejného přerušení, ale pokaždé s jinak vyplněnou strukturou ICI.

4.1.7 Editace zdrojových kódu

Editace zdrojových kódů jednotlivých stavů se provádí v jednoduchém textovém editoru (obr. 4.15).



Obrázek 4.15: Ukázka rozhraní pro editaci zdrojového kódu.

Pro každý stav procesu lze otevřít dvě editační okna (pro *Enter State* a *Exit State*) a po skončení editace se okna zavřou a celý procesní model je nutno překompilovat. Tady je důležité zmínit, že pro správnou funkci OPNET Modeleru je nutné mít nainstalovaný externí kompilátor jazyka C. Pro usnadnění programování existují v rozhraní pro procesní model grafické nástroje pro zadávání globálních proměnných, lokálních proměnných, funkcí, definici hlavičkového souboru, bloku pro diagnostický kód a bloku pro kód vykonávaný při ukončení simulace.

4.1.8 Externí balíčky

Samotný kód procesů je klasickým programem v jazyku C, je samozřejmě možné (a někdy i nutné) k němu tak přistupovat, tedy OPNET Modeler umožňuje využívání vlastních hlavičkových souborů a tak např. sdílení proměnných mezi procesy (i když se v praxi spíše využívá konstant, např. jako hodnota přerušení apod.) a dále využívání úplně externích funkcí, které nemají přímou souvislost s žádným konkrétním procesem. Praktické využití je ukázáno v kapitole 5.

Jak již bylo řečeno v úvodu, uvedený výčet vlastností a možností OPNET Modeleru zdaleka nepostihuje jeho skutečné schopnosti a možnosti. Jde o jen základní přehled, který je navíc omezen jen na zúženou oblast, v které jsem pracoval.

4.2 Praktické používání

V této kapitole jsou, opět z hlediska jednotlivých grafických a logických rozhraní, popsány konkrétní poznatky a zkušenosti s realizací simulačního projektu v OPNET Modeleru.

4.2.1 Projekt

Správa projektů je jednou z ne příliš kvalitních částí OPNET Modeleru. Každý projekt se sestává z mnoha souborů, každé samostatné jednotce (proces, procesní model, síťový model atd.), odpovídá jeden samostatný soubor. OPNET Modeler bohužel nenabízí žádný nástroj na správu těchto souborů. Navíc pokud využíváme standardní modely a komponenty, tak ty zůstávají v systémových adresářích a pokud chceme nějakou jejich část editovat a doplňovat nebo upravovat, je vždy nutné je přejmenovat a uložit jinam, neboť jeden procesní model využívá standardně více síťových modelů a naší úpravou bychom změnili vlastnosti mnoha dalších komponent.

Jediné co OPNET Modeler nabízí z hlediska projektů, je, že si lze nastavit tzv. projektové adresáře, v kterých jsou defaultně projekty[28] hledány. A dále umí nabídnout seznam naposledy otevřených projektů (síťových modelů, procesních modelů atd.).

4.2.2 Knihovna komponent

Knihovna komponent je naopak velice rozsáhlá a kvalitně zpracovaná. Pomocí jednoduchého uživatelského rozhraní se vkládají do projektu prakticky veškeré známé komunikační a síťové prvky. Počínaje různými typy fyzických linek pro přímé propojení, bezdrátové a satelitní komunikační prvky, přes huby, switche,

[28] Myšleno soubory definující projekt

mosty, množství serverů specializovaných na různé aplikace (FTP, WWW[29] servery, databáze, atd.) a konče obecným modelem Internetu jako takového.

Knihovna komponent je samozřejmě rozdělena do tématických sekcí, tím je vyhledání konkrétního prvku je velice jednoduché a rychlé. Také lze vytvořit knihovnu ze vlastních, nebo často používaných systémových komponent. Sadu komponent (v OPNET Modeleru se označuje jako Palette) lze vytvořit pouze z prvků typu *Node model*, *Link model*, *Path model* a *Demand model*. Poslední dva prvky nepopisují žádný reálný prvek síťové komunikace, jde o tzv. popisné komponenty, vyjadřující logické vazby a vztahy mezi ostatními prvky modelu.

4.2.3 Kompilace a simulace

Jak již bylo řečeno dříve, OPNET Modeler využívá externí kompilátor zdrojových kódů, cestu k němu musíme nejprve nadefinovat v menu Preferences. Po zkompilování všech procesních modelů (samozřejmě pouze těch, do kterých jsme nějakým způsobem zasahovali, nebo si je samy tvořili, standardní modely jsou již předkompilovány) si z hlavního okna projektu otevřeme dialogové okno pro nastavení simulace.

Možnosti simulace jsou opět velice široké, z těch základních zmíním jen nastavování simulovaného času (řádově až desítky let), hodnota pro generování statistik, která udává kolik hodnot se bude během simulace sbírat, např. pokud simulujeme 10 vteřin a tato hodnota je 10, pak to znamená, že nová hodnota pro statistiku bude zaznamenána každou vteřinu. Důležitým parametrem při nastavování simulace je také zapínání a vypínání tzv. *Debugovacího režimu*[30] (podrobněji v 4.2.4)

Velice zajímavou a často užitečnou funkcí, je možnost volby simulace mezi tzv. statickou a dynamickou. Dynamická simulace spočívá v tom, že se vezmou všechny použité moduly, externí zdroje, knihovny apod. a jednorázově se vše spojí do jednoho simulačního souboru, který je pak vykonáván jádrem OPNET Modeleru.

Druhou možností je, si tento simulační soubor předpřipravit pomocí nástroje *op_mksim*, kterém lze vytvořit libovolný počet již hotových simulací a pak je pouze spouštět. Toto je výhodné pokud chceme předvádět různé simulace a projekt je tak rozsáhlý, že by každé přenastavování parametrů a opakovaná kompilace příliš zdržovaly.

S tím také souvisí další volba v rozhraní pro konfiguraci simulace a tou je možnost předat simulaci parametry jako v příkazové řádce a dále je zde možnost zvolit vynucenou kompilaci všech použitých modelů (standardně se kompilují

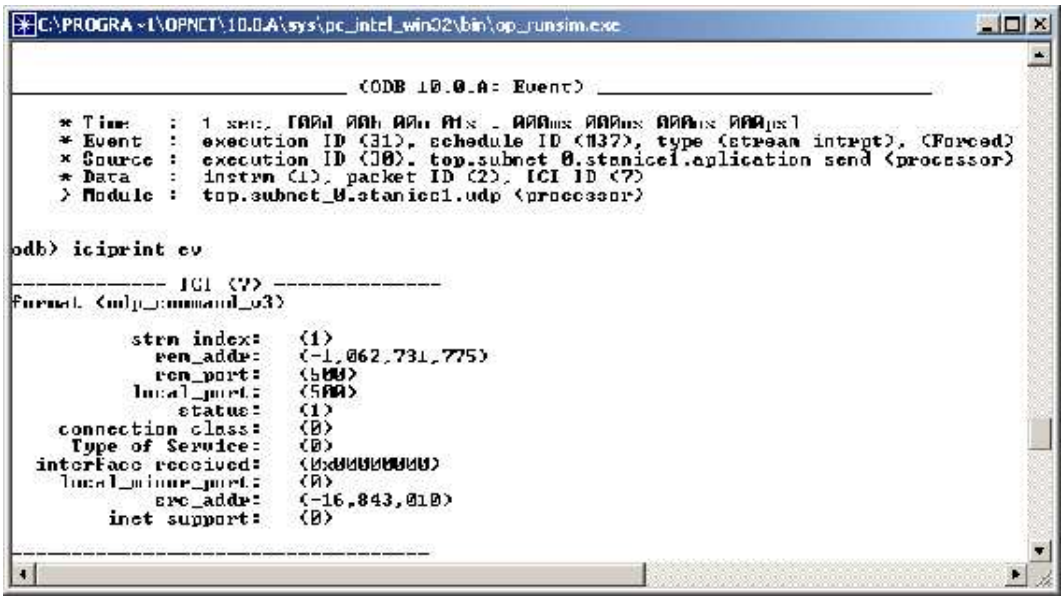
[29] World Wide Web

[30] Z anglického *debug* - *odvšivit*, v přeneseném významu to znamená ladění, zbavování programu chyb.

jen ty, které jsme editovali a nezkompilovali přímo v daném grafickém rozhraní), což ale dosti prodlužuje spuštění samotné simulace.

4.2.4 Debugování

Možnosti ladění a trasování vykonávané simulace jsou opět značně široké. Po zaškrtnutí volby debugovacího režimu v dialogu pro konfiguraci simulace se při spuštění simulace otevře okno příkazové řádky a čeká na příkazy uživatele. Tento režim je také možno detekovat přímo ve zdrojovém kódu a pomocí funkce *op_sim_debug* je možno přidat úseky, které se vykonají jen při tomto režimu, např. aktivovat vypisování obsahu paketů apod.



Obrázek 4.16: Konzole pro ladění a krokování simulace.

Druhou možností jak ladit program je umístit do kódu procesu speciální příkazy pro nastavení určitého bodu, na kterém se simulace zastaví a dále je možnost pokračovat ve debugovacím režimu, nebo pro výpis ladících informací (podrobnější informace v [13], část *Programming Package*.

Opět zde uvedu jen několik z mnoha možností, které jsem ve své práci nejvíce využíval:

- *cont* - Pokračování průběhu simulace standardním způsobem.
- *next* - Skok na následující událost (možno parametrem určit na kolikátou událost může simulace přejít).
- *tstop* - Zastavení simulace v určeném čase (varianty *evstop* pro zastavení na události, *pkstop* pro zastavení při příchodu paketu a další).

- *iciprint_ev* - Výpis ICI přiřazeného k dané události (případně *iciprint_pk* pro paket)
- *pkprint* - Výpis obsahu paketu

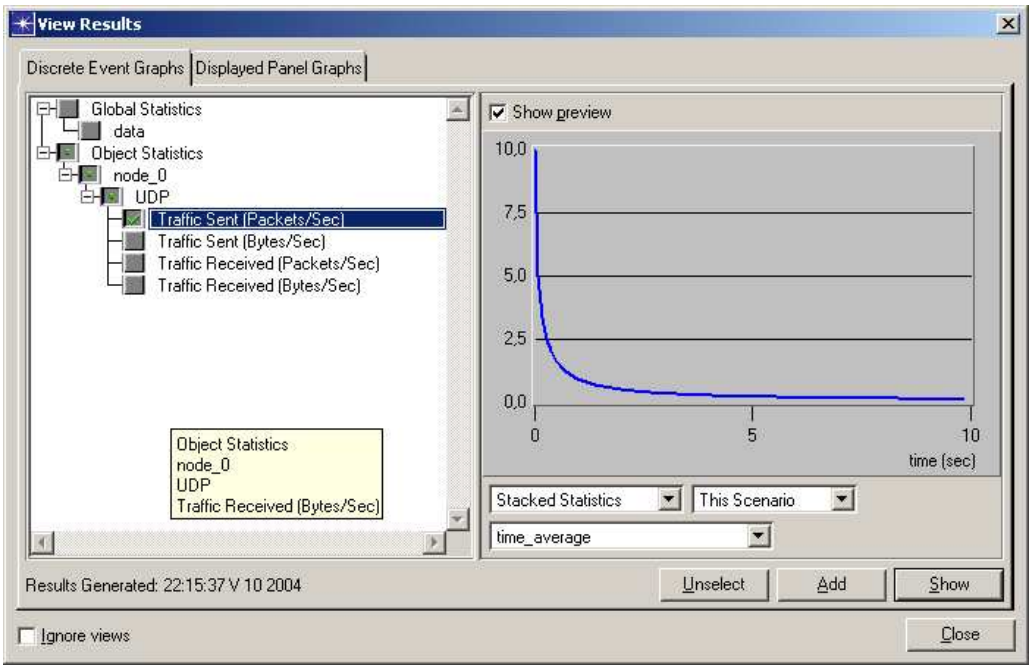
Další příkazy viz. [13].

4.2.5 Analýza výsledků simulace

Pro analýzu výsledků proběhlé simulace máme dva základní nástroje. Jsou to prohlížeč animací (*Animation viewer*) a prohlížeč výsledků (rozhraní *View result*).

Pro to, abychom mohli tyto nástroje používat, je před samotnou simulací nutno nadefinovat jaká data chceme zaznamenávat. OPNET Modeler má mnoho různých předdefinovaných voleb pro záznam nejrůznějších údajů. Od záznamu aktivity nějakého procesu až po počet bytů zaslaných od klienta k serveru.

