Abstract

Active Networks present a network architecture that overcomes limits of traditional computer networks. A data unit of a network flow is associated with a program that is run, every time the data unit traverses a node. This feature enables a rapid adoption of new protocols and services.

The goal of this thesis is a development of new active network server, called Smart Active Node (SAN). Design of the server follows the Active Network concept, yet it learns from known shortcomings of existing implementations.

Poděkování

Rád bych touto cestou poděkoval inženýru Koutnému za cenné rady a připomínky k diplomové práci.

Dále bych chtěl srdečně poděkovat svým rodičům a přátelům, kteří mi vytvořili výborné podmínky pro studium na vysoké škole a kteří mě ve všech směrech podporovali.

V neposlední řadě chci poděkovat pracovníkům Západočeské univerzity v Plzni, které jsem během mého vzdělávacího procesu potkal, a kteří mi předali cenné znalosti a zkušenosti.

Obsah

Prohlášení

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

V Plzni 14. května 2008 ……

Michal Rejda

1 Úvod

Tradiční datové sítě, založené např. na IP protokolu, přenášejí pasivně data z jednoho cílového systému do druhého. Tradiční síť je k datům lhostejná a pouze je přenáší mezi uzly. Ve svém důsledku je v porovnání s aktivní sítí neflexibilní, protože se rozšíření nového protokolu, nebo zavedení nových služeb neobejde bez zdlouhavého procesu standardizace.

Paket aktivní sítě se nazývá capsule a je asociován s programem, který se vykonává při průchodu paketu uzlem sítě. V důsledku je tedy na každém uzlu sítě přítomen program, který rozumí přenášeným datům a je schopen s nimi pracovat. Uživatel takovéto aktivní sítě pak může do jednotlivých uzlů (například směrovačů) zavést svůj vlastní program a požadovat, aby byl vykonáván nad daty, které bude sítí přenášet. Jiným příkladem pak může být vlastní algoritmus směrování, řízení sítě, provádění statistik provozu a nespočetné množství dalších uživatelských programů.

1.1 Koncept aktivních sítí

Koncept aktivních sítí vznikl v Defense Advanced Research Projects Agency (DARPA) jako reakce na limity současných sítí. Jako klíčové limity byly přitom stanoveny:

- Nemožnost rychlého vývoje a nasazování nových protokolů a služeb.
- Model Internetu dovoluje pouze limitované zabezpečení a je stále obtížné vyřešit zabezpečené virtuální privátní sítě skrz něj.
- Mobilní stanice se chovají staticky, neboť není podporována mobilita v síti.
- Nelze vyvíjet služby jako adaptivní překódování dat, neboť jsou veškeré služby přímo vestavěné v jádrech operačních systémů.

Tyto nedostatky vedly k vytvoření nového síťového konceptu nazvaného aktivní sítě [1]. Tento nový koncept si klade za cíle:

- Vytvořit architekturu, která dovolí služby jednoduše vyvíjet a nasazovat.
- Vytvářet množství služeb a ty nabízet na Internetu.
- Povolit aplikacím specifickou kontrolu nad síťovými zdroji.
- Vytvářet a nasazovat zabezpečení už od základu. To například znamená vytvořit takovou infrastrukturu sítě, která se skládá z bezpečných komponent a bezpečnost není zajištěna centrálními prvky sítě jako firewally apod.

Podle [2] existují tři základní výhody, proč založit síťovou architekturu na výměně aktivních programů na místo pasivních paketů:

- Výměna aktivního kódu poskytuje základ pro adaptivní protokoly, dovoluje chytřejší interakci než výměna fixních datových formátů.
- Capsule poskytují prostředek k jemnozrnnému rozmístění aplikačně specifických funkcí na strategických místech sítě.
- Programová abstrakce poskytuje silný prostředek pro přizpůsobení sítě potřebám uživatele, dovoluje nasazování nových služeb mnohem rychleji a bez nutnosti zdlouhavé standardizace.

V dnešní době nejsou ještě aktivní sítě používané, ale už jsou používané aktivní technologie v individuálních koncových systémech a systémech nad end-to-end síťovou vrstvou. Například aby umožnily webovým serverům a klientům výměnu fragmentů programů. Nastává tedy otázka, proč neumožnit počítání na síťové úrovni.

1.2 Oblasti použití

Současně s rozvojem myšlenky konceptu aktivních sítí vznikly různé výzkumné projekty s cílem tuto novou myšlenku využít, nebo k ní přispět. Tyto projekty jsou zaměřeny na vytváření nových protokolů použitelných v těchto sítích, nebo jejich použití jako cílového prostředí, ve kterém bude řešena daná problematika.

1.2.1 Vývoj nových protokolů

Jedním z důvodů, proč používat aktivní sítě, je možnost vytváření a nasazování nových protokolů pokud možno co nejrychleji a bez nutnosti zdlouhavé standardizace. Zároveň tak s vykonáváním vlastního kódu na jednotlivých uzlech sítě vzniká prostředí pro vývoj a používání netradičních protokolů, které by bylo možné v tradičních sítích implementovat jen velice těžko nebo dokonce vůbec.

Jako příklad adaptovaného protokolu, původně vyvinutého pro prostředí mobilních agentů, můžeme uvést protokol NetAnts [3] na vytváření směrovacích tabulek. Algoritmus protokolu vychází z chování mravenců v přírodě. Na začátku protokolu je vytvořen „mravenec“ a náhodně se mu určí směr a doba života. Mravenec pak náhodně prochází síť a sbírá směrovací informace jednotlivých uzlů (každý uzel zná minimálně své sousedy).

Na sklonku svého života vytvoří nového mravence. Nový mravenec převezme nasbíraná data, přenese do inicializačního uzlu a upraví podle nich směrovací informace.

1.2.2 Přerozdělování zátěže pro distribuované výpočty

Jednou z možností, jak urychlit výpočet prováděný distribuovanou aplikací, je jeho rozložení na několik uzlů. Pokud to řešená úloha dovoluje, můžeme tak zkrátit dobu výpočtu. Protože ale může být každý uzel jinak výkonný, nebo jsou některé procesy distribuované aplikace méně náročné na síťové zdroje (paměť, přenosová kapacita, čas procesoru), vyplatí se použít metodu, která tyto okolnosti bere do úvahy. Je-li množina dostupných uzlů dynamická, přerozdělování a řízení výpočtu už není triviální. Aktivní sítě umožňují takové metodě využít vlastnosti, které nejsou v tradičních sítích dostupné.

Problémem rozdělování zátěže je řešen v [4, 5, 6, 7], kde byly zvoleny aktivní sítě jako cílové prostředí. Z tohoto důvodu vznikla implementace aktivních sítí nazvaná Grade32.

1.2.3 Využití pro přenos dat ve vesmíru

Kosmické záření, přímá viditelnost, velká rychlost, špatné počasí a další faktory znepříjemňují komunikaci mezi raketoplány, družicemi a Zemí. Z tohoto důvodu je vytváření sítí v této dynamicky se měnící struktuře značně komplikované, a proto se pro komunikaci používá směs na míru vytvořených protokolů místo řešení založených na protokolu IP.

Z tohoto důvodu se v práci [8] zabývají problémem, jak umožnit použití protokolu IP v takto nehostinném prostředí. Zavádějí IIP (Intelligent Internet Protokol) vycházející z myšlenky aktivních sítí a umožňuje vykonávat a distribuovat vlastní programy uvnitř sítě. Stejně jako aktivní sítě, IIP protokol se zaměřuje na problém s rychle se měnící topologií sítě a umožnit komunikaci mezi jednotlivými uzly ve vesmíru rozšířením protokolu IP.

1.3 Cíl práce

Tato práce představuje koncept aktivních sítí a některé experimentální implementace. Cílem je pak vytvoření serveru aktivní sítě (budeme jej nazývat Smart Active Node - zkratkou SAN) a odstranění některých nedostatků současných implementací. Bude také sloužit jako nástroj pro další výzkum a vývoj v oblasti aktivních sítí. Zajištění bezpečnosti v aktivních sítích je řešeno mimo rámec této práce.

Server SAN by měl být také náhradou serveru Grade32 [9] vyvinutého na Západočeské univerzitě v Plzni. Tento server nebyl primárně vyvíjen za účelem výzkumu aktivních sítí, ale z důvodu vývoje nové metody přerozdělování zátěže distribuovaných

aplikací v heterogenním distribuovaném prostředí. Z tohoto důvodu také nebyly adresovány všechny požadavky na ostrý provoz aktivních sítí, protože je nebylo nutné v rámci původního cíle řešit. Jako příklad můžeme uvést absenci metriky ve směrovací tabulce, nebo údaje o využívání položky směrovací tabulky, díky nimž by bylo možné implementovat důmyslnější algoritmy směrování v prostředí aktivních sítí. Významná je také vlastní interpretace programů Capsulí, která mimo jiné umožní správné přidělování zdrojů serveru nebo bezpečnost vykonávaného aktivního kódu. Ta se ale v prostředí Grade32 také neprovádí.

2 Existující implementace aktivních sítí

V současné době existují nejméně 3 významné implementace konceptu aktivních sítí, které budou popsány v této kapitole. Všechny tyto implementace však spadají do oblasti výzkumu a prozatím neexistuje žádné komerční řešení.

2.1 ANTS

Implementace konceptu aktivních sítí nazvaná ANTS [10, 11, 12] měla za úkol demonstrovat, že je myšlenka aktivních sítí smysluplná a stojí za to se jí i nadále zabývat. Je postavená na programovacím jazyce Java a pro transport dat přímo nepoužívá IP pakety, ale tzv. capsule. Program capsule je pak identifikován MD5 hashem a je tvořen vlastní třídou v jazyce Java.

ANTS se ve své implementaci nezabývají otázkou bezpečnosti. Spoléhá pouze na vlastnosti jazyka Java.

2.1.1 Identifikace aktivního uzlu

Aktivní uzel ANTS předpokládá své použití na IPv4 sítích a tímto je i dána identifikace těchto uzlů. Každý uzel je identifikován právě jedním 32 bitovým číslem, které odpovídá IP adrese tohoto uzlu.

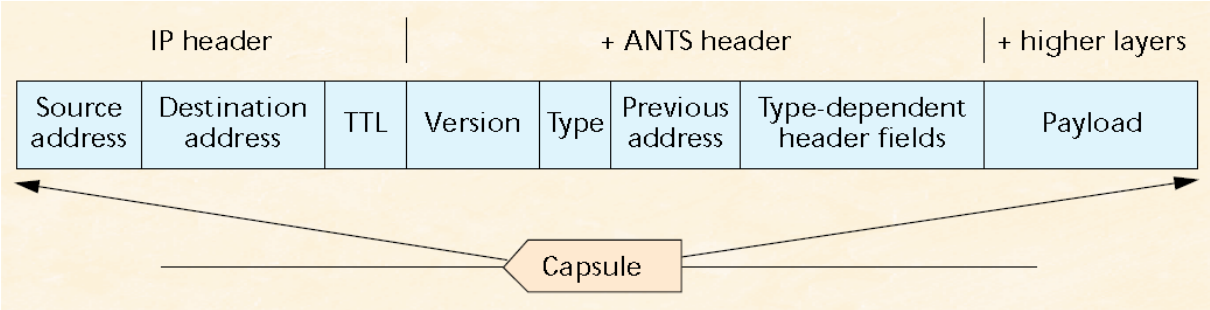
2.1.2 Capsule – jednotka přenášených dat

V ANTS se pro přenos dat používají capsule. Jejich formát vychází z toho, že jsou uzly sítě identifikované IP adresou (ať ve verzi IPv4 nebo IPv6). Dále se v síti předpokládají kromě aktivních uzlů také uzly, dovolující capsule přeposílat jen na základě jejich cílové adresy bez možnosti vykonávání jakéhokoliv programu.

Přeposílací uzly pro svou činnost využívají pouze první část Capsule zvanou IP header. Tato část určitým způsobem koresponduje s hlavičkou IP paketu, ale je značně zjednodušena a omezuje se pouze na tři položky. Odtud dokážou přeposílací uzly jednoduše zjistit směr přeposlání capsule, nebo ji případně zahodit při vyčerpání položky TTL (Time To Live).

Činnost aktivních uzlů je mnohem rozsáhlejší než činnost pasivních uzlů a z tohoto důvodu potřebují kromě základních údajů uvedených v části IP header další informace. Jedná se o položky uvedené v části ANTS header. Aby uzel věděl, jaký program má pro danou capsuli spustit, obsahuje Capsule položku Type. Jedná se o 128 bitový otisk spouštěného programu vytvořený algoritmem MD5.

Protože se může stát, že se potřebný program capsule nebude nacházet na uzlu, kam Capsule přišla, obsahuje Capsule položku Previous address. Tato položka obsahuje adresu aktivního uzlu naposledy vykonávajícího příslušný program. Je to tedy adresa uzlu, odkud se bude aktivní uzel snažit získat příslušný program v případě, že ho nemá ve svém lokálním úložišti.



Obrázek 1: Formát capsule ANTS [11]

Popis jednotlivých položek Capsule:

- Source address – adresa zdrojového uzlu.
- Destination address – adresa uzlu do kterého capsule v daný okamžik míří. Nemusí být stále stejná a asociovaný program ji může v průběhu cesty měnit.
- TTL – označuje maximální počet uzlů, přes které může capsule projít. Každý uzel tuto položku sníží, a když je nulová, capsule je automaticky odstraněna.
- Version – označuje verzi ANTS a položky které se nacházejí v části Type-dependent header fields.
- Type-dependent header fields – položky závislé na polích Version a Type.
- Payload – data vyšších vrstev.

2.1.3 Aktivní programy

Aktivní programy v ANTS představují Javové třídy odvozené od abstraktní třídy Capsule. Na základě toho odvozené třídy implementují sadu metod potřebných pro vlastní činnost capsule. Ty jsou pak volány při zpracování na aktivním uzlu.

2.1.4 Distribuce aktivních programů

Vždy, když chce uživatel přenášet capsule, musí nejprve zajistit dostupnost aktivního programu příslušné capsule na uzlu, odkud chce přenos začít. Tato podmínka musí být

splněna, aby mohly okolní uzly tento aktivní program získat, následně vykonávat, a v případě potřeby distribuovat do dalších uzlů.

V ANTS existuje jednoduchý protokol distribuce aktivních programů. Pokaždé, když data capsule dorazí na některý uzel, pokouší se uzel najít potřebný aktivní program v lokálním úložišti. Pokud už se tam potřebný program nalézá, nemusí se uzel starat o jeho získání od jiného uzlu a může jej začít vykonávat.

V případě, že se ale aktivní program v úložišti nenachází, nastupuje algoritmus distribuce aktivních programů. Tento algoritmus spočívá v odeslání capsule, mající za úkol požadovaný program získat a dopravit na uzel. Tato capsule je odeslána do uzlu, který aktivní program naposledy vykonával (zjistí se z hlavičky capsule). Pokud je uzel funkční, obsahuje ve svém úložišti potřebný aktivní program a capsule dopraví jeho kopii do uzlu, který program požadoval.

2.1.5 Dočasné úložiště stavu

V [10] se předpokládá udržování stavu nových služeb na jednotlivých uzlech uvnitř sítě. Jako příklad je uvedeno uchování dat pro zajištění vlastního směrování capsulí. Proto zavádějí dočasné úložiště pro data – soft-state. V tomto úložišti nejsou data uchovávána permanentně a při vložení nových dat do tohoto úložiště se specifikuje doba, po jejímž uplynutí může jádro uzlu data z úložiště odstranit. Není však ani garantována dostupnost dat do doby vypršení časového kvanta, s nímž byly do úložiště vloženy. Pokud je například server restartován, jsou všechna data v dočasném úložišti ztracena. Spolehlivé uložení dat by sice znamenalo zjednodušení programů některých služeb, ale značně by to zkomplikovalo návrh aktivního uzlu a zvýšení režie. Navíc by se stejně muselo zajistit odstraňování nepožívaných nebo starých položek. Podle [10] použití hard-state neřeší problém u uzlu, který je delší dobu nedostupný. Za tuto dobu se totiž mohla například změnit topologie sítě a s ní spojené směrovací informace.

V ANTS jsou informace ukládané aktivním kódem do soft-state ukládány v podobě dvojice [klíč, hodnota]. Aby se ale předešlo konfliktům klíčů různých protokolů, je každý záznam navíc spojen s jeho vlastníkem (aktivním programem). Informace jsou tedy v soft-state uloženy v podobě trojice, kde identifikace programu společně s klíčem tvoří jedinečný složený klíč. Není tak tedy možné přistupovat a měnit data jednoho protokolu z jiného.

2.2 Grade32

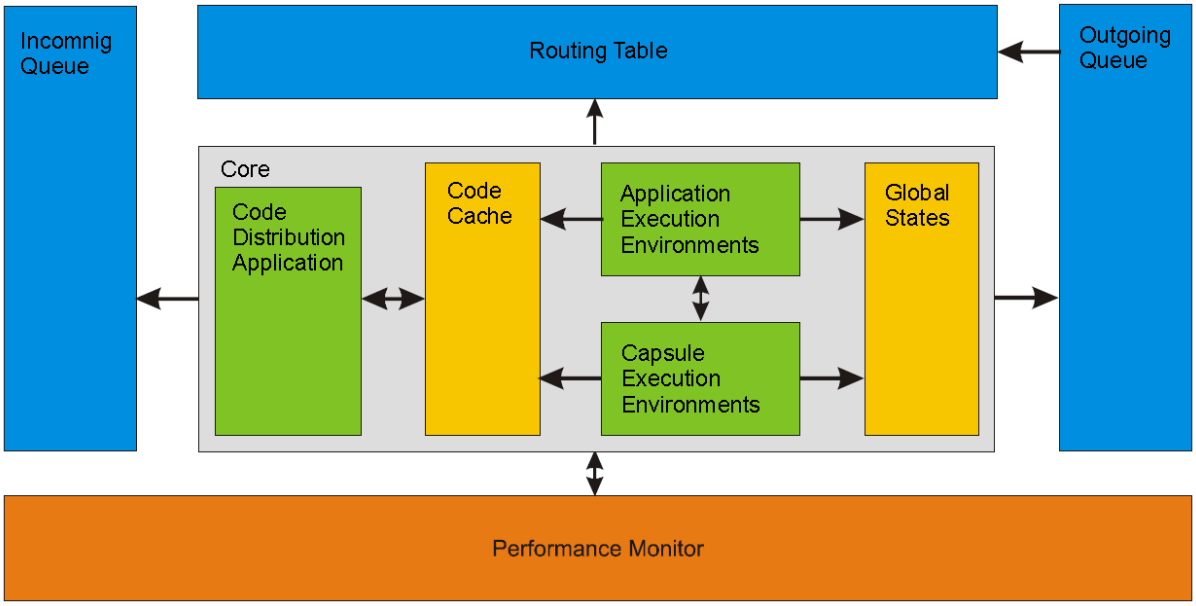
Pro potřeby vývoje nové metody přerozdělování zátěže distribuovaných aplikací v heterogenním distribuovaném prostředí bylo nutné zvolit vhodné cílové prostředí. Bylo rozhodnuto využít aktivních sítí, přičemž se předpokládalo použití některé z existujících implementací (např. ANTS). Postupem času se však dostupné implementace ukázaly jako nepoužitelné pro zamýšlený účel.

