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

Diplomová práce

Vzdálená šifrovaná správa serveru programovatelné sítě

Plzeň, 2013 Pavel Šlechta

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

V Plzni dne .....

Děkuji vedoucímu diplomové práce Ing. Tomáši Koutnému PhD. za odborné vedení a cenné rady při vypracování diplomové práce.

Abstract

This work focuses on the design of Active Networking node and its encrypted remote management for the SAN (Smart Active Node) project. SAN is an active network server that is developed on the University of West Bohemia. This paper describes new architecture of the second version of SAN and its implementation. Furthermore, the security of remote management is covered. Different algorithms for remote user authentication are compared. An implementation of the SRP authentication protocol is provided. In addition, basic principles of encryption and message authentication codes are covered as well. Next part of this work analyses different methods of local communication which is a necessary component in SAN2 design. A cross-platform inter-process communication library is designed and implemented together with SAN management terminal. Finally the design and implementation of encrypted remote management terminal of SAN node is provided.

Obsah

Úvod

Práce se zabývá aktivními sítěmi, konkrétně jednou z jejich implementací - Smart Acive Node (SAN), která je vyvíjena na KIV FAV ZČU. Cílem práce je návrh šifrované správy sítě serveru SAN. První kapitola se zabývá současnými protokoly pro vzdálenou správu. K dosažení cíle je potřeba nejdříve předělat nevyhovující architekturu serveru SAN. K tomu je třeba napsat jádro serveru SAN zcela od začátku. S tím je spojen návrh a implementace přenosových protokolů, které jsou potřeba pro přenos kapsulí mezi uzly. Vzniká tak SAN2, jehož návrhu a implementaci je věnována třetí kapitola. O obecných principech a ostatních implementacích aktivních sítí pojednává druhá kapitola. V druhé kapitole jsou mj. rozebrány Grade32, ANTS a PLANet.

Důraz je kladen na zabezpečení. Autentizaci, šifrování a podepisování včetně implementace autentizačního protokolu SRP a HMAC je věnována čtvrtá kapitola. Čtvrtá kapitola analyzuje vlastnosti jednotlivých autentizačních protokolů, jejich výhody a nevýhody.

Pro fungování nové architektury serveru SAN bylo zapotřebí navrhnout prostředek meziprocesové komunikace. Jednotlivé typy a podpora prostředků meziprocesové komunikace je podrobně popsána v páté kapitole. Prostředky pro lokální komunikaci, jimiž se práce zabývá, jsou: sdílená paměť, sobory mapované do paměti, pojmenované a nepojmenované roury včetně doménových socketů AF_UNIX. Pátá kapitola končí popisem implementace knihovny CPPL (Cross-Platform Pipe Library), která byla navržena a implementována v rámci této práce. Na základě knihovny CPPL je navržen terminál pro správu uzlu SAN. Následuje návrh vzdáleného šifrovaného terminálu. Závěrečná část je věnována přímému použití Ethernetu pro přenos kapsulí a shrnutí dosažených výsledků.

1 Protokoly pro vzdálenou správu

Protokoly pro vzdálenou šifrovanou správu slouží k nastavení a sledování parametrů nějakého programu, který na daném zařízení běží nebo k nastavení hardwaru připojeného k zařízení na dálku. Např. u síťových zařízení můžeme chtít na dálku změnit IP adresu jednoho z rozhraní, nebo změnit položku ve směrovací tabulce, nebo můžeme chtít sledovat parametry nějakého senzoru připojeného k zařízení. V této kapitole budou popsány nejznámější protokoly pro vzdálenou správu.

1.1 SNMP

Simple Network Management Protocol je protokol určený pro monitorování a správu sítě. Protokol je bezstavový a komunikuje datagramově přes UDP. Protokol rozlišuje klienta, *snmp manager*, a server, *snmp agent*. Klient posílá požadavky na server a server mu odpovídá. Jedinou výjimkou je zpráva **`trap`**, která je asynchronně zaslána od serveru klientovi. Zpráva **`trap`** se zasílá při překročení nastaveného rozmezí nějakého parametru (teploty, rychlosti větráku, počtu paketů za sekundu,...). Pro fungování **`trap`** je však nutné nejprve nastavit adresu, kam se zpráva pošle. Protokol SNMP existuje ve třech verzích. Tyto verze se nadále dělí. Především druhá verze má mnoho variant z hlediska použitých zabezpečovacích způsobů. [1]

1.1.1 SNMP v1

Verze 1 začala vznikat v roce 1988. V roce 1989 vyšlo RFC1157 a v roce 1990 byl protokol standardizován institucí Internet Activities Board (IAB). Standard definoval příkazy **`get`**, **`get next`**, **`set`** a **`trap`**. Příkazem **`get`** se klient dotazuje na hodnotu parametru instance nějakého objektu. Příkaz **`get next`** pak vrátí následující hodnotu po hodnotě uvedeném v předchozím datagramu. Příkazem **`set`** lze hodnotu nastavit. Každý SNMP objekt může mít více instancí - např. počítač může mít víc síťových rozhraní stejného typu. Každé takové rozhraní je pak instance daného SNMP objektu. Každá instance má jednoznačný Object Idnetifier (OID). [2]

OID je uvedeno ve všech SNMP zprávách. Objekty jsou hierarchicky řazeny do stromu. Každý SNMP agent obsahuje seznam podporovaných objektů tzv. *Management Information Base* (MIB). Každý záznam MIB obsahuje jméno objektu, datový typ, oprávnění a krátký popis. Protože hodnoty parametrů se přenášejí po síti mezi různými typy počítačů s různou reprezentací datových typů, používá SNMP kódování *Basic Encoding Rules* (BER), což je zjednodušené kódování *Abstract Syntax Notation One* (ASN.1). [2]

Typ	Délka	Hodnota

tab. 1: Kódování BER

<table>
<tr><td colspan="6">SNMP Zpráva (Sekvence)</td></tr>
<tr><td rowspan="2">SNMP Verze (Integer)</td><td rowspan="2">SNMP Community (Octet String)</td><td colspan="4">SNMP PDU (GetRequest, SetRequest,...)</td></tr>
<tr><td>Request ID (Integer)</td><td>Error (Integer)</td><td>Error Index (Integer)</td><td>VarbindList (Sekvence)
Varbind (Sekvence)
Object Identifier (OID) | Hodnota</td></tr>
</table>

tab. 2: Formát SNMP zprávy

RequestID je náhodné číslo, které uvede SNMP Manager v dotazu. V odpovědi pro daný požadavek uvede SNMP Agent číslo totožné. VarbindList je seznam dvojic {OID, {Typ, Hodnota}}, kde při příkazu SetRequest je stanovena hodnota která se má nastavit a při GetRequestu je tato hodnota 0x00 s délkou 0. Z tab. 2 je vidět, že formát SNMP je závislý na kódování BER. Protokol je bezstavový a tak veškerou potřebnou informaci musí obsáhnout jedna zpráva (jeden UDP datagram).

První verze SNMP nebyla nijak zabezpečena. Stačilo znát Community String. Community String je pole obsažené v každém SNMP paketu obsahující textový řetězec. Tento řetězec má funkci hesla a lze zjistit prostým zachycením libovolného SNMP datagramu. Ve výchozím nastavení většiny SNMP agentů je takovým řetězcem řetězec "public".

1.1.2 SNMPv2

Druhá verze SNMP je pouhé vylepšení první verze. Nová verze např. přinesla

- Příkaz **`GetBulk`** - používá se pro přenos většího množství dat.
- Příkaz **`Inform`**
- Předělaná SMI (*Structure of Managemt Information*), která rozšiřuje možnosti MIB
- Možnost přidávat a odstraňovat řádky tabulky
- Přidává 64 bitové čítače

Verze 2 má několik podverzí - SNMPv2p, SNMPv2c, SNMPv2u, SNMPv2* (hvězdička je v názvu protokolu). Odlišnosti jednotlivých variant spočívají v implementaci zabezpečení. [3]

SNMPv2c

SNMPv2c (c = community) používá stejně jako verze 1 pouze community string. [3]

<table>
<tr><td>4B</td><td>Proměnná délka</td><td>Proměnná délka</td></tr>
<tr><td>Integer</td><td>Octet String</td><td rowspan="2">SNMP PDU</td></tr>
<tr><td>Version Number = 1</td><td>Community</td></tr>
</table>

tab. 3: Formát zprávy SNMPv2c

SNMPv2c nepřináší v bezpečnosti proti SNMPv1 žádnou změnu. Knihovna net-snmp, která je k dispozici v repozitářích většiny majoritních linuxových distribucí, implementuje právě jen variantu SNMPv2c. [4]

SNMPv2p

Verze SNMPv2p (p = party) implementuje tzv. *party-based* bezpečnost. V každém datagramu je uvedena *Destination Party* a *Source Party*. [5] Tyto položky obsahují OID příjemce a odesílatele. Třetí položkou je *Context*, který definuje množinu MIB zdrojů přístupných dané entitě. *Context* je opět typu OID.

4B	Proměnná délka			Proměnná délka
Integer	Sequence of Integer	Sequence of Integer	Sequence of Integer	SNMP PDU
Version Number = 2	Destination Party	Source Party	Context	

tab. 4: Formát zprávy SNMPv2p

SNMPv2p bezpečnost nijak nezajišťuje, protože příslušná pole zprávy nejsou nijak zabezpečena a lze je zjistit prostým odposlechem SNMP zprávy.