To lze nastavit buď na globální úrovní pro celý model, nebo lokálně pro každý prvek zvlášť. Po skončení simulace lze pomocí rozhraní (obrázek 4.17) zvolit které z dostupných statistik budou zobrazeny, způsob jejich zobrazení z hlediska grafického (osy grafu, spojité nebo diskrétní zobrazení apod.) i z hlediska matematického (lze zpracovat údaje jako časový průměr, sumu hodnot, pravděpodobnostní rozložení, histogram a další.) Ze zobrazených grafu lze definovat šablony pro zobrazení výsledku simulace s jinými parametry není nutné znovu nastavovat vzhled statistik.



Obrázek 4.17: Okno pro výběr a nastavování parametrů získaných statistik.

Pro zobrazování animací má OPNET Modeler speciální grafický nástroj - *Animation viewer*. Bohužel zaznamenanou animaci nelze nijak uložit, ale i přesto má tento program zajímavé možnosti pro analýzu chování modelu jako např. definice rychlosti posunu času, krokováni po jednotlivých událostech, nebo po časových intervalech a nebo po jednotlivých požadavcích na zobrazení (pohyb v animaci).

Ve výsledném zhodnocení bych OPNET Modeler označil za velice mocný nástroj s rozsáhlými možnostmi simulace, analýzy a prostředků pro návrh a vývoj mnoha simulačních aplikací z oblasti sítí.

Za největší nevýhodu programu považuji právě jeho rozsáhlost a to, že i ve velmi jednoduchém modelu s minimálním počtem prvků, je dosti komplikované naučit se správně nastavovat všechny parametry a všechny využít možnosti analýzy a zpracování výsledků, které program nabízí.

Dokumentace je velice rozsáhlá, ale dosti obtížně se v ní hledají informace o nějakých konkrétních problémech. Např. pokud simulace zobrazí informaci, že proces přijal přerušení typu 5, pak nelze v dokumentaci nalézt o jaký druh přerušení jde. Dokumentace pouze obsahuje symbolické konstanty definující jednotlivé druhy, ale už neříká jakou mají hodnotu.

5 Simulační aplikace

V této kapitole je popsána základní simulační aplikace. Ta se skládá ze tří základních logických částí (obrázek 5.25) :

- *UDP interface*
- *External interface*
- *OPNET - ORTE*

První dvě části tvoří procesní modely OPNET Modeleru, třetí část tvoří externí aplikace realizující rozhraní pro ORTE do OPNET Modeleru.

V této kapitole jsou tyto části popsány jako samostatné jednotky. V následující pak je popsán princip kosimulace a činnost celé aplikace.

Výchozím modelem pro moji aplikaci byl standardní knihovní model stanice připojené na ethernetovskou síť - *ethernet_wkstn_adv.nd.m.* Modul, kterým realizuji proces pro rozhraní do již standardního procesu UDP, je zvýrazněn na obr. 5.18.
Tento model je dále rozšířen o procesní model externího rozhraní (obrázek 5.20), které realizuje propojení z OPNET Modeleru do externí aplikace a zpět.

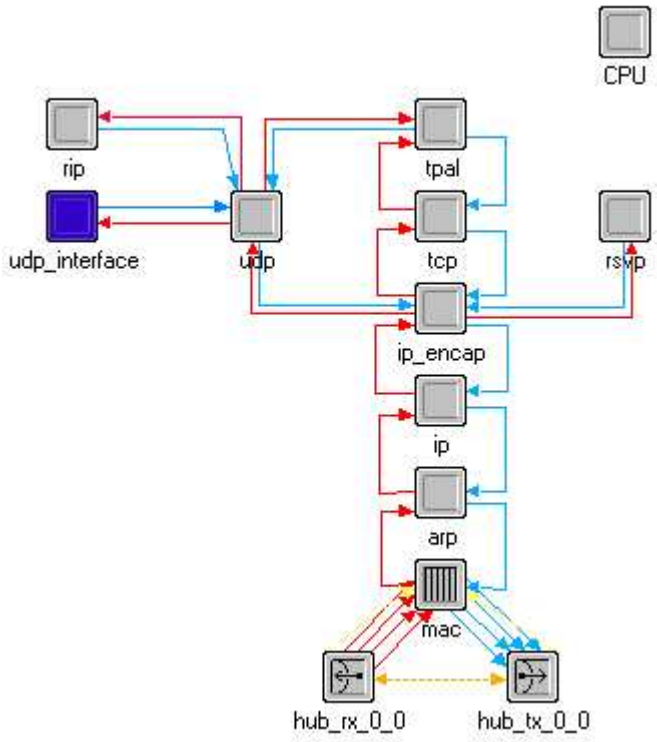
Samotná externí aplikace se skládá ze tří logických částí a to:

- Samotná aplikace (funkce *main*()).
- Sada funkcí realizujících propojení s OPNET Modeler.
- Samotné funkce z ORTE.

Celá výsledná simulační aplikace je koncipována jako jeden spustitelný *.exe* soubor s parametrem, který definuje simulační model v OPNET Modeleru. Výstup simulace je pak stejný, jako když ladíme standardní simulaci, tedy do textové konzole, s tím, že v grafickém prostředí OPNETu pak provádíme analýzu statistik.

5.1 Rozhraní do UDP vrstvy

Ke standardnímu node modelu byl připojen modul pro rozhraní do UDP (zvýrazněn na obr. 5.18). Tento modul[31] řeší základní propojení uživatelské aplikace s již hotovým procesním modelem UDP vrstvy. Z obrázku 5.18 lze vidět dvě datové linky z a do UDP vrstvy a také důležitou vlastnost procesních modelů, kterou je možnost k danému modelu připojit prakticky libovolný počet dalších.



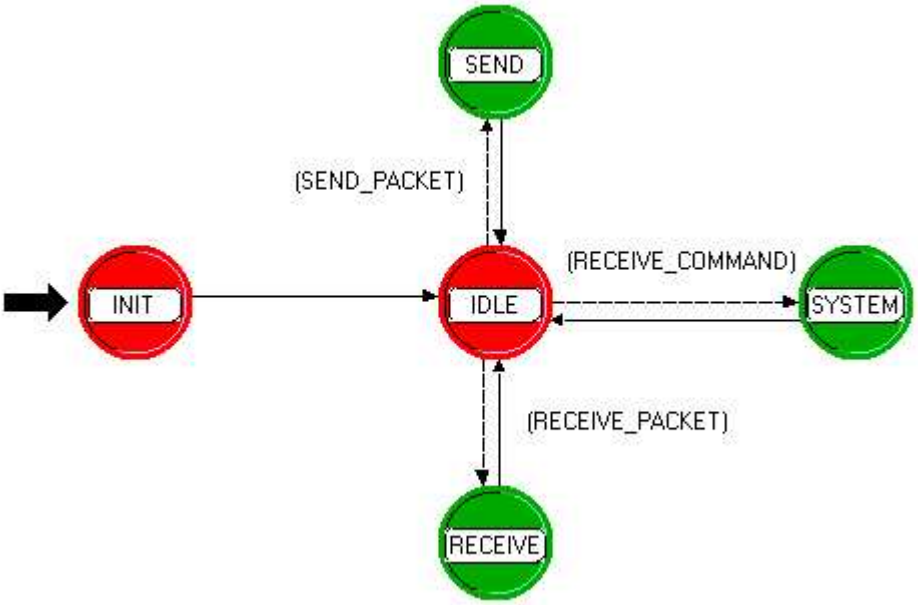
Obrázek 5.18: Node model s vyznačeným rozhraním pro připojení do UDP vrstvy.

Informaci o zdroji příchozích nebo odchozích dat, nese již zmíněný druhý parametr u tzv. *stream interrupt* přerušení, který udává tzv. *stream port*, což je jednoznačný identifikátor daného datového spoje.

Tyto hodnoty se dají zjistit v grafickém rozhraní a je velmi užitečné si v hlavičkovém souboru (viz. dále) převést na pojmenované konstanty, s kterými se lépe pracuje.

[31] Zde je modulem myšlena jedna z uvedených částí simulační aplikace, ve skutečnosti jde o *process model*, jak bylo zmíněno v kapitole 4.1

Vnitřní strukturu procesu *udp_interface* ukazuje obrázek 5.19.



Obrázek 5.19: Struktura procesního modelu *udp_interface*

.

Vidíme, že se process skládá ze čtyř různých stavů:

- INIT
- IDLE
- SEND
- RECEIVE
- SYSTEM

Celý proces využívá následující hlavičkový soubor:

```
#include <udp_api.h>
#include <ip_addr_v4.h>

#define UDPSTRM 0  // rozhrani do UDP vrstvy
#define EXTPSTRM 1  // rozhrani do aplikacni vrstvy
#define CONTROL_CMD 7 // preruseni pro ridici prikaz

#define SEND_PACKET (op_intrpt_type() == OPC_INTRPT_STRM && \
        op_intrpt_code() == EXTSTRM)
```

```
#define RECEIVE_PACKET (op_intrpt_type() == OPC_INTRPT_STRM && \
        op_intrpt_code() == 0)
#define RECEIVE_COMMAND (op_intrpt_type() == OPC_INTRPT_REMOTE \
        && op_intrpt_code() == CONTROL_CMD)
```

První dva řádky vkládají standardní hlavičkové soubory s deklaracemi API funkce pro základní komunikaci do UDP a pro práci s IP adresou. Další tři řádky definují již dříve zmíněné konstanty pro identifikaci z kterého *stream portu* přišla data.

Zbývající řádky pak definují podmínky pro přechody mezi jednotlivými stavy procesu.

U tohoto procesního modelu je též nutno do-deklarovat externí soubor *ip_addr_v4* (v menu *File –> Declare external files...*), protože OPNET Modeler standardně nezná kód funkcí popsaných v *udp_api.h*.

Externí soubor si lze samozřejmě vytvořit nezávisle na OPNETu, pouze je nutné aby měl příponu *.ex.c* a byl zkompilován kompilátorem shodným s tím, který je nastaven v OPNETu.

5.1.1 INIT

Tento stav je označen černou šipkou, což znamená, že jde o tzv. *Initial State*, tedy počáteční stav. Jde o stav do kterého se proces dostane po začátku simulace po vyvolání tzv. *begin_sim_interrupt*.

Je definován jako *unforced*, tedy po vykonání jeho kódu proces samovolně nepřechází do dalšího stavu. To je zde z toho důvodu, že na začátku simulace dostávají ostatní procesní modely speciální signál přerušení, takže je třeba aby můj model tento signál nepovažoval za součást simulace, ale na jeho příchod teprve přejde do stavu IDLE (popsán dále), který již reaguje na standardní simulační přerušení.

Vlastní kód[32] tohoto stavu je jednoduchý a stará se o získáni identifikátoru procesního modelu UDP vrstvy, dále o vytvoření ICI pro zasílání příkazů do UDP a nakonec o alokaci paměti pro proměnné uchovávající některé z parametrů spojení.

```
/* Ziskani vlastniho ID */
my_obj_id = op_id_self();

/* Ziskani ID rodicovskeho objektu*/
parent_obj_id = op_topo_parent (my_obj_id);
```

[32]Zdrojové kódy v dalším textu jsou téměř kompletní, s tím, že jsou vypuštěna různá testovací hlášení, případně operace s proměnnými nebo kód, který pro popis činnosti není důležitý.

```
/* Obtaining id of UPD node*/
udp_obj_id = op_id_from_name (parent_obj_id, \
             OPC_OBJTYPE_PROC, "udp");

/* Create the ICI with opening new session requiring a new port. */
ici_ptr = op_ici_create("udp_command_v3");

rem_port = (int *) malloc (sizeof(int));
rem_addr = (int *) malloc (sizeof(int));
local_port = (int *) malloc (sizeof(int));
```

5.1.2 IDLE

Tento stav je výchozím stavem procesu v kterém se nachází, pokud nezpracovává žádné požadavky a do něj se navrací po každém obslouženému požadavku.

Tento stav neobsahuje žádný kód a je opět *unforced*.

5.1.3 SYSTEM

Do tohoto stavu se proces dostane na základě podmínky pravdivé hodnoty makra RECEIVE_COMMAND viz. oddíl 5.1), která je splněna pokud proces přijme vzdálené přerušení (tedy nejde o přerušení při příchodu dat) a jeho kód odpovídá kódu řídícího příkazu. Proces zpracovává požadavek na zaregistrování nového UDP portu.