Z tohoto důvodu vznikla na Západočeské univerzitě v Plzni implementace aktivního serveru nazvaná Grade32. Tato implementace byla vytvořena v programovacím jazyce Delphi a je použitelná v prostředí operačního systému Microsoft Windows. Protože ale vznikla z důvodu vývoje nové metody přerozdělování zátěže distribuovaných aplikací v heterogenním distribuovaném prostředí (uvedené následně v [4, 5, 6, 7]), byl její vývoj směrován právě tímto směrem a nezabýval se některými detaily, které je nutné pro produkční činnost aktivních sítí řešit. Zároveň tak ale vlastně vzniknul startovní projekt v oblasti aktivních sítí.

2.2.1 Architektura uzlu

Server Grade32 je složen z několika komponent, jak je znázorněno na obrázku Obrázek 2:. Každá komponenta poskytuje specifickou funkcionalitu. Bloky označené modrou barvou zajišťují propojení s ostatními Grade32 servery. Oranžová komponenta (Performance monitor) monitoruje užívání zdrojů poskytovaných serverem. Šedivou barvou je znázorněno jádro aktivního serveru, dělící se dále na několik komponent. Zelené bloky představují prostředí pro vykonávání aplikací, capsulí a vestavěný protokol distribuce programů aplikací (capsulí). Žluté komponenty jsou zodpovědné za ukládání dat. Šipky vedou do komponent, od kterých jsou vyžadovány služby.

Co na obrázku zobrazeno není, je uživatelské rozhraní, sloužící k uživatelsky přívětivé práci se serverem. Uživatelské rozhraní běží v samostatném vlákně, jako i vstupní a výstupní fronta, jádro, distribuce programů a prostředí pro vykonávání aplikací a capsulí.



Obrázek 2: Grade32 Architecture [9]

Síťové spojení aktivních serverů je navázáno na dané IP adrese a portu, přičemž každý server naslouchá právě na jednom portu. Po navázání spojení si po něm hned mohou servery posílat capsule. Všechny příchozí capsule jsou vkládány do vstupní fronty serveru.

Ze vstupní fronty jsou capsule vybírány jádrem a předány k vykonávání v případě, že je jejich aktivní program na serveru dostupný. Pokud aktivní program dostupný není, musí se získat z jiného serveru. K tomu slouží jediný vestavěný protokol, ve kterém se používá pouze dvou typů capsulí. Capsuli s požadavkem na daný aktivní program a capsuli pro přenos požadovaného aktivního programu zpět. Server se snaží získat potřebný program z předchozího uzlu, odkud capsule přišla.

V průběhu vykonávání aplikací a capsulí vznikají nové capsule. Jádro je pak předává výstupní frontě, která se stará o správné doručení. K určení správného směru, kam má capsuli poslat, využívá informace ze směrovací tabulky.

Směrovací tabulka udržuje informace o známých uzlech. Ví, jaké uzly jsou sousední, a dokáže určit správnou bránu pro nesousední uzly. Informace ve směrovací tabulce byly v původní implementaci statické a načítají se z konfiguračního souboru při inicializaci uzlu. Později byl adaptován protokol NetAnts [3].

Výkonnostní monitor periodicky zjišťuje užívání zdrojů serveru a statistické informace o vykonávaných aplikacích a capsulích.

Dočasné úložiště aktivních programů slouží k uložení programů aplikací a capsulí. Je to úložiště, ze kterého se získává potřebný aktivní program při vykonávání aplikací a

capsulí. A je to také úložiště, kam protokol pro distribuci aktivních programů ukládá požadované aktivní programy.

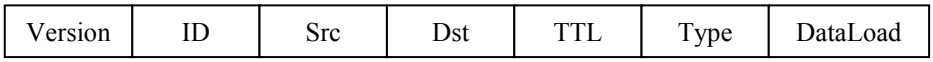
2.2.2 Identifikace aktivního uzlu

Každý uzel Grade32 je jednoznačně identifikován 128 bitovým číslem. Protože je však v Grade32 předpokládáno použití na operačních systémech Windows a k propojení uzlů se tuneluje protokol TCP/IP, je s každým 128 bitovým identifikátorem aktivního uzlu ještě spojena IP adresa, kde se tento uzel nachází a port, na kterém poslouchá. Aktivní aplikace a capsule IP adresu nevidí.

Vlastní identifikátory aktivních uzlů Grade32 díky tomu umožňují vytvoření overlayové sítě a v případě potřeby i simulaci různých topologií na jednom počítači.

2.2.3 Capsule – jednotka přenášených dat

V Grade32, se stejně jako v ANTS, používají pro přenos dat capsule. Formát capsule vychází z toho, že jsou všechny aktivní uzly v síti identifikované 128 bitovým číslem a stejně velkým identifikátorem i každý aktivní program. Úplní formát capsule, používané v prostředí Grade32, je zobrazen na následujícím obrázku Obrázek 3::



Obrázek 3: Capsule Grade32

Význam jednotlivých položek:

- Version – Verze formátu capsule
- ID – Jedinečný identifikátor capsule
- Src – Zdrojový uzel
- Dst – Cílový uzel
- TTL – Doba života capsule
- Type – Identifikátor vykonávaného programu capsule
- DataLoad – Přenášená data

2.2.4 Aktivní programy

Aktivní program capsule je v Grade32 představován DLL knihovnou. Při vytváření aktivního program lze například využít programovacího jazyka Delphi nebo C++. Aktivní server pak volá předem specifikované metody této knihovny.

Protože DLL knihovny představují strojové instrukce procesoru, mohou být procesorem přímo vykonávané a nepotřebují tedy pro svou činnost žádný interpret nebo virtuální stroj.

Ke každé DLL knihovně se vytváří pomocí algoritmu MD5 její 128 bitů dlouhý otisk, který se používá k identifikaci knihovny v rámci celé sítě.

2.2.5 Distribuce aktivních programů

Vždy, když chce uživatel přenášet sítí capsule, musí nejprve zajistit dostupnost aktivního programu příslušné capsule na uzlu, odkud chce přenos začít. Podmínka musí být splněna, aby mohly okolní uzly tento aktivní program získat, následně vykonávat a v případě potřeby distribuovat do dalších uzlů.

V Grade32, stejně jako v ANTS, existuje jednoduchý protokol distribuce aktivních programů. Pokaždé, když data capsule dorazí na nějaký uzel, pokouší se uzel najít potřebný aktivní program v lokálním úložišti. Pokud se tam potřebný aktivní program nalézá, nemusí se uzel starat o získání tohoto programu a může jej začít vykonávat.

V případě, že se ale aktivní program v úložišti nenachází, nastupuje komponenta aktivního uzlu nazývaná Code Distribution. Ta se pokouší získat potřebou DLL knihovnu od uzlu, ze kterého byla capsule přijata. Předpokládá totiž předchozí vykonávání aktivního programu právě na tomto uzlu a tedy i jeho dostupnost v lokálním úložišti. V případě, že tam aktivní program vykonáván nebyl, předpokládá se, že ho tam uživatel uložil, aby mohl být sítí používán.

2.2.6 Dočasné úložiště stavu

V Grade32 je úložiště dočasného stavu reprezentováno blokem Global States a chování je podobné soft-state. Capsule například při ukládání dat do tohoto úložiště nespecifikují, za jak dlouho platnost dat vyprší. Aplikace a capsule pouze předpokládají, že k tomu jednou dojde (tuto funkčnost nebylo nutné řešit s ohledem na původní cíl). Server se proto ani nestará o odstraňování starých dat. Zdálo by se tedy, že data zůstanou v Global States navždy. To ale není pravda, neboť například v případě restartování serveru jsou všechna data v Global States ztracena.

Informace jsou do Global States ukládány jako dvojice [klíč, hodnota]. Protože je klíč, určený pomocí GUID (náhodně generován), neočekává se, že dojde ke konfliktu identifikátorů uložených klíčů. Tímto způsobem nemůže jeden program ovlivňovat uložené stavové informace druhého programu, leda že by znal jeho GUID. Zároveň je tak umožněna spolupráce různých aplikací a protokolů, případně různých verzí jednoho protokolu.

2.3 PLANet

PLANet [16] je implementace aktivních sítí používající dvouúrovňovou architekturu. Jedná se o tzv. switchlety a uživatelské programy. Switchlety poskytují sadu služeb, které lze na serveru využívat. Uživatelské programy jsou popsány jazykem PLAN [13, 14, 15] a přenášejí se společně s daty. Programovací jazyk PLAN byl speciálně vyvinut pro zajištění bezpečnosti aktivních sítí.

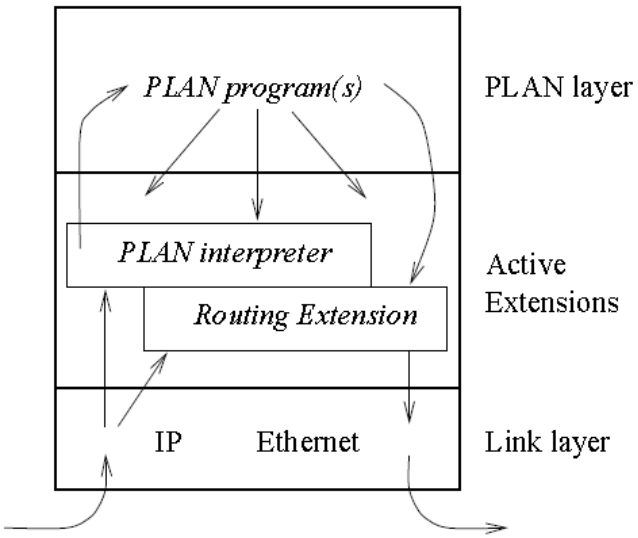
Dříve než je paket s programem zaslán do sítě, musí se určit, kolik prostředků může maximálně použít. Hodnota přidělených prostředků se nikdy nenavyšuje (naopak je snižována) a je tedy garantováno ukončení výpočtu.

2.3.1 Architektura uzlu

Architektura aktivního uzlu PLANet je zobrazena na obrázku Obrázek 4:. Jak je z obrázku patrné, skládá se ze tří hlavních vrstev. Z linkové vrstvy, která se zasluhuje o přijímání a odesílání paketů, z aktivní nadstavby a výkonného prostoru programů popsaných jazykem PLAN.

Pokaždé, když paket dorazí na PLANet uzel, zjišťuje se, zda se jedná o uzle uvedený v poli evalDes. Pokud uzel není cílovým uzlem, zjistí se z položky routFun, která směrovací funkce se má použít a tato funkce zajistí směrování příchozího paketu. Pokud paket dorazil na uzel, který má uveden v položce evalDes, zajistí PLANet uzel načtení a inicializaci programu z části paketu označované souhrnně jako chunk. Po inicializaci programu se začne interpretovat internetem jazyka PLAN. To se provádí na 3. úrovni architektury uzlu PLANet. Vykonávané programy pak mohou volat jiné programy napsané také v jazyce PLAN nebo mohou volat různá rozšíření serveru.

U uzlu PLANet mohou být použity dynamicky načítané aktivní programy, které rozšiřují jeho funkčnost. Tyto programy jsou napsané v jazyce OCaml.



Obrázek 4: PLANet node architecture [16]

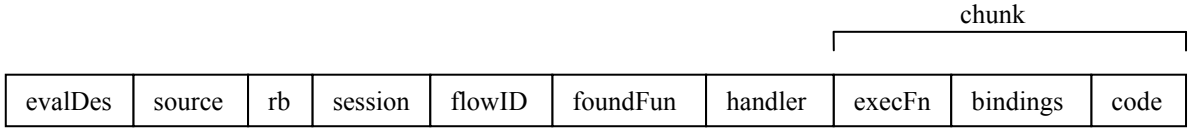
2.3.2 Identifikace aktivního uzlu

Každý aktivní uzel PLANet může vlastnit několik síťových adres, které jsou velké 48 bitů. Typicky se jedná o 32 bitů IP adresy ve verzi 4 a 16 bitů odpovídajících číslu portu. Jako „linkové vrstvy“ se pak používá UDP/IP pro vytvoření komunikačního tunelu mezi fyzicky oddělenými uzly.

Aktivní uzel může vlastnit několik síťových rozhranní, které nemusí korespondovat s fyzickými rozhraními aktivního uzlu. PLANet dovoluje vytvářet virtuální rozhraní, které se odlišují použitým portem.

2.3.3 PLAN paket

V PLANet se o přenášených datech nehovoří jako o capsulích, ale jako o PLAN paketech. Formát takovéhoto paketu je zobrazen na obrázku Obrázek 5:.



Obrázek 5: Formát paketu sítě PLANet

Význam jednotlivých položek PLAN paketu:

- evalDest – adresa cílového uzlu na kterém se má program paketu vyhodnotit
- source – adresa zdroje
- rb – přidělené množství globálních zdrojů

- session – identifikátor sezení
- flowID – identifikátor datového toku
- routFun – jméno směrovací funkce
- handler – jméno obslužné funkce v případě výjimky
- execFn – jméno funkce, která se má vykonat
- bindings – inicializační datové položky programu
- code – program v jazyce PLAN

První dvě pole obsahují adresu cílového a zdrojového uzlu PLANEet. Každá je velká 48 bitů a identifikuje jedno rozhraní uzlu PLANet. V PLANet je adresa definována jako kombinace 32 bitů IP adresy a 16 bitů přiděleného portu. Touto volbou je lehké použít UDP/IP jako „linkové vrstvy“ mezi PLANet uzly.

Pole rb (Resource Bound) můžeme připodobnit poli TTL v IPv4 nebo Hop Limit v IPv6. Jedná se tedy o maximální počet uzlů, jimiž může PLANet paket projít. Po jeho překročení bude ze sítě odstraněn.

Pole session poskytuje identifikaci pro koncové aplikace. V IP paketu jej můžeme připodobnit číslu protokolu vyšší vrstvy.

Pole flowID identifikuje datový tok. Pomáhá tak například směrovačům při zajištění Quality of Services.

Pole routFun obsahuje název směrovací funkce. Tato funkce je použita ke správnému směrování PLANet paketu sítí.

Pole handler obsahuje název funkce zdrojového uzlu, která bude použita v případě výskytu chyby při přenosu PLANet paketu sítí.

Program a jeho data jsou uloženy ve třech polích označovaných společně chunk (zkratka pro code hunk). V prvním poli je uložen název hlavní funkce sloužící jako vstupní bod programu. Následuje pole s uloženými inicializačními argumenty a za ním už je pole s vlastním programem. Když paket dosáhne uzlu uvedeného v poli evalDest, zavede se program z pole code, připojí se inicializační data a začne se vykonávat voláním funkce uvedené v poli execFn.

2.3.4 Aktivní programy

Aktivní kód je představován skriptovacím jazykem PLAN, speciálně vyvinutého pro potřeby aktivních sítí, a protože je používán v síti PLANet, nese ho tato síť ve svém názvu.

PLAN paket přenáší aktivní program v části nazývané chunk. Tento aktivní program není ale spouštěn na každém uzlu, kudy paket prochází, ale až na cílovém uzlu, jehož adresa je uvedena u PLAN paketu.

Při průchodu paketu sítí je nad tímto paketem prováděno směrování, popsané také jazykem PLAN. Směrovací program se ale nepřenáší uvnitř paketu. Přenáší se pouze název směrovací funkce.

Rozdělení vykonávání aktivního programu do směrovací funkce a funkce vykonávané na cílovém uzlu, je dáno představou autorů PLANet sítě, která tkví v tom, že není nutné manipulovat s daty paketu při průchodu sítí, ale je důležité mít možnost, zvolit si vhodnou směrovací funkci, která například zohlední zatížení některých částí sítě.

2.3.4.1 Jazyk PLAN

Jazyk PLAN je skriptovací jazyk používaný pro psaní aktivních programů v PLANet. Tento jazyk byl speciálně vyvinut pro potřeby aktivních sítí. Poskytuje silnou statickou kontrolu a typovou bezpečnost. Kromě standardních klíčových slov jako definování datových typů nebo pro potřeby větvení a cyklů, obsahuje další klíčová slova přímo svázaná s použitím v prostředí aktivního uzlu PLANet. Jedná se například o klíčová slova OnRemote a OnNeighbour. Jejich význam je takový, že dovolují spouštět určitý kód, který je předán parametrem na různých uzlech sítě. Pro zvýšení bezpečnosti vykonávaného kódu obsahuje jazyk systém výjimek a definuje některé základní výjimky jako například DivByZero, HostNotLocal nebo NoRouteEntry.

Jazyk PLAN je kompilován do kódu, který dovoluje vykonávání na různých počítačových architekturách, a stává se tak strojově nezávislý.

2.3.5 Distribuce aktivních programů

V aktivní síti PLANet existují dva typy aktivního programu. Jedná se o aktivní programy provádějící směrování PLANet paketů a aktivní programy vykonávané na cílovém uzlu. Protože je druhý typ aktivního programu přenášen v každém paketu, nemusí se síť starat o jeho distribuci, jako je tomu například v ANTS nebo Grade32.

Oproti tomu je první typ aktivního programu, který není přenášen v každém paketu a přenáší se pouze jeho název. Z tohoto důvodu musí být potřebný aktivní program dostupný na uzlu před jeho použitím nebo dojde k chybě. O jeho distribuci a instalaci se však musí postarat uživatel této sítě sám, neboť neexistuje vestavěný mechanizmus provádějící automatickou distribuci aktivních programů.

2.4 Shrnutí

Zde popsané implementace aktivních sítí tvoří významnou část výzkumu v této oblasti. Jedná se o dvě práce (ANTS, PLANet), které vznikly v době představení konceptu aktivních sítí. Základním důvodem, proč byly tyto práce vytvořeny, bylo otestování, že má cenu zabývat se myšlenkou aktivních sítí i nadále. Třetí práce (Grade32) není primárně určena přímo pro výzkum aktivních sítí, ale pro výzkum nové metody přerozdělování zátěže v distribuovaném heterogenním prostředí. Tato práce byla vytvořena z důvodu, že předchozí implementace nepodporovaly potřebnou funkcionalitu a jejich vývoj byl ukončen.

Jako další implementace aktivních sítí můžeme příkladem zmínit SwitchWare [17, 18], CANEs [19], či NetScript [20].

