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

Diplomová práce

Tunelování IP aplikací v aktivních sítích

Plzeň, 2009 Jakub Sýkora

ORIGINÁL ZADÁNÍ PRÁCE

Prohlášení

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

V Plzni dne 13.8.2009 ….....

Jakub Sýkora

Abstract

Nowadays nearly all computer networks including the Internet are based on the Internet Protocol. It began to evolve in the 60s and 70s of the past century as US Department of Defense initiative, namely the branch that is known as DARPA[1] nowadays. These years were the era of slow machines and so the protocol was designed with performance in mind. Thirty years after 4th version of the Internet Protocol, it begun to reach its limits. The main idea of further development was to enrich current protocol to address its problems. So, 6th version was defined in 1998.

However, IPv6 is not backward compatible. Counting in other drawbacks, such as complexity, learning curve and implementation costs, it is still not a major protocol despite the importance of IP[2].

In 1995, DARPA initiated a work on another concept of computer networking that would overcome shortcomings of entire IP design. The concept is about a programmable network, where each packet is associated with a program code that defines its behavior and thus available network services. The concept has been called Active Networks.

Although development of Active Networks ceased as DARPA cut down research funding, the concept is still attractive for its architectural features, such as flexibility and rapid development of network applications.

The scope of this thesis is to provide possibility to transparently transfer IPv4 services over active network namely the SAN active network implementation, which is being developed at Department of Computer Science and Engineering on University of West Bohemia.

1 Defense Advanced Research Projects Agency
2 http://bgpmon.net/blog/?p=166

Poděkování

Chtěl bych poděkovat především své přítelkyni Anně za vytrvalou podporu ve studiu, která mi vůbec umožnila splnit všechny nutné předpoklady a tuto práci vypracovat.

Poděkovat bych chtěl také Ing. Tomáši Koutnému, Ph. D. za vedení práce a poskytnutí možnosti seznámit se s aktivními sítěmi jako zajímavou alternativou klasických sítí.

Další poděkování patří rodině, kolegům a přátelům za podporu při studiu a práci, bez níž bych studovat nemohl.

Na závěr bych rád poděkoval všem, které jsem měl tu čest poznat během svého studia na Západočeské univerzitě, neboť každý z nich svým způsobem obohatil můj život a pohled na něj.

Obsah

1 Úvod

V dnešní době jsou téměř všechny počítačové sítě včetně Internetu založeny na protokolu IP, který byl navržen a začal se vyvíjet v šedesátých a sedmdesátých letech dvacátého století v DARPA. To byla doba relativně pomalých strojů a cílem bylo přenést data s minimální režií z uzlu A do uzlu B. Po třiceti letech však začali uživatelé čtvrté verze protokolu IP narážet na omezení tohoto protokolu. Hlavní myšlenkou při dalším vývoji bylo logicky rozšířit čtvrtou verzi tak, aby se odstranily nejpalčivější problémy stávající verze protokolu, a tak vznikl IP verze 6. Nicméně IPv6 se stala během procesu standardizace velice komplexním a složitým protokolem, který se těžko implementuje a není snadné jej pochopit. Navíc se nepodařilo odstranit některé problémy spojené například s dlouhou standardizací potřebnou k zavedení nové služby nebo protokolu. Původní IPv4 se za dobu svojí existence stal také velmi komplexním – celkově jeho standardy čítají dle [BS02] přes 4000 RFC[3].

V roce 1995 začali vědci v DARPA pracovat na odlišném konceptu sítí, která neuvažuje mezilehlé uzly v síti jako pasivní zařízení, která pouze předávají pakety z jedné sítě do druhé. Během let vzniklo bezpočet implementací aktivních sítí k různým účelům, ale žádná implementace dosud nedosáhla takového rozšíření, aby bylo možné ji uvažovat jako náhradu IPv4.

Pokud máme uvažovat aktivní síť jako náhradu IPv4 nebo její doplněk, musíme se zabývat myšlenkou provozu nezměněných aplikací pro IPv4 v prostředí aktivních sítí, neboť se nedá předpokládat, že by bylo možné stávající aplikace jakkoliv upravovat pro provoz v nových sítích. Provoz nezměněných aplikací využívajících protokol IP v prostředí aktivních sítí je předmětem této práce. Vzhledem k minimální rozšířenosti protokolu IPv6 se v celé práci uvažuje IP jako IPv4. Pokud bude zmíněn IPv6, vždy bude zapsán takto. Provoz IPv6 aplikací není v této práci řešen, neboť je konceptuálně podobný a s minimálními změnami je možné pro něj připravit podporu.

1.1 Koncept aktivních sítí

Tradiční síť je zpravidla lhostejná k datům, které jí procházejí a chová se staticky – podle předem daných pravidel předává data z uzlu do uzlu. Paket takové sítě obsahuje zpravidla pouze identifikaci zdroje a cíle a data, která zpracuje aplikace na cílovém uzlu. Paket je také uvažován pouze jako obal dat a zaniká při doručení na cílový uzel.

Naproti tomu aktivní sítí rozumíme takovou síť, která je na každém mezilehlém uzlu

3 Request for Comment – dokument IETF pro publikaci nových standardů

vybavena prostředím, ve kterém je možné spouštět kód za předem daných podmínek. Rozdíl mezi klasickou a aktivní sítí je zobrazen na ilustraci 1. Prostředí pro spuštění kódu je v podstatě nezávislý operační systém a označuje se v literatuře běžně jako Execution Environment, zkráceně EE. Jedná se o prostředí, které je spuštěno pouze pro aplikaci a nemůže tak narušit chod vlastního systému aktivního uzlu[4]. Kód vyžadované aplikace nemusí být dokonce ani v uzlu přítomen, neboť je možné jej získat ze sítě.

Na rozdíl od klasické sítě je místo paketů využíváno kapsulí, které s sebou kromě dat nesou i informaci o aplikaci, která ji vytvořila a mohou například požadovat spuštění této aplikace na každém mezilehlém uzlu. Kapsle při průchodu sítě zaniká až v momentě, kdy už není dál potřeba a nemusí to být nutně při dosažení cílového uzlu. Cílový uzel může kapsli například pouze modifikovat, změnit její směr a ponechat ji v síti. Toto chování je demonstrováno například v aplikaci ping uvedené v [RM08]. Zatímco v klasické síti musí mít každá služba přiřazen svůj port a je podle něj identifikována, v aktivní sítí je nahrazena identifikátorem aplikace a EE samo zajistí doručení dat cílové aplikaci. To ve výsledném efektu značně zjednodušuje psaní vlastních aplikací a výměnu dat mezi nimi. Takovýto rozdíl není z hlediska uživatele a programátora nejdůležitější, proto vysvětlíme rozdíl na příkladu.

Příklad *Na ilustraci 1 vidíme v horní části klasickou síť s přepínači, směrovači a koncovými uzly. Pokud bude uzel A chtít vykonat jednoduchou činnost, jakou je zjištění odezvy k uzlu B, využije v RFC definovanou zprávu ICMP echo request[5] a odešle ji do uzlu B. Všechny směrovače po cestě RFC znají a vědí, že na zprávu nemají odpovídat[6], neboť není určena pro ně. Když zpráva dorazí do uzlu B, ten podle RFC ví, že může odpovědět zprávou ICMP echo reply[7]. Pokud tak učiní, zpráva se přenese do uzlu A a uzel A nyní ví, že uzel B je dosažitelný. Pokud budeme uvažovat lehce složitější úkol a tím je vyhledání všech uzlů, kterými paket projde, než dorazí k cíli, zjistíme, že zde je již řešení prováděno malým trikem. Jsou odesílány pakety do cílové stanice s nastavenou dobou života[8], která se postupně zvyšuje od jedné až dojde k dosažení cíle. Vidíme, že se jedná o násilné použití dvou vlastností IP protokolu, aby nebylo nutné definovat další standard pro takovouto aplikaci.*

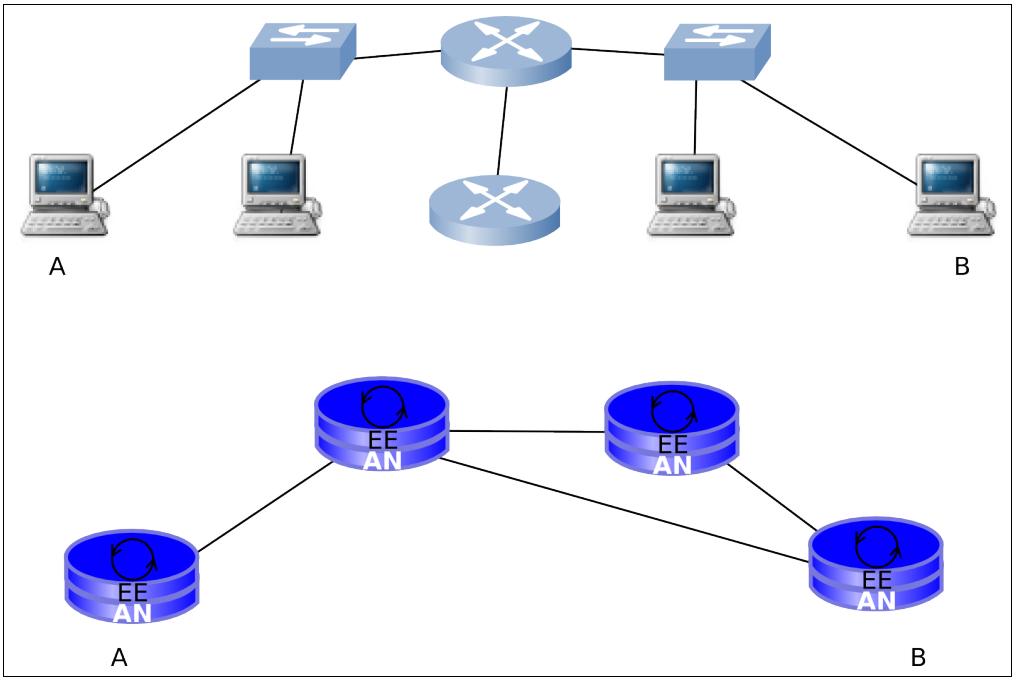
V případě aktivní sítě je situace výrazně odlišná. Uzel A vlastní aplikaci Ping a vloží do sítě kapsuli obsahující identifikaci cílového uzlu a identifikaci aplikace. Aplikace v uzlu A čeká, než se vrátí kapsule s nákladem, který posléze dekóduje. Jakmile je kapsule vložena do

4 tzv. sandbox
5 Požadavek na odezvu
6 Nemají, ale nikdo jim nebrání, aby nemohli
7 Odpověď na požadavek na odezvu
8 TTL – počet přeskoků, které může paket absolvovat – brání zacyklení paketu. Každý směrovač v paketu sníží TTL o 1 a v případě, že je TTL rovno nule, odešle zdrojovému uzlu informaci o nedostupnosti cíle

sítě a dorazí na první uzel, je přečten identifikátor aplikace. Je možné, že aplikace se na uzlu nenachází, a tak si uzel vyžádá od zdroje balík s aplikací. Jakmile je balík v uzlu přítomen je spuštěn identický kód jako na uzlu A a započne se s jeho vykonáváním. Aplikace ping obsahuje pouze jednoduchý test, zda se nachází na cílovém uzlu a pokud ne, ponechá se vložená v síti, dokud nedosáhne cílového uzlu. Na cílovém uzlu aplikace změní cíl kapsle na zdrojový uzel a ukončí se. Po přijetí ve zdrojovém uzlu se kapsule zničí a aplikace skončí.

Aplikace pro vyhledávání průchozích uzlů je pouze drobnou modifikací aplikace Ping s tím, že každý uzel přidá svoji adresu do datové části kapsule. Kapsule má navíc tu výhodu, že může informace do zdrojového uzlu zasílat již v průběhu putování sítí a navíc může odesílat i další doplňkové informace, takže v rámci diagnostiky se dají sledovat i jiné ukazatele, např. aktuální stav směrovacích tabulek.

Všimněte si, že kromě jisté standardizace v ohledu EE není nutné definovat žádný standardní komunikační protokol a aplikaci můžete použít okamžitě, jakmile ji naprogramujete a zavedete na zdrojový uzel. Navíc každý aktivní uzel může mít více než jedno EE, čímž se dá síť velmi dobře škálovat.



Ilustrace 1: Rozdíl mezi klasickou a aktivní sítí

Dalším důvodem, proč má smysl zavádět aktivní síť je ten důvod, že podle [TW96] se v aktivní síti spouštějí identické programy, kdežto v klasické síti, jak ji známe dnes se implementují dva různé programy podle stejného standardu, což vede k daleko větší

náchylnosti k chybám[9] a následné nekompatibilitě. V souladu s trendy[10] v softwarovém inženýrství víme, že je vhodné stavět každý systém tak, aby byl připraven na změny. IP protokol a klasické sítě jsou statické a snaží se všechny problémy řešit předem[11] a jsou dalším změnám víceméně uzavřené. V kontrastu s tím jsou právě aktivní sítě, kde je jednoduchá specifikace rozhraní EE, tedy toho, co aplikace může a nemůže požadovat a provádět. Jakoukoliv funkčnost si pak aplikace může obstarat sama a nemusí se spoléhat na to, zda protokol na kterém komunikuje tuto možnost podporuje.

1.2 Problémy v aktivních sítích

V aktivních sítích se z principu funkce řeší některé jiné problémy, než v sítích klasických. Zejména se aktivně řeší bezpečnost vykonávaného kódu, jeho náročnost na procesorový čas a s tím související plánování spuštěných aplikací. Dále jsou zde řešeny problémy s distribucí aplikací a výkonností navržených řešení.

1.3 Cíl práce

Cílem této práce bylo nalezení způsobu a demonstrace jeho funkčnosti pro provoz nezměněných aplikací využívajících protokol IP na uzlech aktivní sítě SAN. Výzkum v této oblasti byl zaměřen především na obecný koncept a posléze na konkrétní studii na operačním systému Linux, jeho paketovém filtru a protokolovém zásobníku a aktivním uzlu SAN.

9 Do dnešního dne se například opravují chyby v protokolovém zásobníku snad všech operačních systémů
10 Agilní metodiky vývoje software, změnami řízený vývoj aplikací
11 Tento přístup nápadně připomíná vodopádový model vývoje software

2 Teoretická část

2.1 Související práce

Níže uvádím souhrn existujících implementací aktivních sítí a jejich případné řešení problému provozu IP aplikací, které jsem prostudoval, případně jiné problémy vztahující se k problému aktivních sítí. V přehledu nejsou zahrnuta řešení uvedená v [RM08]. Z níže uvedeného seznamu je zřejmé, že do současné doby byly řešeny především jiné aspekty aktivních sítí a provoz aplikací pro IP nebyl zpravidla řešen vůbec nebo pouze okrajově, případně byl řešen problém opačný – jak provozovat aktivní síť nad IP.

2.1.1 Projekt PANDA

Projekt PANDA[12] je projekt naprogramovaný v jazyce Java a postavený nad aktivním serverem ANTS, který byl založen roku 1998 na kalifornské univerzitě a po dobu čtyř let byl financován agenturou DARPA. Důležitou součástí projektu PANDA je právě přenos dat mezi neupravenými aplikacemi prostředím aktivní sítě. V pracovním dokumentu [RP98] je navrhnuta rámcová struktura uzlu, jeho rozdělení na funkční celky a způsob, kterým budou získávána data z aplikace pomocí komponenty PIC[13]. Výsledkem práce bylo demonstrace adaptivního překódování UDP video proudu podle stavu komunikační linky ke klientovi.

Většina výzkumu byla ovšem směrována mimo oblast tunelování aplikací a nebyly o ní publikovány žádné bližší informace. Publikované informace se týkají zejména zabezpečení, plánování aplikací a spolehlivostního modelování.

[FV+02] uvádí některé bližší informace o implementaci PIC a celého projektu. Z důležitých informací vyberme, že PIC zpracovává výhradně UDP a je implementována jako jaderný modul a pakety odesílá přímo do aplikace ke zpracování přes UDP soket. Informace o cílové adrese přenáší separátním kanálem.