Kód procesu jen následující:

```
/* Ziskam z preruseni ukazatel na ICI */
ici_p = op_intrpt_ici();
value = (int *) malloc (sizeof(int));
/* Ziskam hodnotu portu k registraci */
op_ici_attr_get (ici_p,"port_value", value);
   /* Vyplnim informaci do ICI pro UDP */
   op_ici_attr_set(ici_ptr,"local_port",*value);
   /* Nainstaluji ICI */
   op_ici_install(ici_ptr);
   /* inicializace UDP portu pomoci remote preruseni do UDP */
   op_intrpt_force_remote(UDPC_COMMAND_CREATE_PORT, udp_obj_id);
   /* Overeni uspesnosti vytvoreni portu */
   op_ici_attr_get(ici_ptr,"status",&status);
    if (status != UDPC_IND_SUCCESS)
   {
   printf("ERROR - UPD port se nepodarilo vytvorit, ");
   printf("Chyba c.%d \n",status);
```

```
}
else
{
printf("Byl uspesne vytvoren UDP port\n");
};
```

Funkce je velmi jednoduchá. Z přerušení, které vyvolalo přechod do tohoto stavu, získám ukazatel na přidružené ICI a z něj pak hodnotu portu, který chci zaregistrovat. Tu pak vyplním do ICI pro přerušení do UDP a vyvolám vzdálené přerušení pro proces UDP. Z ICI si pak zjistím status, zda byl příkaz úspěšný či nikoliv.

Protože tento stav je *forced* proces se po vykonání kódu vrací zpět do stavu IDLE.

5.1.4 SEND

Do stavu SEND se proces dostane, pokud dojde k tzv. *stream* přerušení, tedy přerušení vyvolaném příchodem datového toku, v tomto případě musí jít od data z procesu *extern_interface* a opět se proces po vykonání kódu vrátí zpět do stavu IDLE. Kód stavu:

```
/* Ziskam pointr na data */
externi_paket = op_pk_get (op_intrpt_strm ());

/*Ziskam informace z~paketu pro odesilani */
op_pk_fd_get (externi_paket, 1, local_port);
op_pk_fd_get (externi_paket, 3, rem_port);
op_pk_fd_get (externi_paket, 2, rem_addr);

/* Nastavim parametry do ICI */
op_ici_attr_set (ici_ptr,"local_port",*local_port);
op_ici_attr_set (ici_ptr,"rem_port",*rem_port);
op_ici_attr_set (ici_ptr,"rem_addr",*rem_addr);

/* Nainstaluju ICI */
op_ici_install(ici_ptr);

/* poslud data do UDP */
op_pk_send_forced(externi_paket,UDPSTRM);

/* Kontrola odeslani */
op_ici_attr_get(ici_ptr,"status",&status);
if (status != UDPC_IND_SUCCESS)
{
printf("ERROR - neni mozne zaslat data do UDP\n");
```

```
printf("Chyba c.%d \n",status);
}
else
{
printf("Paket uspesne odeslan do UPD \n");
};
```

Popis činnosti je zřetelný z jasný z komentářů. Získáme pointer na datový paket (ve formátu pole hodnot typu `int`), z něj potom data týkající se odeslání paketu (cílová adresa, cílový port a lokální port pro odesílání), tyto informace pak nastavíme do ICI a pošleme jako datový tok (*stream*) do UDP procesu.

5.1.5 RECEIVE

Poslední stav tohoto procesu slouží k zpracování příchozích dat. Podmínka pro přechod do tohoto stavu je obdobná jako u SEND, tedy došlo k *stream* přerušení, tentokrát z UDP procesu a stejně jako jako v předchozím případě se proces po vykonání kódu vrací do stavu IDLE.

Tento stav má za úkol pouze převzít příchozí paket od UDP procesu a poslat ho do procesu *extern_interface*

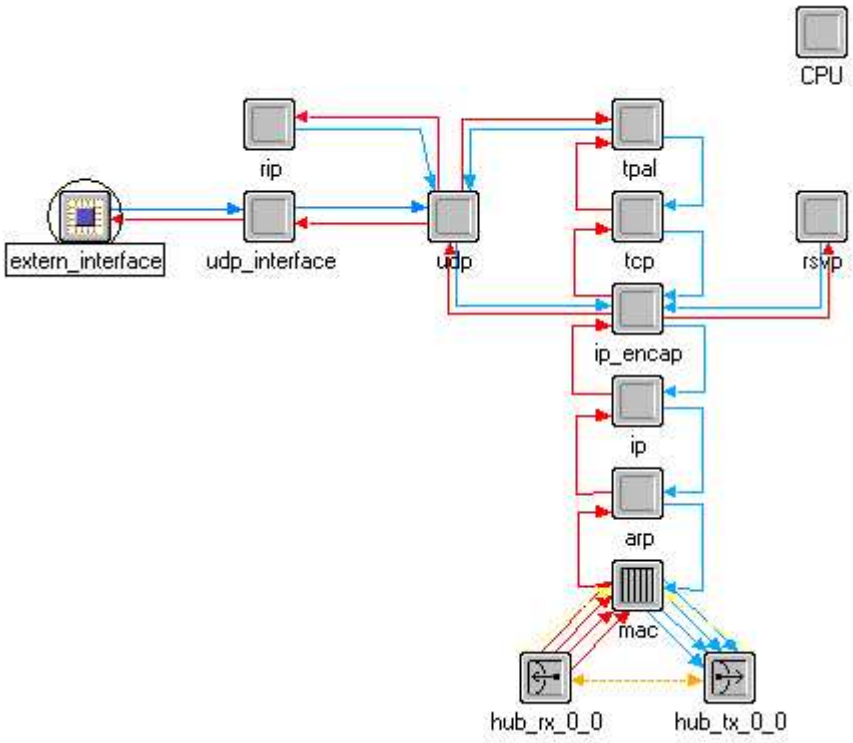
```
/* ziskam pointer na prichozi paket */
rcv_packet = op_pk_get (op_intrpt_strm ());

/* odeslu paket procesu extern_interface */
op_pk_send(rcv_packet,EXTSTRM);
```

5.2 Rozhraní do externí aplikace

Jak jsem již uvedl, druhou základní částí mé práce bylo realizovat rozhraní mezi OPNET Modeler a externí aplikací. Z obrázku 5.20 je vidět, že toto rozhraní je realizováno jako samostatný procesní model propojený s modelem pro *udp_interface*.

Jsou zde opět definované dva datové toky mezi uvedenými modely pro předávání dat. Obrázek 5.20 také ukazuje, že grafická reprezentace tohoto modelu se odlišuje od ostatních procesních modelů. Základní rozdíl je právě v tom, že k tomuto modelu je přiřazen tzv. ESD Model (obr. 5.21), což popis rozhraní mezi simulací v OPNETu a externí aplikací (nebo další simulací OPNET Modeleru, více v kapitole 6).

Ve spodní části lze definovat jednotlivé rozhraní, kterými tento proces komunikuje s externí aplikací. První je jméno rozhraní, dále typ předávaných dat (*integer,double, float, pointer, bit atd.*).



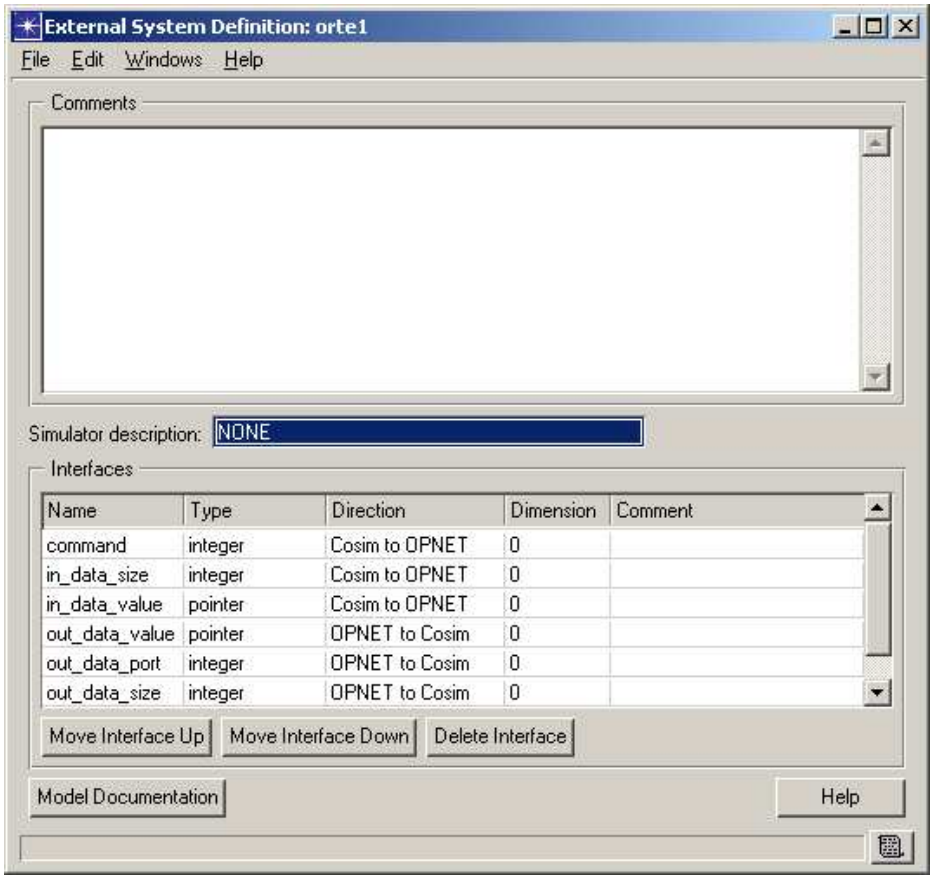
Obrázek 5.20: Připojení procesního modelu pro rozhraní k externí aplikaci.

Dále lze definovat směr kterým mohou být data přenášena a to:

- *OPNET to Cosim* - Data se předávají z OPNETu do externí aplikace.
- *Cosim to OPNET* - Data se předávají z externí aplikace do OPNETu.
- *bidirectional* - Data je možno přenášet obousměrně.

Parametr *dimension* určuje zda se jedná o jednu proměnnou nebo o pole. Pokud je *dimension* větší než nula tak rozhraní s daným jménem představuje pole proměnných daného typu s délkou uvedenou v *dimension*.

Dalším důležitým parametrem je hodnota v poli *Simulator description*, která ukazuje na soubor s popisem simulace a jejím vztahu k externí aplikaci (více v kapitole 6). K jednotlivým rozhraním lze dopisovat komentáře, stejně tak i k popisu celého rozhraní.



Obrázek 5.21: Způsob definic rozhraní do externí aplikace.

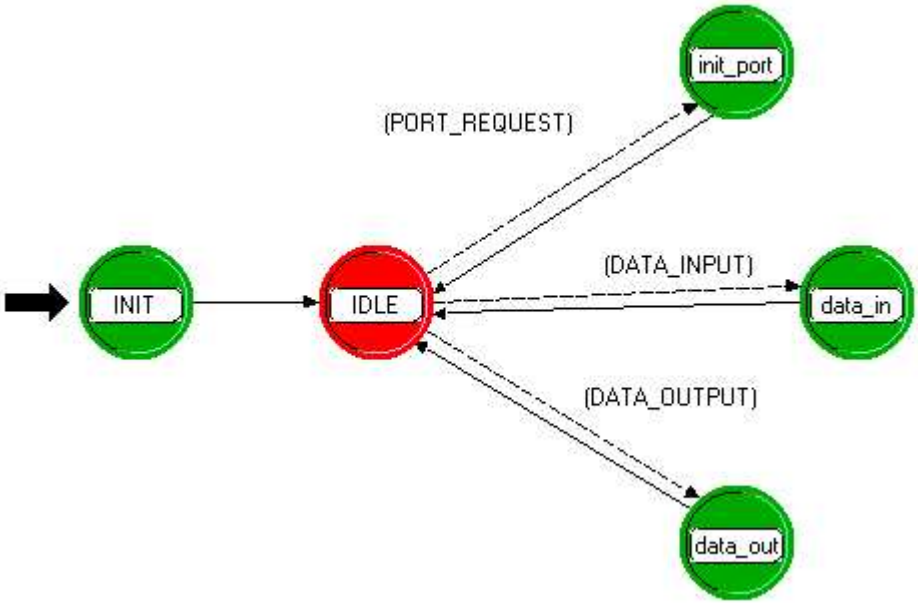
Dále jsou popsány jednotlivé stavy procesu tak, jak jsou znázorněny na obrázku 5.22.