2.4.1 Aktivní programy

V každé implementaci se pro aktivní kód používá jiný jazyk. V ANTS je to jazyk Java, který je také použit při implementaci vlastního aktivního serveru. Tato volba sebou přináší jednoduchost realizace kódu aktivního programu. Aktivní program totiž musí být odvozen od abstraktní třídy, jejíž metody pak může aktivní server volat. Při tomto řešení není možné omezit sadu metod, kterou může aktivní program volat a nejde ani zajistit vlastní přepínání běhu jednotlivých programů. Aktivní program může pak například voláním metody System.exit() předčasně ukončit činnost celého serveru. Stejný princip je použit i v Grade32 s tím rozdílem, že se nepoužívá programovací jazyk Java, ale programy jsou překládány do DLL knihoven, například z programovacích jazyků Delphi a C++. Proti ANTS a Grade32 stojí síť PLANet, zavádějící vlastní jazyk zvaný PLAN. Jedná se o skriptovací jazyk navrhnutý pro potřeby aktivních sítí. Je ale těsně svázán se sítí PLANet a kvůli jeho vlastnostem jej není možné používat v případě vlastní implementace, která by se příliš lišila od PLANet.

Ve většině případů je nutné aktivní programy nějakým způsobem identifikovat. Je to nutné v případě, kdy nepoužívá tlačný princip a programy tak nejsou přenášeny společně s daty v každé capsuli (paketu). Identifikace aktivního programu se pak vkládá do přenášené capsule a server podle ní spouští pro procházející data správný program. V případě ANTS nebo Grade32 se používá pro označování aktivních programů jejich otisk, vypočtený hash algoritmem MD5, jehož výstupem je 128 bitové číslo. V PLANet se tento způsob nepoužívá. Pro označování procedur, vykonávaných nad přenášenými daty, se používá „lidsky čitelný“ textový identifikátor.

Oproti tomuto způsobu je ale použití identifikátoru vytvořeného na základě nějaké hashovaní funkce, díky níž je vytvořen otisk aktivního programu, z mnoha pohledů bezpečnější. V první řadě odstraňuje možné konflikty v pojmenování aktivních programů. Je totiž velice pravděpodobné, že mnoho programátorů pojmenuje podobný (tedy ne identický) aktivní program stejným názvem. Ve druhé řadě zvyšuje použití hashovací funkce bezpečnost. Pokud bychom totiž chtěli podvrhnout nějaký aktivní program, tak by bylo při malé velikosti kódu velice obtížné vytvořit aktivní program, který by měl stejný identifikátor (hash).

2.4.2 Tlačný, tažný a dopředný princip

Při distribuci programů, vykonávaných pro procházející data, jsou v představených implementacích použity tři principy. Tlačný, tažný a dopředný.

V prvním případě je program přenášen společně s daty. To má přínos v tom, že nemusíme zavádět algoritmus na získání potřebného programu, pokud ho server nemá ve svém úložišti. Nevýhoda je ale v neustálé potřebě přenášet stejnou část dat (která nemusí být malá) v každém paketu. Zbytečně se tak zatěžují přenosové linky něčím, co ale může být klidně uložené v dočasném úložišti na serveru. Tento princip se používá v implementaci PLANet. Je zajímavé, že popsaný přenášený program se nevykonává na každém uzlu, ale až na uzlu cílovém.

Tažný princip spočívá v tom, že se společně s daty nepřenáší žádný program, ale pouze jeho identifikátor. Ten se pak použije pro nalezení správného programu v úložišti a jeho následné vykonání. Protože ale server nemusí mít příslušný program ve svém (dočasném) úložišti, musí se postarat o jeho získání. Většinou se ho pak snaží získat z předchozího uzlu, nebo z uzlu, ze kterého data pocházejí. Tím se pak jakoby program táhne za daty. Je pravda, že je tažný princip složitější než předchozí, protože se musí zajistit získání správného programu. Nevýhoda je také v tom, že při prvním použití nového programu chvíli trvá, než se zavede do všech zainteresovaných uzlů. Základní výhoda spočívá ve zmenšení objemu přenášených dat. Používá se v ANTS a Grade32.

Třetí princip je dopředný. Spočívá v tom, že nasadíme potřebný program do všech uzlů před tím, než začneme přenášet data. Používá v implementaci PLENet, v níž se musí dopředu na všechny uzly nasadit program používaný pro zpracování přenášených dat. U dat se pak uvádí pouze název potřebného programu. V PLANet je dopředný princip určen především pro nasazení směrovacích protokolů. Možnost dopředné distribuce programů je ale možná i u ANTS a Grade32. Není totiž velký problém napsat pro tyto sítě vlastní aktivní program,

starající se o rozdistribuování našeho nového programu před tím, než jej začneme využívat. V případě Grade32 ho může administrátor uzlu i sám před-instalovat.

3 Aktivní uzel SAN

V této kapitole bude popsána architektura serveru aktivních sítí SAN. Budou popsány jeho jednotlivé komponenty a řešení některých problémů. Nebudeme se ale zabývat implementačními detaily.

3.1 Architektura

Zmíněné implementace aktivních uzlů ukazují některé problémy, nutné v návrhu serveru aktivních sítí řešit, a ukazují také některé funkční bloky, ze kterých by se měla architektura aktivního uzlu sestávat. Proto budou v kapitole nejprve stanoveny jednotlivé vrstvy aktivního uzlu SAN a následně navrhnuty funkční bloky jednotlivých vrstev.

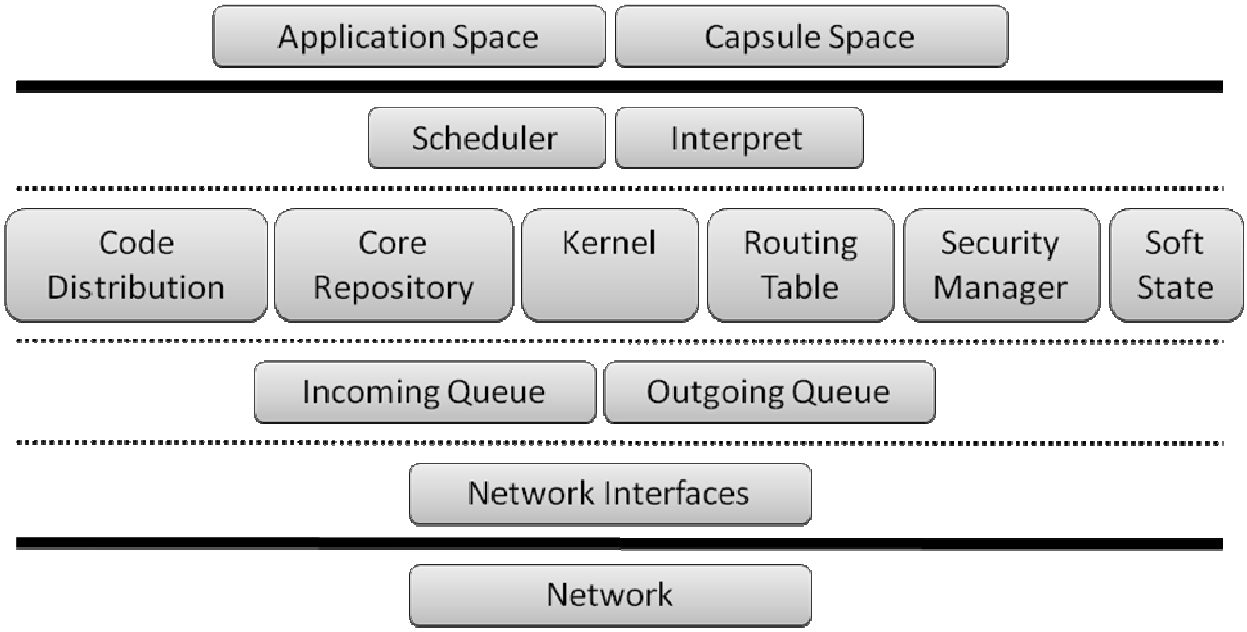
Celá architektura je rozdělena do tří základních vrstev, oddělujících od sebe síť, vlastní aktivní server a uživatelský prostor aplikací a capsulí. Protože by ale bylo toto členění příliš hrubé, je aktivní server dále hierarchicky rozdělen do čtyř podvrstev.

První vrstva serveru je tvořena sítovými rozhraními a stará se o navazování spojení mezi aktivními uzly. Pokud je spojení navázáno, zajišťuje tato vrstva doručování capsulí po daném spojení.

Druhou vrstvou serveru, umístěnou nad síťovou, je vrstva vstupní a výstupní fronty. Zajišťuje příjem příchozích capsulí a správné odesílání odchozích capsulí.

Třetí vrstvu tvoří samotné jádro aktivního serveru. Jádro zajišťuje distribuci a uchování aktivních programů. Stará se o správné směrování odchozích capsulí a zajišťuje dočasné úložiště dat. Řeší problémy spojené s bezpečností aktivního uzlu a přenášených dat.

Čtvrtá vrstva jádra zajišťuje vykonávání aktivních programů aplikací a capsulí.



Obrázek 6: Vrstvy a komponenty aktivního uzlu

Poznámka

Přesto že vrstva běhu aplikací a capsulí a síťová vrstva nejsou součástí serveru, jsou na obrázku zachyceny z důvodu získání globální představy o návaznosti na tyto vrstvy.

3.1.1 Popis jednotlivých komponent

Komponenta Network Interfaces zajišťuje přímou komunikaci s ostatními aktivními uzly sítě. Zároveň tvoří jednu vrstvu architektury aktivního uzlu. Základním úkolem je odstínění síťové technologie, nebo technologií nedostupných na daném uzlu. Stará se o navázání spojení s okolními aktivními uzly a přenos dat mezi nimi. Protože vyšší vrstvy aktivního uzlu pracují s daty v jednotném formátu capsule, zajišťuje vrstva vhodnou konverzi a přenos těchto capsulí po dostupné technologii. Vyšší vrstvy se pak nemusí starat o formáty přenášených dat a způsobu jejich přenosu. Mohou jednotně pracovat s daty v jednotném formátu. Je tedy možné, aby se SAN jednou v budoucnu eventuálně adaptoval např. na IIP.

Vstupní fronta přebírá od síťových rozhranní přicházející data v podobě capsulí a řadí je do fronty na obsloužení aktivním uzlem nebo je může začít zahazovat v případě přetížení aktivního uzlu.

Do výstupní fronty jsou předávány capsule k odeslání do jiného aktivního uzlu. Fronta předá data k odeslání nižší vrstvě, která je pomocí příslušného síťového rozhraní odešle jinému aktivního serveru.

Každá capsule procházející aktivní sítí je spojena s aktivním programem vykonávaným na navštívených uzlech. Protože bylo ale rozhodnuto nepřenášet aktivní program uvnitř capsule z důvodu ušetření přenášených dat, musí být příslušný program dostupný na jednotlivých aktivních uzlech, aby bylo možné capsule zpracovávat. Proto se musí uzly aktivních sítí starat o základní distribuci aktivního programu zajišťované komponentou Code Distribution.

Uchování aktivního programu v rámci aktivního uzlu zajišťuje komponenta Code Repository. Zmenšuje nároky na opětovné přenášení potřebného aktivního programu. Zároveň ale obsahuje možnost zapomínání. Dlouho nepoužívané aplikace zahazuje a uvolňuje tak místo pro nové.

Dočasné úložiště dat slouží nejen pro potřeby aplikací a capsulí uchovávat a sdílet data, ale i samotnému aktivnímu uzlu k uchování některých, například statistických informací. Uložené informace pak mohou capsule nebo aplikace využívat při své činnosti.

Komponenta Routing Table zajišťuje základní směrování mezi aktivními uzly. Směrovací informace pak používá aktivní uzel pro směrování capsulí v případě, že si capsule neurčí směrování sama. Směrovací komponenta také dovoluje změnu směrovacích informací prováděných capsulemi zajišťujícími směrování v této síti.

Komponenta Security Manager zajišťuje bezpečnostní strategii serveru aktivních sítí. Zajišťuje ověřování a zabezpečení různých částí systému a přenášených dat. Jednou z jejích úloh je zajištění ověření uživatelů při přístupu k aktivnímu uzlu na základě předložení uživatelského jména a hesla. Dále ověřuje bezpečnost a důvěryhodnost příchozího aktivního programu, a to i za jeho běhu. Z pohledu dat zajišťuje jejich nepopiratelnost a šifrování. Stará se o správu a distribuci šifrovacích informací a certifikátů.

Plánovač zajišťuje plánování vykonávání aktivních programů interpretem. Interpret pak zajišťuje vlastní interpretaci aktivního programu capsulí a aplikací. Odstiňuje prostor aplikací a capsulí od prostoru aktivního uzlu. Pokud bude SAN v budoucnu podporovat několik typů zápisu aktivního kódu, je to jen otázka přítomnosti potřebných komponent – interpretů. Podpora víceprocesorových systémů je řešena spuštěním instancí interpretů na jednotlivých procesorech a jeden plánovač úkoluje všechny interprety.

Komponenta Core zajišťuje integraci ostatních komponent, nacházejících se ve stejné vrstvě architektury a zajišťuje napojení na nižší vrstvu systému. Zajišťuje také správné spuštění aktivního uzlu a jeho počáteční inicializaci.

3.2 Výběr programovacího jazyka

Při výběru programovacího jazyka pro implementaci Smart Active Node, je přihlíženo k vlastnostem, jako je typová bezpečnost, bezproblémová přenositelnost a objektově orientovaný přístup. Rychlost vykonávání kódu byla obětována na jejich úkor a s přihlédnutím k podpoře víceprocesorových systémů a jejich zvyšující se dostupnosti a výkonnosti.

Z těchto důvodů byl vybrán programovací jazyk Java [21]. Splňuje všechny uvedené vlastnosti. Navíc je to velmi moderní jazyk, který je díky svým vlastnostem značně rozšířený a používaný. Je pravda, že je tento programovací jazyk pomalejší než například jazyk C, nebo některá jeho varianta. Na druhou stranu zaručuje typovou bezpečnost nebo možnost statické kontroly zdrojových kódů. Navíc se sám stará o správu paměti, což také přispívá k bezpečnosti celého serveru.

Pokud bychom přeci jen požadovali rychlejší běh aktivního uzlu, použili bychom jazyk C++, protože syntaxe Javy, v porovnání s ostatními jazyky, má blízko k C++ a lze tedy předpokládat, že by se to kladně projevilo při portaci.

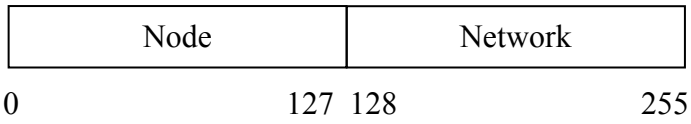
3.3 Identifikace aktivního uzlu

V dnešní době jsou všechny uzly Internetu identifikovány svou IP adresou ať už ve verzi 4, nebo 6. V ANTS se tyto identifikátory přebírají a používají pro identifikaci uzlů. Na rozdíl tomu se v Grade32 zavádějí vlastní identifikátory velikosti 128 bitů a protože je Grade32 používáno na dnešních IP sítích, je tento identifikátor spojen s IP adresou a portem. Je tak vlastně vytvořena overlayová síť a možnost jednoduché simulace různých topologií počítačových sítí na jednom počítači.

V SANu, podobně jako v Grade32, se bude používat vlastních identifikátorů a tím je také možné vytvořit overlayovou síť. Použití overlayových sítí je velmi užitečné pro experimentování a tento postup vede k odstínění vlastní přenosové technologie. Navíc to může být dočasná technika před vlastním nasazením nové technologie, jako je uvedeno v [10]. Tato technika nám umožní například jednoduše simulovat strukturu rozsáhlé počítačové sítě na jednom počítači, nebo vytvořit virtuální topologii nad síťovou topologií sítě. Toto je praktika, která se v dnešním světě hojně používá.

Každý uzel SANu, stejně tak jako každý počítač, může mít více než jedno síťové rozhranní. Každé rozhraní bude mít vlastní identifikátor a celý server pak může být identifikován jakýmkoliv identifikátorem síťového rozhraní, jehož je vlastníkem. Identifikátor

se bude skládat ze dvou částí. Z id uzlu a id sítě. Id uzlu bude identifikovat uzel v rámci celé sítě a síťová část bude identifikovat síť, v níž se uzel momentálně nachází. Obě části budou velké 128 bitů.



Obrázek 7: Identifikace uzlu

Poznámka

Rozdělení identifikátoru aktivního uzlu na dvě sub-id, kde ale samotné id uzlu jednoznačně identifikuje aktivní uzel v rámci celé sítě, má za úkol umožnit snadnou mobilitu aktivního uzlu v rámci sítě. Síťová část by pak měla pomoci se směrováním jednotlivých požadavků. V této oblasti se předpokládá další výzkum.

Identifikace uzlu SANu je dostatečně velká na to, aby mohla obsahovat i adresy IPv4 a IPv6. ID sítě pak bude mít předdefinovanou hodnotu. Tato technika umožňuje převzetí kontroly komunikace u aplikací založených na IP a např. zavedení optimalizací jejich protokolů [22].

3.4 Capsule

V tradičních IP sítích jsou data přenášena v podobě paketů, obsahujících mimo jiné zdrojovou a cílovou adresu, informaci pro protokoly vyšších vrstev a uživatelská data. V aktivních sítích jsou data přenášená v podobě capsulí. Struktura capsule se částečně podobá struktuře IP paketu. Obsahuje tedy mimo jiné zdrojovou a cílovou adresu, čas Time To Live, a uživatelská data. Pro potřeby aktivních sítí se pak v capsuli musí navíc přenášet buďto kód programu vykonávaného na navštívených uzlech, nebo postačí identifikátor aktivního programu uloženého na serveru.

Pokud použijeme první možnost, tj. přenášet program přímo s daty, budeme tak vlastně přenášet určitou část dat pořád dokola a tím zbytečně zatěžovat přenosové kanály. Proto se zdá výhodnější přenášet pouze identifikátor aktivního programu. Pokud se pak příslušný program nachází na navštíveném uzlu, není problém jej použít. Problém může nastat, pokud se vyžadovaný program na uzlu nenachází. Pak se musí tento aktivní program získat z některého jiného uzlu (v úvahu přichází uzel, ze kterého data přišla), což ale znamená zdržení procházejících dat. Pokud se chceme tomuto (inicializačnímu) zdržení vyhnout, nic

nám nebrání napsat „chytrou“ aplikaci spekulativně předem distribuující program capsulí do uzlů, kudy budou procházet naše data. Toto je předmětem dalšího výzkumu.

V implementaci sítě PLANet se v jedné capsuli přenášejí dva druhy programů. Prvním programem je program zajišťující směrování PLANet paketu a druhý program je aplikace, která se zavede a spustí na cílovém uzlu. Program pro směrování PLANet paketu je v paketu uložen v podobě identifikátoru směrovacího programu a program musí být dostupný na celé trase, kudy paket sítí prochází. Pokaždé když paket dorazí na některý uzel, který není cílový (není uveden v evalDes části paketu) je nad paketem vykonán právě ten směrovací program, jehož identifikátor je v paketu uveden. Pokud PLANet paket dorazí na cílový uzel, tj. uzel uvedený v evalDes části paketu, je zaveden program přenášený v datové části zvané chunk.