2.1.2 Aktivní uzel AIPv6

Na Massachusetts Institute of Technology probíhal koncem devadesátých let vývoj uzlu aktivní sítě ANTS. V diplomové práci [MD97] autor rozebírá možnosti aktivních uzlů v Internetu. Zejména se věnuje možnosti využití IPv6 jako aktivního protokolu a konstrukci uzlu, který bude akceptovat jak běžné IPv6, tak i modifikované AIPv6. Cílem práce bylo

12 Peer Agent Negotiation and Deployment of Adapters - http://www.lasr.cs.ucla.edu/panda/
13 Panda Interception Component

ukázat, že je možné vytvořit pomocí IPv6 kapsuli takovou, že ji rozpozná AIPv6 uzel a zpracování na běžném stroji nezpůsobí žádný chybový stav. Dále autor provedl rozdílová měření mezi průchodem běžných paketů IPv6 a AIPv6, kde se nad AIPv6 spouštěla prázdná Java metoda. Jeho práce ukázala, že AIPv6 je nejméně o řád pomalejší, než IPv6.

2.1.3 Projekt Atropos

V roce 1998 započal vývoj projektu aktivních sítí ve firmě GE[14] pod názvem AVNMP. Od počátku byl tento projekt zaměřen na predikci zátěže uzlů aktivní aplikací a simulaci predikce zátěže uzlů pomocí Kolmogorovy složitosti. Provoz stávajících aplikací nebyl podle publikovaných materiálů řešen vůbec. Aktivní síť je vytvářena překrytím nad existující sítí distribuovaných uzlů a ke spouštění aplikací využívá vlastní interpret Magician. Více o projektu hovoří např. [BS02], kde se hovoří i o obecném konceptu aktivních sítí.

2.1.4 Experiment na univerzitě Tianjin

Na čínské univerzitě v Tianjin se s ohledem na současný stav výzkumu rozhodli provést experiment s přenosem IP přes aktivní síť a výsledky shrnují v článku ve sborníku IEEE [ZJ+04] - Podpora tradičních IP aplikací v aktivních sítích. Výzkum byl zaměřen na implementaci prioritní fronty v EE uzlu a byla provedena měření z hlediska výkonnosti uzlů. Experiment byl založen na operačním systému Linux s ASP EE. Vlastní podpora IP aplikací byla prováděna zachytáváním přes knihovnu libipq vlastním modulem jádra aks. Bohužel publikace neuvádí žádné výsledky měření, pouze konstatování, že bez použití prioritní fronty je dvěma datovým tokům přenosové pásmo rozděleno přibližně rovnoměrně a při použití prioritní fronty získá priorizovaný tok získá širší pásmo.

2.1.5 Google Native Client

Google Native Client je open-source výzkum započatý v červnu 2009 zabývající se technologií vykonávání x86 strojového kódu ve webových aplikacích s cílem zachování nezávislosti na prohlížeči, přenositelnosti a bezpečnosti, která se od webových aplikací očekává. Projekt je ve fázi raného vývoje, ale již nyní ukazuje některé silné stránky této technologie. Přestože se nejedná o projekt aktivních sítí, je jím velice blízký, protože výsledky tohoto výzkumu by mohly být použity pro další výzkum v oblasti EE aktivních sítí tak, aby aplikace mohly být napsány například v nativním kódu stroje. Podle dosud prezentovaných výsledků výzkumu je výkonnost tohoto řešení velice blízká přímému spuštění

14 General Electric - http://www.research.ge.com/~bushsf/AVNMP.html

na stroji. Více o implementaci a měřeních výkonu zmiňuje [YB+09].

2.1.6 Aktivní sítě pro podporu mobilních zařízení

Aktivní sítě mohou být také použity pro podporu rozlišeného obsahu pro jednotlivé typy mobilních zařízení. V současné době, pokud je třeba nainstalovat aplikaci na mobilní zařízení, je třeba stáhnout z instalačního serveru balík určený pro dané zařízení, neboť se mobilní zařízení často velmi liší. V [LS05] se autoři zabývají problémem, jak s využitím aktivní sítě a adaptační komponenty dodávat rozlišené J2ME[15] aplikace mobilním zařízením bez nutnosti ručního výběru balíků.

2.2 Smart Active Node

Projekt navazující na experimentální implementaci aktivních sítí Grade32 jako univerzální uzel aktivních sítí. Architektura a základní principy funkce uzlu jsou shrnuty v [RM08]. Jedná se o server napsaný v jazyce Java, který měl za cíl vytvořit univerzálně použitelný server aktivní sítě. Server je možné spustit na libovolné platformě, kde existuje Java Virtual Machine. Samotný server využívá jako jazyk pro psaní aplikací také jazyk Java, který je přímo interpretován z byte kódu – tedy z přeloženého kódu. Mezi klíčové vlastnosti patří vlastní interpret jazyka Java schopný interpretovat vše až do úrovně nativních metod ([SJ09]), distribuce kódu aktivních aplikací ([SP09]) a automatické propojování uzlů na definovaných rozhraních ([RM08]).

V současné době se také intenzivně pracuje na vývoji uzlu SAN v jazyce C++, což by mělo odstranit některé nevhodné vlastnosti současné implementace.

2.3 Architektura konceptu

Tunelováním datových přenosů se rozumí takový přenos dat aktivní sítí, že je pro tunelovaná data transparentní – zdrojová ani cílová aplikace nesmějí být dotčeny manipulací s daty.

Tunelování aplikací využívajících protokol IP je z podstaty platformě závislá činnost, neboť se pohybujeme na nízké úrovni operačního systému a manipulujeme s daty, která jsou obsluhována jádrem operačního systému. Je tedy zřejmé, že implementace některých částí bude různá podle operačního systému.

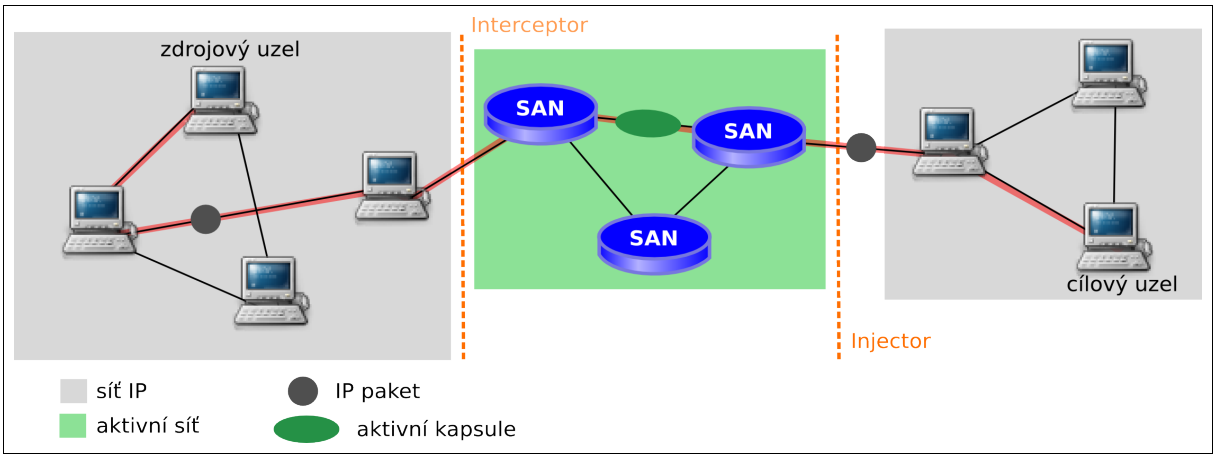
Každá aplikace, která musí zajistit tunelování musí mít nejméně tři funkční celky. Jeden

15 Java Micro Edition – edice pro mobilní zařízení

na zdrojovém uzlu pakety zachycuje a znemožňuje jejich další zpracování operačním systémem a druhou komponentu na cílovém uzlu, která naopak paket vytvoří a vloží jej do síťového zásobníku operačního systému. Třetí komponenta musí zprostředkovat předání dat mezi oběma odloučenými komponentami. V souladu s projektem PANDA, aby nebyl zbytečně zaváděn nový pojem pojmenujeme první komponentu jako interceptor a druhou jako injector. Třetí komponentu budeme nazývat interceptor-injector nebo zkráceně ii. Zatímco injector a interceptor jsou platformě závislé, ii již běží přímo v SAN, tudíž zůstává univerzální.

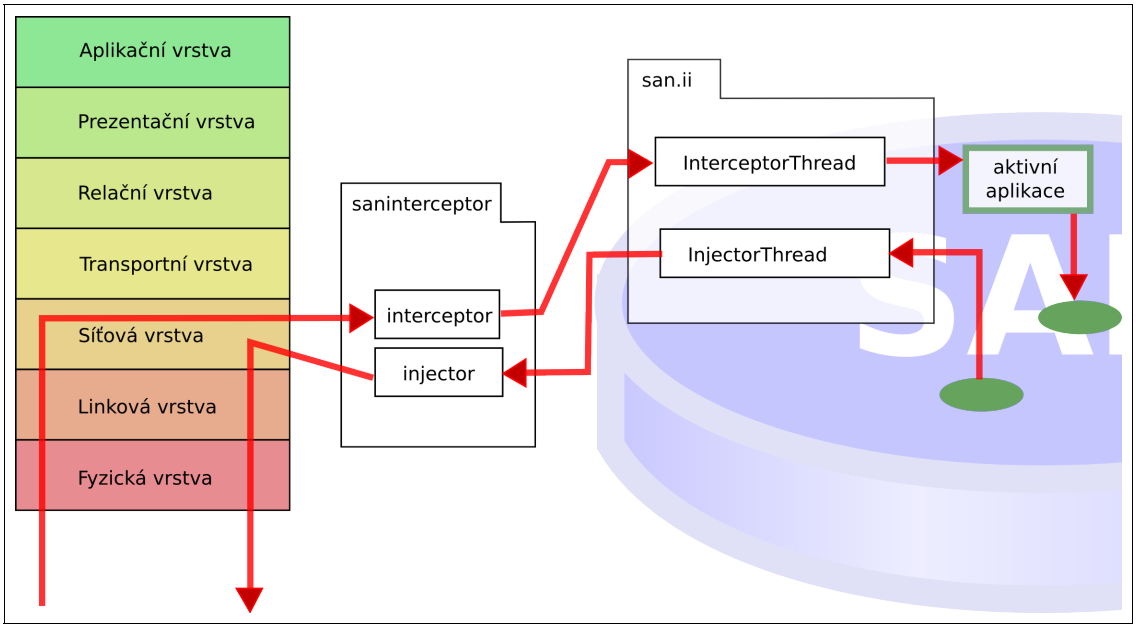
Vzhledem k tomu, že zpravidla potřebujeme zajistit obousměrné spojení uzlů, dá se předpokládat, že řešení musí být taktéž symetrické. Proto na vstupu do aktivní sítě musí být interceptor i injector, stejně jako na výstupním uzlu aktivní sítě. Pro zjednodušení a zpřehlednění ilustrací budeme uvažovat vždy pouze jeden směr toku dat s tím, že opačný tok je analogicky symetrický.

Ilustrace 2 ukazuje základní rozmístění komponent v systému a zapojení sítí pro lepší představu o problematice. Data byla vyslána ze zdrojového uzlu v klasické síti, putovala klasickou sítí k bráně sítě, která je propojena s hraničním uzlem aktivní sítě. Komponenta Interceptor přeruší cestu paketu, předá ho aktivnímu uzlu. Aktivní uzel zajistí předání paketu v kapsli do aktivní sítě. Na uzlu aktivní sítě, který je propojen s cílovou sítí dojde k předání paketu komponentě Injector, která zajistí odeslání paketu do sítě cílového uzlu.



Ilustrace 2: Schéma zapojení sítě

Ilustrace 3 sice poněkud předbíhá, ale pomůže lépe pochopit situaci. Ukazuje detailní pohled na hraniční uzel a přehledně ukazuje rozmístění komponent. Z ilustrace je patrné, které komponenty jsou svoji povahou v uzlu SAN a které mimo něj v operačním systému. Ilustrace dále ukazuje vazbu celého řešení na protokolový zásobník. Níže rozebereme funkčnost jednotlivých navržených komponent.

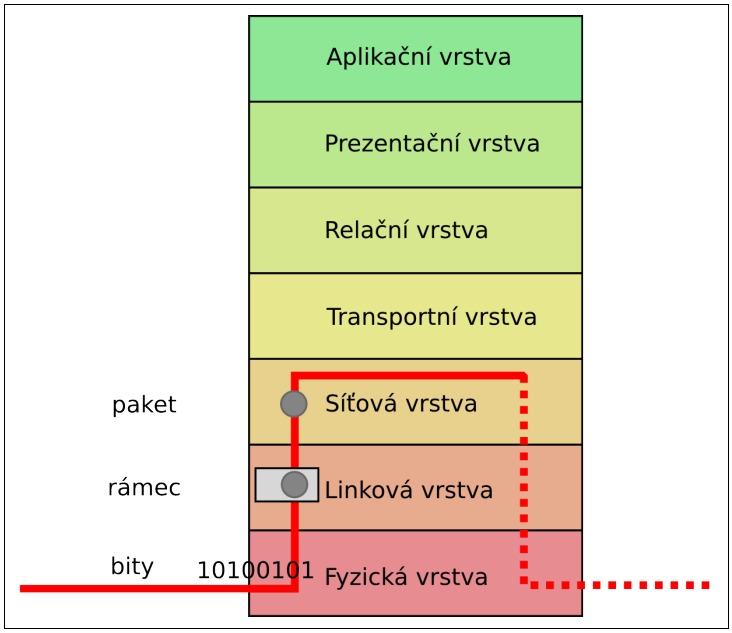


Ilustrace 3: Detailní pohled na rozmístění komponent v systému

2.4 Interceptor

Interceptor je komponenta zodpovědná za přerušení průchodu paketu síťovým zásobníkem. Průchod paketu je třeba přerušit dříve, než jej zpracuje systém, neboť ten by paket zahodil a vyslal by na zdrojový uzel ICMP[16] zprávu o nedostupnosti cílové stanice.

2.4.1 ISO/OSI model síťového zásobníku



Ilustrace 4: ISO/OSI model

Na ilustraci 4 je znázorněn ISO/OSI model[17] síťového zásobníku, který představuje

16 ICMP – Internet Control Message Protocol
17 International Organization for Standardization = Open Systems Interconnection Reference Model

způsob, jakým je možné dělit síťový zásobník na jednotlivé vrstvy podle funkčnosti a jednotek, se kterými pracuje. V tomto modelu narůstá srozumitelnost dat odspoda nahoru. V našem případě platí, že pokud by byl hraniční uzel aktivní sítě pouhým směrovačem, paket by prošel nejméně třemi spodními vrstvami zásobníku a po rozhodnutí o směrování by se opět přes tyto vrstvy vrátil do sítě výstupním rozhraním směrovače po trase naznačené tečkováním. K tomu by ale došlo pouze za předpokladu, že uzel zná cestu do cílové sítě. Podle Ilustrace 2 víme, že tomu tak není, neboť s ní není propojen. Máme tedy v zásadě dvě možnosti, jak zachytit paket, pomineme-li možnost napsání vlastního ovladače hardwaru pro dekódování jednotlivých bitů.

V [TN+02] a [DD+98] byl prováděn průzkum v oblasti hardwarové akcelerace aktivních sítí za účelem dosažení propustnosti srovnatelné se stávajícími směrovači.

2.4.2 Zachycení dat na linkové vrstvě

Zachycením dat na linkové vrstvě získáme rámec závislý na technologii, na které je síť postavena, např. rámec Ethernet nebo FDDI. Pokud bychom takto zachycený rámec chtěli použít, museli bychom předpokládat, že na straně injectoru bude použita stejná technologie a nebo bychom museli pro získání IP paketu znát strukturu rámce pro každý typ technologie. Z tohoto důvodu na linkové vrstvě zachytávat nebudeme. Dalším problémem zůstává fakt, že v této vrstvě nemůžeme data příliš filtrovat, protože by to víceméně znamenalo napsat jednoduchý paketový filtr s konfigurací, abychom mohli určovat, která data se budou smět přes aktivní síť tunelovat a která ne.

2.4.3 Zachycení dat na síťové vrstvě

Pokud zachytíme data v úrovni síťové vrstvy, získáme pouze IP paket včetně hlaviček, ale není zatížen informacemi o přenosové technologii. Navíc v každém moderním operačním systému existuje firewall[18] pracující právě v této vrstvě, kterým můžeme data před předávkou do aktivní sítě filtrovat. Zachycený paket tedy můžeme bez větších problémů vložit v jiném místě sítě, aniž by bylo nutné ho jakkoliv modifikovat.

Protože zachycování paketů na síťové vrstvě je závislé na platformě, budu se dále věnovat pouze případové studii na operačním systému Linux.