5.2.1 INIT

Počáteční stav procesu, opět označen šipkou. V tomto případě není nutno čekat na další signál, takže je možné po vykonání svého kódu přejít automaticky do "pracovního" stavu IDLE. Kód je opět velice jednoduchý:

```
/* ziskam ID rozhrani kam se zapisuje prikaz */
command_iface = op_id_from_name (op_id_self(), \
                OPC_OBJTYPE_ESINTERFACE,"command");

/*Ziskam id procesu udp_interface */
udp_iface_node = op_id_from_name(op_topo_parent( \
                 op_id_self()), OPC_OBJTYPE_PROC,"udp_interface");

/* ziskam id rozhrani pro vstup a vystup dat */
DataValueIn = op_id_from_name (op_id_self(), \
              OPC_OBJTYPE_ESINTERFACE, "in_data_value");
```

```
⋮

DataValueOut = op_id_from_name (op_id_self(),\
               OPC_OBJTYPE_ESINTERFACE, "out_data_value");
```



Obrázek 5.22: Stavový model procesu *extern_interface*

Kód obsahuje pouze příkazy pro zjištění ID každého rozhraní z jména, které je zadáno v popisu (viz. obrázek 5.21). Hodnoty ID se ukládají do globálních proměnných, protože jsou využívány po celou dobu simulace a jejich hodnota se nemění.

5.2.2 IDLE

Tento stav má stejnou funkci jako v procesu *udp_interface*, tedy jde o "klidový" stav procesu. Z něj se proces dostává následkem přerušení a do něj se potom opět vrací.

5.2.3 init_port

Tento stav obsluhuje požadavek externí aplikace na přidělení (zaregistrování) UDP portu. Proces do něj přechází na základě splněné podmínky definované v makru PORT_REQUEST. Tato podmínka říká, že přerušení zdrojem přerušení musí být modul rozhraní k externí aplikaci a zároveň ho musí vyvolat rozhraní s ID odpovídajícím hodnotě proměnné *command_iface*.

Kód tohoto stavu je následující:

```
/* Ziskam data z rozhrani */
iface = op_intrpt_esys_interface();
op_esys_interface_value_get (iface, &port, 0);
/* Vytvorim ICI a nastavim hodnotu portu */
ici = op_ici_create("orte_commands");
op_ici_attr_set(ici,"port_value",port);
op_ici_install(ici);
/* vyvolam preruseni v procesu udp_interface */
op_intrpt_force_remote(CONTROL_CMD,udp_iface_node);
```

Nejprve získám ukazatel na rozhraní, ze kterého přišlo přerušení a pomocí něj přečtu z rozhraní data. Vytvořím ICI a nastavím mu hodnotu portu, který budu chtít registrovat. A nakonec pošlu přerušení do procesu *udp_interface*, s přiřazeným ICI.

5.2.4 data_in

Tento stav se stará o data přicházející z externí aplikace, upravuje je a posílá dál do OPNETu. Přechodová podmínka do tohoto stavu říká, že musí dojít opět k přerušení z modulu pro rozhraní a ID zdroje rozhraní musí odpovídat hodnotě proměnné *DataValueIn*.

```
/* ziskam pointer na interface */
iface = op_intrpt_esys_interface();
op_esys_interface_value_get (DataSizeIn, size, 0);
op_esys_interface_value_get (DataValueIn,&my_data,0);
/* vytvorim neformatovany OPNET paket */
data_packet_in = op_pk_create(0);
/* naplnim ho daty */
for (i=0;i<*size;i++) {
    op_pk_fd_set (data_packet_in, i, \
    OPC_FIELD_TYPE_INTEGER, *(my_data+i), 32);
    };
/* a odeslu skrze datovy tok (stream) */
op_pk_send(data_packet_in,UDP_INT);
printf("posilam data do UDP\n");

:
```

Nejprve z rozhraní získám velikost příchozích dat a poté ukazatel na pole dat v paměti. Vytvořím si neformátovaný OPNET paket a uložím si do něj všechny položky datového pole. Nakonec paket standardním způsobem odešlu do procesu *udp_interface*.

5.2.5 data_out

Poslední částí procesu je stav, který zpracovává příchozí data z procesu *udp_interface* a posílá je na rozhraní pro externí aplikaci.

K přechodu do tohoto stavu ze stavu IDLE je nutné, aby přerušení bylo způsobené příchozím datovým tokem a tento datový tok musí přijít na specifický port (v tomto případě je jen jeden, definován konstantou UDP_INT).

Kód tohoto stavu je opět velice jednoduchý:

```
/* ziskame paket z udp_interface */
temp_packet = op_pk_get (op_intrpt_strm ());
/* ziskam delku datove casti paketu */
op_pk_fd_get (temp_packet, 4,&delka);
/* naplni pole v pameti pro predani zkrze rozhrani ven */
for (i=0;i<delka+5;i++) {
op_pk_fd_get (temp_packet, i,&polozka);
*(data_packet_out+i) = polozka;
};
/* paket z udp_interface zrusim */
op_pk_destroy(temp_packet);
/* naplnim strukturu pro child proces */
data->PortOut = DataPortOut;

:

data->port = *(data_packet_out+3);
/* vytvorim a spustim child proces */
Child = op_pro_create ("tc_esd_child", data);
op_pro_invoke (Child,OPC_NIL);
op_intrpt_schedule_process (Child, op_sim_time(), 1);
```

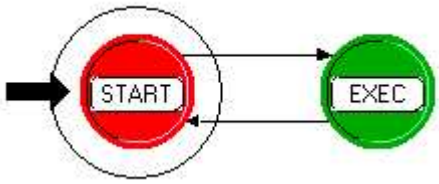
Tento stav je odlišný od všech ostatních a to tím, že v rámci svého kódu spouští tzv. *child process*, který je popsán v další části. Nejprve se z příchozího paketu získá délka dat a následně se vytvoří pole v paměti o této délce, do kterého se data překopírují a paket z ruší.

Nakonec se naplní datová struktura pro *child process* a ten se vytvoří a spustí s touto strukturou jako parametr.

5.2.6 Child process

Jde o zvláštní proces který je vyvoláván procesem *data_out* a jeho hlavní úkol je zapsat data na rozhraní pro externí aplikaci.

Jako parametr je mu předána struktura která obsahuje všechny proměnné, které jsou pro zápis na rozhraní potřebné. Propojení stavů procesu ukazuje obrázek 5.23.



Obrázek 5.23: Stavový model *child* procesu.

Je vidět, že model má jen dva stavy a přechody mezi nimi nejsou nijak podmíněny, tedy jakmile proces dostane přerušení, dojde k přechodu do stavu EXEC a po vykonání kódu se proces ihned vrátí do START.

Stav START neobsahuje žádný kód a stav EXEC obsahuje následující příkazy:

```
/* ziskam data predane z rodicovskeho procesu */
data = (MY_DATA *) op_pro_parmem_access ();
/* pouziju parametry pro zapis do rozhrani */
op_esys_interface_value_set (data->ValueOut, \
                        OPC_ESYS_NOTIFY_NEVER,data->data_out,0);
op_esys_interface_value_set (data->SizeOut, \
                        OPC_ESYS_NOTIFY_NEVER,data->data_delka,0);
op_esys_interface_value_set (data->PortOut, \
                        OPC_ESYS_NOTIFY_IMMEDIATELY,data->port,0);
```

Tento proces od rodičovského procesu (*extern_interface*) dostane jako parametr strukturu, ve které jsou jednak proměnné obsahující ID rozhraní na která se má posílat data a pak samotná data k odeslání.

V tomto případě se na rozhraní zapisují nejen délka dat a ukazatel na pole s daty, ale také port, pro který jsou data určena, protože je potřeba, aby externí aplikace už při zápisu dat na rozhraní věděla, pro jaký port jsou data určena.

Toto a další principy fungování popisuje kapitola 6.

5.3 Externí aplikace - soubor *sock.c*

Cílem externí aplikace bylo implementovat část již zmiňovaného projektu ORTE tak, aby mohl být propojen s OPNET Modeler a bylo možné jej testovat a analyzovat jeho činnost aniž by bylo nutné složitě řešit propojení počítačů a na každém pouštět aplikaci a provádět testování analýzu. To vše by měl za nás udělat OPNET Modeler.

ORTE komunikuje se síťovou vrstvou pomocí tzv. *Soketů*, které pracují přímo s UDP vrstvou. Funkce pro tuto komunikaci jsou obsaženy v souboru *sock.c*

Základním úkolem tedy bylo implementovat varianty funkcí ze *sock.c* tak, aby nepracovali z UDP vrstvou systému, ale s rozhraním definovaným v OPNETu.

V souboru *sock.c* je celkem 11 funkcí, a z nich bylo, pro potřebnou funkčnost, nutné upravit tyto funkce:

- *sock_start()* - inicializace rozhraní
- *sock_bind()* - registrace UDP portů
- *sock_recvfrom()* - příjem dat
- *sock_sendto()* - odesílání dat
- *sock_get_local_interfaces()* - zjišťování počtu a parametrů síťových rozhraní

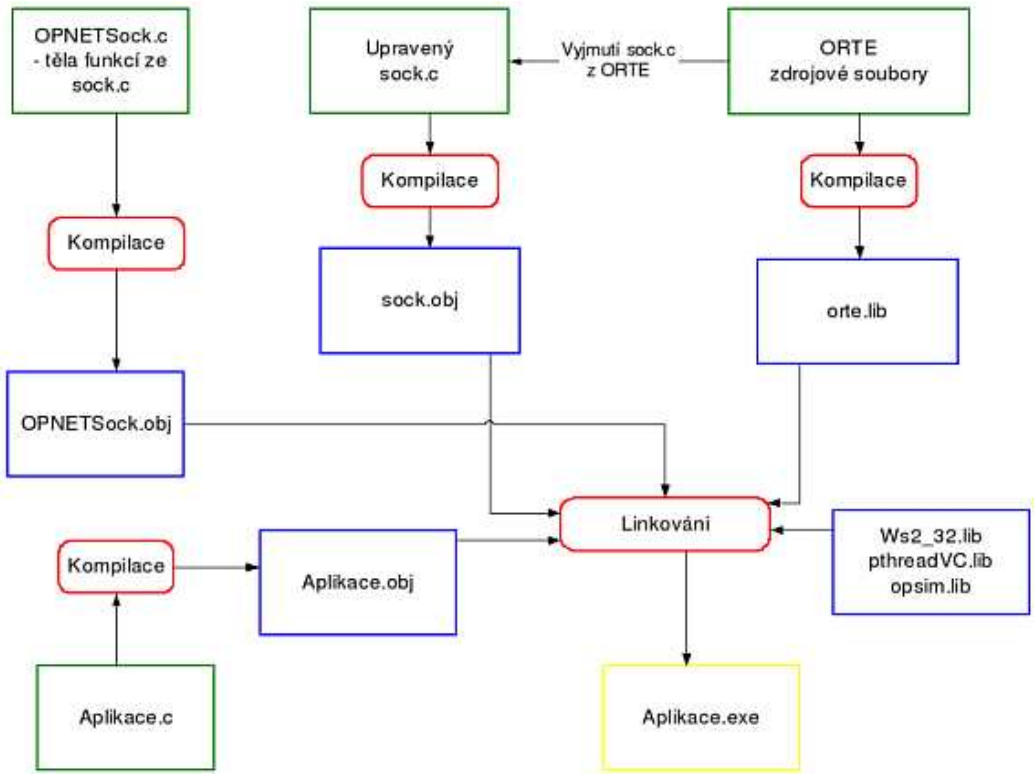
Aby byla zachována původní funkčnost celého ORTE, potřebné úpravy kódu byly prováděny pomocí tzv. direktiv kompilátoru jazyka C (*if defined, elif defined, endif* apod.) tak, že ke stávajícímu kódu uvedených funkcí jsem přiřadil podmínku

```
#if defined (SOCK_OPNET)
```

potom je samozřejmě nutné v některém z hlavičkových souborů tuto konstantu definovat.