U implementace sítě ANTS už se nepřenáší žádné programy uvnitř capsule. Přenáší se pouze identifikátory programů uložených do pole Type. Oproti PLANet nespecifikuje žádný jiný program a program uvedený v poli Type se spouští na každém aktivním uzlu kudy capsule prochází. Stejně se chová i implementace Grade32.

3.4.1 Caspule v SANu

Formát capsule, který se používá v SANu je podobný formátu v ANTS a Grade32. Kombinuje některé jejich myšlenky a vytváří tak nový formát. Struktura capsule je zobrazena na obrázku Obrázek 8:.

ID	Dest	Src	IDC	IDA	HopLimit	Data

Obrázek 8: Struktura capsule SANu

Význam jednotlivých polí:

- ID – udává jednoznačný identifikátor capsule přidělený v době jejího vzniku. ID capsule se po dobu jejího života nemění.
- Dest – (Destination address) udává cílovou adresu uzlu, do kterého capsule míří. Capsule ji může během svého vykonávání měnit a tak měnit i směr, kudy bude sítí putovat.
- Src – (Source address) je zdrojová adresa uzlu, na kterém capsule vznikla. Capsule ji nemůže během jejího vykonávání měnit. Jedním z důvodů je fakt, že je to adresa uzlu, ze kterého se bude jiný uzel snažit získat její aktivní program v případě, že jej nezíská od jiného uzlu.

- IDA – udává identifikátor aplikace, která capsuli vytvořila. Byl aplikaci přiřazen jádrem aktivního uzlu při jejím zavedení. Jeho plánované využití spadá do oblasti výzkumu ohledně tzv. survivability applications / fault tolerant výpočtů, kdy je třeba mít v síti několik replik celého programu aktivní aplikace.
- IDC – identifikuje program capsule spouštěného na serveru. V ANTS je stejná položka označována Type, která udává typ capsule.
- Hop Limit – udává maximální počet uzlů, kudy může capsule projít. Jedná se o vestavěný obranný mechanizmus, používaný jak v IP sítích, tak v ANTS i Grade32. Ale například v ANTS se tato položka nazývá TTL, což historicky pochází z IPv4. V IPv6 je tato položka přejmenována na Hop Limit, což dnes lépe vystihuje způsob jejího použití, a proto je v SANu označena stejně.
- Data – obsahuje užitečná (uživatelská) data. Mohou být zpracována samotným programem capsule nebo je capsule může předat aplikaci.

Poznámka

Při použití protokolu */IP k přenosu capsulí se nemusíme zabývat jinými uzly sítě než je SAN server, protože celou capsuli uvidí jenom jako data a omezí se na IP hlavičku. V capsuli tedy nemusíme přenášet identifikátor uzlu, na kterém byl program capsule naposledy spouštěn jako je tomu například n ANTS. Když totiž capsule přijde na server, je jasné odkud přišla a tedy i adresa uzlu na kterém byla naposledy vykonávána.

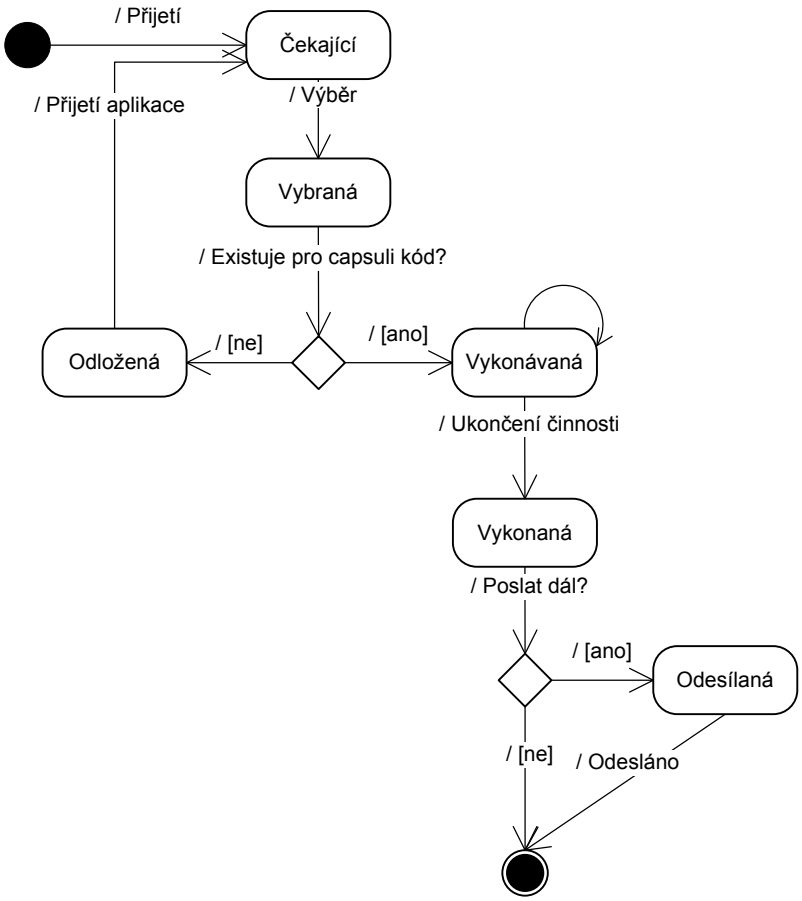
3.4.2 Stavy capsule SANu

Žádná capsule nemůže z ničeho nic vzniknout a začít putovat sítí. Za vznikem capsule vždy stojí nějaká aplikace mající v úmyslu počítat v síti, nebo jiná capsule. Zmíněná aplikace (nebo capsule) tzv. injektuje capsuli do sítě. Capsule pak začne sítí procházet a provádět požadovaný výpočet. Injekce spočívá ve vytvoření capsule, nastavení cílové adresy, aktivního programu (nebo jeho identifikace) vykonávaného na navštívených uzlech a dat. Následně capsule je předána aktivnímu uzlu, který zaručí její doručení správnému aktivnímu uzlu.

Po přijetí capsule aktivním uzlem je capsule zařazena do vstupní fronty a čeká na vykonání. Před vlastním vykonáním příslušného aktivního programu se musí zjistit, zda je požadovaný aktivní program na serveru dostupný. V případě, že tomu tak není, je vykonání capsule odloženo a musí se získat potřebný aktivní program z jiného aktivního uzlu, přičemž

v úvahu přichází aktivní uzel, odkud byla capsule přijata. Musel to totiž být aktivní uzel naposledy vykonávající program capsule, nebo uzel, na kterém capsule vznikla.

Pokud je aktivní program pro vykonávání capsule dostupný, začne se vykonávat. Na konec své práce je buď capsule zaslána dalšímu aktivnímu uzlu ke zpracování, nebo její život končí.



Obrázek 9: Stavy capsule při průchodu uzlem

3.5 Programový kód aplikací a capsulí

V implementaci aktivního serveru ANTS je každá capsule potomkem abstraktní třídy Capsule, což jí dává základní vlastnosti a nutí k implementaci některých metod. Protože je celý server ANTS napsán v programovacím jazyce Java, je tento postup přímočarý a z pohledu návrhu se zdá být čistý.

Ve zmíněném návrhu je ale také každá capsule samostatně vykonávané vlákno, což může být problém. Vláken běžících capsulí může být v jednu chvíli víc a o jejich přepínání se stará virtuální stroj Javy, což může při větším počtu capsulí znamenat problém. My však také potřebujeme nějakým způsobem měřit, kolik systémových zdrojů které vlákno spotřebovalo a

v závislosti na tom, jakou prioritu má, přidělovat více či méně systémových zdrojů. To však nedokážeme určit, pokud použijeme k interpretaci aktivního kódu pouze JVM, nebo nám známé virtuální stroje třetích stran. Dokážeme sice zvětšovat nebo snižovat prioritu vlákna, ale už nedokážeme zjistit, kolik času procesoru vlákno využilo a už vůbec nedokážeme určit, kolik paměti si vlákno alokovalo. Pokud bychom pak chtěli některá vlákna pozastavovat nebo dokonce ukončovat, dostaneme se do úzkých.

Vlastní interpretace aktivního kódu serverem využitá například v implementaci PLANet, se proto zdá být výhodná. Při vlastní interpretaci aktivního kódu můžeme zjišťovat, kolik systémových prostředků se aktivním programem používá a kolik času procesoru spotřebovává. Pokud pak interpretaci spojíme s plánovačem, můžeme výpočet aktivního kódu přerušovat a řídit prioritu jeho vykonávání podle toho, jak uznáme za vhodné. Navíc můžeme jednoduchým způsobem omezit přístup aktivního kódu ke komponentám serveru připojením bezpečnostního monitoru k našemu interpretu.

3.5.1 Jazyk interpretovaný v SANu

Zmíněný jazyk PLAN bohužel nemůže být použit pro potřeby SANu. Je totiž těsně vázán na architekturu uzlu PLANet, a pro potřeby SANu nepoužitelný. Proto musí být buďto vytvořen jazyk vlastní nebo upraven některý z existujících jazyků.

Hlavním požadavkem na tento jazyk je možnost silné statické kontroly a typové bezpečnosti. Kvůli horší možnosti ladění distribuované aplikace je právě statická kontrola a typová bezpečnost velice výhodná a usnadní vývoj nových služeb.

Dále je výhodné použití jazyka kompilovaného do bytecodu. Tato kompilace zajistí strojovou nezávislost. Bylo by také výhodné, aby byla syntaxe tohoto jazyka podobná syntaxi používané dnešními moderními programovacími jazyky jako je například Java nebo C#. To totiž do značné míry usnadní vývoj nových služeb, neboť se vývojáři nebudou muset učit novou syntaxi, ale budou jej moci intuitivně používat.

Zmíněné vlastnosti má programovací jazyk Java. Tento jazyk je navíc použit pro vlastní aktivní uzel SAN, což může také přispět k zjednodušení implementace vlastního interpretu.

Poznámka

Vlastní interpretace přináší i jiné výhody než jen například měření, kolik systémových prostředků se používá nebo řízení přístupu k těmto prostředků. Už nyní se předpokládá

rozšíření jazyka Java, aby podporoval některé další programové konstrukce. Jako příklad můžeme uvést rendez-vous z Ady.

Poznámka 2

Vývoj vlastního interpretu programovacího jazyka Java je velmi složitý a není obsahem této práce. Tato záležitost si zasluhuje zvláštní pozornost a už dnes je jisté, že se jí bude zabývat někdo jiný v některé z navazujících prací. Pro potřeby vykonávání aktivního kódu se proto použije existující interpret BeanShell, který je popsán v [23]. Tento interpret nemá veškeré potřebné vlastnosti, ale pro nynější vývoj je dostačující. Nevýhodou je interpretace přímo zdrojových souborů, což ji značně zpomaluje. Dále neumí interpretovat generické objekty a neexistuje možnost interpretaci pozastavit, uložit její stav, začít interpretovat jiný kód a poté se vrátit k interpretaci předchozího kódu od uloženého místa.

Kromě BeanShellu bude brzy v prostředí SANu možné použít i nativní interpret. Tento interpret se začal vyvíjet odděleně a v současné době je začleňován do prostředí aktivního uzlu SAN. Kvůli oddělenému vývoji vzniklo několik technických problémů při volání jádra aktivního uzlu z uživatelského prostoru aplikací a capsulí. Doufáme, že se problémy brzy vyřeší a bude možné plně přejít na nativní interpret.

3.5.2 Identifikace aktivních programů aplikací a capsulí

V implementaci ANTS a Grade32 je každý aktivní program identifikován jeho otiskem vytvořeným na základě algoritmu MD5. Základem této myšlenky je odstranit problém s pojmenováním a přispět k bezpečnosti. Pokud se totiž instaluje aktivní program na serveru, server si vypočítá jeho otisk, a pod tímto otiskem si ho uloží. Uživatel tedy nemůže uložit na serveru aktivní program s jiným otiskem, než je skutečný otisk ukládaného programu. Implementace PLANet něco takového nezavádí a používá „lidsky čitelné“ identifikátory aktivního programu.

Problém algoritmu MD5, popsaný např. v [24], je v nalezení možnosti útoku proti tomuto algoritmu. Byla nalezena metoda, jak vytvořit zprávu, která bude mít otisk podle našeho přání. Z tohoto není vhodné používat tento algoritmus pro potřeby aktivních sítí, neboť by to přinášelo bezpečnostní rizika.

Pro potřeby identifikace aktivních programů SANu bude použit algoritmus Whirlpool-T [25]. Tento algoritmus (a některé další) byl vybrán v projektu NESSIE pro kryptografické potřeby na území Evropy. Zajišťuje tedy potřebnou bezpečnost a netrpí bezpečnostním

problémem jako například MD5. Díky jeho jednoduchosti je jej možné dokonce jednoduše implementovat i pomocí hardwaru.

3.5.3 Uchování aktivních programů aplikací a capsulí

V každé implementaci aktivních sítí je zapotřebí uchovávat na uzlech aktivní program neboť musí být dostupný minimálně po dobu přenosu dat, s nimiž je spojen. V současnosti je podoba uchování aktivních programů do značné míry závislá na programovacím jazyku, který byl použit při vytváření aktivního uzlu.

V ANTS se aktivní programy capsulí přenášejí a uchovávají jako jednotlivé Javové třídy. Každá takováto třída je oddděděná od abstraktní třídy Capsule definující metody, které nad ní bude server volat a díky tomu bude vykonáván potřebný algoritmus capsule. Uvedený přístup je vzhledem k použitému programovacímu jazyku velmi přímočarý, ale pokud bychom chtěli zároveň přenášet s programem capsule nějaká metadata, budeme je muset komplikovaně přidávat mezi uživatelská data.

Oproti tomu jsou v Grade32 programy capsulí kompilovány do podoby DLL knihoven. Což je zase postup přímočarý při použití programovacího jazyka Delphi, C++, nebo některých dalších. Pokud bychom pak potřebovali, můžeme do těchto knihoven jednoduše uložit potřebné dodatečné informace za použití Embeded Resource.

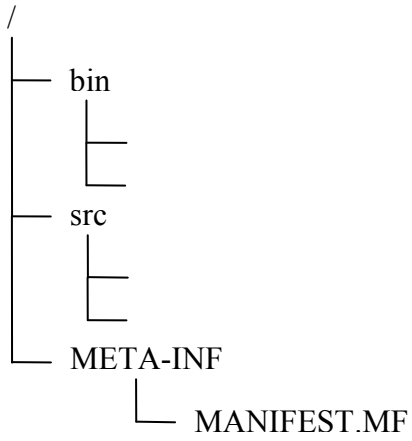
V PLANet sítích se pro propis aktivních programů používá speciální programovací jazyk, který byl vyvinut pro potřeby aktivních sítí. Tento jazyk je překládán do podoby bytecodu a interpretován na serverech. Směrovací programy jsou na serveru uchovány v podobě souborů obsahujících interpretovaný bytecode. Oproti tomu aktivní aplikace, spouštěná až na cílovém uzlu, je přímo přenášená v PLANet paketu společně s daty nebo případnými metadaty.

3.5.4 Uchování aktivních programů aplikací a capsulí v SANu

Programy napsané v jazyce Java se skládají z mnoha tříd, a každá třída je přeložena do samostatného souboru. Aby nebyly Javové programy na disku počítače uloženy v podobě velkého množství souborů, zavádí Java JAR archivy. Do těchto archivů se pak ukládají veškeré třídy programu, ale třeba i obrázky, zdrojové kódy a jiné potřebné informace. Ve své podstatě můžeme do JAR archivů uložit, co budeme chtít. Navíc nám to může ušetřit nemalé množství místa na disku neboť JAR archiv je v podstatě ZIP archiv.

V našem případě se nám tento přístup může velice hodit. Předpokládáme totiž, že se aplikace a capsule budou skládat z více tříd. Navíc budeme určitě potřebovat další doplňující

informace, které lze do takovéhoto balíčku jednoduše uložit. Vlastní struktura JARu nám však zcela nevyhovuje, a proto ji upravíme naším potřebám. Nový balíček budeme nazývat SPK (SAN Package). Základní struktura tohoto nového balíčku je naznačena na následujícím obrázku Obrázek 10:.



Obrázek 10: Struktura SPK balíčku

V adresáři src se nacházejí zdrojové kódy aplikace a capsule . Tento adresář není povinný. V adresáři bin jsou přeložené zdrojové soubory. V adresáři META-INF se nacházejí další potřebné soubory, zejména pak soubor MANIFEST.MF popisující vlastní aplikaci a capsuli. V souboru manifestu (MANIFEST.MF) jsou uloženy základní informace o aplikaci, capsuli ale třeba i zdrojových kódech. Můžeme do něho například také uložit veřejný klíč pro potřeby šifrování.

Soubor manifestu je strukturován tak, že se na každé řádce nachází klíč a hodnota k němu připojená. Základní informace uložené v manifestu shrnuje následující tabulka Tabulka 1:.

Klíč	Význam
Manifest-Version	Verze souboru manifestu.
Application-Name	Lidsky čitelné jméno aplikace.
Capsule-Name	Lidsky čitelné jméno capsule.
Application-Main-Class	Hlavní třída aplikace. Třída, která splňuje rozhraní IAplication.
Capsule-Main-Class	Hlavní třída capsule. Třída, která splňuje rozhraní ICapsule.
Source-Encoding	Formát kódování zdrojových souborů (implicitně se předpokládá UTF-8).

Tabulka 1: Položky manifestu a jejich význam

Pro balíček není problém udělat jeho otisk algoritmem Whirlpool-T. V případě potřeby uložení dalších potřebných informací můžeme tyto informace, v podobě souborů nebo záznamů manifestu, do balíčku uložit. Je však nutné při každé změně obsahu balíčku přepočítat jeho otisk.

3.6 Distribuce aktivních programů

V implementacích ANTS, Grade32 i PLANet existuje určitým způsobem automatická distribuce potřebných aktivních programů. V každé implementaci je ale tato distribuce chápána jinak.

V případě nedostupnosti potřebného aktivního programu capsule v implementaci ANTS se server snaží získat potřebný aktivní program z uzlu, na kterém byla capsule před tím zpracovávána. Adresa takového uzlu je uložena přímo v capsuli v poli Previous Address a server použije, jako výchozí adresu uzlu odkud se snaží chybějící program získat. Pro tento účel se použijí dvě servisní capsule. První nalezne potřebný aktivní program a druhá jej dopraví zpět do uzlu.