2.4.4 Zpracování paketů v operačním systému Linux

Níže popíši, jak se síťovými daty zachází operační systém než se dostanou do cílové

18 Paketový filtr sloužící k ochraně počítače před nežádoucí komunikací

aplikace. Tento popis je nezbytný pro pochopení funkce operačního systému a následnou úvahu o řešení zachytávání paketů.

Úplně na začátku cesty dorazí data po fyzickém médiu do síťové karty. Pokud fyzická adresa síťové karty je ve shodě s adresou v přijatém rámci nebo se jedná o záplavové vysílání nebo je karta přepnuta do promiskuitního[19] režimu, je vyvoláno přerušení[20]. Ovladač síťové karty toto přerušení zpracuje a nějakým[21] způsobem uloží data do operační paměti. Poté pro tato data alokuje strukturu skb – socket buffer.

Skb je struktura obsahující data a dodatečné systémové informace o paketu, které jsou uloženy mimo něj a tudíž platné pouze pro daný uzel[22]. Skb obsahuje odkaz na předcházející a následující skb a odkaz na první prvek seznamu. Jedná se tedy implementaci obousměrně zřetězeného seznamu. Každý skb může být v několika frontách a v několika seznamech. Skb obsahuje také časovou značku, vstupní a výstupní síťové rozhraní.

Jakmile je alokován skb pro příchozí data, je opatřen časovou značkou a v případě, že je přijímací fronta[23] plná, je rovnou zahozen. Pokud fronta plná není, je paket zařazen do fronty a je vyvoláno softwarové přerušení – zde se již nejedná o přerušení na úrovni hardwaru, ale o softwarový aparát poskytující asynchronní zpracování událostí v rámci systému i na víceprocesorových stanicích[24]. Softwarové přerušení je zachyceno na úrovni jádra síťových operací[25] a paket předán ke zpracování paketovému modulu zodpovědnému za daný typ paketu. IPv4 paket je tedy předán ke zpracování modulu IPv4[26]. Modul provede základní testy paketu, jako je CRC[27] a dále jej zpracovává podle cíle IP paketu. Cílem může být lokální stanice nebo vzdálená stanice. Podle typu cíle paketový modul volá ke zpracování skb jednotlivé funkce. Na každou tuto funkci je v závěru jejího zpracování pověšen háček[28] projektu Netfilter.

Po prvotním testu správnosti IP paketu je provedeno rozhodnutí o směrování paketu a následně je paket doručen buď lokálně nebo (pokud je to dovoleno) předán do odchozí fronty.

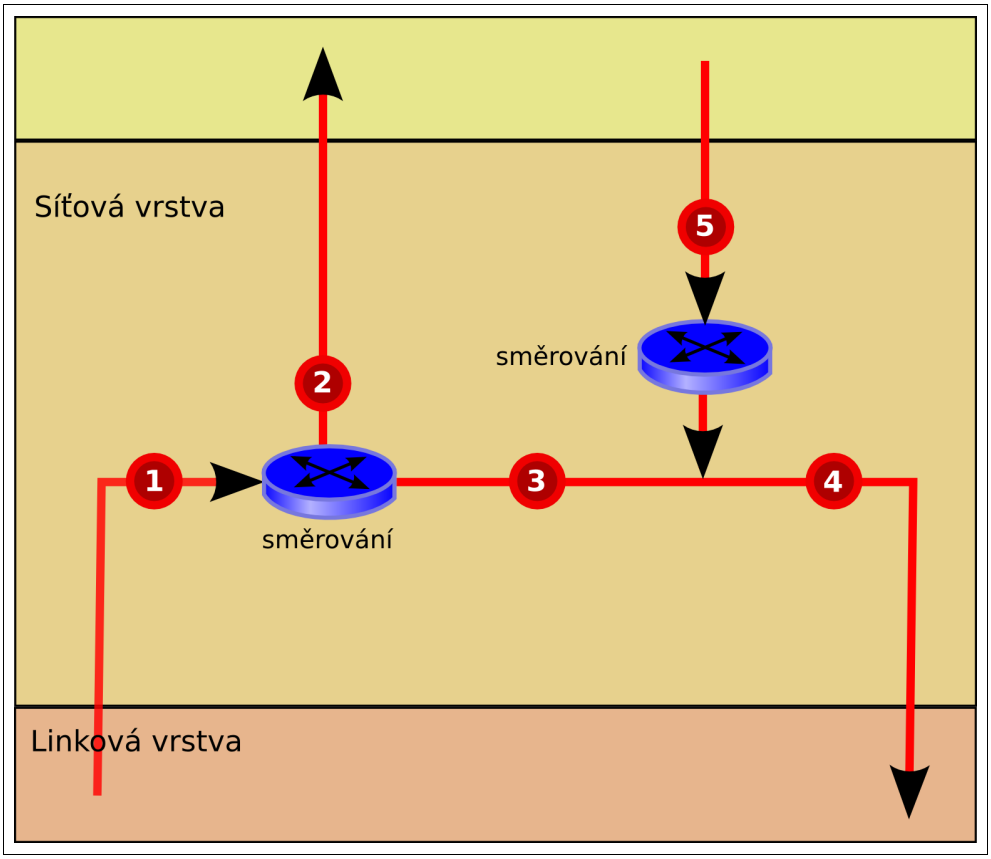
Zde vidíme, že pokud bychom chtěli s paketem manipulovat dříve, než pomocí projektu Netfilter, museli bychom upravovat zdrojový kód jádra, což je ve většině případů velmi nežádoucí akce.

19 Režim, kdy karta zpracovává všechny přijaté rámce bez ohledu na to, zda má být jejich příjemcem
20 Signál vyslaný procesoru, aby asynchronně zpracoval událost
21 Nejčastěji DMA – přímým přístupem do paměti
22 Příkladem budiž značka – mark – pomocí které lze například pakety třídit do front – více např. v [SJ07]
23 Více o frontách v kapitole 2.4.7
24 Tradiční přerušení zastaví všechny procesory a může jej obsloužit pouze jeden procesor
25 net/core/dev.c
26 net/ipv4
27 Cyklická redundantní kontrola
28 Programová konstrukce volající v daném místě jiný modul ke zpracování paketu

2.4.5 Netfilter a iptables

Projekt Netfilter je moduly rozšiřitelný paketový filtr v systému Linux, který obsahuje háčky pro zpracování nejen pro protokol IP, ale i některé další méně obvyklé protokoly a tvoří tak univerzální systém pro správu paketů na síťové úrovni. Iptables jsou pak jeho uživatelská nadstavba (aplikace) pro snadnou manipulaci s pravidly v jednotlivých místech systémových háčků. Vychází z principu, že nad každým háčkem je vytvořena tabulka pravidel, která musí v konečném důsledku buď paket akceptovat nebo jej zahodit.

Háčky projektu Netfilter, jak je uvádí [RRWH02] nebo přímo i hlavičkový soubor netfilter_ipv4.h, jsou zobrazeny na ilustraci 5 jako kroužky na možné dráze paketu. Při příchodu paketu z linkové vrstvy (tedy z jiného stroje) je jako první spuštěn háček NF_IP_PRE_ROUTING (označen 1). Po směrovacím rozhodnutí, zda paket patří této stanici nebo zda bude poslán sítí dále následují háčky 2 a 3 resp. NF_IP_LOCAL_IN, NF_IP_FORWARD. Těsně před odesláním paketu zpět do linkové vrstvy je volán háček 4 NF_IP_POST_ROUTING. Pro pakety vytvořené aplikací přímo na stroji je pak volán háček 5 NF_IP_LOCAL_OUT.



Ilustrace 5: Systémové háčky projektu Netfilter – detail síťové vrstvy

Netfilter tyto háčky dále zpřístupňuje ve svém API[29] dalším svým modulům. Nabízí se tedy možnost využít API Netfilteru pro zachytávání paketu, abychom nemuseli zasahovat do funkce jádra. Pokud dále důkladně prozkoumáme projekt Netfilter, zjistíme, že část tohoto projektu nazvaná libnetfilter_queue[30] poskytuje pro nás velmi zajímavou službu. Umožňuje totiž předávání celých paketů do prostoru uživatelských aplikací – tedy zcela mimo prostor jádra – a zároveň sdělit zpět Netfilteru, co se má s tímto paketem stát v rámci jádra.

Libnetfilter_queue nahrazuje původní implementaci libipq, která pro celý systém dovolovala pouze jednu frontu pro zpracování paketů. Pro jednoduchost si systém představme tak, že pomocí iptables zapíšeme pravidlo s cílem QUEUE a paket je v ten moment přesunut do fronty a čeká, až si jej aplikace voláním knihovny libipq vyzvedne. Libnetfilter_queue se liší od libipq tím, že umožňuje k cíli NFQUEUE v pravidlu iptables přidat číslo fronty, do které má být paket zařazen. Výsledkem je tedy fakt, že můžeme mít několik aplikací, které mohou pracovat současně, a každá může pracovat s jinou množinou paketů.

2.4.6 libnetfilter_queue

Práce s knihovnou netfilter_queue je poměrně přímočará. V první řadě je třeba zavést do iptables pravidlo, které obstará doručení paketů do fronty. Jako příklad uveďme výpis v Kód 1, který zaručí, že všechny ICMP pakety přicházející po rozhraní eth0 směřující na jiný

```
iptables -A FORWARD -i eth0 -p icmp -j NFQUEUE --queue-num 2
```

Kód 1: Příklad použití cíle NFQUEUE

stroj, budou zařazeny do fronty číslo 2. Toto pravidlo okamžitě způsobí, že pakety přestanou v tomto místě procházet zásobníkem a do jisté míry se zatím chová jako cíl DROP, protože na frontu není připojena klientská aplikace a výchozí verdikt pro pakety je v tomto případě zahození.

Ve zdrojovém kódu obslužného programu je pak třeba vložit hlavičkový soubor knihovny a provést stručnou inicializaci fronty pomocí knihovních funkcí. Výsledkem inicializace je deskriptor, ze kterého můžeme číst funkcí read jednotlivé zachycené pakety – celý proces se velmi podobá otevírání běžného BSD soketu. Důležitou součástí programu je vydání verdiktu nad paketem. Máme v zásadě možnosti zahodit paket (okamžitě uvolní alokovanou paměť pro paket), přijmout a ponechat jeho zpracování systému (což je vhodné pokud chceme paket pouze pozměnit), ukrást paket (zůstane v paměti, ale nebude dál

29 Application Programming Interface - rozhraní pro komunikaci s aplikací
30 http://www.netfilter.org/projects/libnetfilter_queue/index.html

zpracováván systémem) nebo vydat verdikt o pokračování průchodu paketu skrze iptables.

2.4.7 Fronty paketů

V kapitole 2.4.4 byl prvně použit termín fronta paketů. Pokusím se nyní vysvětlit, o jaký koncept se jedná a jakou službu může tato vlastnost jádra Linuxu provést.

Je zřejmé, že zpracování paketů v operačním systému musí mít nějaký řád a není možné pakety například odkládat na haldu a vybírat je v náhodném pořadí[31]. Proto existují v operačním systému fronty[32], do kterých je paket zařazen při příjmu z rozhraní a před odesláním do rozhraní. Linux k tomuto přistupuje velmi otevřeně a umožňuje dokonce napsat vlastní metodu fronty pro případ, že by nevyhovovala prioritní fronta vestavěná v jádře[33]. Obecně se vlastní metody používají k řízení a ořezávání datového toku, případně jeho rovnoměrné rozložení mezi uzly atp a pracuje přímo se strukturou skb a očekává se, že fronta určí pořadí zpracování paketů podle nějakého pravidla, které může být i třeba velmi složité[34] a umožňuje jádru sdělit, zda je zaplněna, či nikoliv. Vlastní metodu fronty je možné implementovat jako modul jádra zaregistrováním pomocí systémového volání register_qdisc a unregister_qdisc.

Důvod, proč se o frontách zmiňuji je ten, že v případě potřeby velmi vysokého výkonu aplikace je možnost napsat vlastní frontu tak, že by přijatý skb přímo předávala aktivnímu uzlu. Jednalo by se ale o zpracování na linkové vrstvě, které, zdá se, prozatím není vhodným kandidátem a nebylo dále rozpracováno.

2.4.8 Soket domény AF_PACKET

Nejjednodušší variantou zachycení paketů je otevření soketu typu AF_PACKET, který využívají[35] takové projekty jako tcpdump[36] nebo wireshark[37] a slouží k zachytávání paketů na linkové úrovni – tedy získání datového rámce. Nevýhodou tohoto přístupu je fakt, že pracujeme pouze s kopií paketu (kopií skb) a nemáme žádnou možnost jej pozměnit nebo ovlivnit jeho průchod síťovým zásobníkem. Tento přístup je tedy pro nás nepoužitelný.

31 Obecně žádný řád být nemusí, ale je to žádoucí s hlediska determinismu systému
32 Queuing discipline
33 Priorita je určována podle pole ToS v paketu a všechny fronty s vyšší prioritou musejí být zcela vyprázdněny než se z nich začne vybírat paket
34 Např. HTB nebo HFSC - více v [SJ07]
35 resp. využívají knihovnu pcap, která ovšem používá tento soket
36 Program *pro zachytávání a analýzu paketů – http://www.**tcpdump**.org/*
37 Grafický program pro hlubší analýzu zachycených dat a toků – http://www.wireshark.org/

2.4.9 Shrnutí

Pomocí projektu paketového filtru Netfilter s nadstavbou iptables a za pomoci knihovny netfilter_queue jsme schopni efektivně vytvářet pravidla, kterými můžeme vybírat data k tunelování skrze aktivní síť. Pro změnu těchto pravidel nebude třeba vůbec upravovat komponentu interceptor, která bude fungovat pouze tak, že akceptuje data zařazená do fronty cílem NFQUEUE.

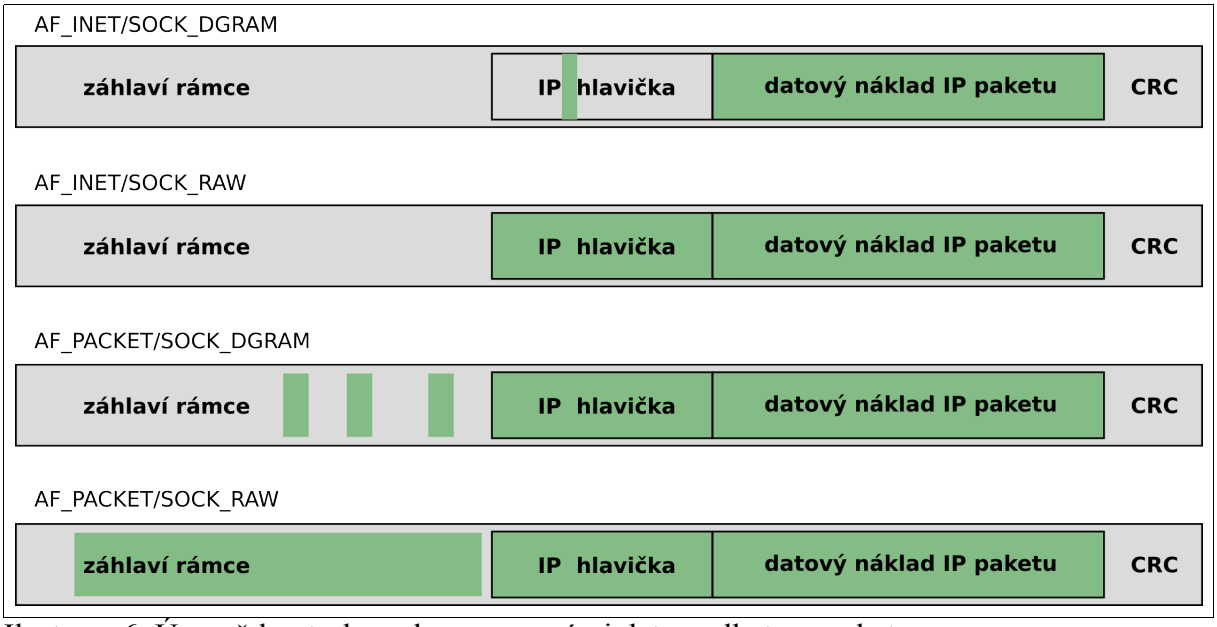
V případě potřeby zvýšení výkonu připadá v úvahu napsat buď vlastní modul pro projekt Netfilter nebo vlastní modul jádra s implementací fronty.

2.5 Injector

Vycházíme z předpokladu, že máme dostupný celý IP paket zachycený komponentou Interceptor. Takovýto paket obsahuje nezměněné hlavičky i datový náklad. Máme dvě možnosti, jak můžeme paket vložit do sítě a to na síťové vrstvě nebo na linkové vrstvě. Rozebereme níže obě možnosti a porovnáme je.