SNMPv2u

Verze SNMPv2u (u = user) implementuje bezpečnost na základě uživatelů.

4B	Proměnná délka	Proměnná délka
Integer	Octet String	SNMP PDU
Version Number = 2 (stejná jako u SNMPv2p)	Bezpečnostní parametry	

tab. 5: Formát zprávy SNMPv2u

Bezpečnostní parametry jsou poměrně rozsáhlé, viz **Chyba! Nenalezen zdroj odkazů.**.

Parametr	**Délka**	**Popis**
Model	1	rovná se jedné pro user-based model.

QoS	1	QoS určuje typy zabezpečení.
		`Bity QoS` — `Význam`
		`.... ..00` — `Bez zabezpečení`
		`.... ..01` — `Ověření ano, šifrování ne`
		`.... ..1.` — `Ověření ano, šifrování ano`
		`.... .1..` — `Povolení PDU typu report`

AgentID	12	Unikátní identifikace agenta. Používá se jako obrana proti útoku opakováním.
AgentBoots	4	Počet restartů agenta
AgentTime	4	Počet sekund od restartu agenta
MaxSize	2	Maximální velikost zprávy, kterou je odesílatel schopen přijmout
UserLen	1	Délka položky Username
Username	1 až 16	Uživatelské jméno
authLen	1	Délka položky authDigest
authDigest	0 až 255	Pokud QoS používá ověřování pak: authDigest = HASH_MD5(ZPRÁVA + HESLO), kde heslo má délku 16B.
contextSelector	0 až 40	Kontext, ke kterému přistupujeme

tab. 6: Bezpečnostní parametry SNMPv2c [6]

Délka klíče pro ověření zprávy je stanovena na 16B. Pro šifrování se používá jiný 16B klíč. Jsou tedy celkem 2 různé klíče na uživatele. Klíč se generuje u klienta z hesla podle algoritmu uvedeného RFC1910 Appendix A2.

Ověření autenticity zprávy se provádí spojením zprávy a 16 bajtového klíče. Z těchto dat je vytvořen otisk MD5, který je následně uveden v poli authDigest.

Šifrování používá algoritmus *Data Encryption Standard (DES)*. Šifrovací klíč je rozdělen na 2x8 bajtů. Prvních osm oktetů je použito jako klíč, a to tak, že prvních 56 bitů se použije jako klíč DES a 8 bitů na paritu, kterou ale protokol SNMPv2u nepoužívá. Druhých osm oktetů je použito na inicializační vektor. Při šifrování se zarovnání do bloků provádí libovolnou hodnotou. Při dešifrování jsou tyto informace zahozeny. Mód operace blokové šifry je *CBC (Cipher Block Chaining)*. [6]

SNMPv2* (SNMPv2 star)

SNMPv2* nebylo nikdo standardizováno. Informace o tomto protokolu jsou velmi omezené. V praxi se tento protokol nepoužívá.

1.1.3 SNMPv3

Jednou z hlavních oblastí, které řeší SNMPv3 je opět zabezpečení zpráv. SNMPv3 podporuje integritu a důvěryhodnost zpráv, zavádí autentizaci a řízení přístupu a podporuje šifrování. Tato verze SNMP umožňuje pracovat s různými modely zabezpečení, a to včetně community based modelu.

Výchozí autentizační protokol dle RFC2574, který musí mít všechny verze implementovány je HMAC-MD5-96. [7] Tedy *Hash Message Authentication Code* s hashovací funkcí MD5, jejíž výstup je zkrácen na 96 bitů. SNMPv3 podporuje také šifrování zpráv.

1.1.4 SNMP shrnutí

SNMP se vyskytuje v mnoha variantách. Za bezpečné lze považovat SNMPv2u a SNMPv3 s příslušným bezpečnostním modelem. Všechny verze používají stejné SNMP PDU.

1.2 Common Management Information Protocol(CMIP)

CMIP je protokol, který vznikal jako konkurence SNMP. Protokol je specifikován ITU Recommendation X.711 (03/91)[1]. V dnešní době je protokol dle ITU již zastaralý. Protokol byl především používán u telefonních zařízení. V sítích TCP/IP byl vytlačen protokolem SNMP.

[1] Specifikace je placená.

1.3 Protokol TELNET

Protokol TELNET se používá pro konfiguraci a správu síťových zařízení (směrovačů apod.). Cílem protokolu TELNET je poskytnout osmibitový obousměrný textově orientovaný terminál (*NVT - Network Virtual Terminal*). TELNET používá protokol TCP s výchozím portem 23. Klient je zodpovědný za překlad kódu z klávesnice do NVT. NVT používá 7 bitové kódy pro znaky. Tyto znaky jsou přenášeny po 8 bitech s nejvýznamnějším bitem nastaveným na hodnotu 0. Od zobrazovacího zařízení (v terminologii telnetu *printer*) je pouze vyžadováno, aby podporovalo dolní sadu ASCII znaků. [8]

Jméno	Kód	Dekadická hodnota	Funkce
NULLL	NUL	0	Žádná operace
Zalomení řádku	LF	10	Přesun na další řádek se zachováním horizontální pozice
Návrat vozíku	CR	13	Přesun na začátek řádku

tab. 7: Základní povinné kódy NVT

Kromě výše uvedených kódů, které jsou povinné, definuje NVT následující kódy:

Jméno	Kód	Dekadická hodnota	Funkce
BELL	BEL	7	Vyvolá zvukový signál, nedojde k posunu kurzoru
Back Space	BS	8	Přesune kurzor o jednu pozici vlevo
Horizontální tabulátor	HT	9	Přesune kurzor na další horizontální zarážku. Pozice zarážek není specifikována.
Vertikální tabulátor	VT	11	Přesune kurzor na další vertikální zarážku. Pozice zarážek není specifikována.
Form feed	FF	12	Přesun na další stránku. Horizontální pozice kurzoru zůstane zachována.

tab. 8: Nepovinné kódy NVT

1.3.1 Příkazy v protokolu Telnet

Příkazy jsou 8 bitové s nejvýznamnějším bitem nastaveným na 1. Všechny příkazy jsou uvozeny kódem IAC (*Interpret as Command*) jehož hodnota je dekadicky 255. Mezi příkazy patří např. smazání řádku (EL 248) nebo smazání znaku (EC 247). [8]

1.3.2 Zabezpečení protokolu TELNET

Z hlediska bezpečnosti není TELNET nijak zabezpečen. Používá se prosté TCP spojení. Pro projekt SAN je protokol TELNET jako takový zcela nevhodný, ale dá se použít v případě, že je přenášen bezpečným kanálem např. protokolem SSH.

1.4 Protokol RLOGIN

RLogin je utilita pro vzdálenou správu UNIXových systémů. Protokol RLOGIN, podobně jako TELNET, poskytuje vzdálený virtuální terminál. RLOGIN explicitně vyžaduje použití TCP. Výchozí port je 513.

Po připojení klient pošle serveru 4 řetězce ukončené nulou.

```
<null>
client-user-name<null>
server-user-name<null>
terminal-type/speed<null>
```

Server odpoví <null> a tím dá najevo, že tyto řetězce přijal. (Poté může následovat dohoda o velikosti okna.)

Protokol rozlišuje dva režimy - *cooked* a *raw*. Komunikace začíná v režimu *cooked*. V tomto režimu jsou znaky START a STOP (většinou ASCII DC1 a DC2) interpretovány jako začátek a konec znaků, které mají být tištěny klientovy na terminál. [9]

V režimu *raw* nejsou znaky START a STOP zpracovávány a jsou poslány dál terminálu. Řídící znak (výchozí je ~) se musí vyskytovat na začátku řádky tj. po znacích CR LF, po něm následuje 1 bytový kód. [9]

Stejně jako v případě TELNET, není protokol RLOGIN nijak zabezpečen. Veškerá hesla se dají odchytit prostým odposlechem TCP spojení.

1.5 Secure Shell (SSH)

Protokol SSH (*Secure Shell*) není protokolem pro správu v pravém slova smyslu, protože přímo neumožňuje sledování parametrů nějakého monitorovaného objektu. Protokol SSH se sestává ze tří částí - zabezpečený terminál, bezpečný přenos souborů (SCP - Secure Copy) a tunelování spojení.

1.5.1 Autentizace v SSH

Předtím, než budou popsány metody autentizace, je nutné si uvědomit zcela zásadní rozdíl, a to rozdíl, mezi autentizací serveru a autentizací uživatele. Při použití SSH je nutné nejprve autentizovat server a až poté teprve uživatele. Po úspěšné autentizaci serveru je možné vytvořit bezpečný komunikační kanál, kterým budou posílána veškerá další data včetně údajů potřebných pro autentizaci uživatele.

Protokol SSH 2.0 je závislý na asymetrické kryptografii. Vyžaduje použití asymetrické kryptografie pro zajištění prvotního vytvoření bezpečného kanálu. To lze realizovat dle RFC4253 u SSH dvěma způsoby. Oba způsoby předpokládají, že na serveru se nachází pár klíčů (veřejný a soukromý).