Logická struktura celé aplikace je vidět na následujícím obrázku (obr. 5.24). Je vidět, že se celý program skládá ze čtyř základních částí:

- Původní zdrojové souboru ORTE.
- Upravený soubor *sock.c*.
- Soubor *OPNETSock.c* s těly OPNET variant funkcí z *sock.c*.
- Kód samotné aplikace.



Obrázek 5.24: Blokové schéma externí aplikace

Z obrázku 5.24 je také vidět postup kompilace a linkování celé aplikace. Hlavičkové soubory zde pro zjednodušení nejsou uváděny.[33]

Postup je následující:

1. Ze zdrojových souborů ORTE byl vyjmut soubor *sock.c* a ze zbytku byla vytvořena knihovna *liborte.lib*

2. Do souboru *sock.c* byl za pomocí direktiv kompilátoru a konstanty OPNET_SOCK doplněn kód pro funkčnost s OPNET Modeler.

3. Byl vytvořen soubor *OPNETSock.c*, který obsahuje těla nových funkcí ze souboru *sock.c* a dalších funkcí pomocných.

4. Na základě ukázkové aplikace z projektu ORTE byla vytvořena aplikace která demonstruje funkčnost a využití komunikačního rozhraní s OPNET Modeler.

Přesně obrázku 5.24 je poté z každého zdrojového souboru vytvořen kompilátorem objekt (soubor *.obj*) a všechny objekty jsou pak s přídavnými knihovnami slinkovány do jednoho *.exe* souboru.

[33]Kompletní výpis kódu včetně hlavičkových souborů je uveden v příloze.

Knihovny nezbytné pro chod aplikace jsou tyto:

- *liborte.lib* - již zmíněná knihovna s celým ORTE.
- *pthreadVC.lib* - knihovna pro práci s POSIX[34] vlákny.
- *opsim.lib* - hlavní knihovna pro propojení s OPNET Modeler.
- *ws2_32.lib* - systémová knihovna pro práci se sokety. (je nutná z důvodu zachování kompatibility s variantami ze *sock.c*.)

Po vytvoření *.exe* souboru se celá simulace spustí příkazem

```
nazev_aplikace.exe -net_name nazev_site_v_opnetu \
                   -ef nazef_ef_souboru
```

Název sítě lze nejjednodušeji zjistit z titulku konfiguračního dialogu pro spuštění simulace v OPNETu, parametr *-ef* definuje tzv. *enviromental file*, což je soubor popisující parametry simulace a který je vytvořen editací konfiguračního dialogu simulace v OPNETu a následným uložením projektu. Fyzicky jde o soubor s příponou *.ef*

Pro zjednodušení celého procesu kompilace a linkování jsem vytvořil dva dávkové soubory[35]. Pomocí prvního souboru jsou vytvořeny objekty z *OPNETSock.c* a *sock.c* a druhý dávkový soubor kompiluje kód samotné aplikace a linkuje všechny objekty a knihovny do výsledného .exe souboru.

Samozřejmě by to šlo udělat i v rámci jednoho souboru, ale toto rozdělení je lepší z hlediska logické struktury aplikace, větší část práce byla na souborech *OPNETSock.c* a *sock.c*, které bylo častěji nutno rekompilovat a upravovat.

Nyní se pokusím popsat činnost nejdůležitějších částí celé externí aplikace s tím, že funkčnost kosimulace s OPNET Modeler bude vysvětlena v následující kapitole.

Na tomto místě je důležité říci, že ačkoliv výsledek je jeden spustitelný soubor, tak zde stále máme dva rozdílné programy a jak je dále popsáno, mohou si vzájemně předávat řízení aplikace, navíc veškerá komunikace probíhá jedině přes již dříve popsané rozhraní a externí aplikace nesmí používat standardní funkce OPNETu.

OPNET poskytuje pro kosimulaci tzv. ESA API, tedy sadu funkcí, které jediné mohou pracovat s OPNETem tedy manipulovat s rozhraním, předávat řízení OPNETu, zjišťovat stav v jakém je OPNET část simulace apod. Pro tyto funkce je nutné aby byl v programu vložen hlavičkový soubor *esa.h* umístěný v adresáři OPNETu, který obsahuje prototypy těchto funkcí.

[34]Portable Operating System Interface

[35]uvedeny v příloze.

Toto je hlavní soubor ORTE, jehož funkcemi ORTE posílá svá data do sítě. Zkráceně zde budou popsány všechny upravené funkce a jejich činnost.

5.3.1 sock_start

Na této funkci je ukázán způsob úpravy pro OPNET kód:

```
#if defined(SOCK_BSD) || defined (SOCK_RTL)
  return 0;
#elif defined (SOCK_WIN) && !defined (SOCK_OPNET)
/* kod pro Windows */
return 0;
#elif defined (SOCK_OPNET)
/* kod pro OPNET Modeler */
return 0;
#endif
```

V části pro OPNET se nejprve inicializuje spojový seznam obsahující strukturu, která popisuje vždy jeden soket, vytvořený dále popsanou funkcí *sock_bind()*. Struktura obsahuje zatím pouze tzv. *file descriptor*, tedy jakési ID soketu, a port na kterém je soket vytvořen.

Dále je zde možnost inicializovat proměnné, které jsou využívány pro práci s vlákny a nakonec je zde inicializována tzv. *call back* funkce, která se stará o příchozí data z OPNETu, a dále je zde volána funkce, pomocí které externí aplikace předává řízení nad programem do OPNETu, a jejíž pomocí může OPNET provést inicializaci své části simulace.

```
/* registrace callbackove funkce pro obsluhu preruseni z OPNETu */
Esa_Interface_Callback_Register(*esa_state_pptr, status, \
   Esa_Interface_Get(*esa_state_pptr, \
   "top.node_0.extern_interface.out_data_port"), \
   opnet_callback,(EsaT_Interface_Array_Callback_Proc)NULL, \
   (void *)NULL);
/* Vykonani inicializacnich akci OPNETU */
Esa_Execute_Until(*esa_state_pptr, status,2, \
   ESAC_UNTIL_INCLUSIVE,time_reached_ptr, num_events_ptr);
```

Registrační funkci pro *call back* je předáván ukazatel na obecný objekt rozhraní, který je vytvářen funkcí *Esa_Init* (popsána dále), dále proměnná, kam se ukládá status o vykonání této funkce, poté je volána další funkce pro získání ID rozhraní, ke kterému má být daná *call back* funkce přiřazena. Dalším parametrem je název *call back* funkce a nakonec je nastaveni parametrů.

Druhá funkce má prví dva parametry shodné, třetí je maximální čas, který je OPNETu poskytnut na vykonání naplánovaných akcí, dalším parametrem je pak konstanta určující, jakým způsobem bude probíhat běh části programu řízené OPNETem a jako poslední jsou dvě proměnné, do kterých je po skončení části řízené OPNETem, uložena hodnota proběhlých událostí a času v OPNETu.

5.3.2 sock_bind

Jedna z důležitých funkcí, která se stará o získávání UDP portů Je zde proto uveden celý kód varianty této funkce pro OPNET Modeler.

```
/* ziskam ID rozhrani pro ridici prikaz */
cmd_int_id = Esa_Interface_Get (*esa_state_pptr, \
             "top.node_0.extern_interface.command");
/* jestlize je port nenulovy, po kontrole ho poslu OPNETU */
if (port && is_port_free(port) && port>=1025 ) {
   Esa_Interface_Value_Set (*esa_state_pptr, status, \
                 cmd_int_id, ESAC_NOTIFY_IMMEDIATELY,port);
   sock->fd = add_socket(port);
   sock->port = port;
 } else if (!port) {
 /* parametr port ma hodnotu nula = pridelujeme automaticky */
   int akt_port = get_free_port();
   Esa_Interface_Value_Set (*esa_state_pptr, status, \
                 cmd_int_id, ESAC_NOTIFY_IMMEDIATELY,akt_port);
   sock->fd = add_socket(akt_port);
   sock->port = akt_port;
 };
 return 0;
```

Vstupem do funkce je *soket*, což je vlastně pouze struktura o položkách *file descriptor* a *port*, který chceme k tomuto soketu přiřadit

Nejprve je tedy získáno ID rozhraní, kam bude poslán řídící příkaz (v tomto případě to je požadavek na registraci portu, ale lze tento port využít i pro ostatní možné řídící příkazy). Jak je vidět, parametrem funkce je název daného rozhraní uvedený v zápisu popisující celou strukturu OPNET modelu.

Pokud je hodnota požadovaného portu nulová, pak to znamená, že systém (tedy naše funkce) si může nastavit port jak potřebuje. Na toto slouží funkce *get_free_port()* popsaná v části 5.4.

Pokud je větší než nula, nejprve je pomocí funkce *is_port_free()* ověřeno, zda je požadovaný port volný a zda je větší nebo roven 1025. Pokud je tomu tak, požadovaný port je přiřazen danému soketu.

Ověřování i nastavování portu je prováděno úpravami již popsaného spojového seznamu soketů.

Poznámka
Bylo počítáno, a v původním *sock.c* taková funkce je, i s funkcí, která bude sokety rušit a UDP porty uvolňovat, ale bohužel v OPNETu je pouze funkce pro registraci a nikoliv pro zrušení portu.

U následujících funkcí *sock_recvfrom* a *sock_sento* je k stejný (samozřejmě kromě jmen volaných funkcí), proto je zde uvedena pouze jedna a to funkce, která slouží k přijímání dat. Obě funkce jsou inline, tedy jsou vkládány při překladu přímo na místo, kde je uvedeno jejich volání.

5.3.3 sock_recvfrom

```
inline int
sock_recvfrom(sock_t *sock, void *buf, int max_len, \
              struct sockaddr_in *des,int des_len) {
#if !defined(SOCK_OPNET)
  return recvfrom(sock, buf, max_len, 0, (struct sockaddr*)des, \
                  &des_len);
#else
  return receive_from_soket(sock,buf,max_len,des);
#endif
```

Vidíme, že OPNETu se týká pouze poslední řádka, na které je volána funkce *receive_from_soket()*, zajišťující zpracování dat z OPNETu. Jí jsou předávány všechny parametry, kromě posledního (stejné je to i u *sock_sento*), který popisuje velikost datové proměnné určující cílovou adresu.[36]

Podrobněji jsou funkce realizující OPNET variantu standardních funkcí *sock_recvfrom* a *sock_sento* popsány v části 5.4.

Vidíme, že změny v souboru *sock.c* nejsou nijak veliké, hlavní je až implementace uvedených funkcí pro propojení do OPNETu popsaná v následující kapitole.

5.4 Externí aplikace - soubor *OPNETSock.c*

Tento soubor obsahuje stěžejní část externí aplikace, a to funkce realizující propojení do OPNET Modeleru a externí aplikace a další pomocné funkce, které jsou pro toto propojení nezbytné.

[36]Toto má vliv, pouze tehdy, pokud máme v daném počítači (nódu) více síťových rozhraní, což není náš případ.

5.4.1 add_soket

Tato funkce se stará o přidávání dalších soketů do již zmíněného spojového seznamu soketů. Jejím parametrem je hodnota přidávaného portu a funkce vrací *file descriptor* přidaného soketu.

Kód funkce je triviální, proto je uveden pouze v příloze B.

5.4.2 is_port_free

Jedna z pomocných funkcí pracujících se spojovým seznamem soketů. Tato funkce prochází seznam a ověřuje, zda číslo portu, předávané jako vstupní parametr není již použité. Pokud ano, funkce vrací nulu, pokud ne, pak vrací jedničku.

Kód uveden opět v příloze B.

5.4.3 get_free_port

Poslední z pomocných funkcí pracující se seznamem soketů. Tato funkce zajišťuje získávání volného portu v tom případě, že při volání funkce *sock_bind* je na místo požadovaného portu uvedena hodnota nula.

Funkce prochází seznam soketů a snaží se nalézt ještě neobsazený port, který je ale menší než 7400, protože na těchto portech komunikují objekty typu *Manager* a porty obsazují tím, že si ho registrují přímo.

5.4.4 send_to_socket

Základní funkce realizující OPNET Modeler variantu odesílání dat. Místo UDP vrstvy systému jsou však data odeslána do OPNETu.

Nejprve jsou získány ID rozhraní pro data a pro velikost dat a poté je vytvořen datový buffer, na jehož začátek jsou uložena data týkající se odesílaných dat (informace o soketu, cílové adrese, portu apod.) a poté je do něj uložen datový buffer, který je předáván funkci jako vstupní parametr.