2.5.1 Vkládání paketů na úrovni síťové vrstvy

Vkládání paketů na úrovni síťové vrstvy je možné pomocí klasických konstrukcí nad BSD sokety. Protože nutně potřebujeme vytvářet pakety s vlastním obsahem hlavičky, nevystačíme s běžnými sokety typu SOCK_DGRAM[38] a SOCK_STREAM[39]. V tento okamžik musíme vytvářet soket typu SOCK_RAW, jak je uvedeno v Kód 2. Doména AF_INET určuje, že bude vytvořen IPv4 paket, typ soketu SOCK_RAW nám umožní vytvořit vlastní hlavičku nebo modifikovat hlavičku vygenerovanou systémem a IPPROTO_RAW je



Ilustrace 6: Úroveň kontroly nad generovanými daty podle typu soketu

uvedena jako číslo protokolu[40], který nese IP paket. Na ilustraci 6 je zelenou barvou znázorněno, s jakými daty uvnitř rámce můžeme pracovat a naopak světle šedou, která modifikovat vůbec nemůžeme v případě použití jednotlivých typů soketů. Z ilustrace je jasně vidět, že nevystačíme s běžným soketem typu DGRAM, neboť většina hlavičky paketu je generována bez možnosti našeho zásahu. Pokud bychom v budoucnu uvažovali o tunelování i jiných protokolů než IP, pomocí RAW soketu AF_INET bychom to provést nemohli, neboť nemůžeme změnit identifikátor EtherType určující typ přenášeného protokolu. Ovšem pro přenos nezměněných IP paketů je RAW soket zcela dostačující a poskytuje nám programátorský komfort. V neposlední řadě bude takto napsaná aplikace schopna běhu v libovolném prostředí, které implementuje BSD sokety.

```
#include <sys/types.h>
#include <sys/socket.h>
...
s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

Kód 2: příklad použití RAW soketu

RAW soket se poté chová podobně jako soket pro UDP a k odesílání dat se využívá funkce sendto(). Funkce sendto() vyžaduje uvést příjemce, kterého si musíme zjistit z hlavičky paketu. To není žádný problém, neboť ten se dle RFC 791 nachází na offsetu 16 Bytů a má délku 4 Byty. Nevýhodou tohoto řešení je, že vkládáme paket na třetí úrovni a je třeba dbát opatrnosti na nastavení paketového filtru, aby nedošlo k zacyklení. Tomu se dá vyhnout například předpokladem, že cíle s frontou se budou zásadně využívat pro zvenčí vytvářená data a budou se tak nacházet pouze v řetězu FORWARD. Testování je lepší provádět pomocí několika oddělených strojů nebo alespoň virtuálních strojů.

2.5.2 Vkládání paketů na úrovni linkové vrstvy

Vzhledem k tomu, že pakety musejí projít skrze linkovou vrstvu, je možné je samozřejmě kromě zachytávání i na této vrstvě vkládat. Toto řešení má ovšem, stejně jako u zachytávání, tu nevýhodu, že se musíme zabývat technologií, kterou je zajištěn přenos po síti. Pro odeslání paketu v síti typu Ethernet musíme znát MAC[41] adresu příjemce. V tento okamžik je nutné využívat např. ARP[42] a v případě nedostupnosti cílové stanice v síti vysílat na uzel uvedený ve směrovací tabulce jako výchozí brána. Toto řešení jsem pokusně

40 RAW protokol samozřejmě neexistuje. Pokud jej ale uvedeme, nebudeme muset nastavovat parametr IP_HDRINCL, který říká soketu, zda má nebo nemá generovat hlavičku paketu.
41 Fyzická adresa zařízení příjemce
42 Address Resolution Protocol – protokol pro překlad IP adres na fyzické adresy

implementoval pomocí soketu AF_PACKET/SOCK_DGRAM pro zjištění přesného chování takto vytvořených paketů. S vytvářením takovéhoto injectoru je velmi pracné a nepřináší žádný užitek oproti RAW soketům, snad s výjimkou plné kontroly nad odesílanými daty. Vzhledem k tomu, že uvažujeme pouze tunelování protokolu IP, není tato kontrola pro nás žádným přínosem, nicméně pokusná implementace ukázala, že pro omezenou množinu technologií a paketů je možné tento soket použít.

2.5.3 Shrnutí

Vkládání paketů bude prováděno pomocí RAW soketu v síťové vrstvě. Pro vkládání na nižší vrstvě není reálný důvod a takové řešení by de facto suplovalo síťovou vrstvu (ARP dotazy, směrovací rozhodnutí, zjištění technologie rozhraní atp.).

2.6 Interceptor-Injector

Interceptor-Injector je univerzální část uzlu SAN, která komunikuje s procesem interceptoru a po přijetí paketu spustí v aktivním uzlu aplikaci, která má za úkol vytvořit kapsuli s datovým nákladem původního zachyceného paketu.

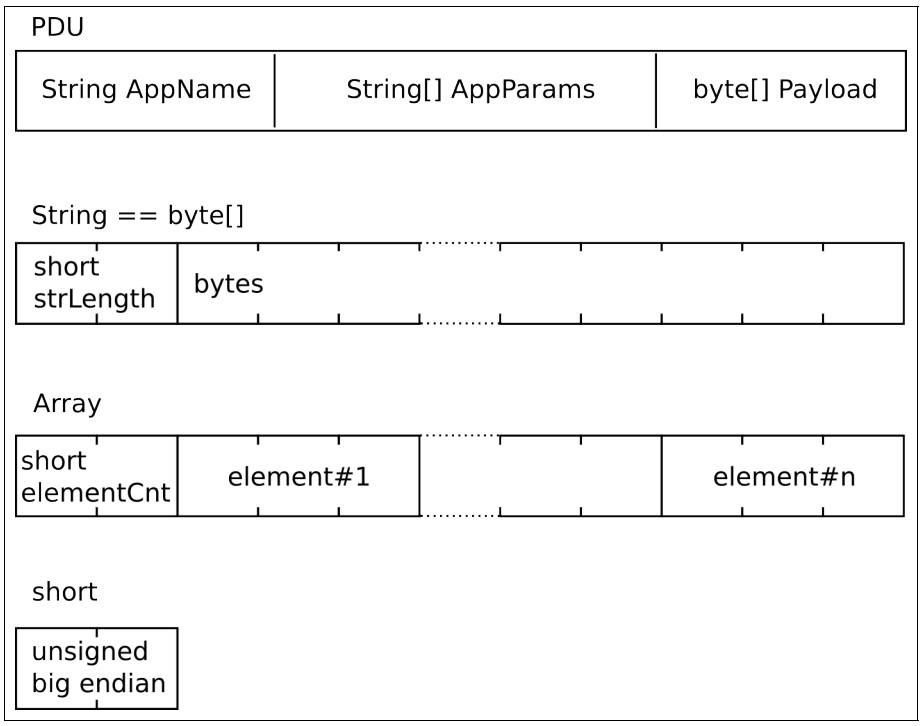
Původní myšlenkou bylo zřídit komponentu ii jako aktivní aplikaci v uzlu. Nicméně toto řešení, i když je možné, by neposkytovalo službu pro žádnou jinou aplikaci a bylo by silně jednoúčelové. Vzhledem k tomu, že dosud neexistovala možnost do uzlu SAN nějakým způsobem zavést externí data, zvolil jsem metodu univerzálního mostu pro předávání dat.

Interceptor-Injector je spuštěn během startu jádra[43] uzlu SAN a pro každý přijatý paket spustí aplikaci definovanou v PDU[44] přijatou pomocí IPC[45] od komponenty Interceptor.

2.6.1 Formát PDU SAN ii

Formát PDU pro SAN ii byl zvolen čistě pragmaticky jako jednoduchý způsob předávání stávájících paketů spolehlivým kanálem s doplňkovou informací tak, aby byl nezávislý na platformě a zároveň nebyl příliš složitý[46]. Aby se z paketu mohla stát kapsule, musíme bezpodmínečně znát alespoň název nebo identifikátor aktivní aplikace, která bude data zpracovávat. Dále je možné každé aktivní aplikaci v SAN předat pole řetězců jako parametry. Bylo by tedy vhodné umět přenést i takovouto informaci. A pak samozřejmě musíme přenést vlastní binární data.

43 san.core.Kernel
44 Protocol Data Unit – datová jednotka komunikačního protokolu
45 Inter Process Communication – meziprocesová komunikace
46 Jako například norma ASN.1



Ilustrace 7: PDU SAN ii a základní použitelné jednotky

PDU SAN ii je znázorněna na ilustraci 7. Základními jednotkami, které je možné přenést jsou řetězec (String), pole elementů (Array), a dvoubajtové číslo bez znaménka uložené v network byte order[47] (short). Takto navržený paket umožňuje předávat řetězce a pole o celkové délce až 65535 znaků, resp. elementů. Celková délka paketu je tedy proměnná a pohybuje se v rozmezí 6 Bytů (prázdný paket) do 4 GB (v mezním případě využití maximálního počtu parametrů s maximální možnou délkou). Datový náklad může mít nejvýše 65kB, což se i podle nejnovějších pokusů se super jumbo frames[48] jeví jako dostatečná kapacita.

PDU se skládá z názvu aktivní aplikace (název uvedený v Manifestu[49]), která bude spuštěna po přijetí dat, jejích parametrů, které jsou předány při spuštění a dat, která jsou do aplikace vložena přes rozhraní ReceiveDataListener.

2.6.2 Komunikace s ostatními procesy

Meziprocesová komunikace v jazyce Java neumožňuje efektivní komunikaci přes sdílenou paměť apod. a veškeré prostředky pro meziprocesovou komunikaci jsou implementovány pomocí soketů. Z tohoto důvodu je i komunikace mezi komponentou Interceptor (resp. Injector) a SAN ii realizována spojením přes soket, nad kterým je vytvořeno

47 Network byte order – pořadí bytů čísla definované pro protokol IP jako tzv. big-endian
48 Rámce obsahující více než 9kB a méně než 64kB dat
49 Soubor obsahující metadata JAR balíku

spolehlivé spojení (TCP). To na druhou stranu umožní obsluhovat více případných klientů standardním způsobem (využití vícevláknového serveru). Očekává se, že předávání dat nebude tak efektivní jako v případě použití sdílené paměti a že se může stát úzkým hrdlem navrženého systému. K využití sdílené paměti nebo jiného podobného principu pro meziprocesovou komunikaci by bylo nutné použít volání nativních metod a server by ztratil možnost snadné přenositelnosti.

3 Realizační část

3.1 Výběr programovacího jazyka a vývojových prostředků

Komponenty Interceptor a Injector budou napsány v jazyce C, neboť jsou relativně malé, málo komplexní a dají se z něj přímo provádět potřebná systémová volání a je vyžadována vysoká rychlost provádění. Pro sestavení aplikací je použita utilita make[50].

Komponenta SAN ii bude zapsána v jazyce Java, který byl v [RM08] vybrán pro implementaci serveru.

Vzhledem k neuspořádanosti stávajícího úložistě byly provedeny změny ve struktuře a byl vytvořen sestavovací skript pro ANT[51], aby bylo snadné sestavit celý projekt včetně aktivních aplikací.

Vzhledem k potřebě ladit současně aplikaci napsanou v jazyce C a Java jsem učinil dobrou zkušenost s vývojovým prostředím Netbeans[52]. Celý vývoj probíhal na operačním systému Ubuntu[53] Linux a některé testy byly prováděny pod operačním systémem Microsoft Windows XP.

K analýze síťových přenosů jsem využil program Wireshark[54], což je nástupce známého projektu Ethereal a poskytuje grafické rozhraní k analýze zachycených dat. Při psaní síťových aplikací je to nepostradatelná pomůcka.

3.2 Program saninterceptor

V kapitole 2.3 jsem zmínil nutnost symetrického spojení mezi uzly. Z tohoto důvodu vznikla aplikace saninterceptor v jazyce C, která zastřešuje komponenty Interceptor a Injector. Tato aplikace je implementována v souboru ii.c a hlavičkovém souboru ii.h. Obsahuje metodu main() pro spuštění. V hlavičkovém souboru jsou definována variadická[55] makra pro výpis chybových hlášení a ladicích výpisů. Tato makra se nazývají printd() a printe(). Volají se stejně jako funkce pro formátovaný výstup printf(). Rozdíl mezi makry a funkcí printf je v tom, že printd vypisuje pouze v případě, že je proměnná označující ladění vyhodnocena jako pravda a druhá vypisuje hlášení na standardní chybový výstup.

50 http://www.gnu.org/software/make/
51 http://ant.apache.org/
52 http://www.netbeans.org/
53 http://www.ubuntu.com/
54 http://www.wireshark.org/
55 makra obsahující elipsu (…) pro předem neznámý počet parametrů

Po spuštění aplikace přečte konfigurační soubor, pokusí se spojit se serverem SAN ii uvedeným v konfiguraci a pokud uspěje, spustí dvě vlákna. Jedno vlákno vykonává Interceptor a druhé Injector. Aplikace pak pomocí volání thread_join nad oběma vlákny čeká na jejich ukončení.

Aplikace vyžaduje parametr *-c* s cestou ke konfiguračnímu souboru. Dále je možné aplikaci přepínačem *-v* sdělit, že má provádět ladicí výpisy.

3.2.1 Sestavení

Pro přeložení a sestavení aplikace je v prvé řadě mít nainstalovánu knihovnu POSIX[56] vláken (pthread), knihovnu netfilter_queue a utilitu make. Obě jsou zpravidla dostupné v balících pro každou distribuci Linuxu. Pro sestavení je vytvořen Makefile. Aplikaci je možné sestavit ve dvou verzích – ladicí (Debug) a produkční (Release), které se odlišují nastavením překladače. Pokud chcete přeložit obě verze využijte cíl all – tedy make all. Pro vyčištění adresáře od přeložených souborů zvolte cíle clobber nebo clean, kde první z nich odstraní i vygenerované spustitelné soubory.

3.2.2 Konfigurace

Konfigurační soubor je velice jednoduchý a obsahuje pouze dvojice klíč=hodnota na samostatných řádcích. Klíče mohou nabývat následujících hodnot:

- host – IPv4 adresa uzlu, kde naslouchá SAN ii
- port – číslo portu, na kterém naslouchá SAN ii
- gatewaynode – identifikátor hraničního uzlu SAN připojeného ke vzdálené IP síti
- gatewaynetwork – identifikátor sítě hraničního uzlu SAN
- application – jméno aplikace, která se má spustit a které má být předán paket
- nfqueue – číslo fronty, ze které budeme vybírat pakety

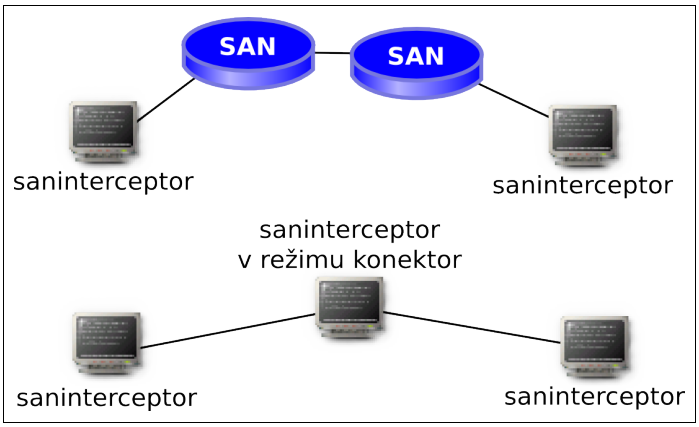
Všechny klíče jsou povinné, neboť jsou nutné ke korektnímu běhu aplikace. V případě, že kterýkoliv chybí, aplikace skončí chybovým hlášením a vypíše chybějící parametry.

Pakety, které bude program zachytávat se specifikují pomocí pravidel v paketovém filtru v souladu s kapitolou 2.4.6.

56 Portable Open System Interface (for UNIX) – standard aplikačního programovacího rozhraní

3.2.3 Spuštění

Program vyžaduje spuštění s přepínačem *-c* za kterým následuje cesta ke konfiguračnímu souboru. Pokud uvedete navíc přepínač *-v*, program bude vypisovat ladicí informace na konzolu. Před spuštěním aplikace musí být spuštěn SAN server, jinak spuštění skončí chybou. Pravidlo v iptables s cílem NFQUEUE může být zavedeno jak před spuštěním, tak i po spuštění programu.



Ilustrace 8: Standardní spuštění aplikace a spuštění v režimu konektor

Druhou možností spuštění určenou pro ladění aplikace a testování propustnosti a výkonnosti je spuštění s parametrem *-C*, které provede spuštění komponenty Connector a slouží pro propojení dvou instancí saninterceptoru dohromady bez využití SAN.ii. Ilustrace 8 zobrazuje rozdíl mezi standardním spuštěním s uzly SAN a spuštěním s konektorem.

3.2.4 Komponenta Interceptor

Největším problémem při prvních pokusech o vytvoření aplikace pro zachytávání paketů byl celkový nedostatek informací o knihovně libnetfilter_queue. Ve většině dostupných zdrojů je popisována práce se starší knihovnou libipq a zjistit tak vůbec, že existuje novější a lepší knihovna je velmi obtížné. Ke knihovně libnetfilter_queue do března tohoto roku v podstatě neexistovala oficiální dokumentace a vývojář byl nucen informace skládat z útržků na e-mailových konferencích. Zatímco některá volání jsou naprosto přímočará a není pochyb o jejich přesné funkci a parametrech, u jiných bylo třeba analyzovat zdrojový kód knihovny a provádět pokusy s funkčností. Současná verze knihovny již obsahuje dokumentaci ve zdrojových kódech pro použití v doxygen[57]. Bohužel dokumentaci samotnou není možné snadno přeložit a je nutné přeložit celou knihovnu, popřípadě ji vygenerovat ručně. Pro získání dokumentace je tedy nejprve třeba stáhnout archiv se

57 Nástroj podobný JavaDocu pro generování dokumentace ze zdrojových kódů

zdrojovým kódem (nejméně verze 0.17) z domovské stránky Netfilter a rozbalit jej na pevný disk. Poté nainstalovat program doxygen a spustit jej v adresáři src rozbaleného archivu s parametrem libnetfilter_queue.c . Doxygen vytvoří adresáře html a latex ve kterém se nachází přehledně zpracovaná dokumentace ke knihovně. Celý postup pomocí běžných nástrojů na systému Linux shrnuje úsek kódu Kód 3.