První způsob je že, když se chce klient přihlásit na server a je mu nejprve zaslán serverem jeho veřejný klíč. Program klienta pak následně spočte jeho otisk, a pokud ho nezná, tak jej zobrazí na obrazovku s požadavkem, aby uživatel rozhodl, zda daný otisk skutečně patří pravému veřejnému klíči a žádný nebyl podstrčen útokem *Man in the Middle*. Pokud uživatel souhlasí s pravostí klíče, je úspěšně vytvořen zabezpečený tunel. Variantou tohoto způsobu je mít tabulku <server, veřejný klíč serveru>.

Druhým způsobem je využití PKI (*Public Key Infrastructure*). Klient má pak nainstalovaný kořenový certifikát certifikační autority. Po připojení k serveru je klientovi zaslán certifikát. Klient ověří platnost certifikátu - především zda-li ho skutečně podepsala důvěryhodná certifikační autorita, datum platnosti a zda nebyl odvolán (k tomu je ale potřeba získat *Certificate Revocation List* (CRL)).

Ve výsledku je cíl obou metod stejný - verifikace pravosti veřejného klíče serveru. Celý problém vytvoření kanálu mezi serverem a klientem je pak problém distribuce klíče. Problémem distribuce klíče se architektura SSH vůbec nezabývá:

"This protocol makes no assumptions or provisions for an infrastructure or means for distributing the public keys of hosts. It is expected that this protocol will sometimes be used without first verifying the association between the server host key and the server host name. Such usage is vulnerable to man-in-the-middle attacks. This section describes this and encourages administrators and users to understand the importance of verifying this association before any session is initiated."

[10 str. 16]

Po úspěšném ověření pravosti veřejného klíče lze vytvořit bezpečný kanál. Nicméně server pořád nezná identitu klienta. Proto musí proběhnout autentizace uživatele. Protokol SSH umožňuje použít různé metody autentizace:

Identifikátor autentizační metody	Podpora implementace	Popis
`publickey`	Musí být implementován	Ověření na základě znalosti soukromého klíče uživatele
`password`	Volitelný	Ověření na základě hesla poslaného v otevřené podobě
`hostbased`	Volitelný	Ověření na základě zdrojové adresy
`none`	Nedoporučovaný	Žádné ověření

tab. 9: Metody ověření uživatele dle [11]

Metoda *Password Authentication* spoléhá na to, že transportní kanál je bezpečný a klient heslo posílá v otevřené podobě tímto kanálem. Pokud heslo souhlasí, je klient autentizován.

Metodou *Publickey* ověřuje server uživatele na základě znalosti jeho pravého veřejného klíče. Uživatel/klient zná pak svůj soukromý klíč.

Protokol SSH připouští rozšíření autentizačních metod. Další metodou uživatelské autentizace může být využití protokolu SRP (Stanford *Secure Remote Password Protocol*).

Existuje patch do OpenSSH doplňující autentizační mechanismus o protokol SRP[2]. Protokol SRP je podrobně popsán v sekci 4.3.4.

1.5.2 Šifrování a integrita zpráv

RFC4521 definuje několik metod šifrování a metod zabezpečení zpráv proti pozměnění během cesty. Jediným povinným algoritmem šifrování je 3DES[3] v režimu *Cipher Block Chaining*. Stejně tak jediným povinným algoritmem zajištění integrity zpráv je HMAC-SHA1. Jak šifrování, tak i integritu zpráv lze zcela vypnout.[4] Možnosti šifrování a zabezpečení zpráv jsou poměrně rozsáhlé a vzhledem k použitým algoritmům (AES, Twofish a Serpent) je zabezpečení na vysoké úrovni.

1.5.3 Použití v projektu SAN

Výhodou protokolu je jeho rozšířenost na platformách Linux. Na Windows lze jako SSH klient použít např. program PuTTy, který je možné používat zdarma a je k dispozici i se zdrojovým kódem. Existují i servery pro Windows, např. Bitvise WinSSHd nebo freeSSHd.

Jméno	Linux	Windows	Otevřený kód	Zdarma
Bitvise SSH Server	Ne	Ano	Ne	Ne
freeSSHd	Ne	Ano	Ano	Ano
GSW SSH2 Server	Ne	Ano	Ne	Ne
Kpym	Ne	Ano	Ano	Ano
MobaSSH Home	Ne	Ano	Ne	Ano
MobaSSH Professional	Ne	Ano	Ne	Ne
OpenSSH	Ano	Pouze Cygwin	Ano	Ano

tab. 10: Srovnání SSH Serverů

Nevýhodou při použití SSH je závislost naprosté většiny implementací na transportním protokolu TCP/IP. Projekt SAN však staví na principu aktivních sítí, které měly nahradit právě protokol IP. Protokol by se tedy dal provozovat pomocí SSH tunelování [12], ale bylo by to ad-hoc řešení, o které není usilováno. Cílem je šifrovaná správa SAN serveru pomocí nativního protokolu SAN sítě.

[2] http://freecode.com/projects/opensshsrp
[3] Data Encryption Standard použitý 3x po sobě s různým klíčem
[4] Není to však z pochopitelných důvodů doporučováno.

2 Aktivní sítě

Jedno z možných dělení počítačových sítí je dělení sítí na sítě aktivní (nebo také programovatelné) a sítě pasivní. Úkolem pasivní sítě je přenos zpráv z místa A do místa B. Typickým příkladem protokolů pro pasivní sítě jsou IPv4 a IPv6. Naproti tomu aktivní síť spolu přenáší tzv. kapsule. Kapsule asociuje s přenášenými daty kód programu, který může být vykonáván na uzlech, směrovačích a jiných aktivních prvcích během jeho přenosu. Aktivní sítě tak umožňují mnohem rychlejší nasazení protokolů než pasivní sítě.

Aktivní sítě vznikly z důvodu rychlého nasazení nových služeb. Koncept programovatelných sítí vznikl v DARPA - Defense Advanced Research Project Agency.[12] Cílem bylo vytvořit síť, která by překonala současná omezení tradičních sítí jako např. IP.

Jako klíčové problémy byly identifikovány [13]:

- Exitující sítě nepodporují rychlé nasazení nových protokolů a služeb.
- Síťová funkcionalita je roztroušena mezi směrovače, přepínače, koncové uzly, spojovací zařízení a aplikace, které zamezují získání koherentního stavu a vytváří tak síť, která se velmi obtížně spravuje.
- Přenosná zařízení musejí být považována za statická, protože neexistuje způsob, jak zajistit transparentní přenos zařízení mezi jednotlivými sítěmi.
- Není možné nasadit nové služby. V dosavadních sítích jsou služby pevně zabudovány do jader operačních systémů zabraňujíc tak dynamickému přizpůsobení konkrétním aplikacím.

Pokud zvážíme síť založenou na protokolu IP jako reprezentanta tradičních sítí, první nevýhodou je potřeba vzájemné globální dohody jak vše bude fungovat, což je důsledek pasivity sítě. Učebnicový příklad je IPv6. Každý paket se skládá ze dvou částí - hlavička a obsah. Zatímco obsah jsou uživatelsky definovaná data, hlavička obsahuje informaci, jak data zpracovat. Pokud zúčastněné uzly nerozumějí hlavičce, není možné uskutečnit spojení. Problém může nastat i s položkou Next Header protokolu IPv6, pokud by ho uzel sítě neznal a z tohoto důvodu by paket zahodil. Stejným problém má i IPv4, VLAN enkapsulace, ... [13]

Aktivní sítě tento problém nepostihuje, protože každá datová jednotka je spjata se spustitelným kódem, který je vykonán na každém uzlu. Kód definuje, jak zacházet s jednotkou. Tím pádem každý uzel ví, jak s daným paketem zacházet. Nové protokoly a služby můžou být ihned nasazeny podle potřeb uživatele. Pouze dvě věci je nutno standardizovat, a to kód a jeho distribuci.

Klíčovým problémem aktivních sítí je to, že neexistuje implementace, která by byla použitelná a dostatečně bezpečná.

Na to aby vše mohlo fungovat spolehlivě, je třeba mít vybudovaný bezpečný *Code Execution Environment*. Ten musí kromě oprávnění také zajišťovat přidělení zdrojů - strojového času a množství paměti.

2.1 Možné aplikace

2.1.1 Vysílání multimedií

Mezi možné oblasti použití patří např. streamování multimedií velkému počtu účastníků s minimálním datovým tokem. Dnes je většina vysílání na internetu realizována vícenásobným unicastem, který zabere velmi velkou kapacitu linek. IP multicast lze prakticky realizovat jen ve velmi omezené oblasti sítě. Většina směrovačů multicast ignoruje. Multicast musí být explicitně nastaven na každém směrovači a to pochopitelně není možné realizovat na celém internetu. Jednak z bezpečnostních důvodů a jednak z politických. Naproti tomu v aktivních sítích lze síť naprogramovat např. tak, že v daném uzlu se data rozdělí tak, že na každý port, za kterým bude alespoň jeden účastník, bude vysílán jen jeden proud dat nehledě na počet účastníků skupiny. Toto opatření výrazně šetří kapacitu linek.