⋮

```
/*Ziskam ID rozhrani pro data a jejich velikost */
data_iface = Esa_Interface_Get (*esa_state_pptr, \
             "top.node_0.extern_interface.in_data_value");
size_iface = Esa_Interface_Get (*esa_state_pptr, \
             "top.node_0.extern_interface.in_data_size");

/* vytvorim datovy paket pro odeslani*/
*buffer= sock->fd;
```

```
*(buffer+1) = sock->port;
*(buffer+2) = ntohl(des->sin_addr.s_addr);
*(buffer+3) = ntohs(des->sin_port);
*(buffer+4) = len;
for (i=0;i<len;i++) {
  *(buffer+i+5) = *(data+i);
  }
```

⋮

Při zadávání cílové adresy a portu je důležité převádět předávané hodnoty pomocí funkcí *ntohl*[37] a *ntohs*, protože ORTE převádí adresu i port do tzv. *network* formátu pomocí opačné funkce *ntohl*.

OPNET však tento formát nezná, proto je třeba touto funkcí převést formát zpět na pouhé převedení řetězce IP adresy na číslo, protože výstup funkce, která převádí řetězec IP adresy na číslo je shodný s výstupem obdobné funkce z OPNET Modeler.

V další části se již odesílají data na rozhraní do OPNETu, protože nejde pouze o zápis jedné hodnoty, ale dvojice hodnot, která musí být zpracována současně, je tato část kódu chráněna pomocí tzv. *mutexů*,[38] které zabraňují přístupu k specifické části programu případným dalším vláknům, které by chtěly odesílat data a mohlo by se stát, že v daný okamžik je na rozhraní do OPNETu jedna hodnota patřící k jednomu vláknu a druhá hodnota k druhému.

⋮

```
pthread_mutex_lock(&mutex_send);
Esa_Interface_Value_Set (*esa_state_pptr, status, size_iface, \
                         ESAC_NOTIFY_NEVER,len+5);
Esa_Interface_Value_Set (*esa_state_pptr, status, data_iface, \
                         ESAC_NOTIFY_IMMEDIATELY,buffer);
pthread_mutex_unlock(&mutex_send);
```

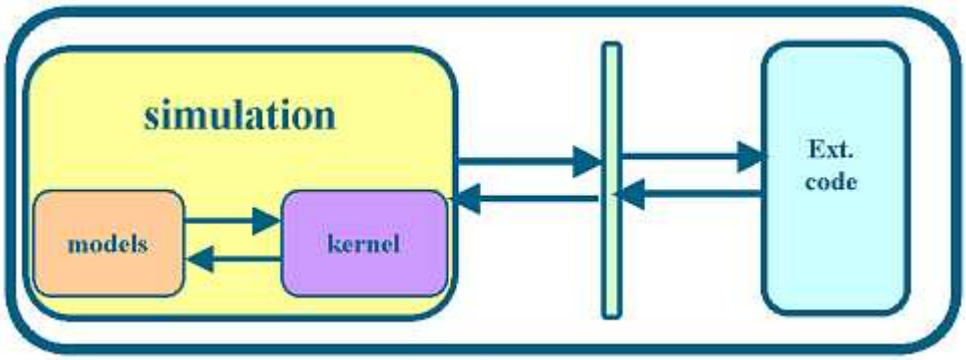
⋮

Opět zde využívám funkce z ESA API pro zapsání hodnot na rozhraní, kterým předávám již zmiňovaný ukazatel a proměnnou status, dále potom ID rozhraní, na které budu zapisovat, způsob upozornění OPNETu na to, že došlo k změně hodnoty na rozhraní (vysvětleno v kapitole 6) a posledním parametrem je datová proměnná.

[37] jde o zkratku *network to host long* - pro adresu a *network to host short* pro port

[38] mutual exclusion object, více informací v [14]

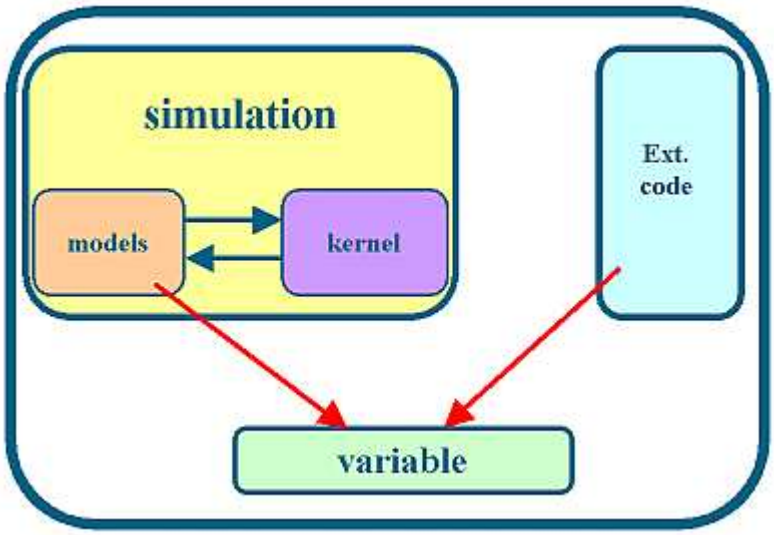
Způsob, jak lze předávat data mezi OPNET Modelerem a externí aplikací ukazují následující obrázky.



Obrázek 5.25: Předávání dat skrze definované rozhraní.

Na obrázku 5.25 je znázorněna komunikace mezi OPNET Modeler a externí aplikací pomocí oboustranného zápisu a čtení z(na) definované rozhraní.

Další možností výměny dat je pomocí tzv. sdílené proměné, kdy například pomocí hlavičkového souboru nadeklarujeme proměnnou a tento soubor potom vložíme jak do OPNETu tak do externí aplikace (obr. 5.26). Tento způsob sice



Obrázek 5.26: Předávání dat pomocí sdílené proměnné.

zjednodušuje celou výměnu dat, protože nemusíme používat žádné speciální funkce, na druhou stranu přináší dosti nevýhod, jako je ošetření současného zápisu do sdílené proměnné, nemožnost efektivně zjistit, zda proměnná obsahuje již aktuální data apod.

v této práci je využíván první způsob a částečně i druhý. Jak bylo uvedeno dříve, je možné na rozhraní zapisovat i pole hodnot. Bohužel se nepodařilo docílit optimální funkčnosti předávání pole dat tímto způsobem a tak je zde předávání dat vyřešeno tak, že externí aplikace alokuje v paměti programu místo, které naplní daty a na rozhraní je zapsán pouze ukazatel na tyto data.

OPNET potom pomocí tohoto ukazatele data získá a po zpracování místo v paměti uvolní. Stejný způsob funguje i opačným směrem. OPNET zapíše data do alokované paměti a externí aplikaci předá ukazatel. Aplikace data zpracuje a paměť uvolní.

5.4.5 receive_from_soket

Funkce realizující přijetí a zpracování dat z OPNETu, návratovou hodnotou je délka přijatých dat a dále funkce naplňuje předávaný buffer a do proměnné *des* ukládá adresu a port z kterého přišla data.

Základní rozdělení funkce je na dvě části - čekání na data a zpracování dat. Kód první části je:

⋮

```
pthread_mutex_lock(&mutex_c1);
\* pokud neni zprava pro pozadovany port cekej *\
 while (sock->port != cond_port) {
       pthread_cond_wait(&cond_receive,&mutex_c1);
       };
pthread_mutex_unlock(&mutex_c1);
```

Vidíme, že pokud není proměnná *cond_port* shodná s portem, na kterém očekáváme data (určený parametrem soket), přijímací vlákno usne a čeká na probouzecí signál. Pokud ten přijde, znovu se ověří podmínka shodnosti portů a pokud je splněna pokračuje se dalším kódem.

⋮

```
Esa_Interface_Value_Get (*esa_state_pptr,status, \
  Esa_Interface_Get(*esa_state_pptr, \
  "top.node_2.extern_interface.out_data_size"),&delka);
Esa_Interface_Value_Get (*esa_state_pptr,status, \
  Esa_Interface_Get(*esa_state_pptr, \
  "top.node_2.extern_interface.out_data_value"),&data);

if (delka > max_len ) max_delka = max_len; else max_delka = delka;

for (i=5;i<5+max_delka;i++) {
    *(((char*)buf)+i-5)=*(data+i);
  }
```

```
des->sin_addr.s_addr =htonl(*(data+2));
des->sin_port = htons(*(data+3));

return max_delka;
```

⋮

Nejprve tedy získáme z rozhraní délku dat a ukazatel na první položku z datového pole. Porovnáním skutečné délky dat a maximální požadované délky přijímaných dat se získá délka přijímaných dat a cyklicky se prochází celé pole dat(kromě hlavičky). Po přetypování se vše ukládá do bufferu. Nakonec se nastaví ostatní proměnné.

5.4.6 opnet_callback

Poslední důležitou funkcí ze souboru *OPNETSock.c* je funkce starající se o oznámení příchozích dat. Stejně jako když externí aplikace zapíše na rozhraní data a v OPNETu je vyvoláno specifické přerušení, tak i po zápisu OPNETu na rozhraní je možné dát to externí aplikaci nějak vědět, o což se právě stará tato funkce.

Jak je uvedeno u popisu funkce *sock_start*, je *opnet_callback* funkce přiřazena k rozhraní pro port příchozích dat, a tak je vždy vyvolána pokud OPNET na tento port zapíše data.

⋮

```
Esa_Interface_Value_Get (*esa_state_pptr, status, \
   Esa_Interface_Get(*esa_state_pptr, \
   "top.node_0.extern_interface.out_data_port"),&cond_port);
pthread_cond_signal(&cond_receive);
```

⋮

Vidíme, že kód je opravdu velice jednoduchý, základem je načtení hodnoty portu na který přišla data do proměnné *cond_port* a zavolání signálu, kterým probudíme všechna případné přijímací vlákna a to, které čeká na data právě z daného portu se probudí a pokračuje v jeho zpracování.

5.5 Externí aplikace - soubor *aplikace.c*

Toto je hlavní soubor aplikace, který obsahuje funkci *main()* a v kterém probíhá celý uživatelský program. Zde jsou uvedeny pouze základní funkce a proměnné, které jsou potřeba pro kosimulaci nezávisle na konkrétní aplikaci. Ukázkový aplikace pak bude popsána v následující kapitole.

```
⋮

int main( int argc, char * argv[] ) {

/* OPNET initicializace */
 Esa_Main (argc, argv, 0);
/* Inicializace simulace */
Esa_Init (argc, argv, ESAC_OPTS_NONE, esa_state_pptr);
/* Nahrani modelu site a pripadnych dalsich modulu */
Esa_Load (*esa_state_pptr,ESAC_OPTS_NONE);

/* Zde je kod samotne externi aplikace */

/* Ukonceni OPNETovske casti simulace */
Esa_Terminate (*esa_state_pptr, ESAC_TERMINATE_NORMAL);

return 0;
}
```

První z funkcí ESA API *Esa_Main* určuje počátek kosimulace, jde v podstatě o OPNET variantu funkce *main*, je vidět, že používá vstupní argumenty funkce *main*.

Druhá funkce *Esa_Init* vykonává základní inicializaci a vytváří globální datovou strukturu popisující kosimulaci a navrací pointer na ni. Ten který je pak používán ve všech funkcích z ESA API.

Poslední funkce se stará o nahrání a nastavení simulované sítě a ostatních přidružených souborů. Poté jej již možné vykonávat externí program. Ukázky dalších funkcí z ESA API pro práci s rozhraním jsou ukázány v kapitole 7.

Celkově lze říci, že funkční implementace samotného komunikačního rozhraní mezi externí aplikací a OPNET Modelerem není příliš složitá a funkce lze jednoduše modifikovat a upravovat podle aktuální potřeby, jak je popsáno v následující kapitole. Je popsána funkčnost celé kosimulace z globálního hlediska s informacemi o problematických částech.