```
wget
http://www.netfilter.org/projects/libnetfilter_queue/files/libnetfilter_queue-0.0.17.tar.bz2

tar -xvf libnetfilter_queue-0.0.17.tar.bz2

cd libnetfilter_queue-0.0.17/src/

doxygen libnetfilter_queue-0.0.17
```

Kód 3: Postup získání dokumentace knihovny netfilter_queue

Komponenta je implementována ve zdrojovém souboru interceptor.c a hlavičkovém souboru interceptor.h. Vstupním bodem komponenty je interceptor_run(), která jako parametr přijme deskriptor soketu se spojením k SAN ii a je spuštěna při vytvoření vlákna.

```
h = nfq_open();

nfq_unbind_pf(h, AF_INET);

nfq_bind_pf(h, AF_INET);

qh = nfq_create_queue(h, nfqueue, &cb, NULL);

nfq_set_mode(qh, NFQNL_COPY_PACKET, 0xffff);

nh = nfq_nfnlh(h);

fd = nfnl_fd(nh);

while ((rv = recv(fd, buf, sizeof (buf), 0)) > 0) {
        nfq_handle_packet(h, buf, rv);
}
```

Kód 4: Osa systémových volání libnetfilter_queue

Ve výpisu kódu 4 je zachycena hlavní osa po které se ubírají systémová volání směřující k zachycení paketu uloženého ve frontě. Funkce nfq_open() získá programátorovi deskriptor pro práci s knihovnou a nastavení jejích parametrů. Pár funkcí nfq_bind_pf a nfq_unbind_pf slouží programátorovi k vyfiltrování potřebných paketů podle adresních rodin. V našem případě máme zájem pouze o IPv4 pakety, proto zvolíme rodinu AF_INET. Doporučuje se před vlastním nastavením toto zresetovat voláním funkce nfq_unbind_pf. Pomocí volání nfq_create_queue se pak přímo připojíme k některé z front a zároveň předáme odkaz na

callback[58] funkci, která bude zavolána, jakmile bude paket zpracován a dále je možné specifikovat odkaz na data, která budou této callback funkci předána při jejím volání. Tuto funkčnost nepotřebujeme, proto je v příkladu uveden ukazatel NULL.

Funkce nfq_set_mode umožňuje nastavit, jaká data z paketu budou kopírována do prostoru uživatelských programů. Na výběr je žádná, data hlavičky a celý paket. Poslední parametr udává limit délky předaných dat.

Funkce nfq_nfnlh vrátí deskriptor netlinku pro danou frontu, který potřebujeme jako parametr pro další funkci nfq_fd, která nám získá deskriptor pro čtení paketů.

Dále můžeme ve smyčce číst data z deskriptoru. Tato data, což mohou být i fragmenty paketů, předáme funkci nfq_handle_packet, která jakmile získá kompletní paket, tak zavolá callback funkci, které předá kompletní paket.

Callback funkce obsahuje pouze volání funkce pro odeslání paketu SAN ii a vydání verdiktu NF_STOLEN. Tento verdikt sděluje systému, že může uvolnit paměť paketu a nemá dále pokračovat v jeho zpracování. Veškerá spojení asociovaná s tímto paketem však v paměti i nadále zůstávají.

Ve zdrojovém souboru se dále nachází funkce send_packet, která datový paket odešle přes soket do SAN ii. Tato funkce obsahuje vyrovnávací paměť, do které je zapsán celý paket před odesláním a je pak odeslán najednou a v ideálním případě pak dojde při přenosu sítí k přenosu pouze jednoho IP paketu. Kdyby vyrovnávací paměť nebyla použita, odcházely by z aplikace jednotlivé bajty v separátních paketech, což by se negativně projevilo na síťovém provozu a posléze i na výkonnosti.

3.2.5 Komponenta Injector

Komponenta Injector je implementována v souboru injector.c a souvisejícím hlavičkovém souboru injector.h. Podobně jako komponenta Interceptor má komponenta Injector vstupní bod aktivovaný při vytvoření vlákna. Vstupní bod komponenty se nazývá injector_run().

V metodě injector_run() je prvně vytvořen AF_INET/RAW soket a posléze je ve smyčce volána metoda receive_packet(), která přečte ze soketu SAN ii paket. Pokud se shoduje jméno aplikace se jménem v konfiguračním souboru, předá jej metodě inject_packet(), jež jej vloží do protokolového zásobníku.

58 odkaz na funkci, která je spuštěna při nějaké události tak, aby se zamezilo potřebě aktivního čekání ze strany konzumenta

Když jsem prováděl analýzu chování soketů v oblasti vytváření paketů, dospěl jsem prvně k mylnému závěru, že pomocí RAW soketu AF_INET není možné použít hlavičku paketu v binárním tvaru, kterou mám k dispozici a že je nutné buď získaný paket zpracovat do struktury nebo je nutné vložení paketu udělat pomocí soketu PF_PACKET/SOCK_DGRAM. Z obav před nedokonalým namapování hlavičky paketu (která obecně může být jakákoliv) na strukturu a tím i možnou změnu paketu jsem se uchýlil k analýze chování soketů na linkové vrstvě, kde jsem byl přesvědčen, že najdu řešení. Během pokusů na druhé vrstvě jsem dosáhl funkčního řešení, které však vyžadovalo komunikaci s ARP vyrovnávací pamětí pomocí systémových volání ioctl a dokonce i vysílání ARP požadavků. Bylo také nutné používat volání závislé na technologii a složitě provádět směrovací rozhodnutí. V tento okamžik jsem svůj předchozí výzkum v této oblasti přehodnotil a podrobil zpětné kontrole. Zjistil jsem svůj omyl a následná implementace pomocí soketu AF_INET/SOCK_RAW byla o poznání kratší a jednodušší, neboť o veškeré výše uvedené činnosti se stará systém sám a i přesto paket zůstane naprosto nezměněn.

3.2.5.1 Injector na linkové vrstvě

Přesto, že samotné řešení na linkové vrstvě není nyní využito, obsahovalo několik zajímavých úseků kódu. Ukazuje, jak se provádějí systémová volání pomocí rozhraní ioctl[59] a jak je možné vytvářet i jiné než IP pakety. Také přečtení této pasáže pomůže osvětlit a ucelit pohled na celou problematiku BSD soketů. Zejména pokud se člověk setká pouze s programováním soketů na síťové vrstvě a programuje pouze pro IP, těžko pochopí, proč se které konstanty a struktury jmenují tak, jak se jmenují a jaký je jejich přesný význam.

V první řadě vytvoříme soket pro linkovou vrstvu AF_PACKET[60]/SOCK_DGRAM jemuž jako parametr předáme konstantu ETH_P_IP. Jedná se tedy o analogii k soketu na síťové vrstvě, kde je posledním parametrem také protokol nesený na vyšší vrstvě.

Namísto cílové adresy ve struktuře sockaddr_in[61], kterou používáme při komunikaci na síťové vrstvě protokolem IP musíme logicky mít o stupeň nižší adresu. Struktura pro fyzické adresy se nazývá sockaddr_ll[62] a je definována systémovém hlavičkovém souboru packet.h.

59 Ioctl je univerzální systémové na platformě linux, kterým se kód v uživatelském prostoru může dotazovat ovladačů v systému a nastavovat jejich parametry. Ve Windows této funkci zhruba odpovídá funkce DeviceIoControl z win32 API

60 V různých zdrojích se často zaměňuje Protocol Family a Address Family. Oddělení těchto pojmů je spíše historické a ve většině systémů je dnes naprosto jedno, zda používáte AF_ nebo PF_ neboť tyto konstanty jsou ve zdrojových kódech většinou mapovány na sebe navzájem, ale přednostně by se mělo používat označení začínající AF_, neboť PF_ pochází z BSD systému a i tam se od něj upouští

61 socket address inet – adresa pro použití při komunikaci soketem protokolem IP

62 socket address link layer – adresa linkové vrstvy určená pro použití se soketem

Kromě cílové adresy uložené v položce struktury sll_addr[63], její délky sll_halen[64] je třeba také vyplnit nesený protokol v položce sll_protocol a samozřejmě rozhraní, do kterého má být rámec vložen – položka sll_ifindex.

Většinu parametrů snadno získáme. Problém nastává se zjištěním cílové adresy, protože nevíme, kam paket vlastně chceme poslat. Jednou z nejjednodušších možností je odeslat rámec záplavou, což ale bude většina strojů tiše ignorovat[65]. Každý stroj, který má zásobník protokolu IPv4, musí mít implementován také protokol ARP[66] a páry IP adresa – fyzická adresa si ukládá do vyrovnávací paměti, aby zabránil prodlení při doručování paketů na stejnou adresu. Proto když chceme zjistit, na jakou adresu paket odeslat, je třeba provést dotaz na vyrovnávací paměť ARP. Toho dosáhneme pomocí dříve uvedeného univerzálního systémového volání ioctl.

Funkce ioctl potřebuje ke své funkci tři parametry – prvním je soket, nad kterým bude volání provedeno. Dalším je identifikátor dotazu – konstanta definovaná v některém ze systémových hlavičkových souborů. A posledním je struktura pro předání dat vztahujícím se k volání. V našem případě se budeme dotazovat nad soketem typu AF_INET, který pro tento účel vytvoříme, typ dotazu bude SIOCGARP[67] a jako dotaz předáme strukturu arpreq ve které vyplníme pouze položku protokolové adresy, do položky hardwarové adresy musíme vyplnit pouze technologii rozhraní např. ARPHRD_ETHER[68] a jméno zařízení na kterém potřebujeme dotaz vykonat.

Zde se již chování jednotlivých Linuxových distribucí odlišuje. Některé považují tento dotaz pouze za dotaz, zda položka ve vyrovnávací paměti existuje a pokud ne, neprovádějí žádné další akce. Jiné distribuce v případě neexistence záznamu ve vyrovnávací paměti okamžitě vysílají po síti ARP dotaz a až v případě neúspěchu vracejí prázdný záznam.

V příloze A na straně 53 naleznete výpis souboru arp.c, který obsahuje výše uvedený princip získání fyzické adresy z adresy IP s tím, že před samotným dotazem na vyrovnávací paměť je do sítě odeslán ARP požadavek pro danou adresu, aby se setřel rozdíl mezi chováním jednotlivých distribucí a jader. V případě neúspěchu dotazu je vracena fyzická adresa brány a v případě neexistence brány je rámec rozeslán záplavou. Toto řešení je omezeno na technologii Ethernet.

63 socket link layer address
64 socket link layer hardware address length -
65 Je to nejjednodušší možnost, ale praktický pokus ukázal, že například na ICMP echo paket určený do jiné sítě odeslaný záplavou neodpověděla žádná stanice, ani výchozí brána
66 Address Resolution Protocol – protokol pro zjištění fyzické adresy z adresy IP
67 socket ioctl get ARP
68 ARP hardware – samozřejmě podle typu hardwaru, který používáme

3.2.6 Komponenta Connector

Komponenta Connector vznikla čistě za účelem testování paměťové náročnosti, náročnosti na procesorový čas a celkové propustnosti výše zmíněných komponent. Komponenta je implementována ve zdrojovém souboru connector.c. Komponenta vytvoří dva naslouchající servery a na každém z nich akceptuje právě jedno spojení. Po úspěšném připojení dvou klientů pouze předává data mezi klienty, jinak nevykonává žádnou činnost. Konfigurace portů se z důvodu, že se jedná o vývojový prostředek neuvažuje a je nutné případně přepsat hodnotu ve zdrojovém kódu.

3.3 Práce na projektu SAN

Vytvoření balíku SAN ii byl celkově časově nejnáročnější úsek práce. Zaprvé bylo nutné se seznámit s architekturou uzlu SAN, která se od doby publikace [RM08] poněkud změnila a změny nejsou prozatím nikde zdokumentovány. Další nepříjemnou překážkou byl fakt, že pro celý server do té doby neexistoval jednoduchý způsob jak jej přeložit, sestavit a spustit, původní dokumentace nekorespondovala s realitou. Vzhledem k tomu, že v poslední době také probíhal intenzivní vývoj vlastního interpreteru jazyka Java, aby uzel SAN mohl mít v budoucnu plnohodnotné EE, nebylo možné ani použít dodané aplikace, neboť ty byly určeny pro staré aplikační rozhraní. Nově byly také dokončeny práce na distribuci kódu aplikací, a tak každá chyba v nastavení nebo balících aplikací vyústila v nekonečné chybové výpisy.

Celý server se překládal spuštěním překladače nad celým adresářem a výsledkem byla směs zdrojového kódu, přeložených tříd a aplikací zabalených v balících SPK[69] a BSH[70]. Pro spuštění bylo nutné doplnit cestu ke knihovnám a nebylo vůbec zřejmé, odkud server získává vlastní kód pro svůj běh.

Protože tento stav byl neúnosný pro můj vlastní vývoj aplikace v serveru, rozhodl jsem se pro vyčištění serveru a nastavení jasných pravidel pro sestavení a vytvoření distribuovatelné verze v balíčcích. Mé činnosti na zvýšení použitelnosti balíku byly následující:

- kompletní vymazání přeloženého kódu, ponechání pouze zdrojových kódů
- vytvoření samostatného adresáře pro aktivní aplikace
- vytvoření samostatného adresáře pro přeložený kód

69 SAN package – aplikace pro interpretaci v novém interpreteru
70 BeanShell package – aplikace pro interpretaci v BeanShellu

- vytvoření skriptu pro utilitu ANT pro překlad a spuštění serveru
- vytvoření JAR balíku se serverem
- vytvoření SPK a BSH balíků aplikací a jejich otisků
- vytvoření readme souboru se základními informacemi o práci s projektem

Teprve po těchto činnostech bylo možné započít produktivní práci na projektu a začal vznikat balík san.ii.

3.3.1 Struktura úložiště projektu

Úložiště projektu bylo chaotické, nemělo žádný pevný řád a neobsahovalo aktuální kód. Vzhledem k havarijnímu stavu úložiště jsem byl nucen změnit jeho strukturu tak, aby odpovídala trendům v softwarovém inženýrství a byla přehledná a jasná pro všechny uživatele projektu. Úložiště projektu se nachází na serveru pro open source projekty – sourceforge.net. Jedná se o úložiště spravované verzovacím systémem Subversion[71], jež nahrazuje starší CVS[72]. Vzhledem k tomu, že v oblasti SAN probíhá další vývoj, vznikly další navazující projekty a tudíž bylo nutné úložiště změnit z jednoprojektového na víceprojektové[73]. Každý projekt tak má v kořenové složce svoji vlastní složku. Ve složce každého projektu se pak