2.1.2 Sítě odolné proti poruchám

Další oblastí jsou sítě odolné proti poruchám. V aktivních sítích to lze řešit např. tak, že data jsou posílána několika různými cestami. Při výpadku jedné cesty jsou zbylé cesty k dispozici. V IP sítích toto lze řešit jen v omezené míře, protože nemůžeme předem určit jakou cestou (přes které směrovače) pakety půjdou. Toto lze v omezené míře uskutečnit v IPv6, ovšem je pravděpodobné, že většina správců tuto funkci z bezpečnostních důvodů ponechá vypnutou, stejně jako je tomu u IPv4 v případě strict „source and record route" a "loose source and record route".

Internet ve vesmíru je velmi náchylný na různé rušení a navíc poloha družic se mění v čase. Dochází k častým výpadkům linek. Proto je nutné realizovat záložní mechanismy, alternativní cesty. To nelze klasickými IP sítěmi řešit. Na IP směrovačích musí být nainstalován speciální software.

2.1.3 Překódování signálu během cesty

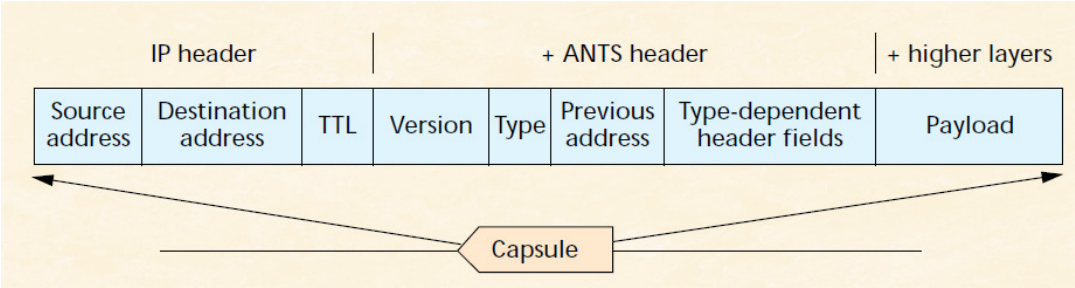
Překódování signálů lze v aktivních sítích řešit přímo na daném uzlu aktivní sítě. Příkladem může být video signál ve vysokém rozlišení, který bude šířen drátovou sítí, ale uživatelé připojení bezdrátově nemají dostatečnou kapacitu linky. Na hraničním směrovači pak může být snížena kvalita, aby video hrálo plynule.

2.2 Existující implementace

2.2.1 ANTS

Cílem projektu bylo zjistit, zda má vůbec cenu se aktivními sítěmi zabývat. Jeho hlavní oblastí zájmu byly směrovací algoritmy založené na roji mravenců (*ARA - The Ant-Colony Based Routing Algorithm*) a dynamické algoritmy směrování pro ad-hoc mobilní sítě. Síť funguje nad IP a identifikátory uzlů odpovídají jejich IP adresám. Uzly jsou dvojího typu, a to aktivní a pasivní. Pasivní uzly data pouze přeposílají. [14]

Implementace ANTS je založena na programovacím jazyce JAVA. K transportu dat používá tzv. kapsule, jejichž formát je na obr. 1.



obr. 1: Formát kapsule ANTS. [14]

Na obr. 1 je vidět formát kapsule ANTS. Každá kapsule začíná běžnou IP hlavičkou. ANTS hlavička obsahuje verzi, typ přeposílacího podprogramu (*forwarding routine*), který říká, jak se má zacházet s kapsulí na aktivním uzlu a adresu předchozího uzlu. Další část hlavičky je závislá na jejím typu. Na konci kapsule jsou samotná data.

Protože se program na daném uzlu nemusí nacházet, předpokládá se, že uzel, ze kterého kapsule přišla, ji musel vykonat a tudíž na něm program je. Program je tedy nahrán z předchozího uzlu.

V ANTS vývojář aplikace postaví síťovou službu kombinací různých kapsulí a vytvoří tím protokol. V každé kapsuli je pak odkaz na směrovací podprogram, který říká, jak se s danou kapsulí bude na aktivním uzlu zacházet. Odkaz je ve formě MD5 otisku programu, konkrétně se jedná o třídu v Javě. Kapsule v rámci stejného protokolu můžou komunikovat mezi sebou na základě stavu uloženého na aktivních uzlech. Např. jeden typ kapsule může nastavit informaci o umístění mobilního klienta.

ANTS obsahuje mechanismus oddělení kapsulí různých protokolů, tak aby se navzájem neovlivňovaly. Zásadní problém tkví v tom, že směrovací podprogram může být vir. I když omezíme funkce, které je možno z podprogramu volat a omezíme přístup k paměti, je pořád třeba se zabývat otázkou, jak úlohy plánovat. Cílem viru může být uzel zahltit např. posíláním dat do smyčky nebo vytížit procesor natolik, že nezbude čas na ostatní úlohy.

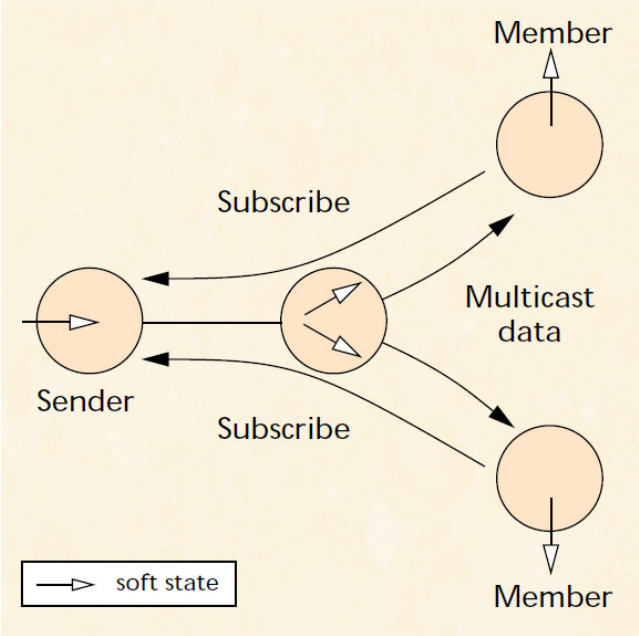
ANTS má 10 základních/primitivních volání, které může aktivní kód kapsule obsahovat. [14] Tyto rutiny ANTS dělí na:

- *Environmental Calls* - vracejí informaci o místním uzlu. Např. jeho adresu.
- *Storage Calls* - modifikují stav uživatelsky definovaných objektů. Používá se pojem *soft-store*, protože není garantováno, jak dlouhou budou data uložena.
- *Control Operations* - směřují kapsule na různá rozhraní

Čím víc přidáme systémových volání, tím více bude aktivní síť flexibilnější a tím většímu bezpečnostnímu riziku se vystavujeme.

Distribuce kódu

V rámci aktivní sítě musíme zajistit distribuci kódu kapsulí, protože nelze předem znát všechny programy. ANTS pro distribuci kódu používá odděleně. Maximální velikost kódu, kterou ANTS povoluje je 16kB.



obr. 2: Distribuce kódu v ANTS. K distribuci je používán multicast. Soft-store ukazatele ukazují, kam se bude kód posílat. [14]

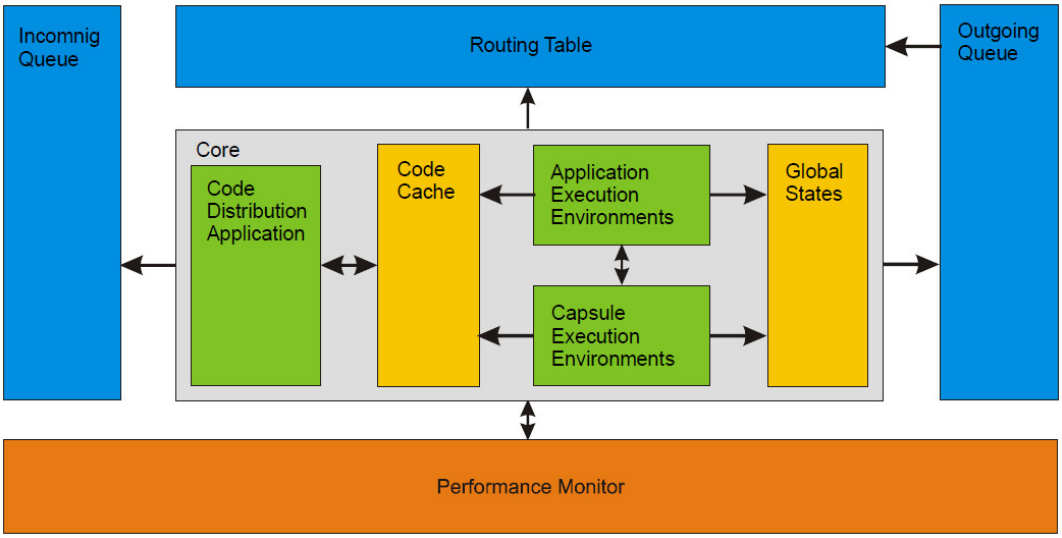
Bezpečnost ANTS

Implementace ANTS se nezabývá otázkou bezpečnosti. Zcela spoléhá na vlastnosti jazyka Java.

2.2.2 Grade32

Z důvodů otestování nové metody přerozdělování zátěže distribuovaných aplikací v heterogenním prostředí vznikla na Západočeské Univerzitě v Plzni implementace aktivního uzlu Grade32.