6 Kosimulace

Celý problém kosimulace mezi externí aplikací a OPNETem (případně mezi OPNETem a OPNETem) lze řešit dvojím způsobem[39] a to buď:

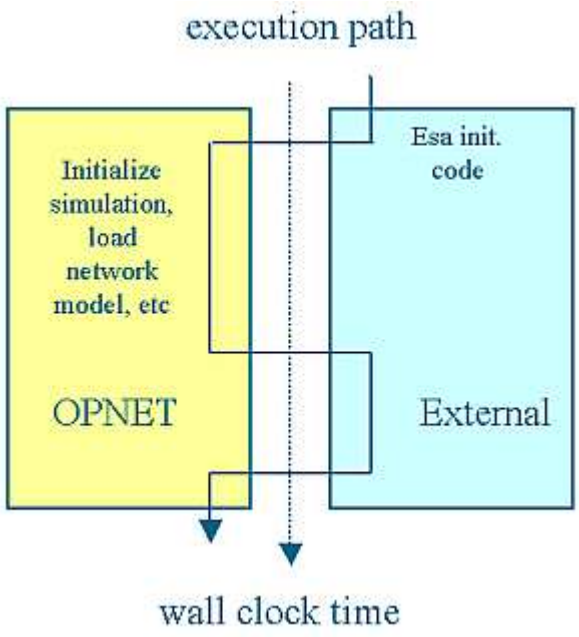
- Externí aplikace jako knihovna při-linkovaná do OPNET Modeleru
- OPNET simulace je zahrnuta v externím programu

V našem případě byla zvolena první varianta. Pro tuto variantu je ještě možnost kompilace celého programu za použití funkcí z OPNET Modeleru (pro tuto variantu je nutno mít v modelu rozhraní, popsaném v úvodu kapitoly 5.2, definován soubor typu *simulator description*) nebo pomocí externího kompilátoru, jak je použito v této práci.

V této krátké kapitole je popsán průběh celé kosimulace, s ohledem na fakt, že princip fungování jednotlivých prvků a funkcí byl vysvětlen již v předchozích částech.

Zde se zaměřím pouze na popis jednotlivých akcí a událostí během kosimulace.

Celý průběh kosimulace si lze představit podle následujícího obrázku (obr. 6.27):

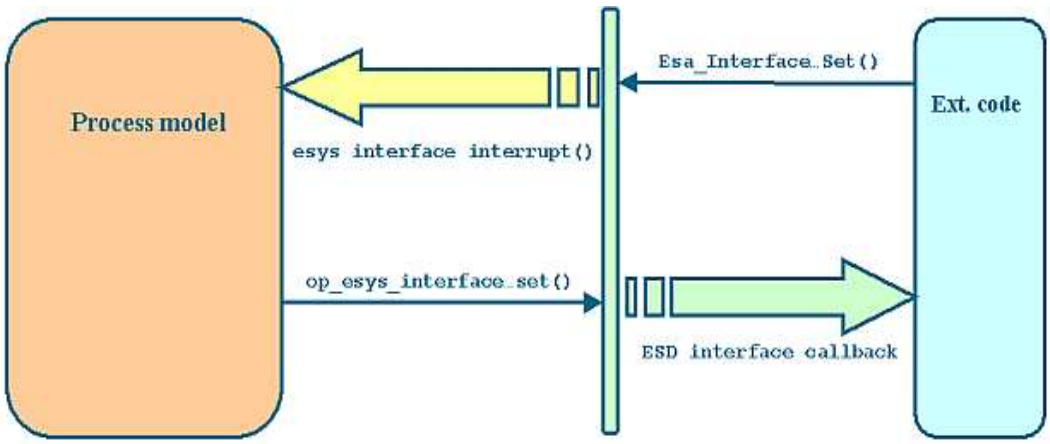


Obrázek 6.27: Průběh kosimulace

[39] Více informací o způsobu kosimulace viz. [12]

Nejprve se v externí aplikaci zavolají inicializační funkce popsané v závěru předchozí kapitoly. Tím se na čas předá řízení OPNETu aby mohl vykonat svoji inicializaci. Poté už celá kosimulace probíhá na základě přepínání řízení mezi externí aplikací a OPNETem.

Toto přepínání se děje na základě přerušení, která jsou vyvolávána jednotlivými zápisy na rozhraní, a to buď ze strany externí aplikace nebo ze strany OPNETu, jak to ilustruje obrázek 6.28.



Obrázek 6.28: Předávání řízení pomocí vzájemných přerušení.

Celý proces vzájemné komunikace funguje následovně[40]:

1. Externí aplikace připraví data, která bude chtít poslat do OPNETu. Jde o ukazatel na pole dat a dále jejich velikost. U funkce pro zapsání dat na dané rozhraní (*Esa_Interface_Value_Set*) existuje parametr, který určuje, zda bude OPNET informován o nové hodnotě dat na rozhraní či nikoliv.

   Protože obě hodnoty zapisované na rozhraní spolu souvisí a potřebujeme je zpracovat najednou, je na rozhraní zapsána nejprve jedna proměnná bez upozornění OPNETu a až při zápisu druhé proměnné nastavíme upozornění OPNETu na nová data.

2. Na základě změny dat na rozhraní je v OPNETu vyvoláno přerušení v procesu *extern_interface*, které spustí kód podle toho, na které rozhraní byla nová hodnota zapsána. Pro ilustraci předpokládáme, že byla zapsán ukazatel na data a dále jejich velikost. Tedy spustil se kód popsaný v části 5.2.4.

   Z těchto dat je procesem *extern_interface* vytvořen OPNET paket, který je pomocí standardní OPNET funkce *op_pk_send* odeslán procesu *udp_interface*. Ten z něj zjistí informace o cílové adrese a portu a odešle jej

[40]Popis není obecný, týká se implementace vzájemné komunikace realizované v této práci.

do procesu UDP vrstvy. Tímto končí kód vytvořený v rámci této práce. O zbytek se starají standardní modely procesů jednotlivých vrstev, linek apod.

3. Při příchodu paketu z modelu UDP procesu se vyvolá kód stavu *RECEIVE* z procesního modelu *udp_interface* (obr. 5.18), který datový paket pouze přepošle procesu *extern_interface*.

   Ten z něj vezme nejprve informace o tom, z které adresy a portu data přišly a pro který port jsou určena. Poté samotná data zapíše do pole alokovaného v paměti.

   Nyní by se měly všechny tyto informace zapsat na rozhraní. Problém ale může nastat v tom, že na straně externí aplikace se zpracovává informace o změnách na rozhraní *call back* funkce a pokud není dokončena, proces *extern_interface* zůstává ve stavu, ve kterém došlo k zápisu na rozhraní a nemůže přijímat další data.

   Jak víme z odstavce 5.4.6, v našem případě *call back* funkce pouze posílá signál pro probuzení přijímacího vlákna a po jeho dokončení může být externí aplikací spuštěno vlákno pro odeslání dat (jako reakce na data přijatá) a dostaneme se do výše popsané situace, kdy je celá kosimulace zablokována.

   Proto je k procesu *extern_interface* přiřazen tzv. *child process*, kterému se jako vstupní parametr předají ID všech rozhraní pro zápis a také všechny hodnoty dat (viz. oddíl 5.2.5) a který je pouze na rozhraní zapíše.

   Hlavní proces *extern_interface* se mezi tím může vrátit do stavu *IDLE* a zpracovávat případná další příchozí data. *Child* procesů může být v jeden okamžik spuštěno několik.

   Callback funkce je přiřazena právě k jednomu rozhraní (v našem případě k rozhraní pro hodnotu portu příchozích dat), takže samotný zápis hodnot pro externí aplikaci probíhá podobně jako pro OPNET. Nejprve je zapsán ukazatel na pole dat, potom jejich velikost a nakonec port na který přišly data a na základě tohoto zápisu je spuštěna *call back* funkce.

4. *Call back* funkce pomocí *Esa_Interface_Value_Get* uloží hodnotu portu, na který přišla data a spustí signál pro probuzení přijímací funkce.

5. Přijímací funkce, která očekává data, pomocí stejné ESA API funkce jako je v *call back* funkci, získá velikost příchozích dat a ukazatel na ně a provede jejich zpracování.

Představíme-li si celý průběh dle obrázku 6.27, tak dochází k tomu, že při zápisu proměnné na rozhraní s nastaveným upozorněním je řízení předáno OPNETu, který vykoná všechny operace, které z tohoto přerušení plynou a vrátí řízení zpět aplikaci.

Obdobně i naopak, pokud OPNET zapíše data na rozhraní s připojenou *call back* funkcí, je přerušeno vykonávání hlavního programu a vykoná se kód *call back* funkce.

Řízení lze OPNETu předat i přímo funkcí ESA API a to pomocí funkce *Esa_Execute_Until*, která umožní vykonat všechny plánované OPNET akce a po skončení (nebo uplynutí maximálního povoleného času) je řízení předáno zpět.

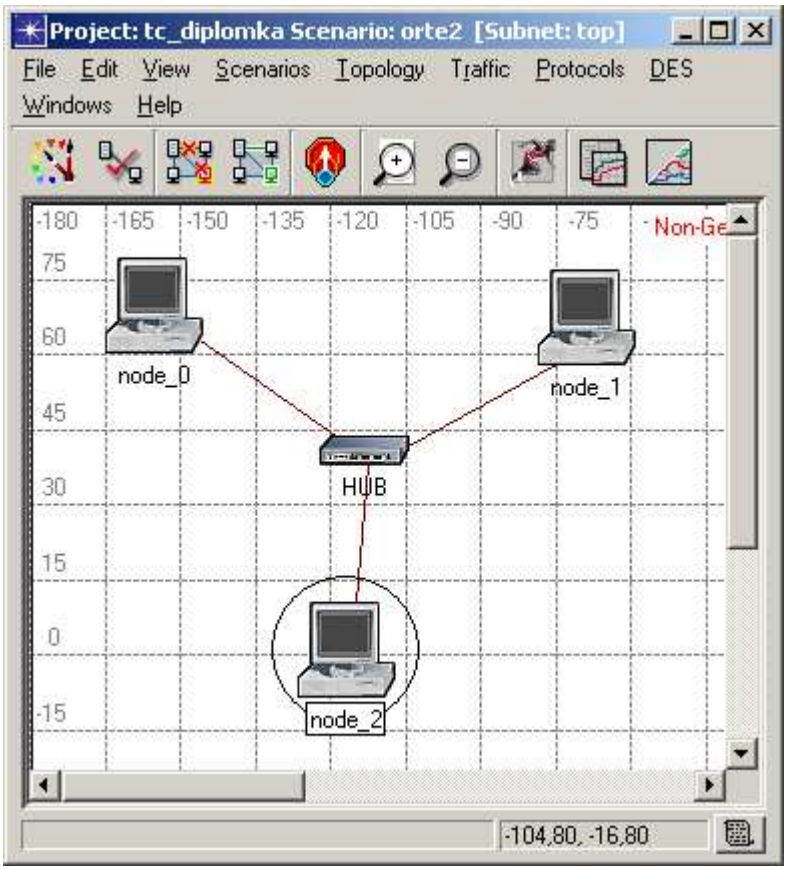
V této práci je tato funkce použita na začátku externí aplikace v rámci volání funkce *ORTEInit* pro nastavení OPNET modelu do pracovního stavu (většinou jde o procesní stavy IDLE) a dále kvůli testovacím výpisům, protože funkce pro výpis na obrazovku mají při běhu programu nízkou prioritu a chceme-li vypisovat testovací informace z OPNETu průběžně, je třeba po každé akci, která OPNET ovlivňuje, předat řízení OPNETu, aby tyto informace vypsal.

7 Ukázková aplikace

V této poslední části je popsána ukázková aplikace vytvořená pro demonstrační účely funkčnosti rozhraní mezi externí aplikací a OPNET Modelerem s využitím upravených variant funkcí pro implementaci aplikace projektu ORTE.

Tato aplikace na rozdíl od ORTE nepoužívá vlákna, takže musely být upraveny některé funkce.

Schéma zapojení simulované sítě je na obrázku 7.29.



Obrázek 7.29: Topologie testovací sítě.

Nódy *node_0* a *node_1* jsou vysílací a *node_2* přijímací. HUB je čtyř portový a vše je propojeno 100MBitovým Ethernetem

Pro účely prezentace funkce kosimulace byla aplikace navržena tak, že každý nód má své rozhraní, jehož struktura je ale identická a ke každému nódu je registrován jeden soket.

Vzhledem k tomu byla i *call back* funkce přesunuta do hlavni aplikace a její registrace k danému portu na nódu *node_2* do funkce *main*, hned za inicializační funkce ESA API.

Funkce upravené *call back* funkce spočívá v tom, že zavolá ORTE funkci *sock_recvfrom*, která data přijme a zpracuje.