```
|-- build                       adresář s přeloženým kódem
|   |-- classes                   adresář obsahující pouze přeložené
|   |   |-- applications        třídy
|   |   `-- server                  aplikací
|   `-- dist                        serveru
|       |-- applications          adresář s JAR balíkem serveru a
|       `-- codeRepository      aplikací
|-- conf                            SPK a BSH balíky
|-- doc                             SPK a BSH balíky s hash kódem
|-- lib                         adresář s konfiguračními soubory
|-- log                         adresář s dokumentací
`-- src                         adresář s knihovnami třetích stran
    |-- applications            adresář pro výpisy běhů serveru
    `-- server                  adresář se zdrojovými kódy
                                  aktivních aplikací
                                  serveru
```

Ilustrace 9: Adresářová struktura hlavní vývojové větve SAN

nacházejí tři podsložky se speciálním významem. Složka trunk je přítomna vždy a obsahuje poslední vývojové změny – je to hlavní vývojová větev - kmen. Trunk projektu SAN je

71 http://subversion.tigris.org
72 Concurrent Versioning System – http://www.nongnu.org/cvs/
73 Jedná se pouze o formální změnu a doporučení, nicméně dodržení těchto pravidel vede k většímu řádu v úložišti

zobrazen a významy jednotlivých složek vysvětleny na Ilustraci 9. Složka tags obsahuje snímky hlavní větve v určitém čase. Dá se říci, že se jedná o takové snímky, kdy je projekt v konzistentním stavu a je připraven k vydání v této podobě. V adresáři branches pak nacházíme kopie hlavní větve, kde se vývoj ubírá jiným směrem nebo jsou zde prováděny experimenty. Tyto vývojové větve je pak možné dle potřeby sloučit do hlavní vývojové větve.

3.3.2 Konfigurace sestavení

Sestavovací skript build.xml je konfigurační soubor pro utilitu Apache ANT. Více o aplikaci ANT a jejím použití lze zjistit v technickém manuálu [BS+08]. Konfigurační soubor podobně jako konfigurační soubor známé utility make sestává z cílů (target), které na sobě mohou být závislé. V každém cíli je pak možné spouštět úlohy. Výhodou oproti utilitě make je, že konfigurační soubor build.xml neobsahuje implementaci úloh, ale pouze jejich opis, proto je nezávislý na platformě. To je příjemná vlastnost obzvláště u projektu typu SAN, kde se předpokládá nasazení v heterogenním prostředí a pokud by se měla udržovat sada sestavovacích skriptů pro jednotlivé operační systémy, bylo by to velmi nepohodlné a vedlo by to v konečném důsledku k nekonzistenci.

Ve výpisu kódu 5 můžeme porovnat variantu dávkového sestavovacího skriptu a zápis pro ANT. Oba skripty provádějí stejnou činnost pouze s tím rozdílem, že první z nich je určena pouze pro OS Linux a např. na OS Windows se vůbec neprovede, druhá varianta funguje na všech systémech, pro které existuje program ANT.

```
rm -rf build/dist/codeRepository
mkdir build/dist/codeRepository

cp  build/dist/applications/init.bsh  build/dist/codeRepository

java -cp build/dist/san.jar san.core.CodeStoreManager build/dist/codeRepository/init.bsh
```

```
<delete dir="build/dist/codeRepository"/>
<mkdir dir="build/dist/codeRepository"/>

<copy file="build/dist/applications/init.bsh"
todir="build/dist/codeRepository"/>

<java classname="san.core.CodeStoreManager">
  <classpath>
    <path location="build/dist/san.jar"/>
  </classpath>
  <arg value="build/dist/codeRepository/init.bsh"/>
</java>
```

Kód 5: Porovnání zápisu v dávkovém skriptu a v build.xml

Pro sestavení a práci s projektem SAN jsem zapsal následující cíle, které odpovídají akcím, které je nejčastěji nutné provádět:

- clean – cíl smaže veškeré přeložené a vygenerované soubory, ponechá pouze zdrojové
- servercompile – přeloží všechny zdrojové soubory serveru do adresáře build classes/server
- appcompile – přeloží všechny aktivní aplikace do adresáře build/classes/applications
- compile – závisí na předchozích dvou cílech, přeloží tedy server a všechny aplikacemi
- serverpackage – z přeložených tříd vytvoří JAR balík san.jar v adresáři build/dist/
- apppackage – z přeložených aplikací vytvoří SPK nebo BSH balíčky v adresáři build/dist/applications
- package – závisí na předchozích dvou
- run – spustí balíkovou verzi serveru s konfiguračním souborem settings1.xml
- run2 – dtto, ale se souborem settings2.xml, což je vhodné pro základní testování
- apppackagehash – všechny balíky aplikací opatří hash kódem a překopíruje je do adresáře codeRepository, odkud je načítá server

3.3.3 Balík SAN ii

Balík san.ii obsahuje všechny třídy, které SAN ii vyžaduje ke své funkci a tvoří tak jeden logický funkční celek serveru. Sestává z tříd InterceptorInjector, Packet, PacketSerializer, InterceptorThread a InjectorThread.

3.3.3.1 Třída InterceptorInjector

Třída InterceptorInjector z balíku san.ii je třída zodpovědná za vytvoření naslouchajícího serveru na portu určeném v konfiguraci serveru a může obsluhovat najednou více klientů. Třída nemá veřejný konstruktor a je vytvořena podle vzoru singleton[74] a instanci třídy tak zle získat pouze voláním statické metody *getInetrceptorInjector()*. To má smysl z toho hlediska, že na daném portu může naslouchat pouze jeden server a není důvod povolit jakékoliv aplikaci získat více instancí tohoto serveru. První instance je vytvořena přímo ve vstupním bodu serveru, ve třídě san.core.Kernel. Poté, co se InterceptorInjector spustí, vytvoří pro každého přípojeného klienta samostatné vlákno InterceptorThread a InjectorThread. Zatímco není potřeba udržovat referenci z InterceptorInjector na InterceptorThread, protože v

74 Jedináček – od dané třídy lze získat nejvýše jednu instanci

tomto směru se nebudou předávat žádná data, při vytvoření každého InjectorThread se reference uloží do rozptylové tabulky.

Třída dále poskytuje spuštěným vláknům služby odeslání dat do aktivní aplikace – metodu storePayload – a odeslání paketu meziprocesovou komunikací klientům – metodu sendPayload.

Metoda sendPayload je jednoduchá a pouze data pošle záplavou na všechny připojené klienty. To je z toho důvodu, že bez hlubší analýzy přenášených dat a udržování stavů komunikace, nejsme schopni poznat, kterému klientovi data patří. K rozeslání dat projde metoda všechna vlákna InjectorThread v rozptylové tabulce a předá jim data k odeslání.

```
synchronized public void storePayload(Packet payload) throws InterruptedException {
    Guid guid;

    try {
      //run application with Init as its parent - to innherit console etc.
      guid = kernel.runApplication(payload.getApplicationName(),
                                   payload.getApplicationParameters(),
                                   kernel.getInitGuid());

      //get process of application run
      Process p = kernel.getScheduler().getProcess(guid);

      if (p != null && p instanceof ProcessApplication) {
        ProcessApplication pa = (ProcessApplication) p;
        Reference thisRef = kernel.getInterpretManager().getRunningApplication(guid);
        while (thisRef == null) { //this can take a while - wait. Who waits, gets :]
          thisRef = kernel.getInterpretManager().getRunningApplication(guid);
          Thread.sleep(1L);
        }

        //get application's ClassManager
        ClassManager cman = new ClassManager((SanCodePkg) pa.getCodePackage());
        try {
          // fire application's receiveData method
          cman.runMethod(thisRef.getClassInfo().getName(),
                         "receiveData",
                         "(Lsan/ident/Guid;Lsan/data/TransmitData;)V",
                         new Object[]{kernel.getInitGuid(),
                         new TransmitData(payload.getPayload())},
                         thisRef,
                         false);
        } catch (Exception ex) {
          System.out.println("Cannot execute methods on application" +
                  payload.getApplicationName() + " with state " +
                  p.getState() + "\n" + ex);
          ex.printStackTrace();
        }
      }

    } catch (ExecuteException ex) {
      System.out.println("Cannot run application " + payload.getApplicationName());
    }
}
```

Kód 6: Zdrojový kód metody storePayload

Funkce metody storePayload pro předání paketu a parametrů aktivní aplikaci je složitější a hlavní myšlenkový tok uvedu dále – pro lepší pochopení je vhodné sledovat

zdrojový kód metody uvedený v Kód 6. Metoda nejprve spustí pomocí metody jádra aktivní aplikaci a získá identifikátor běžící aplikace. Povšimnete si, že jako referenci nadřazené aplikace předáváme identifikátor aplikace Init, kterou jádro spouští při svém zavedení. Je to z toho důvodu, že v serveru SAN aplikace od svých rodičů dědí výstupní a vstupní proud. Pokud chceme aplikaci ponechat možnost něco vypsat, je to vhodné provést takto. Navíc i z hlediska hierarchie procesů je to čisté řešení. Takto spuštěná aplikace je plánována interním plánovačem a postupně interpretována vnitřním interpreterem. Pomocí systémového volání plánovače postupně získá referenci samotného procesu. Vzhledem k tomu, že plánovači nějakou dobu trvá, než aplikaci naplánuje a dojde k její inicializaci, je vloženo aktivní čekání, neboť jinak bychom náhodně nemuseli referenci získat, pokud by aplikace ještě nebyla inicializovaná – na tento problém jsem narazil záhy, když při ručním krokování aplikace fungovala bezchybně a při neasistovaném běhu vyhazovala výjimku, neboť reference thisRef byla stále null. Za pomoci reference procesu získá referenci na instanci interpreteru. Dále získáme manažer tříd pro danou aplikaci a nad ním můžeme interpretovat libovolnou metodu z aplikace.

Interpreteru sdělíme název metody, typy předaných parametrů a referenci na proces běžící aplikace. Metoda se vykoná v kontextu běžící aplikace, popřípadě nastane výjimka, pokud nelze metodu z jakéhokoliv důvodu vykonat.

3.3.3.2 Třída Packet

Třída Packet je datový kontejner na san.ii PDU. Obsahuje pouze proměnné pro uskladnění PDU, konstruktor a gettery[75].

3.3.3.3 Třída InterceptorThread

Stejně jako v implementaci programu saninterceptor jsou zvlášť implementována část pro příjem a zvlášť část pro odesílání dat tak, jak je zobrazeno již na ilustraci 3 na straně 17. Třída implementuje rozhraní Runnable, aby mohla být spuštěna jako samostatné vlákno.

3.3.3.4 Třída PacketSerializer

Třída PacketSerializer poskytuje metody nad vstupním nebo výstupním proudem pro serializaci třídy Packet, která je logickou obálkou SAN ii PDU. PacketSerializer obsahuje pouze dvě veřejné metody readPacket a writePacket. První načítá bajty ze vstupního proudu a po přečtení celého paketu vrátí inicializovaný objekt třídy Packet a druhá zmíněná metoda

75 Getter je metoda, která pouze zpřístupňuje privátní proměnnou objektu

provádí přesně opačnou činnost, tedy zapisuje paket do výstupního proudu. Zápis paketu do výstupního proudu je prováděn přes vyrovnávací paměť tak, aby po komunikačním kanálu byl paket přenášen v celku a nikoliv po jednotlivých bajtech.

3.3.3.5 Nastavení san.ii

Nastavení san.ii se provádí v centrálním konfiguračním souboru settings.xml. Zde je zavedena entita interceptorInjector s jediným atributem listenPort, který určuje, na jakém portu má san.ii naslouchat. V případě vynechání entity v konfiguračním souboru nedojde ke spuštění san.ii vůbec.

3.3.4 Úpravy SAN interpreteru

Během své práce jsem byl nucen několikrát zasáhnout do samotného SAN interpreteru. Většina způsobených problémů a jejich náprava byla dána neschopností interpreteru na různých místech získat zdrojový kód aplikace.

3.3.4.1 ClassManager.java a HardClass.java

Třída ClassManager z balíku san.interpreter.sanint.classLoading má na starosti načítání souborů tříd v případě, že ještě nejsou známy interpreteru. V původním stavu využívala nedokonalosti úložiště a kód pro aktivní aplikace paradoxně vůbec nezískávala z úložiště aplikací a jejich balíků, ale přímo z přeložených tříd. Po úklidu v úložišti se tento problém naplno projevil, protože adresáře s přeloženými třídami nejsou zahrnuty v class path[76]. Tím došlo k tomu, že při pokusu interpretovat jakoukoliv aplikaci došlo k výjimce ClassNotFoundException a interpretace skončila. To bylo způsobeno chybným zpracováním URL pro načítanou třídu, jež se chovalo na každé platformě jinak.

V původním řešení jak je naznačeno v Kód 7 se ClassManager spoléhal na fakt, že třída aplikace se nachází v class path a tudíž ji ClassLoader nalezne. Tento předpoklad je ovšem mylný, neboť sice toto řešení může fungovat pro všechny systémové třídy a třídy serveru, nicméně nikoliv pro aktivní aplikace. Vzhledem k tomu, že mnohdy nejsou ani přítomné při spuštění serveru a objeví se až za běhu, je třeba tuto situaci ošetřit. Dalším problémem je fakt, že funkce url.toString() vrací na různých platformách pro archivy JAR různé počáteční lokátory a navíc může vracet v zásadě i jiné než jar:file: . Na OS Windows vrací jar:file:cesta a na OS Linux pouze file://cesta, a proto nelze využít odřezávaní počátků cest jako řešení.

76 Class path – cesta k souborům, archivům a adresářům, kde Java Virtual Machine hledá objekty tříd

ClassManager.java:

```
URL url = ClassLoader.getSystemClassLoader().getResource(
                                type.replace('.', '/') + ".class");
String str = UrlUtil.decode(url.toString());
cl = new HardClass(str, this);
```

HardClass.java:

```
String jarFileName = filename.substring(9, ind);
String fileName = filename.substring(ind + 2);
```

Kód 7: Původní kód ClassManager.java a odpovídající kód HardClass.java

Proto jsem navrhl a implementoval řešení na platformě nezávislé, které počítá s tím, že aktivní aplikace nebude systémovým Class Loaderem nalezena. Vhodnou kombinací systémových volání zjistí cestu k souboru s třídou nezávisle na platformě. Toto řešení je uvedeno v Kód 8 a níže jej dovysvětlím.

ClassManager.java

```
URL url = ClassLoader.getSystemClassLoader().getResource(
                                type + "." + SanCodePkg.BIN_EXTENSION);

if (url == null && sanPackage.getPkgName().equals(
                                type.substring(0,type.indexOf('/')))) {

      String path = sanPackage.getAplPkgZipFilename() + "!/" +
            SanCodePkg.BINARY_ENTRY +
            type.substring(type.indexOf('/')) + "." +
            SanCodePkg.BIN_EXTENSION;

            cl = new HardClass(path, this);
} else { // system class loader located the file
      String str = new File(
      new URI(url.getPath())).getAbsolutePath());
      cl = new HardClass(str, this);
}
```

HardClass.java