Grade32 je napsaný v programovacím jazyku Object Pascal a funguje na operačním systému Microsoft Windows. Strukturu Grade32 znázorňuje obr. 3.



obr. 3: Architektura aktivního uzlu Grade32

Modré bloky představují konektivitu k ostatním uzlům. Oranžový *Performance Monitor* zaznamenává množství zdrojů, které je k dispozici. Šedý střed představuje jádro uzlu. Jádro obsahuje implementaci protokolu na distribuci kódu, kódovou mezipaměť pro zrychlení načítání kódu, prostředí pro spouštění aplikací, prostředí pro spouštění kódu a globální úložiště stavu. Příchozí fronta, odchozí fronta a distribuce kódu mají svá vlastní vlákna.

Komunikace s ostatními uzly probíhá přes IP. Je testován otisk příchozích kapsulí proti databázi kódů známých protokolů. Příchozí kapsule se řadí do fronty. Jedná se o problém producent-konzument. Pokud je fronta prázdná a přijde kapsule, je signalizována jádru událost. Jádro vybírá z fronty kapsule. Pokud byla fronta plná a jádro vyjmulo kapsuli z fronty, signalizuje událost. Kapsule je reprezentovaná objektem, který je serializovaný a později jsou z něj data extrahována. Jádro posílá kapsule k odeslání do odchozí fronty, odkud jsou přeposílány na příslušné rozhraní podle směrovací tabulky.

Performance monitor periodicky zaznamenává informace o využitých zdrojích a prostředích pro vykonávání kapsulí.

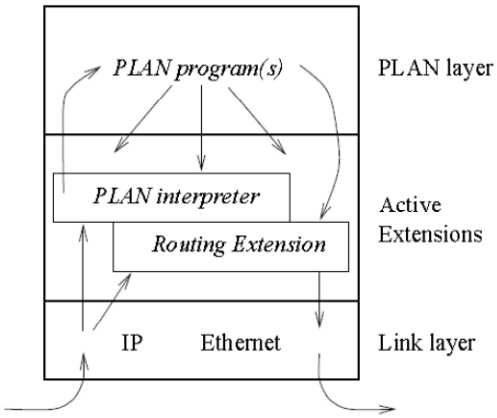
Distribuce kódu je řešena zabudováním jednoho protokolu do jádra. Protokol má jenom dva typy kapsulí. Jednu pro požadavek na kód a druhou jako odpověď, která nese kód.

Aktivní uzel udržuje globální stav, který leží mimo *code-execution environment*, takže přetrvává i po ukončení vykonávání kódu. Stavy jsou identifikovány pomocí GUID, které jsou náhodné. Pro modifikaci stavu proměnné je třeba znát její GUID. Z bezpečnostních důvodů není povolen výčet přidělených GUID.

Jelikož Grade32 byl vyvinut zejména pro testování nové metody rozdělení zátěže, není dále vyvíjen. Grade32 demonstruje základní architekturu serveru programovatelné sítě.

2.2.3 PLANet

V síti PLANet jsou uzly identifikovány 48 bitovou adresou. 32 bitů tvoří IPv4 adresa a 16 bitů port UDP. Komunikace probíhá přes UDP/IP. Při přijetí paketu zjišťuje uzel, zda se jedná o uzel uvedený v poli evalDes. Pokud uzel není cílovým uzlem, pak se z položky routeFun zjistí, která směrovací funkce se má použít. Ta potom zajistí směrování paketu. Pokud paket dorazil na uzel, který má uveden v položce evalDes (viz obr. 5), načte se program z pole chunk do interpretu PLANetu a spustí se na daném uzlu. PLANnet implementuje jako jeden způsob směrování kapsulí distance vector algoritmus založený na protokolu RIP.



obr. 4: Architektura uzlu PLANet [15]

Hlavička							Chunk sekce		
evalDes	source	rb	session	flowID	foundFun	handler	execFn	bindings	Code

obr. 5: Struktura paketu sítě PLANet [15]

- evalDes - adresa (cílového) uzlu na kterém bude paket spuštěn
- source - zdrojová adresa
- rb - Resource Bound - přidělené množství globálních zdrojů
- session - ID sezení
- flowID - identifikátor toku
- routFun - směrovací funkce, která bude použita

- handler - obslužná funkce, která se volá v případě výjimky
- bindings - inicializace proměnných
- code - program v jazyce PLAN

Identifikátor toku (flowID) se využívá v QoS (Quality of Services) a identifikuje toky, které k sobě patří.

Síť PLANet umožňuje vykonávání paketu pouze v cílovém uzlu. Chování během cesty se dá měnit pouze výběrem směrovací funkce. Není tedy přímo možná žádná manipulace s daty v paketu během cesty.

Jazyk PLAN byl vyvinut pro potřeby aktivních sítí. PLAN je funkcionální interpretovaný skriptovací jazyk s omezením zdrojů. [16] Umožňuje volat jen předem dané služby po určitou dobu a s určitým množstvím paměti. Například není možné udělat nekonečnou smyčku či iteraci, jejíž délka není pevně dána. Každý program je od ostatních izolovaný a jsou nastaveny čítače, kolik paketů už daný program odeslal. Při překročení některého z bezpečnostních čítačů je program ukončen. Jazyk PLAN již není vyvíjen. [16]

2.3 Shrnutí

V současnosti je k dispozici několik experimentálních serverů aktivních sítí. Grade32 není primárně určena jako aktivní síť, ale pro ověření funkčnosti metody přerozdělování zátěže. PLANet se zabývá myšlenkou specializovaného jazyka pro spouštění kódu v aktivních sítích.

Implementace se od sebe liší především použitým prostředím pro spouštění aktivního kódu. ANTS používá Java třídy a neřeší, jaké metody bude tato třída volat a nijak nezamezuje nekonečným smyčkám.

PLANet používá interpretovaný jazyk PLAN. PLAN omezuje veškeré iterace, počet odeslaných dat a metody, které lze volat.

Grade32 používá DLL knihovny. Spoléhá se na databázi otisků. Pokud program není v databázi otisků, pak není spuštěn.

3 Projekt Smart Active Node

3.1 SAN1 a SAN2

Z důvodů výrazných změn v architektuře uzlu SAN bylo rozhodnuto v rámci této práce napsat nový server označovaný jako SAN2. SAN2 se nezakládá na kódu SAN1.[5] SAN2 vychází z poznatků získaných během vývoje SAN1. Následující text v této kapitole se bude výhradně zabývat nově navrženým a implementovaným jádrem uzlu SAN2. Důraz bude kladen na síťovou vrstvu, jejíž implementace byla realizována.

3.2 Architektura SAN2

Uzel SAN2 se skládá z několika vrstev. Nejnižší vrstvu tvoří vrstva síťová, jež zajišťuje komunikaci se sousedními uzly. Vstupní fronta a výstupní fronty obsahují kapsule. Účel front je nezdržovat směrovač uzlu zbytečným čekáním na odeslání či příjem kapsulí.

obr. 6: Architektura uzlu SAN2

Distribuce kódu zajišťuje, aby kód pro danou kapsuli byl na serveru. Jednoduchým protokolem pro distribuci kódu je požádat sousední uzel, ze kterého kapsule přišla o zaslání kódu, protože můžeme předpokládat, že pokud kapsule přišla z daného uzlu, uzel ji musel vykonat a k tomu musel mít k dispozici kód.

Repozitář obsahuje spustitelné kódy s jejich otisky[6]. Jedná se o vyrovnávací paměť. Bylo by velmi neefektivní pro každou kapsuli získávat kód znovu.

[5] Změny starého kódu na novou architekturu by zabraly více času, než napsat server od začátku.
[6] SHA1 hash

Prostředí pro vykonávání kapsulí zajišťuje bezpečný běh kapsulí v sandboxu. Při spouštění kódu kapsulí musí být omezené systémové prostředky. Zejména doba vykonávání, množství odeslaných dat, vytížení procesoru a omezený nebo zcela znemožněný přístup k systémovým voláním. Měl by také existovat obecný mechanismus, jak spouštět kapsule na různých operačních systémech a architekturách.

Plánovač úloh zajišťuje rovnoměrné rozdělení práce mezi tzv. workery. Workery jsou samostatné procesy, které vykonávají kód kapsulí v sandboxu mimo proces uzlu SAN.

Aplikace poskytují služby. Aplikace mohou spouštět libovolný kód a injektovat do systému libovolné množství kapsulí. Příkladem aplikace může být i směrovací algoritmus, který mění dynamicky směrovací tabulku. Příkladem další aplikace může být vzdálená správa systému.

Pro potřeby vzdálené šifrované správy serveru SAN je nutné minimálně implementovat síťovou část spolu s vstupními a výstupními frontami včetně jednoduchého směrovacího algoritmu. Implementovat prostředí pro spouštění kapsulí je mimo oblast a časové prostředky této práce stejně jako dynamické směrovací protokoly a plánovače úloh. V následujících odstavcích bude popsán návrh a implementace síťové vrstvy SAN2 s vysvětlením použitých prostředků a protokolů.

3.3 Síťová vrstva

V této sekci bude popsán návrh přenosového protokolu pro přenos kapsulí mezi sousedními uzly.