Další úpravy ve funkcích ze souboru *OPNETSock.c* spočívají v tom, že funkce pro příjem dat čte hodnoty přímo z rozhraní nódu *node_2* a funkce zapisuje data na rozhraní podle toho, pro který port jsou data určeny. Podobná úprava je i v kódu *sock.c*, kte je nutné zasílat požadavek na registraci portu na odpovídající rozhraní.

Všechny upravené soubory jsou okomentovány v příloze.

Upravená definice *call back* funkce v hlavním programu před funkcí *main*:

⋮

```
void opnet_callback(void *state_ptr, double time, \
                    va_list vararg) {
sock_recvfrom(soket3,incomming_data,100,destination3,8);
return;
};
```

⋮

Ve funkci *main* jsou potom nadefinovány tři proměnné typu soket a tři proměnné typu sock_addr_in (struktura obsahující mj. adresu a port) a dále dvě proměnné *data1* a *data2* obsahující 5 a 500 znaků, sloužící jako testovací data.

Zbývající část programu vypadá následovně:

⋮

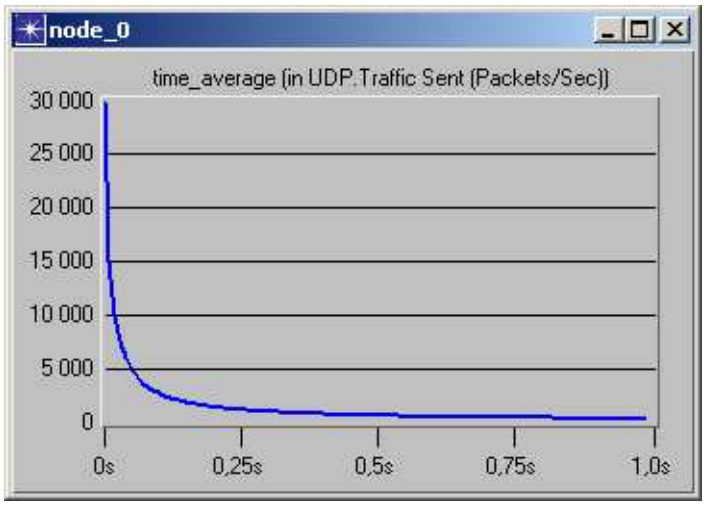
```
/* inicializace ORTE */
 ORTEInit();

/* prirazeni portu k~soketum */
sock_bind(soket1,1025);
sock_bind(soket2,1026);
sock_bind(soket3,8000);

/* odesilani dat ve 2 cyklech
for (i=0;i<200;i++) {
  sock_sendto(soket1,data1,500,destination3,sizeof(destination3));
  };
/* Zde je mozne vlozit zpozdeni mezi odesilanim */
for (i=0;i<100;i++) {
  sock_sendto(soket1,data1,500,destination3,sizeof(destination3));
  sock_sendto(soket2,data2,5,destination3,sizeof(destination3));
  };
```

```
/* predam rizeni OPNETu pro vypis hlaseni */
Esa_Execute_Until(*esa_state_pptr, status,5, \
    ESAC_UNTIL_INCLUSIVE, time_reached_ptr, num_events_ptr);
```

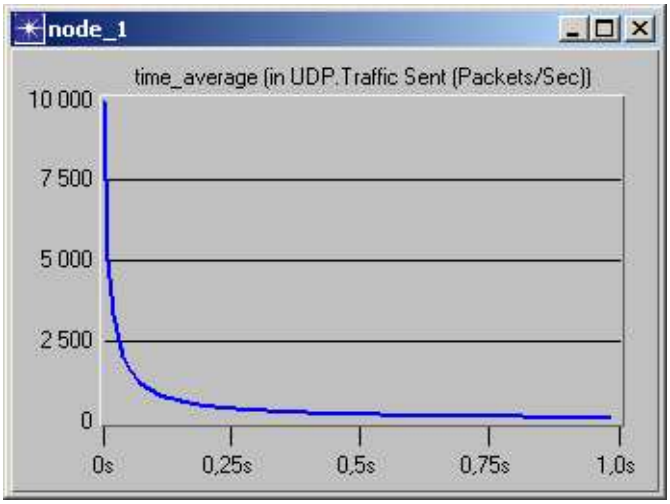
⋮

Následující obraázky ilustrují možnosti analýzy síťového provozu pomocí standardních statistik.

Je zobrazen provoz na UDP vrstvě obou nódů při odesílání dat. Další údaje zjistitelné ze standardních statistik UDP vrstvy jsou: počet přijatých paketů, počet odeslaných Bytů a počet přijatých Bytů.



Obrázek 7.30: Průměrný počet odesílaných paketů z nódu *node_0*.



Obrázek 7.31: Průměrný počet odesílaných paketů z nódu *node_1*.

8 Závěr

Tato práce přináší souhrn mnoha nových poznatků o simulačním prostředí OPNET Modeler, zvláště v oblasti programování kódu procesů, spolupráce a vzájemné komunikace jednotlivých procesů a používání externího kódu v OPNET Modeleru.

Byl vytvořen OPNET Model procesu umožnující přímou komunikaci s UDP procesem, díky němuž lze nyní v OPNETu vytvořit jakoukoliv vlastní aplikaci, která komunikuje skrz UDP vrstvu a velice efektivně analyzovat její vlastnosti a chování.

Na tento procesní model byl připojen model procesu pro komunikaci s externí aplikací. Na tomto procesu byla ověřena funkčnost rozhraní mezi OPNET Modelerem a externí aplikací. Ze získaných poznatků lze toto rozhraní implementovat mezi jakýmkoliv jiným modelem procesu OPNET Modeleru a tak jej v podstatě ovlivňovat a řídit z vnějšku.

Dále byla realizována implementace funkcí z programu ORTE, které zajišťují komunikaci s UDP vrstvou systému. Implementace byla realizována tak, aby aplikace nepoznala, že nekomunikuje skrze reálnou UDP vrstvu.

Funkčnost propojení z ORTE do OPNETu byla ověřena na testovací aplikací s třemi síťovými nódy. Testování a ladění přímo na nějaké konkrétní ORTE aplikaci nebylo možné, protože samotné ORTE je ještě ve fázi testování a bylo by dosti časově náročné analyzovat jeho kompletní funkčnost vzhledem k propojení do OPNETu a ani to nebylo cílem práce.

Rizikový je v tomto propojení reálné aplikace a simulačního prostředí fakt, že běžně komunikuje aplikace pouze s rozhraním lokálního nódu, kdežto při kosimulaci je třeba, aby aplikace rozesílala data na různá rozhraní podle toho, kolik nódů sítě je třeba simulovat. Jedno z možných řešení je naznačeno právě v ukázkové aplikaci, kdy je v odesílací funkci rozhraní vybráno na základě portu na který budeme odesílat i když by bylo asi vhodnější použít dělení dle IP adresy.

Dalším velkým přínosem pro budoucí práci s OPNET Modelerem jsou poznatky týkající se spolupráce OPNETu s externí aplikací a porozumění funkčnosti jak samotné simulace v OPNET modeleru, tak i chování OPNETu pokud je zakomponován do standardní Windows aplikace.

Ze získaných zkušeností mohu říci, že tento způsob využití OPNET Modeleru je velice efektivní a věřím, že bude i přínosem pro další práci na projektech katedry řídící techniky ČVUT FEL.

Literatura

[1] Stan Schneider, Gerardo Pardo-Castellote, Mark Hamilton *Can Ethernet Be Real Time?* Real-Time Innovations, Sunnyvale, CA 94089

[2] Dostálek L., Kabelová A. *Velký průvodce protokoly TCP/IP a systémem DNS*. Třetí aktualizované a rozšířené vydání. Praha: Computer Press, 2002. 542 s. ISBN 80-7226-675-6

[3] Klaška L. *CSMA/CD je když...* [online]. 15. září 2003. [cit. 2004-04-13]. <http://www.svetsiti.cz/Technologie.asp?clanekID=249>

[4] *Co (ne)najdete ve slovníku* [seriál online]. Computerworld č.32/93, 1993. [cit. 2004-04-13]
Dostupné z <http://www.earchiv.cz/a93/a332c120.php3>. ISSN 1213-077X

[5] *Komunikace v distribuovaných aplikacích* [online]. 2. srpna 2000. [cit. 2004-04-21] Dostupné z <http://www.herkules.cz/doc/sp/node11.html>

[6] *NDDS and Tools* [online]. [cit. 2004-05-03] Dostupné z <http://www.rti.com/products/ndds/index.html>

[7] *OCERA - ORTE* [online]. 1. května 2004. [cit. 2004-05-03] Dostupné z <http://www.rti.com/products/ndds/index.html>

[8] *RTI - Real Time Inovations* [online]. [cit. 2004-05-04] Dostupné z <http://www.rti.com>

[9] *projekt OCERA* [online]. 1. května 2004 [cit. 2004-05-04] Dostupné z <http://www.ocera.org>

[10] *OPNET Technologies–Making Networks and Applications Perform* [online]. 9. dubna 2004 [cit. 2004-05-05] Dostupné z <http://www.opnet.com>

[11] *NDDS Getting Started Guide, Version 3.0*, RTI - Real-Time Innovations, USA 2002

[12] *Session 1532 - Interfacing Multiple Simulators Using the Cosimulation API*, prezentace z konference OPNETWORK 2003, USA

[13] *OPNET Modeler - Product Documentation*, Dokumentace k OPNET Modeler verze 10.0

[14] *POSIX Threads Programming* [online] 1. února 2003, [cit. 2004-05-18] Dostupné z http://www.llnl.gov/computing/tutorials /workshops/workshop/pthreads/MAIN.html

A Seznam použitých zkratek

API	Aplication interface Aplikační rozhraní.
CORBA	Common Object Request Brooker Architecture
CSMA/CD	Carrier Sense Multiple Access/Collision Detection Vícenásobný přístup s odposloucháváním nosného kmitočtu s detekcí kolizí.
DCOM	Distributed extension of the Component Object Model
FTP	File Transfer Protocol Protokol pro přenos souborů.
HTML	HyperText Markup Language Značkovací jazyk pro psaní internetových stránek.
ICI	Interface Control Information Rozhraní pro předávání informaci mezi procesy OPNET Modeleru.
IP	Internet Protocol
ISO	International Standard Organization Mezinárodní organizace pro standardizaci.
MUTEX	mutual exclusion object objekt využívaný v práci s vlákny pro definování kritických sekcí.
OSI	Open System Interconnection Propojení definované normami ISO pro výměnu dat pomocí sedmi vrstev.
OCERA	Open Components for Embedded Real-time Applications Projekt pro vývoj komponent s otevřeným kódem pro real-time aplikace.
ORTE	OCERA Real Time Ethernet Implementace middleware pro real-time komunikaci.
POSIX	Portable Operating System Interface
RTOS	Real Time Operation Systems Prostředí real-time operačních systémů.

RTPS	Real Time Publish Subscribe
SMS	Short message Krátká zpráva (z oblasti telefonické komunikace).
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WWW	World Wide Web
XML	Extensible Markup Language

B Adresářová struktura přiloženého CD

/aplikace_ukazka

> **/orte_rozhrani** Obsahuje soubory s modifikovanými funkcemi pro rozhraní mezi OPNETem a ORTE přizpůsobené ukázkové aplikaci a dále dávkový soubor pro jejich kompilaci.
>
> **/program** Obsahuje zdrojový kód ukázkové aplikace s využitím funkcí z ORTE a dávkový soubor pro kompilaci.

/aplikace_orte

> **/orte_rozhrani** Obsahuje soubory s modifikovanými funkcemi pro rozhraní mezi OPNETem a ORTE připravené pro ORTE aplikaci komunikující na jednom nódu a dávkový soubor pro jejich kompilaci.
>
> **/program** Obsahuje zdrojový kód aplikace připravené na doplnění kódu real-time aplikace založené na ORTE a dávkový soubor pro jeho kompilaci.

/orte

Obsahuje soubory projektu ORTE verze 0.2.2, pro který byla práce navržena.

/opnet

> **/node_model** Obsahuje model nódu s implementovaným rozhraním do externí aplikace.
>
> **/proces_models** Obsahuje modely a zdrojové kódy modelů procesů které jsou ve výše uvedeném nód modelu použity.

/dokumentace

Obsahuje převážnou většinu off-line materiálů využitých v této práci.

/README.txt

Základní informace o struktuře CD.

/ABOUT.txt

Informace o autorovi a účelu CD.

/dp_2004_ciganek_tomas.pdf

Tento dokument ve formátu .pdf