```
String jarFileName = filename.substring(0, ind);
String fileName = filename.substring(ind + 2).replace('\\', '/');
```

Kód 8: Změněný kód ClassManager.java a HardClass.java

Nejprve se pokusíme nalézt třídu pomocí systémového Class Loaderu. Pokud se to nepodaří a jméno balíku třídy se shoduje s názvem interpretovaného balíku, pak je zřejmé, že

zdrojový kód musíme hledat v hashovaném JAR archivu v codeRepository. V případě, že Class Loader nalezne balík se třídou, pak jej pomocí objektu třídy URI převedeme na absolutní cestu k souboru.

3.3.4.2 ReferenceCounter.java

Ve třídě ReferenceCounter autor předpokládal, že každá třída má veřejný konstruktor a pokud se nepodaří jeho referenci najít, že je to důkaz chyby a důvod pro eskalaci výjimky, která přeruší interpretaci kódu. Vzhledem k tomu, že jsem při vývoji san.ii využil návrhový vzor singleton, který nemá veřejný konstruktor docházelo k chybám a přerušení interpretace. Do kódu tak bylo nutné přidat podmínku, která testovala výsledek funkce a pokud nebyl konstruktor nalezen, vrací referenci na null, který je legitimní referencí.

3.3.5 Úpravy rozhraní serveru SAN

Vzhledem k tomu, že v původním návrhu serveru nebylo v podstatě počítáno s výměnou dat mezi aplikací a serverem popřípadě kapsulí a serverem, bylo nutné lehce poupravit rozhraní, kterým aplikace má možnost komunikovat se serverem a také rozhraní, kterým komunikuje kapsule se serverem.

Úpravy se týkají rozhraní IApplicationAPI a ICapsuleAPI. Do rozhraní přibyla metoda sendPayload(), která umožňuje aplikaci a kapsuli předat datový paket serveru ke zpracování. Ve třídách implementujících tato rozhraní byla metoda implementována přímým předáním san.ii metodě sendPayload(). V případě požadavku na rozšíření není ale problém implementovat metodu jiným způsobem a rozhodnout, jak bude s datovým nákladem naloženo.

3.3.6 Aktivní aplikace tunneler

Aplikace tunneler byla napsána jako testovací aplikace celého navrženého konceptu. Je velmi jednoduchá. Dále si vysvětlíme její funkčnost na výpisu kódu uvedeném v příloze B.

V momentě, kdy aplikaci spustíme příkazem runApplication v jádře serveru SAN, je vytvořena instance třídy Process pro tuto aplikaci a tento proces je dále plánován plánovačem. Jakmile je aplikace naplánována, započne její interpretace. Vstupním bodem každé aktivní aplikace je metoda main. V ní je jako parametr předán i odkaz na rozhraní, kterým aplikace může komunikovat se serverem.

V našem případě si aplikace pro kontrolu vyžádá identifikátor aplikace Init, získá ze

serveru proud pro standardní výstup a dále zpracuje parametry, které interpretuje jako identifikaci cílového uzlu.

Následující kód čeká, dokud nebudou aplikaci předána data. Všimněte si, že kód čeká aktivně, namísto toho, aby využil synchronizaci wait-notify. Bohužel prozatím není tato synchronizace možná vzhledem k jistým omezením v SAN interpreteru. Způsob předávání dat pomocí rozhraní ReceiveDataListener byl sice původně určen pro předávání dat kapsulemi do aplikace, ale v zásadě nám nic nebrání toto rozhraní využít pro zaslání jakýchkoliv dat aplikaci.

Data jsou přijímána metodou receiveData a pouze jsou předána do proměnné objektu v okamžiku, kdy je metoda volána. V tento okamžik je do sítě vložena kapsule a aplikace skončí, neboť neočekává žádnou odpověď.

Nyní se podívejme, co se stane po vložení kapsule do sítě. Kapsule obsahuje kód uvedený v příloze C. Při každém příchodu kapsule na uzel je spuštěna metoda main() a začne se interpretovat. Protože kapsule tunneleru se potřebuje pouze dostat k cíli, na mezilehlých uzlech nevykonává žádný kód a opouští interpretaci voláním return. Všimněme si zde rozdílu oproti klasické síti. Zde by obyčejný paket skončil a byl by zničen. Naproti tomu kapsule dál putuje sítí, dokud se sama nezničí.

Na koncovém uzlu si kapsule vyzvedne svůj binární datový náklad a vytvoří instanci třídy Packet, do které uloží parametry, svojí identifikaci a datový náklad. Svůj úkol tak splnila a může tedy požádat skrze rozhraní o ukončení svého života.

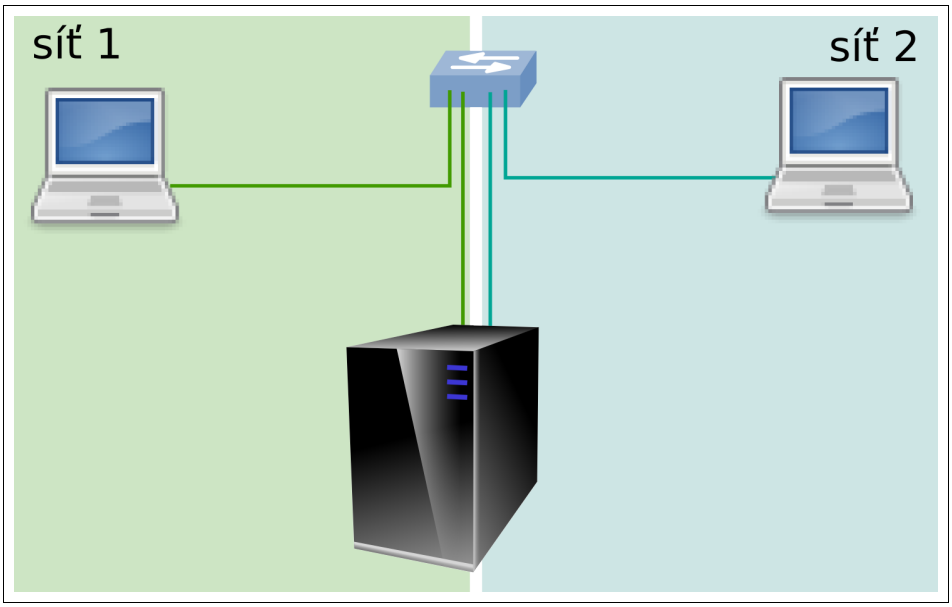
3.4 Testování aplikace

Aplikace byla testována ve dvou fázích z hlediska výkonnostního a poté z hlediska funkčnosti jako celek. V první fázi byl proveden test v zapojení s konektorem[77] a v druhé v zapojení se dvěma servery SAN na jedné stanici.

77 Komponenta Connector programu saninterceptor

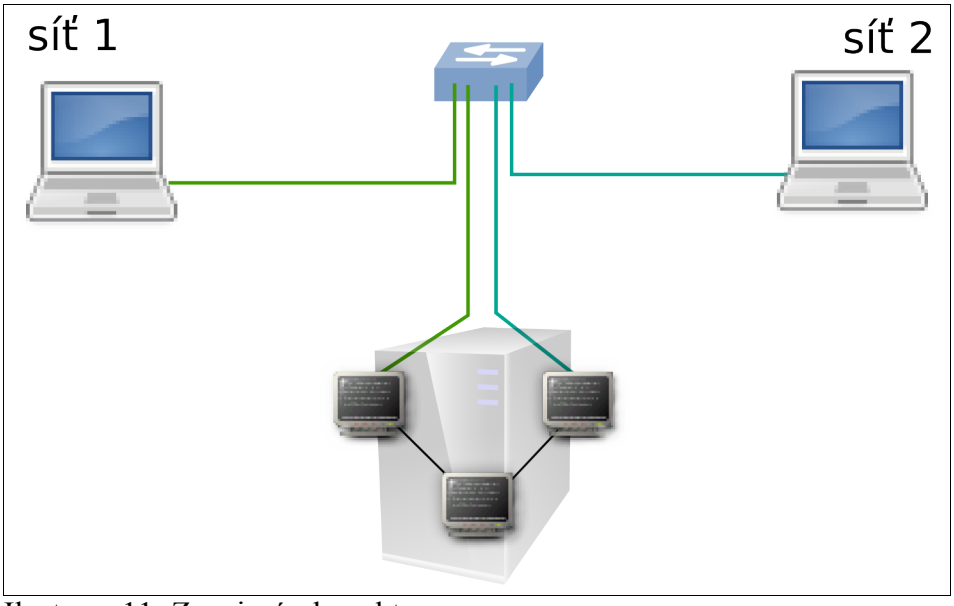
3.4.1 Testovací síť

Pro testování jsem využil jednoho stolního počítače a dvou starších notebooků



Ilustrace 10: Fyzické zapojení testovací sítě

propojených spravovatelným přepínačem. Cílem testování nebylo zjištění maximálních propustností a časů na dostupném nejvýkonnějším hardware, nýbrž sledování zátěže prvků a rozdíl oproti vykonávání bez tunelování. Zapojení testovací sítě je znázorněno na ilustraci 10. Stolní počítač nebyl vybaven dvěma síťovými kartami, proto bylo použito virtuální sítě, aby došlo k úplnému oddělení koncových stanic tvořených notebooky. Na stolním počítači bylo



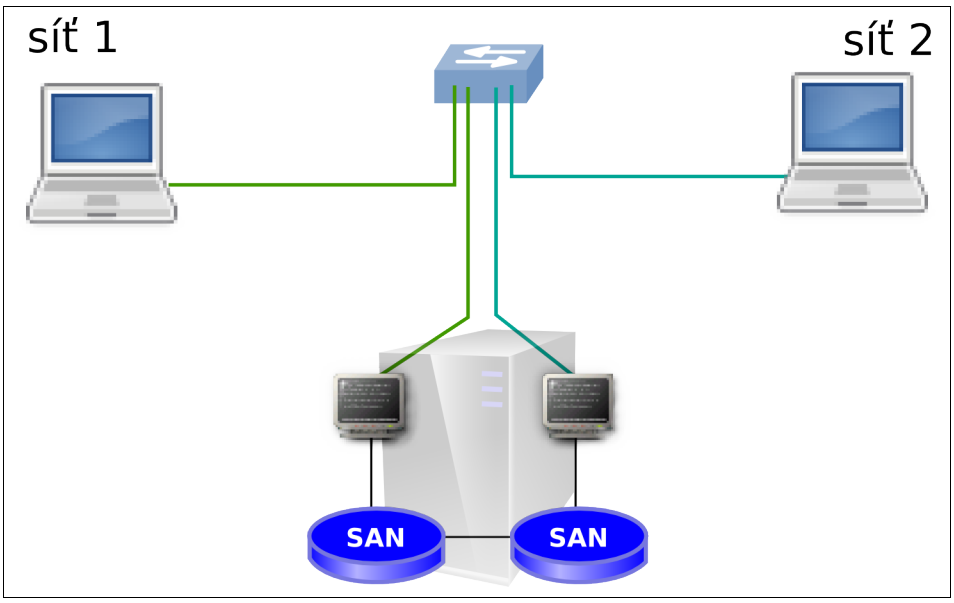
Ilustrace 11: Zapojení s konektorem

zapnuto směrování IP, spuštěny příslušné aplikace. Do paketového filtru byla zavedena

pravidla s cílem NFQUEUE. Pro obě síťová rozhraní byla vytvořena samostatná fronta[78] a filtr byl nastaven na protokoly TCP, UDP a ICMP.

Aplikace spuštěné ve fázi jedna a vytvořené logické spojení je zobrazeno na ilustraci 11. Jedná se o zapojení testující čistou propustnost skrze saninterceptor za použití konektoru a není zde vykonáván žádný aktivní kód, pouze kód spojený s kódováním a předáním paketu v SAN PDU. Toto zapojení umožňuje při dlouhodobé zátěži získat charakteristiku přenosu bez ohledu na aktivní uzel a empiricky zjistit, zda nedochází k únikům paměti a zda je řešení stabilní. Také umožňuje snadno sledovat, zda dochází opravdu k transparentnímu tunelování IP.

Ve fázi dvě byl konektor nahrazen párem aktivních serverů SAN se spuštěným san.ii. V tomto zapojení se již projeví zpoždění dané vykonáváním aktivního kódu na SAN serverech. Zapojení a konfigurace použitá ve druhé fázi testování je vyobrazena na ilustraci 12.



Ilustrace 12: Zapojení s párem aktivních serverů

3.4.2 Výsledky testů

Před vlastním zahájením testů byly zaznamenány hodnoty propustnosti a odezvy sítě pro pozdější srovnání. Propustnost prosté sítě byla vyšší než 94 Mbps a odezva mezi koncovými stanicemi byla nejvýše 2 ms, zátěž CPU nebyla pozorována. Kromě zátěžového testu byl proveden subjektivní uživatelský test přihlášením ke vzdálené stanici přes SSH[79] a

78 --queue-num parametr cíle NFQUEUE
79 Secure Shell – bezpečná skořápka – protokol využívaný pro vzdaálené přihlášení a bezpešné tunelování služeb

prohlížení stránek na webovém serveru na vzdáleném uzlu.

3.4.2.1 Saninterceptor

V první fázi byl proveden zátěžový test na síti podle ilustrace 11. Po spuštění konektoru a obou instancí saninterceptoru byl vytvořen saturující datový tok z prvního koncového uzlu do druhého koncového uzlu. Během přenosu byla monitorována odezva, spotřeba paměti programu saninterceptor, vytížení CPU a propustnost. Výsledky testu ukázaly, že odezva se zvýšila oproti prosté síti na 2 ms, maximální datový tok kolem 90 Mbps, spotřeba paměti konstantně 18 kB na jeden proces a vytížení CPU nebylo pozorovatelné[80].

Při vzdáleném přihlášení nebyl pozorován rozdíl oproti použití v klasické síti. Při testu prohlížení webových stránek nebyl pozorován žádný rozdíl.

3.4.2.2 Saninterceptor s párem aktivních serverů SAN

Ve druhé fázi byl proveden zátěžový test dle ilustrace 12. Po spuštění páru aktivních serverů a obou instancí saninterceptoru byl stejně jako v předchozím případě vytvořen saturující datový tok mezi koncovými uzly. Výsledky testu ukázaly, že odezva se zvýšila na průměrnou hodnotu cca 200 ms, maximální datový tok 120 kbps a vytížení CPU 100%.

Při vzdáleném přihlášení byl pozorován rozdíl ve smyslu zhoršené odezvy v případě požadavku na větší objem dat, jako je například výpis adresářové struktury atp. Při prohlížení webových stránek na cílovém uzlu bylo znatelné pomalé načítání grafiky, jinak beze změny.

3.4.3 Zhodnocení výsledků

Z výsledků je patrné, že již použití cíle NFQUEUE a předání paketu do uživatelského prostoru vyžaduje jisté systémové prostředky, ale v zásadě je pro projekt bez problému použitelné. V případě, že by se toto místo stalo úzkým hrdlem projektu, vždy existuje možnost přímé manipulace s paketem v režimu jádra pomocí jaderného modulu.

Výsledky použití samotného saninterceptoru jsou velmi dobré. Lze dosáhnout bez optimalizací dostatečného datového toku s minimálním zpožděním. Konstantní využití malého množství paměti je taktéž dobrým výsledkem. Oproti předpokladu, že meziprocesová komunikace skrze sokety nebude příliš rychlá je toto překvapivý výsledek a ukazuje se, že ani toto není prozatím úzké hrdlo systému.

Výsledek plného nasazení není nijak překvapivý. Již autor serveru SAN v [RM08]

80 Proces saninterceptor podle výpisu utility top strávil přes 90% času ve stavu wait I/O

popisuje ve výsledcích velkou odezvu při použití aktivní aplikace Ping. I přesto, že došlo k vývoji vlastního interpreteru, ukazuje se, že dochází stále k výraznému zpoždění při vykonávání aplikací i když se jedná o velmi krátké programové kódy. Datová propustnost je velmi malá a již při interaktivních službách typu SSH představuje nepříjemné úzké hrdlo.

4 Závěr

Vývoj aktivních sítí sahá do roku 1995 a jedná se o zcela odlišný koncept oproti klasickým sítím. I když se vyvíjí 14 let, je stále ve fázi výzkumu a neexistuje žádné univerzálně použitelné řešení. Podle [CK06] je však za tím třeba hledat i jiné aspekty, než je architektura aktivních sítí. I přesto se však jedná o koncept, který v budoucnu má velkou šanci na úspěch, neboť používání síťových protokolů typu IP a podobných nebude efektivní nejen z hlediska přenosových kapacit, ale hlavně z hlediska rychlosti nasazení nových protokolů a služeb. Oproti klasickým sítím nabízejí aktivní sítě velmi obecný a snadno pochopitelný koncept pro vývojáře, o čemž jsem se přesvědčil při zpracovávání této diplomové práce.

Největším problémem současného řešení aktivního uzlu SAN vidím v pomalé rychlosti zpracování aktivních aplikací na uzlech sítě, nicméně například v projektu PANDA bylo ukázáno, že toto je problém, který je řešitelný. V současné době již existuje analýza, která ukazuje, že pravděpodobným úzkým hrdlem SAN je plánovač ve spojení s implementací vláken EE jako fibres v jazyce Java, tj. vlákny interpreteru SAN. Tato oblast bude dále řešena i s možným využitím Google Native Client.

4.1 Splnění cílů práce

Během zpracovávání práce jsem se seznámil s konceptem aktivních sítí a z dostupných poznatků jsem sestavil úvodní kapitolu a získané poznatky využil při návrhu konceptu práce.

V této diplomové práci byly představeny možnosti, jak zajistit provoz aplikací pro IP protokol v prostředí aktivních sítí a pro důkaz funkčnosti konceptu byla provedena implementace konceptu v rámci operačního systému Linux a aktivního uzlu SAN a zároveň byly ukázány směry, kterými se je možné vydat a kterými nikoliv v případě potřeby změny stávajícího řešení.

Při implementaci byla vytvořena i ukázková aktivní aplikace s kapsulí pro přenos IP paketů mezi sítěmi propojenými aktivní sítí na uzlech SAN a provedena měření. Implementace je funkční a jistá omezení použití vyplývají z aktivního uzlu SAN, nikoliv z práce samotné.

4.2 Vlastní přínos

Při vypracovávání úvodu a teoretické části jsem se nad dostupnými informacemi o aktivních sítích prováděl vlastní úvahy a neomezil jsem se pouze na citace publikovaných děl.

Jako příklad mohu uvést srovnání klasických a aktivních sítí z pohledu softwarového inženýrství a jejich přirovnání k metodikám vývoje používaných v softwarovém inženýrství, které ukazují, proč jsou aktivní sítě lepším konceptem, než sítě klasické.

Při zpracování práce jsem narazil na problémy, které znemožňovaly implementaci konceptu. Z tohoto důvodu jsem provedl několik zásadních zásahů do aktivního uzlu SAN a zároveň jsem vytvořil automatizovaný sestavovací skript pro snadné přeložení a spouštění aktivního uzlu, který dosud chyběl. Tím jsem také odhalil zásadní problémy s dostupností kódu aktivních aplikací, který jsem se snažil vyřešit. Dále jsem upravil strukturu úložiště aplikace tak, aby se stalo přehlednějším a použitelnějším.

Tento přínos vidím jako důležitý hlavně pro další vývojáře aktivního uzlu, neboť rapidně zkracuje dobu nutnou ke zprovoznění aktivního uzlu a umožní soustředit se na vlastní výzkum.

4.3 Další práce

Na konceptu aktivních sítí i na aktivním uzlu SAN existuje ještě mnoho problémů, které bude nutné řešit.

V první řadě je to úprava aktivního uzlu tak, aby nezpůsoboval markantní zpoždění při vykonávání aplikací a byl tedy celkově rychlejší i na běžně dostupném hardware. V současné době probíhá vývoj aktivního uzlu SAN i v jazyce C++, který by měl odstranit například problém v těžkopádné meziprocesové komunikaci a v konečném důsledku by mohl vést i k rychlejší interpretaci kódu aktivních aplikací. Dalším prací spojenou se zvýšením výkonu by mohlo být například využití Google Native Client ke spouštění částí kódu aktivních aplikací, které by byly přeloženy metodou JIT[81] a uloženy ve vyrovnávací paměti.

Další práce v oblasti tunelování IP aplikací by se mohla soustředit například na oblast aktivního zjišťování dostupných SAN.ii a jimi dostupných sítí, aby nebylo nutné do san.ii předávat pakety včetně adres cílového aktivního uzlu.

Prozatím otevřeným problémem zůstává otázka bezpečnosti vykonávaného programu aktivní aplikace. I přesto, že v minulém roce došlo k výraznému posunu v této oblasti, stále chybí implementace bezpečnostního monitoru, který zastavil případné spouštění škodlivého kódu. Otázku bezpečnosti nicméně řeší všechny dosud založené projekty aktivních sítí a v této oblasti bylo dosaženo velkého pokroku a tento rok bude dále řešena formou diplomové práce i v projektu SAN.

81 Just-in-time – metoda překladu bytecode za běhu do strojového kódu

Literatura

[BS02] Stephen F Bush. *Introduction to Active Networks*, 2002, <http://videolectures.net/contrib07_bush_ian/>

[RM08] Rejda Michal. *Server aktivní sítě*, Plzeň, 2008, Diplomová práce na ZČU Plzeň. Vedoucí práce Ing. Tomáš Koutný, PhD.

[TW96] Tennenhouse David L., Wetherall David J. . *Towards an Active Network Architecture*, In *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings*, 1996. s. 2-15

[RP98]: Reiher Peter. *Preliminary Panda Architecture*, 1998, <http://www.lasr.cs.ucla.edu/panda/preliminary_arch.ps>

[FV+02] Ferreria, V.; Rudenko, A.; Eustice, K.; Guy, R.; Ramakrishna, V.; Reiher, P.;. *Panda: middleware to provide the benefits of active networks to legacy applications*, In *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings*, 2002. s. 319 - 332

[MD97] Murphy David M.. *Building an Active Node on the Internet* , Massachssetts, 1997, Diplomová práce na MIT. Vedoucí práce John V. Guttag

[ZJ+04] Zhigang Jin, Yongmei Luo, Yantai Shu, Zhifeng Fu. *Supporting traditional IP applications in active networks*, In *Electrical and Computer Engineering, 2004. Canadian Conference on*, 2004. s. 121- 124

[YB+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. *Native Client: A Sandbox for Portable, Untrusted x86 Native Code*, In , 2009. s.

[LS05] Laurent Lefevre, Aweni Saroukou. *Active network support for deployment of Java-based games on mobile platforms*, In *Proceedings of the First International Conference on Distributed Frameworks for Multimedia Applications*, 2005. s.

[SJ09] Syrovátka Jan. *Code Execution in Active Networks*, Plzeň, 2009, Diplomová práce na ZČU Plzeň. Vedoucí práce Ing. Tomáš Koutný, Ph.D.

[SP09] Štěpánek Petr. *Code Distribution in Active Networks*, Plzeň, 2009, Diplomová práce na ZČU Plzeň. Vedoucí práce Ing. Tomáš Koutný, Ph.D.

[TN+02] Noriyuki Takahashi, Toshiaki Miyazaki, Takahiro Murooka. *APE: Fast and Secure Active Networking Architecture for Active Packet Editing*, In , 2002. s.

[DD+98] Dan Decasper, Guru Parulkar, Bernhard Plattner. *A Scalable, High Performance Active Network Node*, In , 1998. s.

[SJ07] Sýkora Jakub. *Internetová gateway*, Plzeň, 2007, Diplomová práce na ZČU Plzeň. Vedoucí práce Ing. Ladislav Pešička

[RRWH02] Russell Rusty, Welte Harald. *Linux netfilter Hacking HOWTO*, 2002, <http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html>

[BS+08]: Bailliez Stephane et. al. *Apache Ant User Manual*, 2008, <http://ant.apache.org/manual/index.html>

[CK06] Ken Calvert. *Reflections on Network Architecture: an Active Networking Perspective*, In *ACM SIGCOMM Computer Communication Review*, 2006. s. 27-30

Přílohy

Příloha A: Výpis zdrojového souboru arp.c