Pro spolehlivý přenos je tedy nutné nad kapsulemi (viz obr. 7) vybudovat spolehlivý přenosový protokol.

3.3.1 Formát kapsule

Nejprve je třeba zvolit vhodný formát kapsule.

16B	16B	1B	1B	20B	Proměnná délka
dstAddr	srcAddr	hop	flags	appHash	data

obr. 7: Formát kapsule SAN

dstAddr - cílová SAN adresa[7]
srcAddr - zdrojová SAN adresa
hop - maximální počet uzlů, kterými může kapsule projít
flags - vlajky

[7] SAN adresa a IP adresa mezi sebou nemají žádnou přímou vazbu

appHash - identifikátor aplikace
data - obsah kapsule

V současné implementaci SAN2 existují pouze 2 vlajky. Vlajka DX (Destination eXecute) říká, že aktivní kód se má spustit až na cílovém uzlu. Vlajka DS (Destination Send) říká, že kapsule nebude vykonávána a data budou přenášena ze zdroje do cíle výchozím směrovacím protokolem.

Při použití vlajky DS se datová sekce skládá ze tří částí - cílového portu aplikace, zdrojového portu aplikace a samotných dat. Port je 16 bitové neznaménkové číslo.

3.3.2 Přenosový protokol

SAN používá jako transportní protokol TCP/IP. Transportní protokol TCP je proudový a nerozlišuje hranice zpráv. Zároveň protokol TCP rozlišuje klienta a server (klient používá funkci connect(), kdežto server listen() a accept()). Oba tyto problémy musí přenosový protokol řešit.

Protokol UDP rozlišuje hranice zpráv a nerozlišuje klient a server. Protokol UDP je nespolehlivý a tak data můžou dorazit v jiném pořadí, duplicitně, poškozená nebo vůbec. UDP umožňuje maximální velikost dat jenom takovou, jako maximální MTU nižší ISO/OSI vrstvy. Ethernet má MTU 1500B z toho je třeba odečíst veškeré hlavičky - IP a UDP. Pro přenos větších kapsulí bychom při použití UDP museli vytvořit spolehlivý přenosový protokol. Logickým závěrem je, že se jako výhodnější jeví spolehlivý transportní protokol TCP.

3.3.3 Řešení problému TCP Klient-Server

Kapsule jsou přenášeny mezi rovnocennými uzly. Protokol TCP je nesymetrický. Rozlišuje server a klient. Pokud budeme uvažovat uzly serveru SAN A a B, pak lze uvažovat, že kdykoliv budeme chtít vytvořit spojení mezi uzly A a B, tak nevíme, který z uzlů bude spuštěn jako první. Proto je nutné zajistit, aby se uzel A mohl připojit k uzlu B a naopak.

Řešením je symetrizace TCP spojení. Té docílíme tak, že oba uzly se zároveň chovají jako TCP klient a TCP server. Z toho vyplývá, že je nutné vytvořit dvě TCP spojení mezi uzly. Jedno spojení, které jen vysílá kapsule a druhé, které jen přijímá kapsule. Kapsule vždy vysílá TCP klient a přijímá je TCP server.

3.3.4 Dělení zpráv v TCP

Pro rozlišení začátku a konce zpráv v TCP lze použít několik metod. Jednoduchou metodou je na začátku každého vysílání zprávy uvést délku této zprávy. Pole délka zprávy musí mít pevně danou délku. Pro potřeby SAN bylo zvoleno 32 bitů. Délka se přenáší jako unsigned integer v netorder, tj. big endian. Př. viz obr. 8.

Start	Délka	Zpráva	Start	Délka	Zpráva
1B	4B	10B	1B	4B	3B
0x53	0x0000000A	abcdeabcde	0x53	0x00000003	ghi

obr. 8: Příklad přenášení zpráv v proudovém transportním protokolu

Synchronizační preambule slouží jako ochrana, proti ztrátě synchronizace – jenže jako jeden byte je krátká, 16B by bylo výrazně lepších. V případě TCP lze vynechat, no, vždy asi ne, a ušetřit množství přenesených dat.

3.3.5 Vrstva RPC

Výše uvedené dělení zpráv by stačilo pro přenos kapsulí mezi sousedními uzly. Zprávy by byly prosté kapsule. Uzel SAN potřebuje také komunikovat s aplikacemi na lokálním systému. Aplikace na lokálním systému může přijímat a injektovat kapsule do sítě. K tomu potřebuje např. znát svoji zdrojovou adresu, která je rovna adrese některého z rozhraní uzlu. Aplikace si také může zaregistrovat příchozí port. Z tohoto důvodu se nejeví jako rozumné omezovat obsah zpráv pouze na kapsule - zejména pak u komunikace aplikace-uzel.

Z tohoto důvodu jsem navrhl jednoduchou mezivrstvu RPC. Vrstva RPC pracuje s datagramy. Je jedno jakou používá nižší vrstvu, stačí implementovat jednoduché rozhraní:

```
class San2::Rpc::CIRpcChannel
{
	// TRUE = success
	virtual bool sendDatagram(const San2::Utils::bytes &data)=0;
	virtual bool recvDatagram(San2::Utils::bytes &out, unsigned int timeoutMsec)=0;
}
```

Samotný protokol RPC má jednoduchý formát:

ID funkce	Data
2B (unsigned int BE)	Proměnná délka
<Číslo>	<Data Funkce>

obr. 9: Formát RPC zprávy

ID funkce určuje, která funkce se má zavolat. Samotná data v RPC zprávě jsou specifická pro danou funkci. Funkce je v systému definována dvěma třídami, které dědí příslušné bázové třídy. Jeden objekt používá dotazovatel (RpcInvoker) a jeden vykonavatel (RpcExecutor). Bázové třídy a jejich virtuální funkce:

```
class San2::Rpc::CIRpcSyncFunctionOut
{
	virtual unsigned int getUniqueId()const=0;
	virtual bool pack(San2::Utils::bytes &out)=0;
	virtual bool parseResponse(const San2::Utils::bytes &in)=0;
};

class San2::Rpc::CIRpcSyncFunctionIn
{
	virtual unsigned int getUniqueId()const=0 ;
```

```
        virtual bool operator()(void)=0;
        virtual bool unpack(const San2::Utils::bytes &in)=0;
        virtual San2::Utils::bytes getResponse()=0;
};
```

Pořadí volání	Jméno členské funkce	Význam
	getUniqueId	ID funkce, je v systému unikátní a pro třídy _In a _Out musí být totožné. Podle tohoto čísla hledá příslušnou funkci CRpcExecutor.
1	Pack	Serializace dat. CRpcInvoker volá tuto funkci, aby získal data, která má poslat
2	Unpack	Desrializace dat. Voláno CRpcExecutorem těsně před voláním samotné funkce operator().
3	operator()	Tělo samotné funkce, která se vykoná.[8] Zároveň nastaví příslušné atributy objektu, ze kterých bude sestavena funkcí getResponse() odpověď, která se má zpět poslat klientovi.
4	Bytes getResponse	Voláno třídou CRpcExecutor. Ta si tak vyžádá odpověď, kterou má poslat klientovi.
5	parseResponse	Voláno třídou CRpcInvoker. Slouží k předání odpovědi do funkce/objektu. Tato metoda většinou nastaví odpověď do atributů objektu. Klient si ji pak vyžádá příslušným getrem.

Příklad Funkce na násobení dvou čísel:

```
#define MULTIPLY_ID 124

// ----- MultiplyOut -----
MultiplyOut::MultiplyOut(int x, int y) : m_x(x),m_y(y)
{
}

unsigned int MultiplyOut::getUniqueId()const
{
        return MULTIPLY_ID; // unikatni id funkce
}

bool MultiplyOut::pack(San2::Utils::bytes &out)
{
        out = San2::Utils::CDataPack::pack(m_x) + San2::Utils::CDataPack::pack(m_y);
        return true;
}

SAN_INT32 MultiplyOut::getResult()
{
    return m_result;
```

[8] V případě funkce SendCapsule, vloží operator() kapsuli do příslušné vstupní fronty na uzlu. Fronta je předána funkci tak, že je předána v konstruktoru a je zkopírován odkaz na frontu do atributů objektu. Zároveň funkce SendCapsule není implementována jako "sync" tj. nevrací hodnotu, takže neblokuje volajícího, který nemusí čekat na odpověď.

```
}

bool MultiplyOut::parseResponse(const San2::Utils::bytes &in)
{
    m_result = San2::Utils::CDataPack::unpackInt32(in, 0);
    return true;
}

// ----- MultiplyIn -----
unsigned int MultiplyIn::getUniqueId() const
{
       return MULTIPLY_ID;
}

bool MultiplyIn::unpack(const San2::Utils::bytes &in)
{
       if (in.size() != 2 * sizeof(SAN_INT32)) return false;
       m_x = San2::Utils::CDataPack::unpackInt32(in, 0);
       m_y = San2::Utils::CDataPack::unpackInt32(in, sizeof(SAN_INT32));
       return true;
}

bool MultiplyIn::operator()(void)
{
    SAN_INT32 result = m_x * m_y;
       printf("Multiply: %d * %d = %d\n", m_x, m_y, result);
    m_response = San2::Utils::CDataPack::pack(result);
       return true;
}

San2::Utils::bytes MultiplyIn::getResponse()
{
    return m_response;
}
```