V implementaci Grade32 je automatická distribuce aktivních programů capsulí zajištěna podobně jako v ANTS. Rozdíl je ale v tom, že v capsuli není přenášen identifikátor předchozího uzlu, na kterém byla capsule zpracovávána. Grade32 si ji dokáže zjistit sám během přenosu capsule po síti. Uzel, od kterého tedy capsule přišla, je uzlem, ze kterého se Grade32 snaží získat potřebný aktivní program. Druhý rozdíl spočívá ve vytvoření speciální komponenty uzlu zajišťující distribuci programů capsulí. Nepoužívají se tedy speciální capsule, ale uzel obsahuje vestavěný mechanizmus, který distribuci zařídí.

Velkým rozdílem od předchozích je implementace PLANet. Ta zajišťuje automatickou distribuci přenosem potřebného aktivního programu přímo uvnitř paketu. Program je sítí přenášen v paketu a vykoná se až na cílovém uzlu. Současně je ale také v paketu přenášen identifikátor směrovacího programu. Tento program není ale automaticky distribuován a uživatel se musí dopředu postarat o to, aby byl na všech potřebných uzlech dostupný, než začne sítí posílat data.

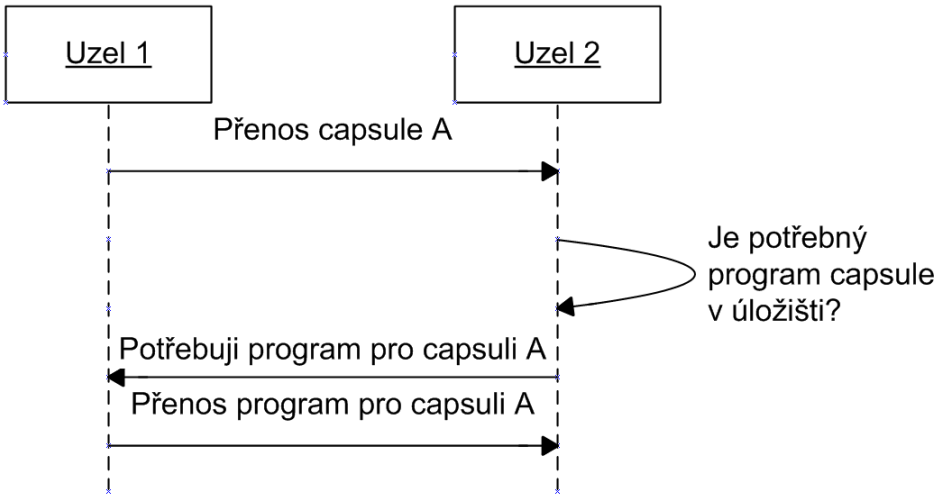
3.6.1 Distribuce aktivních programů v SANu

Každý aktivní uzel musí mít možnost získat potřebný aktivní program aplikace nebo capsule z některého sousedního uzlu. Pokud by tohoto schopen nebyl, musel by to zařídit uživatel sám dříve, než začne šířit své capsule sítí. Tímto by se však myšlenka aktivních sítí

degradovala, a proto by měl mít server aktivních sítí vestavěný protokol pro distribuci aktivních programů.

Když tedy přijde capsule, pro kterou na uzlu není potřebný program, musí ho získat z jiného uzlu. Jednou otázkou je, ze kterého uzlu ho má získat. A odpověď na ní je jednoduchá. Z uzlu odkud ji přijal. Na něm byla capsule určitě také vykonávána a potřebný program tam tedy musel být, nebo to byl uzel ve kterém capsule vznikla a uživatel musel zajistit dostupnost potřebného aktivního programu na tomto uzlu.

Když už víme, ze kterého uzlu aktivní program získat zbývá otázka: „Jak?“ Na tuto otázku už není odpověď přímočará. Můžeme totiž v serveru vytvořit komponentu, která to bude zajišťovat, anebo se pokusíme nějakým způsobem využít něco, co už máme. A když jsme v aktivních sítích, proč nevyužít capsule? Právě ony se hodí na získání aktivních programů z jiného uzlu a doručení do potřebného uzlu. Jediné, co je třeba zajistit, je vhodné API, přes které budou capsule schopné aktivní programy získat a uložit na jiném uzlu. Program těchto capsulí by pak měl být dostupný na všech uzlech sítě.



Obrázek 11: Automatická distribuce programů

Výhodou použití capsulí je také možnost modifikovat způsob získání aktivního programu bez nutnosti měnit způsob práce všech aktivních serverů v síti. Může nás totiž napadnout, co se stane, pokud se požadovaný aktivní program na předchozím serveru nebude nacházet? Aktivní aplikace jako celek se navíc nemusí skládat pouze z jednoho programového balíku. Pokud bude algoritmus capsule pro distribuci aktivních programů chytřejší, může v případě nenalezení požadovaného aktivního programu na předchozím serveru zkusit štěstí přímo u uzlu, na kterém capsule vznikla. Nebo to může zkoušet u uzlů ležících na cestě ke zdrojovému uzlu.

Další zajímavá myšlenka by byla v dopředné distribuci aktivních programů. Zatím jsme totiž vždy uvažovali, že se aktivní program získává až když capsule dorazí do uzlu a on se tam nenachází. Pokud ale některý uzel už potřeboval získat potřebný aktivní program, je pravděpodobné, že bude vyžadován i dalším uzlem. Proč tedy tento aktivní program rovnou neposlat do dalšího uzlu, přes který bude capsule procházet. Výběr uzlů, na které se má aktivní program s předstihem distribuovat, stejně tak jako použitá technika je předmětem dalšího výzkumu.

3.7 Dočasné úložiště stavu

K možnosti ukládat stavové informace na aktivním serveru se každá ze zmíněných implementací staví jinak. V ANTS je možno stavové informace ukládat do lokálního úložiště založeného na soft-state. Tato je podporováno také v Geade32, kde ale úložiště není zcela založené na soft-state, protože se nespecifikuje doba, na kterou jsou data do úložiště ukládána. Na druhou stranu není možné uložená data ze serveru získat po jeho restartování nebo výpadku. V implementaci PLANet se žádná komponenta serveru nestará o ukládání stavových informací.

V SANu použijeme také, jako v ANTS, úložiště založené na soft-state, kam budou capsule nebo aplikace v případě potřeby ukládat svá data v podobě [klíč, hodnota]. Aby nemohlo dojít ke konfliktům klíčů různých capsulí (aplikací), bude navíc každý údaj spojen s identifikátorem této capsule (aplikace). Tímto způsobem od sebe navíc oddělíme uložená data a různé capsule (aplikace) nebudou moci ovlivňovat soukromá data jiných capsulí (aplikací). Uvědomujeme si však také potřebu sdílet data jedné capsule (aplikace) s jinou. Proto zavedeme podobný mechanizmus jako je v Grade32 a dovolíme ukládat data v podobě [klíč, hodnota], která však nebudou spojena s capsulí ani s aplikací. Pokud pak uživatel přímo zadá požadovaný klíč, budou související data uložena pod tímto klíčem nezávisle na tom, zda je pod stejným klíčem uložil už někdo jiný. Pokud uživatel klíč nezadá, bude klíč automaticky náhodně vygenerován.

Stejně jako v Grade32 nedovolíme možnost výčtu klíčů a hodnot dočasného úložiště. V opačném případě by to totiž vedlo ke snížení bezpečnosti. Tímto způsobem zajistíme, aby capsule nebo aplikace nemohly přistupovat k privátním datům jiných capsule (aplikací). Pokud ale budou potřebovat, mohou být některá data sdílená a využívaná skupinou capsulí a aplikací, jako například vlastní tabulka směrování. Některé capsule pak mohou zapisovat data do této tabulky a jiné capsule mohou být podle těchto dat směrovány. Nakonec, například pro potřeby multicastu, může být využita vlastnost ukládání sdílených dat pod automaticky

vygenerovaný klíč. Tato vlastnost je výhodná, pokud se doručovací skupina dynamicky mění a uzly se do ní mohou přihlašovat nebo se z ní mohou odhlašovat.

3.8 Plánování aplikací a capsulí

Než se podíváme na to, jakým způsobem budeme plánovat vykonávání aplikací nebo capsulí, shrneme si nejdříve jejich předpokládané chování a délku života.

Protože jsou capsule programy vykonávané na každém uzlu, přes který procházejí, předpokládáme, že se bude jednat o malé programy s velmi krátkou dobou života. Předpokládáme, že se činnost těchto programů omezí pouze na směrování dat, předání dat aplikacím nebo jednoduché modifikaci přenášených dat. Pokud by tomu tak nebylo, začalo by docházet k nárůstu latence. Bohužel se ale nemůžeme na tento předpoklad stoprocentně spolehnout. Určitě se najde nemalé množství uživatelů sítě, kteří budou chtít využít výpočetní výkon aktivních uzlů pro své vlastní účely. Tito uživatelé pak například mohou do sítě vyslat capsule, které jim budou v této síti např. podporovat vědeckotechnické výpočty nebo distribuovat video. Další skupinou uživatelů mohou být uživatelé snažící se síť napadnout, nebo zneužít. Z tohoto důvodu bude Security Monitor hlídat jejich spotřebu systémových zdrojů.

Druhým typem vykonávaný aktivních programů na aktivních uzlech jsou aplikace. U aplikací, na rozdíl od capsulí, primárně nepředpokládáme velmi krátkou dobu života. Naopak předpokládáme, že některé mohou žít po celou dobu provozu aktivního uzlu. Těmito aplikacemi mohou být například aplikace starající se o směrování nebo aplikace shromažďující statistická data v síti, případně aplikace poskytující služby jiným aplikacím.

3.8.1 Plánovací strategie

Podívejme se nyní, jaké plánovací strategie můžeme použít při plánování vykonávání aplikací a capsulí. Protože aplikace a capsule spadají podle vlastností uvedených v předchozím odstavci do oblasti tzv. interaktivních nebo real-time procesů, uvedeme pouze plánovací strategie vhodné pro tyto procesy. Nebudeme však uvádět ani plánovací strategie real-time systémů neboť použitím programovacího jazyka Java a jeho vykonávání v prostředí běžných operačních systémů bychom real-time zpracování nebyli schopni docílit.

3.8.1.1 Round robin

Jedná se o jednoduchou plánovací strategii s předbíháním. Připravené procesy čekají ve frontě typu FIFO. Plánovač postupně vybírá procesy z této fronty a přiděluje jim časová

kvanta. Po uplynutí časového kvanta (pokud ještě proces neskončil) je zařazen na konec fronty. Velikost časového kvanta je důležitý parametr a v případě potřeby je jím možné řídit prioritu procesů.

3.8.1.2 Priority scheduling

V této plánovací strategii má každý proces přidělenou prioritu. Připravené procesy se pak podle své priority řadí do prioritních tříd a CPU je přidělována procesům z nejvyšší neprázdné prioritní třídy. V této prioritní třídě jsou procesy plánovány metodou Round Robin. Nevýhodou plánovací strategie je hladovění procesů s nízkou prioritou, což se může například řešit zvýšením priority příliš dlouho čekajícího procesu.

Varianty prioritního plánování:

- Bez předbíhání
- S předbíháním
    - Statická priorita
    - Dynamická priorita
    - Fixní časové kvantum pro všechny prioritní třídy
    - Různě velká časová kvanta pro různé prioritní třídy

3.8.1.3 Shortest process next

Jedná se o modifikaci plánování Short Job First pro interaktivní systémy. U procesů se odhaduje potřebná doba výpočtu na základě minulého chování. Plánovač pak vybírá proces s nejmenší odhadnutou dobou výpočtu. Problémem strategie je možnost hladovění dlouhých procesů, respektive procesů, u kterých je stále odhadována dlouhá doba výpočtu.

3.8.1.4 Fair-Share scheduling

V tomto prioritním plánování znamená vyšší numerická hodnota nižší prioritu. Každý proces k má nastavenu pevnou základní prioritu B_k a v každém časovém intervalu i používal proces k CPU po dobu $CPU_k(i)$. Proces má v časovém intervalu i prioritu $P_k(i)$ a na začátku každého časového intervalu je nastavena hodnota $CPU_k(i)$ na polovinu předchozí hodnoty. Hodnota $P_k(i)$ je pak dána jako součet základní priority a nové hodnoty $P_k(i)$. Plánovač vybere proces s nejnižší hodnotou $P_k(i)$.

Tato plánovací strategie, jak už tomu její název napovídá, zajišťuje poctivější sdílení procesorového času než například prioritní plánování. Odstraňuje navíc problém s hladověním procesů přímo svým návrhem a nemusí ho řešit nějakým dodatečným mechanizmem.

3.8.1.5 Plánování použitím epoch

Plánovací strategie používaná např. v operačním systému Linux [26]. Rozděluje čas procesoru do tzv. epoch. V každé epoše má každý proces přiděleno časové kvantum vypočítané na začátku každé epochy. Toto časové kvantum nemusí proces vyčerpat najednou. Pokud se například uspí při čekání na I/O operaci a nevyčerpal své časové kvantum, bude znovu naplánován ve stejné epoše. Epocha končí, když všechny běhu schopné procesy vyčerpaly svá kvanta. Následně jsou vypočtena nová časová kvanta všech procesů (nejen běhu schopných) a začíná nová epocha. Délka časového kvanta závisí na prioritě procesu.

3.8.2 Výběr plánovací strategie

V implementaci ANTS není použit žádný vlastní plánovací mechanizmus a spoléhá se vestavěné plánování JMV. Grade32 využívá plánovacího mechanismu hostitelského operačního systému Windows NT [27] založeného na prioritním plánování.

Protože předpokládáme existenci velmi krátkých, ale i velmi dlouhých procesů, není vhodné použít strategii Shortest process next. Pravděpodobně by totiž docházelo k vyhladovění dlouhých procesů. U strategie Round-Robin k vyhladovění procesů nedochází. Primárně se v ní ale nepoužívají priority, a pokud bychom je chtěli používat, vedlo by to na ne úplně poctivé přidělování času procesoru.

Ze zbývajících třech uvedených algoritmů použijeme v naší implementaci Fair-Share scheduling. Tato plánovací strategie nemá problémy s hladověním procesů a předpokládáme, že bude mít z pohledu rychlosti odezvy lepší vlastnosti než plánovací strategie používající epochy. Tento předpoklad vychází z toho, že bude velké množství velmi krátkých procesů (capsulí). Použitím epoch by pak ale pravděpodobně byla nemalá skupina připravených procesů, které by musely čekat na novou epochu. Předpokládáme tedy, že použitím Fair-Share scheduling nebude vznikat nárazové zpracování skupiny procesů, ale průběžné zpracovávání procesů (například procházejících capsulí).

4 Z pohledu administrátora Smart Active Node

4.1 Instalace, konfigurace a spuštění serveru

Celá instalace serveru SAN není příliš složitá a provádí se ve třech krocích. Předpokladem pro úspěšný provoz je mít nainstalovánu Javu minimálně ve verzi 1.5 (5.0). Pro samotný běh serveru stačí JRE.

V prvním kroku je nutné přeložit všechny zdrojové soubory serveru. Je nutné mít nainstalován i JDK, ale nemusí se provádět, pokud jsou zdrojové soubory již přeložené.

V druhém kroku je nutné nakopírovat do úložiště serveru základní aplikace. Proto je nutné ve stejném adresáři, jako je adresář s přeloženými soubory, vytvořit adresář **codeRepository** a do něho nakopírovat všechny aplikace, které jsou se serverem distribuovány.

Ve třetím kroku se vytvoří soubor nastavení serveru. Tento soubor musí mít název **settings.xml,** a jak už jeho přípona napovídá, je jeho obsah ve formátu XML. V tomto souboru je uloženo celé nastavení aktivního serveru SAN a jako příklad jeho obsahu může být následující:

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="apolo">
  <interface name="eth0">
    <identity>
      <node>
        00000000-0000-0000-0000-ABCD00000001
      </node>
      <network>
        00000000-0000-0000-0000-987600000001
      </network>
    </identity>
    <broadcast ip="230.0.0.1" port="5000" />
    <method value="0" priority="1" />
    <method value="1" priority="0" />
  </interface>
  <logging output="console" level="debug" />
  <console type="graphic" />
</node>
```

Tento konfigurační soubor je postačující pro plné nakonfigurování serveru s jedním síťovým rozhraním.

Značkou na nejvyšší úrovni je `node`. Smí být v dokumentu použita právě jednou a popisuje nastavení aktivního uzlu. Její atribut nazvaný `name` popisuje jméno aktivního uzlu.

Značka `interface` popisuje jedno konkrétní síťové rozhraní serveru. Těchto rozhraní může být libovolný počet, ale doporučuje se, aby bylo minimálně jedno. Atributem je `name` vyjadřující název síťového rozhraní. Může (ale nemusí) korespondovat s názvem síťového rozhraní, které je dostupné na daném zařízení (počítači).

Značka `identity` vyjadřuje adresu síťového rozhraní, která tak tvoří adresu serveru. Tato adresa je rozdělena na id uzlu (`node`) a síťovou část (`network`), které jsou popsány v jednotlivých značkách. Každá část je tvořena 128 bitovým číslem zapsaným v hexadecimálním tvaru, jehož formát je patrný z uvedeného příkladu.

Značka `broadcast` popisuje, na jaké adrese a portu se jednotlivé servery budou kontaktovat, aby mezi sebou následně vytvořili propojení. Protože Java ve verzi 1.5 neumí používat broadcast vysílání, musí se použít multicastová adresa.

Značka `method` udává způsob komunikace na daném rozhraní. Hodnota 0 atributu `value` znamená použití TCP komunikaci a hodnota 1 UDP komunikaci. Atribut `priority` udává, s jakou prioritou se bude komunikovat daným způsobem. Větší číslo znamená větší prioritu. Každý server může mít nastavenu dostupnost obou metod, jako v tomto příkladě, nebo jen jednu z nich. Důsledkem toho je, že některé servery budou mezi sebou komunikovat použitím TCP a některé UDP.

Značka `logging` specifikuje způsob logování. Jako výstup může být buďto soubor (`file`) nebo konzole (`console`) a je specifikován v atributu `output`. Úroveň logovacích informací lze nastavit v atributu `level` a povolené hodnoty jsou: `debug`, `information`, `warning`, `error`.

Mód, v jakém se otevře konzole pro ovládání serveru, je definován atributem `type` uvnitř značky `console`. Pro grafické konzolové okno lze použít hodnotu `graphis` a pro textový režim `text`.

Po správné instalaci a konfiguraci serveru můžeme přejít k jeho spuštění. To lze jednoduše provést zadáním příkazu (do příkazové řádky) v kontextu adresáře serveru:

```
java san.core.Kernel
```

4.2 Aplikace LOGIN

Při spuštění serveru SAN je nejprve zavedena aplikace init, která provede počáteční inicializaci, vytvoří okno konzole a zavede aplikaci login. Aplikace login slouží k přihlášení uživatele k serveru. To se provádí zadáním uživatelského jména a hesla. Po úspěšném přihlášení předá aplikace login řízení aplikaci shell.

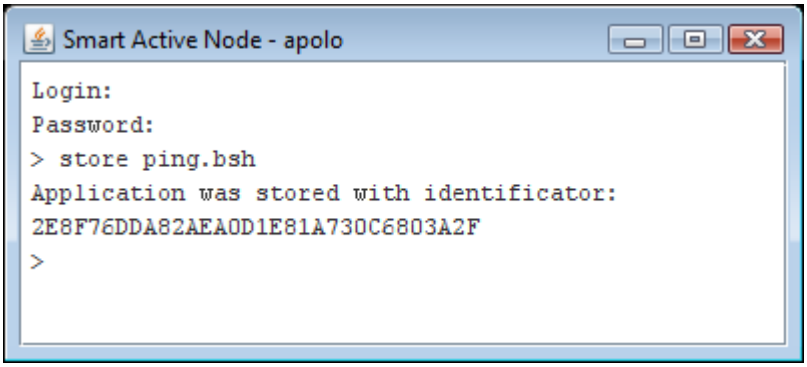
Poznámka

Aplikace login používá ke své činnosti Security Monitor. Tento monitor však není v současné době plně implementován. Proto lze při přihlášení zadat jakékoliv údaje a uživatel bude přihlášen.

4.3 Aplikace SHELL

Aplikace shell slouží k základnímu ovládání aktivního uzlu a v současné implementaci podporuje tři základní příkazy.

Před použitím aktivního programu v prostředí aktivních sítí musíme tento aktivní progran uložit na jednom ze serverů této sítě. K tomu slouží příkaz **store**. Má jediný parametr a tím je cesta k novému aktivnímu programu. Po vykonání příkazu store je do konzole pro kontrolu vypsaný identifikátor nově uloženého aktivního programu. Od této doby můžeme náš nově zavedený aktivní program začít používat.



Obrázek 12: Základní okno konzole a příkaz store

Pro spuštění aplikace, která je uložená na serveru, slouží příkaz **run**. Tento příkaz má minimálně jeden parametr, kterým je název aplikace. Název může být buďto 128 bitový identifikátor aplikace zapsaný v hexadecimálním tvaru, nebo lidsky čitelný název aplikace uvedený v souboru manifestu aplikace. Za identifikátorem nebo názvem aplikace pak můžou následovat další parametry, které budou aplikaci předány při jejím spuštění.

Posledním ze základních příkazů je **exit**. Příkaz exit zajistí správné ukončení činnosti aktivního serveru a je volán bez parametrů.

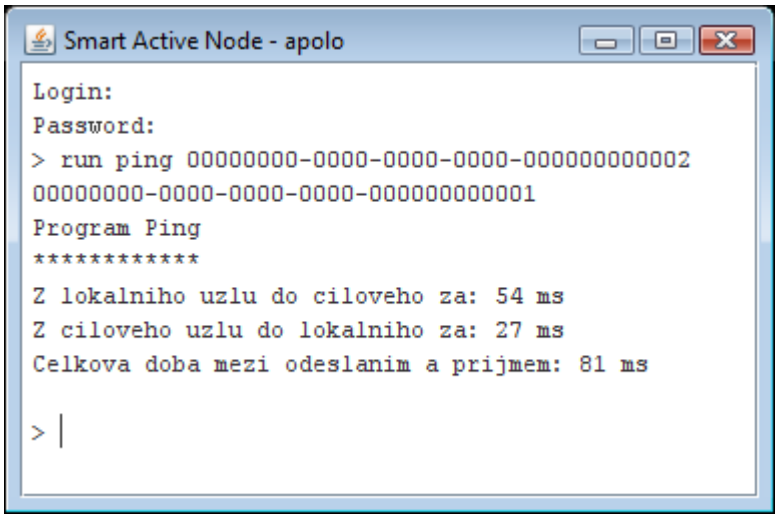
4.4 Aplikace PING

Aplikace ping slouží k otestování dosažitelnosti jiného uzlu. Aplikace ping vyšle capsuli s časem lokálního uzlu do cílového. Na cílovém uzlu capsule k tomuto času přidá čas vzdáleného uzlu a vrátí se zpět. Na zdrojovém uzlu předá data aplikaci, která vypíše výsledku.

Aplikace ping se spouští z příkazové řádky se dvěma parametry. První parametr je identifikátor cílového uzlu a druhý je identifikátor sítě, ve které se uzel nachází. Konkrétní volání z příkazové řádky (shell) může být:

```
run ping 00000000-0000-0000-0000-ABCD00000002
                  00000000-0000-0000-0000-987600000001
```

Základní použití aplikace ping dokumentuje následující obrázek Obrázek 13::



Obrázek 13: Použití aplikace ping

5 Z pohledu vývojáře pro Smart Acive Node

Pro vývoj aplikací použitelných v prostředí Smart Active Network je možné použít oblíbené vývojové prostředí (Eclipse, NetBeans, …), ve kterém lze psát Javové programy. V tomto prostředí si pouze připojíme knihovnu san.jar. Díky které můžeme používat jednotlivá rozhraní serveru.

Každá aplikace musí obsahovat veřejnou třídu implementující rozhraní `san.application.IApplication`. Toto rozhraní obsahuje veřejnou metodu `main` sloužící jako vstupní bod do aplikace a server jí volá při spuštění aplikace. Metoda `main` má dva vstupní parametr. Prvním je pole textových řetězců uchovávající parametry, se kterými byla aplikace spuštěna. Druhým parametrem je rozhraní `san.core.IApplicationAPI` tvořící vstupní bránu k serveru SAN. U této hlavní třídy naší aplikace není vhodné používat konstruktorů pro počáteční inicializaci dat. Server případné konstruktory ignoruje a nebude je nikdy volat.

U psaní programů capsulí je situace podobná jako u aplikací, jen je nutné používat jiná rozhraní. Každá capsule musí obsahovat veřejnou třídu, která implementuje rozhraní `san.application.ICapsule`. Toto rozhraní obsahuje veřejnou metodu `main`, která je vstupním bodem do capsule a server jí volá při spuštění této capsule. Metoda `main` má pouze jeden vstupní parametr. Je jím rozhraní `san.core.ICapsuleAPI`, které tvoří vstupní bránu k serveru SAN a datům samotné capsule. Tedy datům, jejich příchod na server zapříčinil vyvolání programu capsule. Ani u této hlavní třídy naší capsule není vhodné používat konstruktorů pro počáteční inicializaci dat. Server případné konstruktory, stejně jako u aplikací, ignoruje a nebude je nikdy volat.

Poté, co napíšeme a přeložíme naši novou aplikaci, musíme vytvořit SPK balíček, který je možné zavést do serveru SAN. Struktura tohoto balíčku je popsána v kapitole 3.5.4. Zároveň je nutné do tohoto balíčku uložit soubor manifestu, který je popsán ve stejné kapitole. Připomeňme také, že SPK balíček je stejně jako JAR balíček obyčejný ZIP archiv. Z tohoto důvodu může být vytvářen přímo vývojovým prostředím. Je však nutné změnit příponu na `spk`. Použitím jiné přípony by se server snažil pro aplikaci v něm uloženou nalézt jiný interpret, což by skončilo chybou, neboť zatím není podpora žádného dalšího interpretu.

Když máme vytvořen nový balíček, můžeme jej zavést do prostředí aktivní sítě. Například to jednoduše provedeme v příkazovém okně serveru. Použijeme příkaz `store`, a jako jeho parametr dáme cestu k vytvořenému balíčku. Pokud je balíček správně zaveden, je do konzole vypsán jeho otisk. V opačném případě je vypsáno chybové hlášení.

Po úspěšném zavedení nové aplikace do serveru můžeme začít používat tuto aplikaci v síti. Její vyvolání může být buďto použitím jejího otisku, nebo můžeme použít „lidsky čitelný“ název této aplikace použitelný v případě, že byl uveden v manifestu. Je důležité vědět, že voláním názvu aplikace namísto použití jejího otisku může dojít k vyvolání jiné aplikace. Server vždy vykoná poslední uloženou aplikaci se stejným názvem. Proto je vhodné používat toto volání spíše při vývoji nových aplikací, anebo při volání aplikací, u nichž je jejich název uznáván v globálním měřítku.

5.1 Popis rozhraní pro přístup k serveru

Protože záleží na tom, v jakém kontextu k serveru přistupujeme, existují proto dvě základní rozhraní. Pokud k serveru přistupujeme v kontextu aplikace, budeme volat metody z rozhraní `san.core.IApplicationAPI`. V případě přístupu z capsule rozhraní `san.core.ICapsuleAPI`. Metody těchto dvou rozhraní včetně jejich popisu jsou uvedené v následujících tabulkách.

5.1.1 Popis rozhraní san.core.IApplicationAPI

```
void injectCapsule(CodePkgIdent, NetIdentifier, TransmitData) throws
                                                DestinationNotAvailableException
```

Vytvoří novou capsuli, které nastaví podle prvního parametru identifikátor programu capsule, který se má na navštívených uzlech spouštět. Dále nastaví podle druhého parametru cílový uzel této capsule a nakonec data. Pokud není cílový uzel znám nebo není dostupný, vyhazuje metoda výjimku.

```
void addOnSendDataListener(ReceiveDataListener listener)
```

Tato metoda zaregistruje na serveru posluchače pro příchozí data. Posluchač bude volán v případě zaslání dat aplikaci, která jej zaregistrovala.

```
boolean removeOnSendDataListener(ReceiveDataListener listener)
```

Odregistruje posluchače příchozích dat. Metodu není nutné volat před koncem života aplikace. Server po ukončení aplikace posluchače automaticky zruší.