```
#include <arpa/inet.h>
#include <linux/if_packet.h>
#include <net/if_arp.h>
#include <netinet/ether.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/socket.h>

#include "ii.h"

int ioctlsocket, arpsocket;

struct sockaddr_ll me, he; //hardware ie MAC addresses of me and destinations

struct in_addr src, dst; //IP addresses

void arp_init(int ifindex) {
    int on = 1;
    struct sockaddr_in saddr;
    socklen_t alen;

    /* Get an internet domain socket for ARP cahce ioctl */
    if ((ioctlsocket = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        printe("Cannot open ioctl socket");
        return;
    }

    /* Get an PF_PACKET socket for ARP requests  */
    if ((arpsocket = socket(PF_PACKET, SOCK_DGRAM, 0)) == -1) {
        printe("Cannot create ARP request socket");
        return;
    }



    memset(&saddr, 0, sizeof (saddr));
    saddr.sin_addr = src;
    saddr.sin_family = AF_INET;


    alen = sizeof (saddr);

    saddr.sin_port = htons(1025); //connect whereever
    inet_aton(ipgateway, &saddr.sin_addr);

    if (setsockopt(ioctlsocket, SOL_SOCKET, SO_DONTROUTE,
            (char*) & on, sizeof (on)) == -1)
        printd("setsockopt(SO_DONTROUTE)");
    if (connect(ioctlsocket, (struct sockaddr*) & saddr, sizeof (saddr)) == -1){
        printe("Cannot connect to ioctl socket");
        return;
    }
    if (getsockname(ioctlsocket, (struct sockaddr*) & saddr, &alen) == -1) {
        printe("Cannot get my IP address");
        return;
    }

    src = saddr.sin_addr;



    me.sll_family = AF_PACKET;
```

```
    me.sll_ifindex = ifindex;
    me.sll_protocol = htons(ETH_P_ARP);

    if (bind(arpsocket, (struct sockaddr*) & me, sizeof (me)) == -1) {
        printe("Cannot bind to socket");
        return;
    }

    alen = sizeof (me);

    // if got address of bound socket
    if (getsockname(arpsocket, (struct sockaddr*) & me, &alen) == -1) {
        printe("Cannot obtain interface MAC address");
        return;
    }

    if (me.sll_halen == 0) {
        printe("Interface is not ARPable (no ll address)");
        return;
    }

    he = me;
    memset(he.sll_addr, -1, he.sll_halen); //set he address to broadcast



}

/**
 * sends ARP request packet
 */
int send_arp_packet(int s, struct in_addr src, struct in_addr dst,
        struct sockaddr_ll *ME, struct sockaddr_ll * HE) {
    int srbytes;
    unsigned char buf[256];
    struct arphdr *ah = (struct arphdr*) buf;
    unsigned char *p = (unsigned char *) (ah + 1);

    ah->ar_hrd = htons(ME->sll_hatype);
    if (ah->ar_hrd == htons(ARPHRD_FDDI))
        ah->ar_hrd = htons(ARPHRD_ETHER);
    ah->ar_pro = htons(ETH_P_IP);
    ah->ar_hln = ME->sll_halen;
    ah->ar_pln = 4;
    ah->ar_op = htons(ARPOP_REQUEST); //send type of ARP REQUEST

    memcpy(p, &ME->sll_addr, ah->ar_hln); //copy source MAC address
    p += ME->sll_halen;

    memcpy(p, &src, 4); //copy source IP address
    p += 4;


    memcpy(p, &HE->sll_addr, ah->ar_hln); //copy destination MAC address
    p += ah->ar_hln;

    memcpy(p, &dst, 4); //copy destination IP address
    p += 4;

    if ((srbytes = sendto(arpsocket,
                          buf,
                          p - buf,
                          0,
                          (struct sockaddr*) HE,
                          sizeof (*HE)))
            == -1) {
        printd("Error sending ARP request");
        return -1;
    };

    /* if ((srbytes = recvfrom(arpsocket, packet, sizeof (packet), 0,
            (struct sockaddr *) & from, &alen)) < 0) {
        printd("Cannot receive ARP reply");
        return NULL;
    } */
```

```
    return srbytes;
}

/**
 *Get ARP entry from cache with ioctl
 */
char *get_cache_entry(char *ipaddress) {
    char *macaddress;
    struct arpreq areq;
    struct sockaddr_in *sin;
    struct in_addr ipaddr;

    /* Make the ARP request. */
    memset(&areq, 0, sizeof (areq));
    sin = (struct sockaddr_in *) & areq.arp_pa;
    sin->sin_family = AF_INET;

    if (inet_aton(ipaddress, &ipaddr) == 0) {
        printe("Invalid format of IPv4 address %s", ipaddress);
        return NULL;
    }

    sin->sin_addr = ipaddr;
    sin = (struct sockaddr_in *) & areq.arp_ha;
    sin->sin_family = ARPHRD_ETHER;

    strncpy(areq.arp_dev, interface, strlen(interface));

    if (ioctl(ioctlsocket, SIOCGARP, (caddr_t) & areq) == -1) {
        printd("Unable to issue ARP request for %s", ipaddress);
    }

    if (areq.arp_flags & ATF_COM) {
        macaddress = strdup(
                ether_ntoa((struct ether_addr *) areq.arp_ha.sa_data));
        printd("ARP resolution %s -> %s", ipaddress, macaddress);
        return macaddress;
    } else {
        printd("ARP resolution for IP %s incomplete", ipaddress);
        return NULL;
    }
}

char *sprint_hex(unsigned char *p, int len)
{
        int i;
        //allocate 2 bytes for each byte, 5x : and 1x \0
        char *retstr = calloc(2*i+5+1, sizeof(char));
        char *retstrp = retstr;

        for (i=0; i<len; i++) {
                sprintf(retstrp,"%02X", p[i]);
                retstrp+=2;
                if (i != len-1) {
                        sprintf(retstrp, ":");
                        retstrp++;
                }
        }

        return retstr;
}

char *get_arp(char *ipaddress) {
    char *mac;

    he = me;
    memset(he.sll_addr, -1, he.sll_halen); //set his mac address to broadcast


    inet_aton(ipaddress, &dst);

    if (src.s_addr == dst.s_addr) { //if he is me, ARP will not function
        mac = sprint_hex(me.sll_addr, me.sll_halen);
        return mac;
```

```
    }

    send_arp_packet(arpsocket, src, dst, &me, &he);

    mac = get_cache_entry(ipaddress);

    if (mac == NULL) {//if we cannot figure it out, it can be in distant network
        mac = get_cache_entry(ipgateway); // send frame to gateway
    }

    if (mac == NULL) { //OMG even getting gateways MAC failed!
        mac = "ff:ff:ff:ff:ff:ff"; // As a LR broadcast the packet!
    }

    return mac;
}
```

Příloha B: Výpis zdrojového kódu aplikace Tunneler.java

```
public class Tunneler implements IApplication, ReceiveDataListener {

  private TransmitData received = null; //data prijata z kapsle nebo z Initu
  Guid initGuid = null; //guid initu
  Guid sender = null; //guid odesilatele dat
  //final Object lock = new Object(); //locking zatim nejde

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

    initGuid = applicationAPI.getServerControl().getInitGuid();

    PrintWriter out = applicationAPI.getStdOut(); //ziskame si vystup do stdout

    try {
      Guid toAddr = null; //komu posilame echo
      Guid toNet = null; //do jake site
      NetIdentifier tunnelEnd = null; //identifikator cile=adresa site+adresa nodu
      try {
        toAddr = Guid.parseGuid(args[0]); //prvnim argumentem je cilovy node

        toNet = Guid.parseGuid(args[1]); //druhym je cilova sit

        tunnelEnd = new NetIdentifier(toAddr, toNet); //slozime identifikator
      } catch (NumberFormatException e) {
        out.print(e); //muze se stat, ze se nepodari parsovat args
        return;
      }

      while (this.received == null) {
        //synchronized (lock) {
          Thread.sleep(1); //notify-wait does not function because we need to write custom
classloader to reference application packages
          //wait();
        //}
      }

      if (sender.equals(initGuid)) {
        //odesila init - data jdou kapsli do ciloveho uzlu
        applicationAPI.addOnSendDataListener(guid); //zaregistrovat se pro prijem dat od
kapsle
        CodePkgIdent capsulePkgIdent = applicationAPI.getExecutedCodeIdent(); //pro kapsli
vratime identifikaci prave bezici aplikace
        applicationAPI.injectCapsule(capsulePkgIdent, tunnelEnd, received); //vlozime kapsli
do site

      } else {
        //data odeslala uplne jina aplikace!
        return;
      }

    } catch (Exception e) {
      out.println(e);
    }

  }

  /**
   * The capsule returns carried payload back to application
   * @param  sender
   * @param  data
   */
  public void receiveData(Guid sender, TransmitData data) {
      this.sender = sender;
      this.received = data;
      //synchronized (lock) { //receive - notify nelze
      //  notifyAll();
      //}
  }
}
```

Příloha C: Výpis zdrojového kódu kapsule TunnelerCapsule.java

```
public class TunnelerCapsule implements ICapsule {

  public void main(ICapsuleAPI capsuleAPI) {
    try {
      if (!(capsuleAPI.isThisServer(capsuleAPI.getDestination()))) { // dokud nejsme v cilovem
uzlu, tak se nic nedeje a nechame se pouze smerovat
        return;
      }

      TransmitData data = capsuleAPI.getDataLoad(); //ziska datovy naklad kapsle



      //tunneler hardocded, because getClass.getName() does not work in SAN int yet - we need
to write ClassLoader
      Packet p = new Packet("tunneler",
                            new String[]{
                                   capsuleAPI.getDestination().getAddress().toString(),
                                   capsuleAPI.getDestination().getNetwork().toString()
                            },
                            data.getData()); //get data packet

      capsuleAPI.sendPayload(p); //send deserialized packet to ii for injection



      capsuleAPI.finished();
      return; //ukoncime zivot kapsle

    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```