Volání funkce se provede následovně:

```
// inicializace komunikačního kanálu
m_rpci = new San2::Rpc::CRpcInvoker(<odkaz na implem. CIRpcChannel>, timeout);
// registrace funkce (dealokace proběhne automaticky)
ret = m_rpci->registerSyncFunction([](){return new MultiplyOut;});

// samotné volaní
if (ret) printf("reg success\n"); else printf("reg fail\n");
MultiplyOut mp(3, 5);
if (!m_rpci->invokeSyncFunction(mp)) printf("Invoke fail\n");
std::cout << "Výsledek součinu je: " << (int) mp.getResult() << std::endl;

// další volání
mp.mp_x = 4;
mp.mp_y = 6;
if (!m_rpci->invokeSyncFunction(mp)) printf("Invoke fail\n");
std::cout << " Výsledek součinu je:" << (int) mp.getResult() << std::endl;
```

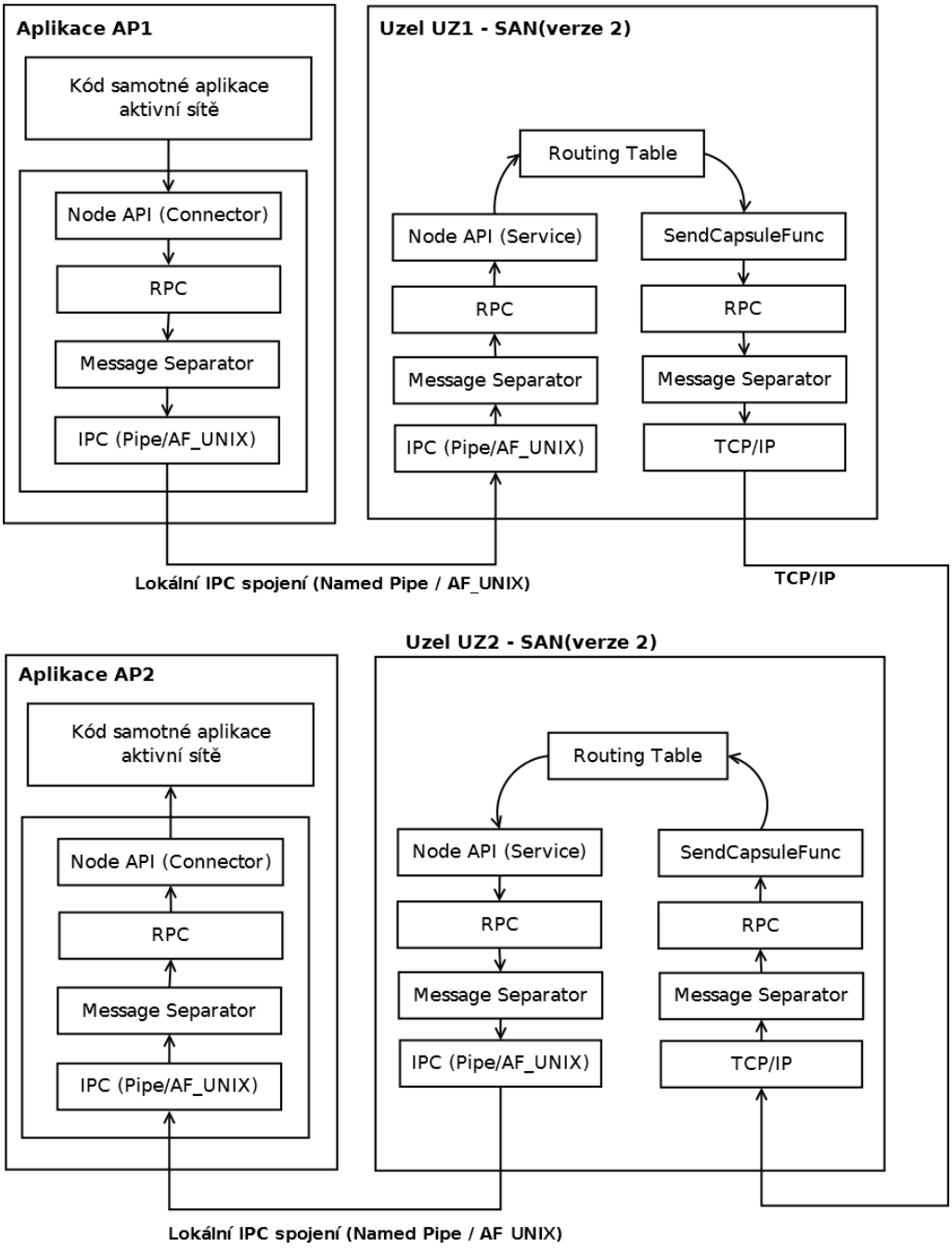
Psaní funkcí a jejich serializace

Psaní RPC funkcí může být poněkud zdlouhavé. Je to z toho důvodu, že není implementován obecný mechanismus serializace a deserializace dat. Program **rpcgen**,

který se používá pro generování serializačních a deserializačních funkcí toto řeší pro ONC RPC. Podobný princip by se dal aplikovat na současnou implementaci RPC v SAN. Mohlo by to vypadat tak, že z funkce v C by se automaticky vytvořily příslušné třídy.

Implementace RPC v SAN je ve srovnání s ostatními implementacemi velmi jednoduchá, ale pro potřeby SAN je dostačující.

3.3.6 Posílání kapsulí mezi aplikacemi mezi různými uzly

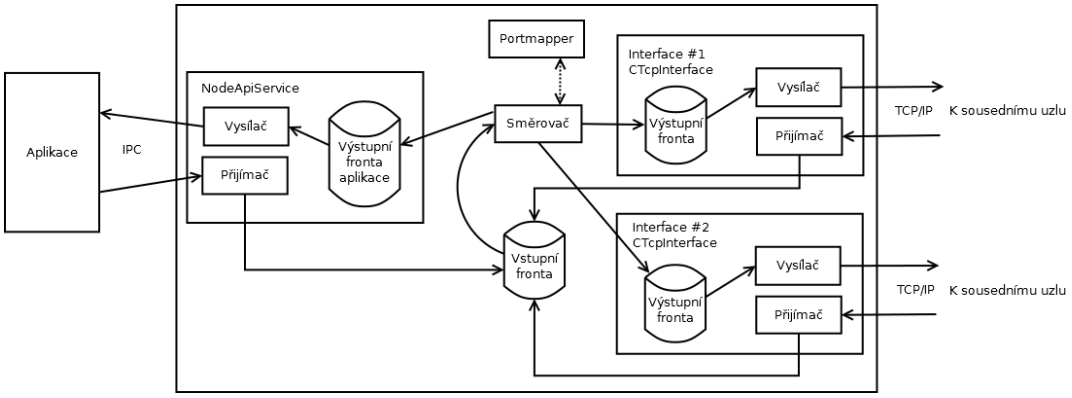


obr. 10: Schéma posílání kapsule z aplikace AP1 připojené k uzlu UZ1 aplikaci AP2 připojené k uzlu UZ2. Šipky ukazují směr, kudy kapsule prochází. (Na obrázku není znázorněn portmapper, který je používán směrovačem.)

Způsob posílání kapsulí mezi jednotlivými aplikacemi aktivní sítě je znázorněn na obr. 10. Uzly UZ1 a UZ2 jsou v rámci SAN sítě sousední.

3.3.7 Zacházení s příchozími a odchozími kapsulemi

Síťová vrstva se stará o síťová rozhraní. Síťová rozhraní navazují TCP/IP spojení s ostatními uzly SAN. Při příchodu kapsule je zařazena kapsule do vstupní fronty. Vstupní fronta je jedna pro celý server. Pokud je vstupní fronta plná, je paket zahozen - tzv. tail-drop. Je to z toho důvodu, aby se server nezahltil. Síť SAN je tedy typu best-effort stejně jako síť IPv4 a IPv6. Podrobné schéma kapsulí je na obr. 11.



obr. 11: Implementovaný tok kapsulí v SANv2 přes komponenty serveru. Každý interface se skládá ze dvou vláken - jedno přijímací a jedno odesílací. Aplikace běží v samostatném procesu a lokálně komunikují s procesem SAN uzlu přes jeho API. (Na obrázku je znázorněna jen jedna aplikace, ale je možno připojit libovolný počet aplikací. Každá aplikace má svoji vyrovnávací frontu. Cílem je, aby se uzel SAN nezdržoval. Zařazením do fronty uzel nečeká na vyzvednutí kapsule aplikací.)

Po vložení kapsule do příchozí fronty je (po nějaké době) jádrem kapsule z fronty vybrána. Kapsule je pak deserializována a podle flagů se s ní zachází. V SAN existují momentálně 2 flagy. Flag *Destination eXecute(DX)* říká, že kapsule bude vykonávána pouze na cílovém uzlu. Druhý flag *Destination Send(DS)* říká, že kapsule nebude vykonávána a bude se chovat jako pasivní datagram jdoucí ze zdrojové do cílové adresy podle výchozího směrovacího algoritmu.