```
Guid runApplication(CodePkgIdent ident, String[] args) throws
                                                ExecuteException
```

Spustí aplikaci na základě jejího identifikátoru a předá jí parametry. V případě, že se aplikace na serveru nenachází, nebo dojde ke komplikacím při spouštění, vyhodí metoda výjimku.

`Guid runApplication(String name, String[] args) throws ExecuteException`

Spustí aplikaci na základě jejího jména a předá jí parametry. V případě, že se aplikace na serveru nenachází, nebo dojde ke komplikacím při spouštění, vyhodí metoda výjimku.

`void waitForApplication(Guid applId) throws InterruptedException`

Voláním této metody je volající pozastaven a čeká, dokud neskončí aplikace, jejíž identifikátor je uveden v parametru metody. V případě předčasného vzbuzení je vyvolána výjimka.

`void waitForApplication(Collection<Guid> applId) throws InterruptedException`

Voláním této metody je volající pozastaven a čeká, dokud neskončí všechny aplikace, jejichž identifikátory jsou uvedeny v kolekci v parametru metody. V případě předčasného vzbuzení je vyvolána výjimka.

`InputStream getStdIn()`

Vrátí standardní vstup. Například z konzole nebo souboru.

`PrintWriter getStdOut()`

Vrátí standardní výstup. Například do konzole nebo souboru.

`Guid getId()`

Vrátí identifikátor aplikace, který jí byl přidělen při jejím spuštění. Tento identifikátor není závislý na identifikátoru balíčku aplikace a pro každé spuštěné se generuje nový.

`CodePkgIdent getExecutedCodeIdent()`

Vrátí identifikátor balíčku, jehož aktivní program je vykonáván.

`ISoftState getSoftState()`

Vrátí rozhraní, díky kterému je možné přistupovat k dočasnému úložišti hodnot na serveru. Úložiště se chová jako soft-state, a proto není dostupnost uložených hodnot garantována. Zároveň je při ukládání hodnot musí specifikovat doba jejich života.

`IServerControl getServerControl()`

Vrátí rozhraní, díky kterému je možné přistupovat k dalším částem serveru jako například ke směrovací tabulce, vytvářet další konzole nebo ukončit činnost serveru.

5.1.2 Popis rozhraní san.core.ICapsuleAPI

`NetIdentifier getSource()`

Vrátí identifikátor serveru, ve kterém capsule vznikla.

`NetIdentifier getDestination()`

Vrátí identifikátor serveru, na který je capsule směrována.

`void setDestination(NetIdentifier dest)`

Nastaví nový server, kam má být capsule směrována.

`Guid getSourceApplication()`

Vrátí identifikátor aplikace, která capsuli vytvořila.

`TransmitData getDataLoad()`

Vrátí data, která jsou uvnitř capsule přenášena.

`boolean isThisServer(NetIdentifier ident)`

Vrátí hodnotu true, pokud na serveru existuje síťové rozhraní, jehož identifikátor je parametrem této metody. V opačném případě vrací hodnotu false.

`void sendData(Guid toProcess, TransmitData data)`

Pošle data aplikaci, jejíž identifikátor je uveden v parametru metody.

`void sendData(CodePkgIdent processCode, TransmitData data)`

Pošle data kterékoliv aplikaci, která je vykonávána a jejíž identifikátor typu aktivního programu je uveden v parametru metody.

`setDataLoad(TransmitData data)`

Nastaví nová data, která se budu v capsuli přenášet.

`ISoftState getSoftState()`

Vrátí rozhraní, díky kterému je možné přistupovat k dočasnému úložišti hodnot na serveru. Toto úložiště se chová jako soft-state, a proto není dostupnost uložených hodnot garantována. Zároveň je při ukládání hodnot musí specifikovat doba jejich života.

`finished()`

Touto metodou capsule oznámí, že její činnost skončila a nechce být dále zpracovávána. Neznamená to však, že by následný kód nebyl vykonán. Pouze po skončení vykonávání bude capsule zahozena a nebude přeposlána na další uzel.

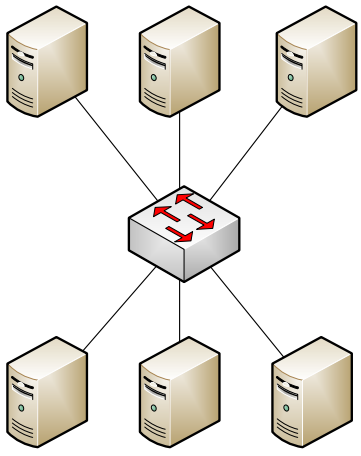
Pokud není metoda nikdy v programu capsule volána, neznamená to její nekonečné směrování a vykonávání. Díky vestavěnému mechanismu doby života bude capsule po čase ze sítě odstraněna.

6 Ověření funkčnosti

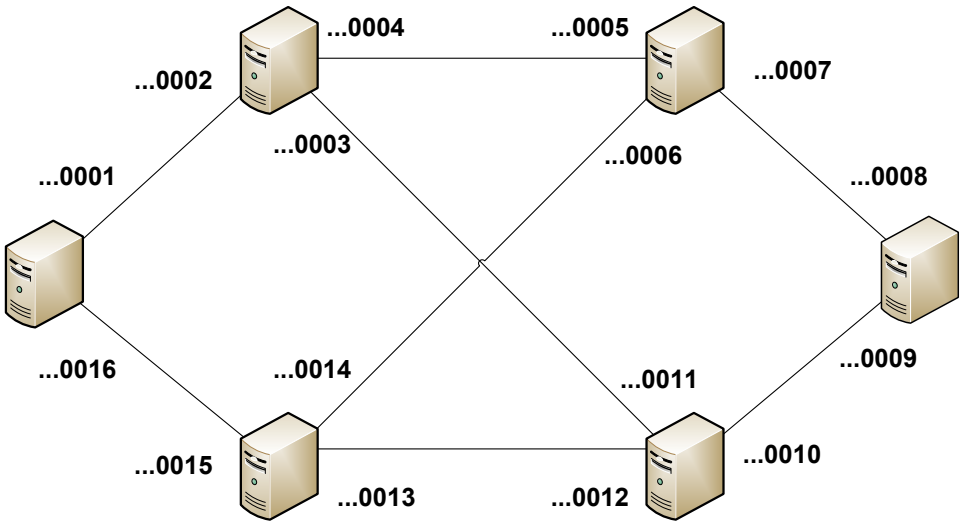
Ověření funkčnosti Smart Active Node bylo provedeno aplikací ping a sítí tvořenou šesti aktivními servery SAN. Síť byla nejdříve modelována na jednom počítači a následně přenesena do prostředí šesti počítačů propojených počítačovou sítí typu ethernet. Centrálním propojovacím prvkem byl přepínač. Zároveň byla také testována na různých operačních systémech provozovaných na počítačích uvedení sítě. Texty zdrojových souborů aplikace ping jsou uvedeny příloze B.

6.1 Testovací síť

Fyzická počítačová síť, použitá pro testování uzlů SAN (kromě testování na jednom počítači), je znázorněna na obrázku Obrázek 14:. Virtuální síť, tvořená uzly SAN, je pak postavena nad fyzickou sítí a zobrazena na obrázku Obrázek 15:. Tvoří jí šest uzlů SANu navzájem propojených podle uvedeného obrázku a každý uzel SAN je nainstalován na jednom počítači. Každý uzel SAN vlastní několik síťových rozhraní a jejich adresy jsou na obrázku vždy uvedeny u příslušného uzlu. Z důvodu přehlednosti nejsou uvedeny celé adresy, ale vždy jen jejich koncová část, která je pro každé rozhraní různá. Celou adresu uzlu můžeme získat doplněním nul na začátek. Platí také, že jsou všechny uzly umístěné ve stejné síti a z důvodu přehlednosti není číslo sítě uvedeno.



Obrázek 14: Fyzické síťové propojení počítačů



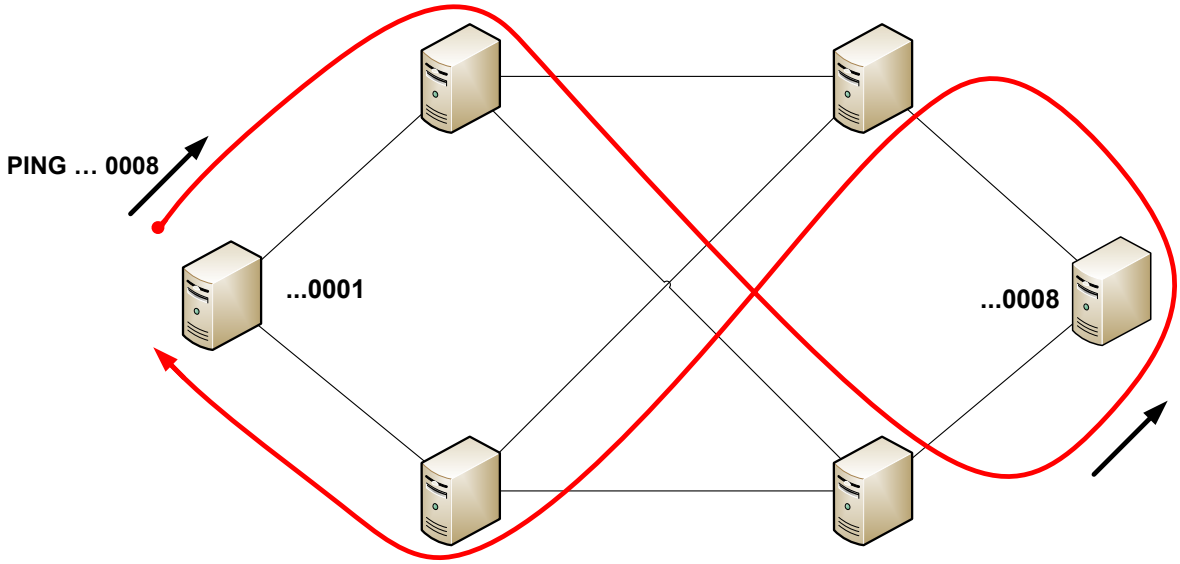
Obrázek 15: Virtuální sít uzlů SAN

Činnost aplikace ping v prostředí SANu spočívá ve zjištění dosažitelnosti některého z aktivních uzlů sítě. Pro zjištění dostupnosti požadovaného uzlu používá vlastní capsuli.

Poté, co je aplikace spuštěna, vytvoří capsuli kterou následně injektuje do sítě. Capsule prochází sítí a zjišťuje, zda se nenachází na cílovém uzlu. Ve chvíli, kdy dorazí na cílový uzel, zjistí lokální čas tohoto uzlu a zamění svou zdrojovou a cílovou adresu. Tím otočí svůj směr a začne se vracet do zdrojového uzlu. Až dojde na zdrojový uzel, předá data aplikaci, která ji vytvořila a ukončí svou činnost. Aplikace zpracuje získané informace a vypíše výsledky o dostupnosti cílového uzlu.

Na obrázku Obrázek 16: je zachycena cesta, kudy se capsule aplikace ping při testování ubírala. Cesta není náhodná, jak by se z obrázku mohlo zdát, a ani si jí neurčuje samotná capsule. Je dána statickým směrováním, které bylo úmyslně takto zvoleno. Aplikace ping byla spuštěna na uzlu s adresou končící 0001 a jako cílový uzel byl zvolen uzel, který vlastní síťové rozhraní s adresou končící na 0008. Vytvořená capsule procházela sítí po vyznačené cestě ve směru šipek.

Při bližším prozkoumání obrázků 15 a 16 zjistíme, že capsule na cílový uzel s označením …0008 nedorazila přes rozhraní …0008, ale přes rozhraní …0009. Podobně pak při jejím návratu na uzel …0001 byla přijata přes rozhraní …0016. Jedná se o test, zda jsou uzly SAN a aplikace schopny akceptovat toto chování.



Obrázek 16: Cesta capsule aplikace ping

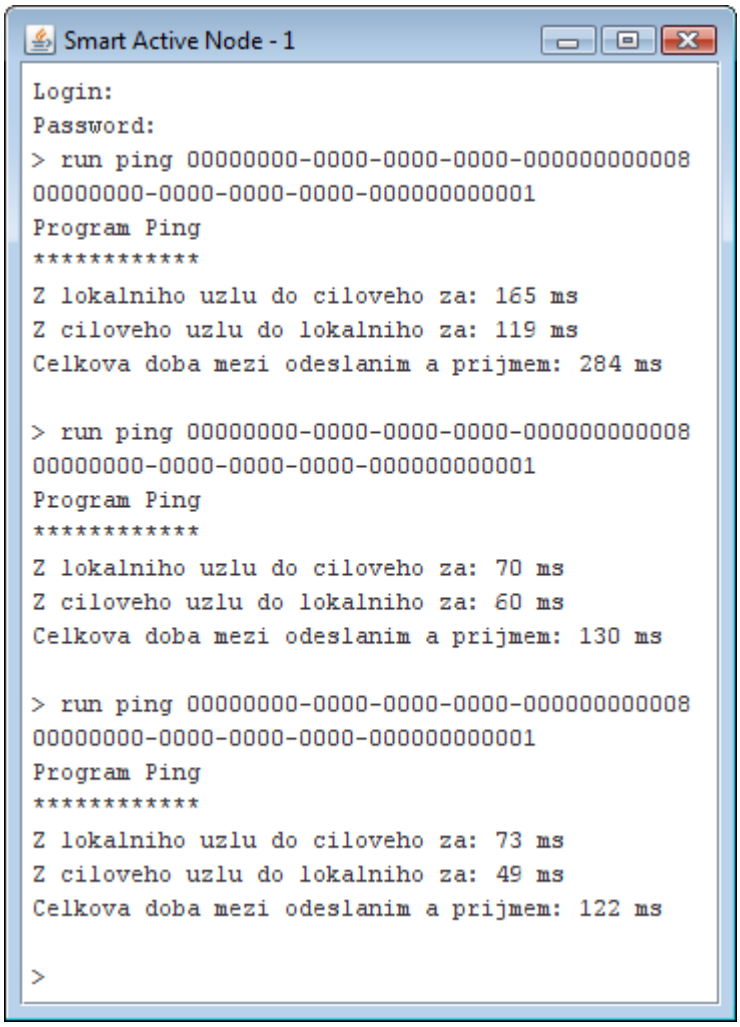
6.2 Test na jednom počítači

Při prvním testování byla virtuální síť namodelována na jednom počítači. Konfigurace tohoto počítače je shrnuta v tabulce 2.

CPU	Intel Core 2 Duo T7300 @ 2,0 GHz
Velikost operační paměti	4 GB
Operační systém	Windows Vista Bussiness, SP 1
Typ operačního systému	64 bitový
Verze JVM	Java SE Runtime Environment 1.6.0-b105

Tabulka 2: Konfigurace při testu na jednom počítači

Výsledek tohoto testu je nejlépe vidět na obrázku 17. Aplikace ping byla opakovaně spuštěna a na uvedeném obrázku je vidět reprezentativní vzorek testu. Při prvním spuštění byl aktivní program capsule dostupný pouze na uzlu, na kterém byla aplikace spuštěna. Aktivní program se tedy při průchodu capsule sítí distribuoval a proto je i čas, za který se capsule vrátila zpět do výchozího uzlu přibližně dvakrát delší než časy při následných spuštěních aplikace ping.



Obrázek 17: Výsledek testu na jednom počítači

6.3 Test na více počítačích se systémem Windows

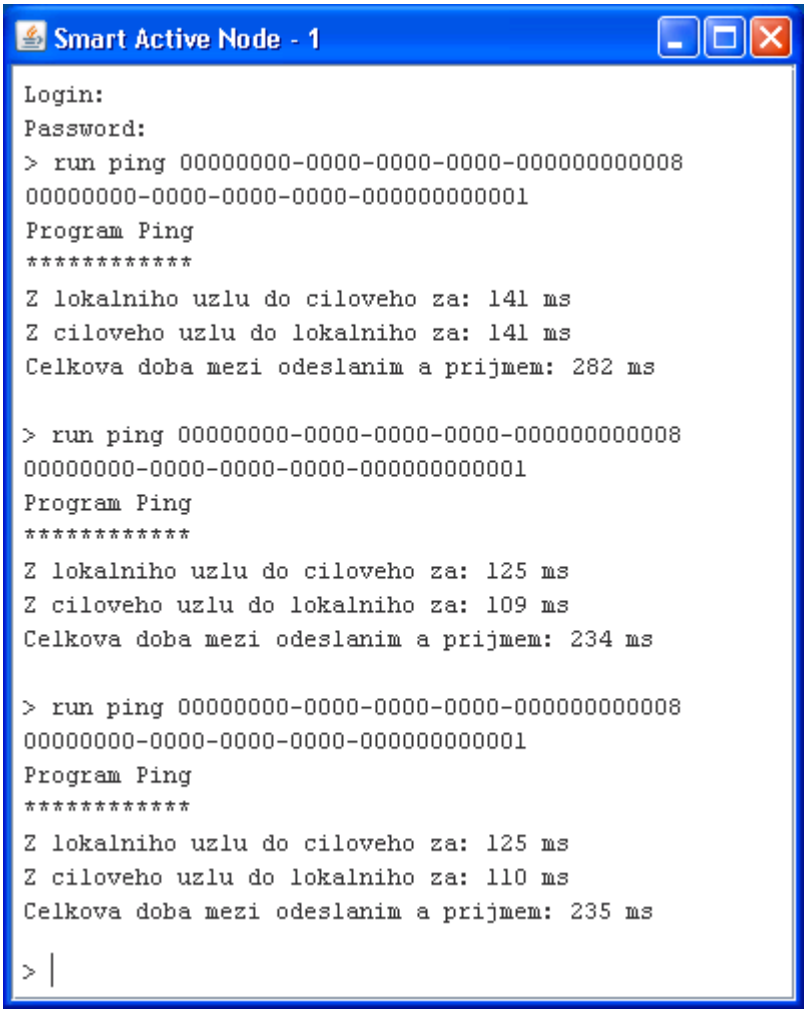
Při druhém testu bylo testováno chování SANu v síťovém prostředí s počítači s operačním systémem Windows. Konfigurace počítačů, použitých při testu, byla stejná a je zachycena v tabulce 3.

CPU	Intel Pentium 4 @ 2,6 GHz
Velikost operační paměti	512 MB
Operační systém	Windows XP Profesional, SP 2
Typ operačního systému	32 bitový
Verze JVM	Java SE Runtime Environment 1.6.0_04-b12

Tabulka 3: Konfigurace při testu na více počítačích se systémem Windows

Reprezentativní výsledek testu je vidět na obrázku 18. Při testování sítě uzlů SAN na šesti počítačích se systémem Windows a oddělených počítačovou sítí byly časy získané

aplikací ping delší než na jednom počítači. Při testu byla aplikace ping dostupná ve všech uzlech sítě a nebylo třeba jí distribuovat.



Obrázek 18: Výsledek testu na šesti počítačích se systémem Windows

6.4 Test na více počítačích se systémem Linux

Při třetím testu bylo testováno chování SANu v síťovém prostředí s počítači s operačním systémem Linux. Konfigurace počítačů, použitých při testu, byla stejná a je zachycena v tabulce 4.

CPU	Intel Pentium 4 @ 2,6 GHz
Velikost operační paměti	512 MB
Operační systém	Linux version 2.6.21.3-grsec (Debian 4.1.1-21)
Typ operačního systému	32 bitový
Verze JVM	Java SE Runtime Environment 1.6.0_04-b105

Tabulka 4: Konfigurace při testu na více počítačích se systémem Linux

Reprezentativní výsledek testu je vidět na obrázku 19. Při testování sítě uzlů SAN na šesti počítačích se systémem Linux a oddělených počítačovou sítí byly časy získané aplikací ping trochu menší než při stejném testu a konfiguraci počítačů se systémem Windows. Při testu byla aplikace ping dostupná ve všech uzlech sítě a nebylo třeba jí distribuovat.

Obrázek 19: Výsledek testu na šesti počítačích se systémem Linux

6.5 Zhodnocení výsledků

Na první pohled vidíme, že naměřené výsledky nejsou moc pozitivní. Co je ale příčinou takto vysokých čísel?

Jedním z hlavních výkonnostních problémů je použitý interpret BeanShell [23]. Jeho hlavním nedostatkem je přímá interpretace zdrojových souborů. Interpret prochází zdrojový soubor a přímo interpretuje, co je v něm napsáno. Nesnaží se o optimalizaci vykonávaného kódu. Tento problém byl znám už při výběru nástroje na interpretaci aktivního kódu. Nebyl to však jediný důvod, kvůli kterému se začal vyvíjet vlastní interpret. Ten je zatím současné době ve stádiu integrace s aktivním uzlem SAN.

Dalším problémem může být technika serializace javových tříd, která se používá v samotné aplikaci ping. Technika serializace nemusí být použita. Je možné zajistit vlastní uložení přenášených dat. Klade to však větší nároky na programátora a znepřehledňuje zdrojový kód.

Jako další faktor zmenšující rychlost je aplikace ping samotná. Pokud se podíváme na její zdrojový kód, zjistíme, že není zcela optimální. Navíc se přenáší víc dat, než je zcela nutné.

Zcela určitě má na naměřených časech svůj podíl samotná Java, ve které je naprogramován uzel SAN. S tímto problémem bylo počítáno už při návrhu aktivního uzlu SAN a rychlost vykonávání byla obětována ve prospěch jiných, dobrých vlastností.

Z provedených testů je také vidět rozdíl při používání SANu v prostředí počítačů s operačním systémem Linux nebo Windows. Výsledky testů jsou při použití stejných počítačů s operačním systémem Linux o něco málo lepší než v případě Windows.

Poznámka

Testování SANu zároveň v prostředí se systémy Windows a Linux nemá význam zde uvádět. Naměřené výsledky jsou podobné zde uvedeným. Jsou závislé na zastoupení jednotlivých operačních systémů a podle toho se přiklánějí k uvedeným výsledkům.

Pro případ distribuce aplikace, pokud se na ostatních uzlech nenachází, jsou výsledky testů pro danou konfiguraci vždy přibližně dvakrát horší než v případě, že se aplikace nachází na všech uzlech sítě.

7 Závěr

Za vznikem konceptu aktivních sítí stojí agentura DARPA. Tento koncept si klade za cíl vytvořit síť, která odstraní nedostatky současných sítí, jako například pomalé nasazování nových protokolů a služeb, nebo problémy s bezpečností přenosu a mobilitou prvků sítě.

V této diplomové práci byly představeny vybrané implementace aktivních sítí, které vznikly době, kdy byl koncept aktivních sítí představen. Byla také představena implementace vytvořená za účelem vývoje nové metody přerozdělování zátěže distribuovaných aplikací v heterogenním distribuovaném prostředí.

Zatím nepřevážila žádná konkrétní implementace aktivního serveru. Důvodem mohou být rozdílná API, nebo fakt že financování programu skončilo dřív, než se dostatečně rozšířil nebo se jedná o důsledek, že vývoj nových síťových architektur je prostě náročný [28]. Také je zapotřebí vyřešit otázku bezpečnosti systému proti různým formám útoků a nekorektně naprogramovaným aplikacím. Každopádně se díky flexibilitě a univerzálnosti aktivních sítí stále objevují projekty směrované na tuto platformu [29, 30, 31, 32].

7.1 Vlastní přínos

V návaznosti na implementaci aktivních sítí Grade32 vznikla tato práce, jež se zaměřila na implementaci serveru aktivních sítí (nazývaného Smart Active Node; zkratkou SAN). Tato práce tak spojuje poznatky z předchozích implementací a odstraňuje některé jejich nedostatky. Díky použitému modernímu jazyku Java, který navíc zajišťuje přenositelnost mezi různými systémy, je tak usnadněn výzkum a vývoj aktivních sítí při použití této implementace.

Aktivní server SAN je po jednoduchém počátečním nastavení schopný automaticky vytvářet spojení s ostatními aktivními servery SAN. Automaticky se tak dokáže začlenit do sítě a plně v ní fungovat. Díky použití jednoduchého a známého jazyka Java, který je použit pro vytváření aplikací a capsulí, můžeme jednoduše implementovat programy a služby, které pak nasadíme do aktivní sítě. Nasazení stačí provést na uzlu, ze kterého chceme začít používat náš aktivní program. V případě potřeby bude aktivní program automaticky distribuován do ostatních uzlů, které jej budou potřebovat.

7.2 Otevřené problémy

V této práci byly otevřeny některé otázky, které jsou řešeny už v současné době, nebo se předpokládá jejich řešení v nadcházejících projektech.

Jako první můžeme uvést problém s identifikací aktivního uzlu. Ve Smart Active Node zavádíme vlastní identifikaci každého síťového rozhranní aktivního uzlu jako 256 bitové číslo. Toto číslo je rozděleno na id uzlu a sítě, kde by měla samotné id uzlu jednoznačně identifikovat aktivní uzel v rámci celé sítě. U síťové části předpokládáme její využití zvláště pro účely směrování při mobilitě aktivních uzlů.

Na některých místech této práce bylo poukázáno na otázku bezpečnosti a některé bezpečnostní problémy byly vyřešeny. Vzhledem k tomu, že je ale otázka bezpečnosti v aktivních sítích značně široké téma, není řešena v této práci. V rámci projektu SAN je řešena v diplomové práci Alexandre Junqueira, která by měla být dostupná přibližně ve stejné době, jako tato diplomová práce.

Jak bylo v kapitole o interpretaci aktivního kódu uvedeno, nezabývá se tato práce vytvořením vlastního interpretu, ale využívá se již existujícího interpretu BeanShell. Ten však nesplňuje všechny naše požadavky, a proto je používán jen jako dočasná varianta. Už nyní je vyvíjen nativní interpret.

7.3 Následná práce

Kromě představených otevřených problémů by bylo vhodné zabývat se v následné práci optimalizací samotného aktivního serveru SAN nebo rozšíření počtu aplikací pro tento server. Určitě by bylo vhodné vytvořit aplikaci pro získání a vykreslení topologie sítě, aplikaci pro trasování cesty z jednoho uzlu do druhého, aplikaci pro přístup ke konzoli jiného uzlu, případně jednoduchou síťovou hru. Zajímavou úlohou by bylo vytvořit distribuovaný souborový systém nad sítí aktivních serverů SAN.

8 Přehled zkratek

SAN	Smart Active Node
DARPA	Defense Advanced Research Projects Agency
SPK	Smart Active Node Package
JAR	Java Archiv
PLAN	Packet Language for Active Networks
DLL	Dynamic Link Library
FIFO	First In First Out
JVM	Java Virtual Machine
ANTS	Active Node Transfer System

9 Použitá terminologie

Aktivní síť	– programovatelná síťová architektura
Aktivní uzel	– síťový uzel aktivní sítě
Aktivní kód	– kód popisující chování aktivního programu
Aktivní program	– kód, který je potencionálně vykonáván aktivním uzlem
Capsule	– paket aktivní sítě, je asociována s aktivním programem, který se vykonává při průchodu capsule aktivním uzlem
Aktivní aplikace	– aktivní program, jehož instance je vykonávána po celou dobu svého života pouze na jednom uzlu aktivní sítě

10 Literatura

[1] STRATEGIC TECHNOLOGY OFFICE. *Towards Active Networks* [on line]. 1997 [cit. 4. prosince 2007]. Dostupný z WWW: < http://www.arpa.mil/STO/strategic/an.html >

[2] TENNEHOUSE, D., WETHERALL, D. Towards an Active Network Architecture. In *DARPA Active Networks Conference and Exposition 2002*. San Francisco, 2002.

[3] NAJAFI, K., LEON-GARCIA, A. A Novel Cost Model for Active Networks. In *Proceedings of Int. Conf. on Communication Technologies, World Computer Congress 2000*. Beijing, 2000.

[4] KOUTNÝ, T. *Load-Balancing in Distributed Environment*. Plzeň 2004. Technická zpráva DCSE/TR-2004-06. Fakulta aplikovaných věd Západočeské univerzity v Plzni.

[5] KOUTNÝ, T., ŠAFAŘÍK, J. Gradient Method with Topology Discovery for Load-Balancing in Active Networks. *Proceedings of 11th Annual IEEE International Conference on the Engineering of Computer Based Systems*. Brno, Czech Republic, 2004.

[6] KOUTNÝ, T., ŠAFAŘÍK, J. Load Redistribution in Heterogeneous Systems. *Proceedings of the Third International Conference on Autonomic and Autonomous Systems.* Athens, Greece, 2007.

[7] KOUTNÝ, T., ŠAFAŘÍK, J. Simulating Distributed Applications in an Active Network. *Proceedings of 6th Eurosim Congress*. Ljubljana, Slovenia, 2007.

[8] MALAKOOTI, B., THOMAS, I. A freamwork for an Intelligent Internet Protocol for a Space-Based Internet. *22nd AIAA International Communications Satellite Systems Conference and Exhibit 2004 (ICSSC)*. Monterey, California, 2004.

[9] KOUTNÝ, T. *Reference Implementation of Grade32 AN Server* [on line]. 2004 [cit. 10. října 2006]. Dostupný z WWW: <http://www.kiv.zcu.cz/~txkoutny/download/grade32.zip>

[10] WETHERALL, D. *Service Introduction in an Active Network*. [s.l.], 1999. 150 s. Dizertační práce.

[11] WETHERALL, D. J., GUTTAG, J., TENNENHOUSE, D. L. ANTS: Network Services Without the Red Tape. *Computer*, April 1999, vol. 32, no. 4, s. 42-48.

[12] WETHERALL, D., GUTTAG, J., TENNENHOUSE, D. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE Open Architectures and Network Programming Conference*, 1998. s. 117-129.

[13] KAKKAR, P. *The specification of PLAN* [on line]. 12. července 1999. Dostupný z WWW: <http://www.cis.upenn.edu/~dsl/PLAN/spec/spec.ps>

[14] KAKKAR, P., HINKS, M., MOORE, J., GUNTER, C. Specifying the PLAN Network Programming Language. *Electronic Notes in Theoretical Computer Science*, 1999, vol. 26. s. 87-104.

[15] HINKS, M., KAKKAR, P., MOORE, J., GUNTER, C., NETTLES, S. PLAN: A Packet Language for Active Networks. *ICFP 1998*. Baltimore, Maryland, USA, 1998.

[16] HINKS, M., MOORE, J., ALEXANDER, S., GUNTER, C., NETTLES, S. PLANet: An Active Internetwork. *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*. New York, 1999.

[17] ALEXANDER, D. The SwitchWare Active Network Architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 1998, vol. 12, no. 3, s. 29-36.

[18] ALEXANDER, D. The SwitchWare Active Network Implementation. *The 1998 ACM SIGPLAN Workshop on ML*. Baltimore, Maryland, USA, 1998.

[19] MERUGU, S., BHATTACHARJEE, S., CHAE, Y., SANDERS, M., CALVERT, K., ZEGURA, E. Bowman and CANEs: Implementation of an Active Network. Invited paper at *37th Annual Allerton Conference*, September 1999.

[20] YEMINI, Y., DA SILVA, S. Towards Programmable Networks. *Proceedings of IFIP/IEEE International Workshop on Distributed System: Operations and Management*. October, 1996.

[21] GOSTLING, J., JOY, B., STEELLE, G., BRACHA, G. *The The Java™ Language Specification.* 3. vydání. Prentice Hall PTR, 2005. Dostupný také z WWW: <http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>. ISBN 0-321-24678-0.

[22] RAMAKRISHNA, V., ROBINSON, M., EUSTICE, K., REIHER, P. An Active Self-Optimizing Multiplayer Gaming Architecture. *Cluster Computing*, 2006, vol. 9, no 2. s. 201 – 215.

[23] NIEMEYER, P. *BeanShell 2.0* [online]. 2005 [cit. 8. ledna 2007]. Dostupný z WWW: <http://www.beanshell.org/>

[24] WEIS, R., LUCKS, S. Cryptographic Hash Functions – Recent Results on Cryptanalyis and their Implications on System Security. In *SANE 2006*. Delft, 2006.

[25] BARRETO, P. S. L. M., RIJMEN, V. The Whirlpool Hashing Function. *First open NESSIE Workshop*. Leuven, Belgium, 2000.

[26] BOVERT, D. P., CESATI, M. *Understanding the Linux Kernel*. 1. vyd. O'REILLY, 2000, 704 s. ISBN 0-596-00002-2.

[27] RUSSINOVICH, M. E., SOLOMON, D. A. *Microsoft Windows internals : Microsoft Windows Server 2003, Windows XP, and Windows 2000*. [s.l.] : Microsoft Press, 2005. 976 s. ISBN 0-7356-1917-4.

[28] ACM SIGCOMM Computer Communication Review, 2006, vol. 36, no. 2, s. 27 – 30.

[29] FRITSCH, W., MEYER, K., KIRSTEIN, P., BHATTI, S., SACKS, L. Programmable Active Networks for Next Generation Multimedia Services. *European Space Agency (ESA)*. 25. listopadu 2005.

[30] CHENG, L., GALIS, A. Security Protocol for Active Networks. In *14th IEEE International Conference on Networks*. Singapore, Malaysia, 2006. s. 1 – 6.

[31] RAMAKRISHNA, V., ROBINSON, M., EUSTICE, K., REIHER, P. An Active Self-Optimizing Multiplayer Gaming Architecture. *Cluster Computing*, 2006, vol. 9, Issue 2. s. 201 – 215.

[32] NIE FEI, LI ZENG-ZHI. Congestion Control Model Based on Hop-by-hop Feedback for Active Networks. In *8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. Tsingtao, China 2007. s. 490 – 494.

Příloha A: API capsulí a aktivního uzlu ANTS

API poskytované ANTS aplikacím capsulí shrnuje následující tabulka. V první části tabulky jsou vyjmenovány metody používané capsulemi pro získání základních informací o aktivním uzlu, získání rozšíření uzlu nebo zapsání ladících zpráv. Ve druhé části tabulky jsou metody pro uchování objektů v dočasném úložišti aktivního uzlu a ve třetí části metody pro doručení capsule do lokální aplikace nebo jiného uzlu sítě. Všechny v této příloze popsané API jsou převzaty z [10].

Metoda	Popis
`int getAddress()`	Vrátí adresu aktuálního uzlu.
`ChannelObject getChannel()`	Vrátí kanál, odkud byla capsule přijata.
`Extension findExtension(string ext)`	Najde požadované rozšíření uzlu.
`long time()`	Vrátí lokální čas.
`void log(String msg)`	Zapíše ladící zprávu do logu.
`Object put(Object key, Object val, int age)`	Vloží objekt do dočasného úložiště.
`Object get(Object key)`	Vrátí objekt z dočasného úložiště.
`Object remove(Object key)`	Odstraní objekt z dočasného úložiště.
`void routeForNode(Capsule c, int n)`	Pošle capsuli do uzlu.
`void deliverToApp(Capsule c, int a)`	Doručí capsuli do lokální aplikcae.

Tabulka 1: API pro přístup k aktivnímu uzlu z Capsule

Následující API je používání aplikacemi pro přístup k aktivnímu uzlu. Jsou zde také metody pro registraci nových protokolů nebo odesílání a příjem capsulí.

Metoda	Popis
`void attachNode(Node n)`	Připojení se k lokálnímu uzlu.
`Node getNode()`	Vrátí připojený uzel.
`short getPort()`	Vrátí přidělený port.
`int getDefaultResources()`	Vrátí výchozí maximální limit zdrojů.
`void setDefaultResources()`	Nastaví výchozí maximální limit zdrojů.
`void register(Protocol p)`	Zaregistruje protokol.
`void unregister(Protocol p)`	Odregistruje protokol.
`void send(Capsule c)`	Odešle capsuli přes připojený uzel s výchozím maximálním množstvím zdrojů.
`void send(Capsule c, int l)`	Odešle capsuli přes připojený uzel s definovaným maximálním množstvím zdrojů.

`void receive(Capsule c)`	Přijme capsuli z připojeného uzlu.

Tabulka 2: API pro přístup k aktivnímu uzlu z Aplikace

Při vykonávání programů capsulí může dojít k několika zásadním výjimečným situacím. Tyto výjimky shrnuje následující tabulka.

Výjimka	Význam
`ResourceLimitException`	Na požadovanou operaci nemá capsule dostatečné množství prostředků.
`TimeLimitException`	Přeposílací program vyčerpal přidělené časové kvantum.
`NoSuchRouteException`	Neexistuje implicitní cesta do požadovaného uzlu.
`NoSuchApplicationException`	Požadovaná lokální aplikace neexistuje.

Tabulka 3: Výjimky při vykonávání programů capsulí

Následující tabulka shrnuje metody, které je možné volat nad abstraktními třídami Capsule a DataCapsule. Uvedené třídy se používají přenos jako obalové třídy pro přenos dat nebo pro spouštění aktivních programů.

Metoda	Popis
`int getSrc()`	Vrátí adresu zdrojového uzlu.
`int getDst()`	Vrátí adresu cílového uzlu.
`void setDst()`	Nastaví cílový uzel.
`int getResources()`	Vrátí zbývající prostředky.
`void prime(Capsule parent)`	Převede zbývající prostředky na novou capsuli.
`int  getPrevious()`	Vrátí adresu předchozího aktivního uzlu, na kterém byla capsule zpracovávána.
`byte[] getCapsuleID()`	Vrátí typ capsule.
`byte[] getGroupID()`	Vrátí typ skupiny.
`byte[] getProtocolID()`	Vrátí typ protokolu.
`short getSrcPort()`	Vrátí zdrojový port.
`void setSrcPort()`	Nastaví zdrojový port.
`short getDstPort()`	Vrátí cílový port.
`void setDstPort(short port)`	Nastaví cílový port.
`ByteArray getData()`	Vrátí užitečná data.
`void setData(ByteArray data)`	Nastaví užitečná data.

Tabulka 4: Metody abstraktních tříd Capsule a DataCapsule

Příloha B: Příklad aplikace pro Smart Active Node – Ping

V tomto příkladu je popsána aplikace ping zjišťující, za jak dlouho se dostane capsule z jednoho uzlu na druhý. Naměřené doby vypíše do konzole aktivního uzlu.

Příklad se skládá ze dvou částí. V první je třída `Ping` napsaná v jazyce Java, která tvoří aplikaci běžící v prostředí aktivního uzlu SAN. Ve druhé části je třída `PingCapsule` napsaná také v jazyce Java tvořící capsuli použitou v této aplikaci.

Aplikace ping je odladěná v prostředí interpretu BeanShell. Až bude plně dostupný nativní interpret SANu, měla by být aplikace běhuschopná i v něm.

Třída Ping

```
/**
 * Aplikace ping.
 *
 * Tato aplikace vysle capsuli na cilovy uzel a zjistuje, jak dlouho trva
 * cesta k tomuto uzlu a zpet.
 *
 * Trida splnuje rozhrani IApplication aby se z ni stala aplikace v
 * prostredi aktivnich siti SAN.
 *
 * Implementovane rozhrani ReceiveDataListener je pouzite z duvodu prijmu
 * dat od capsule.
 */
public class Ping implements IApplication, ReceiveDataListener {

  // Zamek pouzity pri cekani na zaslani dat od capsule.
  private Object lock = new Object();

  // Datovy naklad, ktery bude zaslan capsuli.
  private TransmitData received = null;

  public void main(String[] args, IApplicationAPI applicationAPI) {

    // Zaregistrovani posluchace pro prijem datoveho nakladu
    // od capsuli. V pripade, ze capsule zasle nejaka data teto
    // aplikaci, bude to umozneno prave diky tomuto rozhrani.
    applicationAPI.addOnSendDataListener(this);

    // Ziskani standardniho vystupu. Muze to byt napriklad
```

```
// konzole nebo soubor.
PrintWriter out = applicationAPI.getStdOut();

try{
  // Ziskani cilove adresy pro zaslani capsule.
  // Tato adresa je parametrem pri spusteni programu.
  Guid toAddr = null;
  Guid toNet = null;
  NetIdentifier pingDst = null;
  try{
    // Naparsovani adresni casti adresy ciloveho uzlu.
    toAddr = Guid.parseGuid(args[0]);
    // Naparsovani sitove casti adresy ciloveho uzlu.
    toNet = Guid.parseGuid(args[1]);
    // Vytvoreni identifikatoru ciloveho uzlu.
    pingDst  = new NetIdentifier(toAddr,toNet);
  }catch(NumberFormatException e) {
    // Uzivatel zadal adresu v chybnem tvaru.
    // Vypiseme chybove hlaseni.
    out.print(e);
    return;
  }

  // Definovani objektu pro uchovani jednotlivych casovych znacek.
  Date sendingTime = new Date();
  Date destinationTime = null;
  Date receivedTime = null;

  // Vytvoreni datoveho nakladu pro capsuli
  // pouzitim serializace objektu.
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream(bos);

  // Nastaveni informace, ze je capsule na ceste k cilovemu uzlu.
  oos.writeBoolean(false);
  // Zapsani casu, kdy byla capsule vytvorena.
  oos.writeObject(sendingTime);

  oos.close();

  // Vytovreni datoveho nakladu pro capsuli.
```

```
TransmitData data = new TransmitData(bos.toByteArray());

// Ziskani identifikatoru kodu capsule.
CodePkgIdent capsulePkgIdent =applicationAPI.getExecutedCodeIdent();

// Vytvoreni a odeslani capsule do ciloveho uzlu.
applicationAPI.injectCapsule(capsulePkgIdent,pingDst,data);

// Cekani na prijem dat od prichozi capsule.
synchronized (lock) {
  lock.wait();
}

// Aplikace byla vzbuzena pri cekani na data od casule.

// Zjisteni, zda byla prijata potrebna data.
if(received != null) {

  // Deserializace prijatych dat.
  ByteArrayInputStream bais = new
                          ByteArrayInputStream(received.getData());
  ObjectInputStream ois = new ObjectInputStream(bais);

  // Precteni hodnoty smeru, kterym se capsule odebirala.
  ois.readBoolean();
  // Precteni casu odeslani capsule z tohoto uzlu.
  sendingTime = (Date) ois.readObject();
  // Precteni casu odeslani capsule z ciloveho uzlu.
  destinationTime = (Date) ois.readObject();
  // Ziskani casu prijmu.
  receivedTime = new Date();

  // Vypis hodnot na standardni vystup.
  out.println("Program Ping");
  out.println("************");
  out.println("Z lokalniho uzlu do ciloveho za: " +
          (destinationTime.getTime() - sendingTime.getTime()) +"
          ms");
  out.println("Z ciloveho uzlu do lokalniho za: " +
          (receivedTime.getTime() - destinationTime.getTime()) +"
          ms");
```

```
          out.println("Celkova doba mezi odeslanim a prijmem: " +
                  (receivedTime.getTime() - sendingTime.getTime()) +" ms");
          out.println();
        } else {
          out.println("Chyba pri prijmu dat od capsule.");
        }
      } catch(Exception e) {
        // Vypis chyby, ktera mohla vzniknout pri praci s aktivnim uzlem.
        out.println(e);
      }
    }

    /*
     * Metoda, ktera se zavola, kdyz capsule odesle data teto aplikaci.
     */
    public void receiveData(Guid sender, TransmitData data) {
      // Nastaveni prijatych dat.
      received = data;

      // Probuzeni aplikace.
      synchronized (lock) {
        lock.notify();
      }
    }
}
```

Třída PingCapsule

```
/**
 * Capsule programu Ping.
 *
 * Tato capsule putuje na cilovy uzel, kde zjisti lokalni cas, ktery na
 * ceste zpet preda aplikaci, ktera tuto capsuli pouzila.
 *
 * Trida splnuje rozhrani ICapsule, aby se z ni stala capsule v prostredi
 * aktyvnich siti SAN.
 */
public class PingCapsule implements ICapsule {

  public void main(ICapsuleAPI capsuleAPI) {
    try{
      // Ziskani dat v podobe jednotlivych bytu,
```

```
// ktere capsule prenasi.
TransmitData data = capsuleAPI.getDataLoad();

// Prevedeni bytovych dat na data v podobe objektu.
ByteArrayInputStream bais = new
                              ByteArrayInputStream(data.getData());
ObjectInputStream ois = new ObjectInputStream(bais);

// Ziskani informace, zda je capsule na ceste k cilovemu
// uzlu nebo na ceste zpet.
boolean returnBack = ois.readBoolean();

// Rozhodnuti o dalsi cinnosti na zaklade toho,
// kterym smerem se capsule ubira.
if(returnBack) {
  // V tomto pripade je capsule na ceste zpet.

  // Zjisteni, zda uz capsule dorazila na uzel
  // ze ktereho byla puvodne vyslana.
  if(capsuleAPI.isThisServer(capsuleAPI.getDestination())) {
    // Capsule se nachazi na uzlu, ze ktereho byla puvodne vyslana.

    //Ziskani identifikatoru aplikace, ktera capsuli vytvorila.
    Guid sourceApp = capsuleAPI.getSourceApplication();
    // Zaslani tak aplikaci, ktere capsuli vytvorila.
    capsuleAPI.sendData(sourceApp, data);

    // Oznameni o ukonceni cinnosti teto capsule.
    // Capsule nebude jiz nadale zpracovavana a
    // bude ze site odstranena.
    capsuleAPI.finished();
  }
} else {
  // V tomto pripada je capsule na ceste k cilovemu uzlu.

  // Zjisteni, zda capsule dosahla ciloveho uzlu.
  NetIdentifier pingDestination = capsuleAPI.getDestination();
  if(capsuleAPI.isThisServer(pingDestination)){
    // Capsule se nachazi na cilovem uzlu.

    // Zmena smeru, kterym bude capsule smerovana.
```

```
            // Capsule bude smerovana na uzel, na kretem vznikla.
            capsuleAPI.setDestination(capsuleAPI.getSource());

            // Precteni casu odeslani capsule. Tento cas nastavila
            // aplikace na uzlu, ze ktereho capsule prisla.
            Date senderDate = (Date) ois.readObject();

            // Vytvoreni noveho nakladu pro capsuli.
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(bos);

            // Zapsani informace, ze je capsule na ceste zpet.
            oos.writeBoolean(true);
            // Zapsani casu odeslani capsule ze zdrojoveho uzlu.
            oos.writeObject(senderDate);
            // Zapsani nynejsiho casu.
            oos.writeObject(new Date());

            oos.close();

            // Vytvoreni novych dat, ktera bude capsule prenaset.
            TransmitData dataBack = new TransmitData(bos.toByteArray());
            // Prirazeni novych dat k capsuli.
            capsuleAPI.setDataLoad(dataBack);
          }
        }
      }catch(Exception e) {
        // Chyby, ktera vznikla pri praci s aktivnim uzlem,
        // do lokalniho logu.
        LoggerFactory.createLogger().Log(e, LogType.ERROR);
      }
    }
}
```