3.4 Shrnutí

Byla navrhnuta a implementována síťová funkcionalita uzlu SAN. Byly implementovány obousměrné TCP rozhraní na bázi jednoduchého RPC. Každé rozhraní má 2 vlákna - přijímací a odesílací. Server funguje paralelně a je schopen obsloužit více sousedních uzlů. Při výpadku uzlu či linky je schopen uzel automaticky navázat spojení znovu.

Zároveň byla implementována komunikace mezi uzlem a aplikacemi na lokálním systému přes pojmenované roury.

Výsledkem je, že je možné z Aplikace připojené k jednomu uzlu posílat kapsule druhé aplikaci na vzdáleném uzlu a naopak viz obr. 10.

Nebyla implementována část spouštějící samotný kód kapsulí, plánovač, repozitář kódů a protokol pro distribuci kódu. Tyto komponenty jsou již mimo oblast této práce a ani časově by je nebylo možné realizovat.

4 Návrh autentizace a šifrování

Autentizace je proces ověření proklamované identity. Autentizaci lze provádět mnoha způsoby od předložení občanského průkazu až po autentizační tokeny s certifikáty. Základní dělení autentizace v počítačových systémech je na:

- Vlastnictvím něčeho - např. občanský průkaz, smart karty, hardwarové tokeny, mechanický klíč
- Biometrie (něco čím jsme) - např. otisk prstu, duhovky, sítnice, tvar ušního boltce, tvar ruky
- Znalostí něčeho - typicky heslo

Biometrii ve většině případů nelze bezpečně použít na větší vzdálenosti. Snímač musí být fyzicky chráněn proti napadnutí, protože stačí jednou získat informaci o dané vlastnosti (např. snímek otisku prstu) a ten pak poslat po drátě. Na druhém konci nikdo nepozná, zda informaci vyslal snímač po přiložení prstu, nebo jsme jí nějakým zařízením injektovali do kabelu.

Vlastnictví v počítačových sítích může být realizováno hardwarovými tokeny, které generují heslo na základě času a seedu – vysvětlit seed, nebo tokeny obsahující soukromý klíč osoby. Osoba se pak prokazuje platným certifikátem. Hardwarové tokeny někdy také přímo obsahují smart kartu. Jednou z nevýhod použití takového řešení je pak nutnost zavedení PKI (Public Key Infrastructure). Klíčovým prvkem je certifikační autorita, která musí být důvěryhodná.

Znalostí se ve většině počítačových případů myslí heslo, které uživatel zadá na klávesnici. Důležité je, aby heslo bylo dostatečně dlouhé a nebylo snadno uhodnutelné nebo ze slovníku.

4.1 Autentizace heslem

Pro autentizaci na základě hesla existují dnes standardizované autentizační protokoly. Jelikož se realizuje ověření totožnosti po síti, tedy vzdáleně, a zároveň potřebujeme následnou komunikaci zabezpečit, vyžaduje se od autentizačního algoritmu, aby po úspěšném ověření identity sdílely obě strany společné tajemství, které pak posléze použijí jako klíč pro zabezpečení následné jejich komunikace.

Ověření identity nemusí nutně být jen mezi dvěma účastníky. Lze se i autentizovat ve skupině. Jelikož cílem práce je vzdálená správa serveru, uvažujeme pouze ověření dvou účastníku a to klienta a serveru.

Pokud se autentizujeme heslem, pak se třída těchto protokolů nazývá *Password-authenticated key agreement (PAKE)*.

4.2 Klasická výměna hesel

Předpoklad: Klient K a Server S, mají mezi sebou předem dohodnuté heslo P.

Cíl: Získat klíč pro sezení *session_key*

1. $R \coloneqq random;\ K \longrightarrow S\colon P(R)$
2. $R = P^{-1}\big(P(R)\big);\ \ S \longrightarrow K\colon R("OK")$

Slovy řečeno, klient vygeneruje náhodný klíč pro sezení R, ten zašifruje svým heslem P a pošle to serveru. Server rozšifruje R, protože zná P. A odpoví klientovi. Klient a server nyní sdílí společný klíč pro sezení.

Tento protokol může být bezpečný v případě dlouhých hesel. Problém protokolu tkví v tom, že pokud během komunikace jsou odposlechnuty zprávy $P(R)$ a $R("OK")$, může útočník jednoduše hesla zkoušet off-line tzv. slovníkovým útokem (*offline dictionary attack*) a vůbec k tomu nepotřebuje komunikovat po síti.

Útočník si zvolí kandidáta na heslo P' tímto heslem dešifruje $R' = P^{-1}\big(P(R)\big)$

a pokud $R("OK") = R'("OK")$, pak máme správné heslo.

4.2.1 Kerberos

Tímto způsobem je dodnes možné napadnout i Kerberos nevhodnou preautentizací na bázi časových značek. Časová preautentizace funguje tak, že uživatel svým heslem zašifruje aktuální čas a pošle ho na server. Útočník zprávu odposlechne a může zkoušet offline hesla, protože aktuální čas zná taky. Proto je důležité, aby při použití tohoto protokolu s časovou preautnetizací byla od uživatelů vyžadována složitá a dlouhá hesla. Jedním z možných řešení je použít například protokol SRP pro preautentizaci.

4.3 Požadované vlastnosti PAKE

Je vhodné předem určit, které vlastnosti od ověřovacího algoritmu vlastně požadujeme.

1. Odolnost vůči odposlechu – heslo není prozrazeno při komunikaci (snooping imunity)
2. Imunita vůči opakovacím útokům (replay attack imunity)
3. Výsledkem autentizace je společné tajemství
4. Vzájemné ověření bez prozrazení tajemství toho druhého (mutual authentication)
5. Odolnost vůči slovníkovým útokům
6. Dopředná bezpečnost (forward secrecy)
7. non-plaintext equivalence

4.3.1 Dopředná bezpečnost

Dopředná bezpečnost je u PAKE vlastnost, která říká do jaké míry je protokol odolný, je-li prozrazena část tajemství. Jedním z nejdůležitějších faktorů je, jestli při prozrazení hesla dojde k prozrazení veškerých předchozích komunikací. Zjednodušeně řečeno, jestli při prozrazení hesla bude útočník schopen dopočítat všechny klíče pro předchozí sezení.

Příklad:

1. Alice pošle náhodnou výzvu R Bobovi
2. Bob pošle náhodnou výzvu S Alici
3. Oba spočítají $K = HASH(R, S, P)$, kde P je heslo

Veškerou komunikaci odposlouchává Eva. Kdyby se Evě, podařilo nějakým způsobem získat/vylákat heslo P, pak může dopočítat K, protože z předchozích komunikací zná R i S. Důsledkem toho je, že Eva je schopna rozšifrovat veškerou předchozí komunikaci.

4.3.2 Non-plaintext equivalence

Informace I je ekvivalentní heslu P, když pomocí I obdržím stejnou úroveň oprávnění jako při použití P.

Příklad č. 1:

Přihlášení se do UNIXu přes síť. Místo plaintext hesla pošlu jeho otisk (hash). Tzn., nepotřebuji znát heslo, stačí mi jeho otisk. P je tedy ekvivalentní $HASH(P)$.

Příklad č. 2: Alice se identifikuje vůči Bobovi

1. Bob pošle Alici náhodnou výzvu R
2. Alice spočítá HASH(R, f(P))
3. Bob spočítá HASH(R, f(P)) na základě jeho uložené hodnoty f(P)
4. Pokud souhlasí, tak ověření proběhlo úspěšně

Potíž je v tom, že stačí znát hodnotu f(P). P je tedy ekvivalentní f(P).

Podle plaintext-ekvivalence lze protokoly rozdělit na:

- **Balanced PAKE:** Klient i server používají stejné tajemství (heslo P) pro vzájemné ověření. Informace je ekvivalentní.
    - EKE - Encrpyted Key Exchange
    - PAK
    - PPK
    - SPEKE - Simple Password Exponential Key Exchange
    - J-PAKE - Password Authenticated Key Exchange by Juggling

- **Augmented PAKE:** Server neukládá data ekvivalentní s heslem. Důsledek je, že krádeží samotného serveru útočník nezíská taková privilegia, jako kdyby měl heslo. Např. na serveru není uloženo samotné heslo ale jeho hash osolený[9] náhodným řetězcem spolu s použitou solí. Mezi protokoly řadící se do Augmented PAKE patří např.:
  - AMP
  - Augmented-EKE
  - B-SPEKE
  - PAK-Z
  - SRP - Secure Remote Password Protocol (viz 4.3.4)

4.3.3 Protokol EKE

Protokol EKE (*Encrpyted Key Exchange*) řeší jak problém slovníkových útoků, tak i, jak bude později ukázáno, dopřednou bezpečnost. Řada ostatních protokolů je postavena na bázi EKE (např. *Augemented EKE* a *SRP*), proto je zásadní ukázat myšlenku jak EKE funguje.

1. Alice vygeneruje **pro každé sezení náhodný** pár veřejného a privátního klíče E_A a D_A a zašifruje svůj veřejný klíč heslem P, tzn. $P(E_A)$ a pošle $P(E_A)$ Bobovi.
2. Bob pak snadno dešifruje E_A, protože zná P. Vygeneruje náhodný klíč sezení R. Zašifruje ho E_A a to celé ještě zašifruje P, takže vznikne $P\big(E_A(R)\big)$ a to pošle Alici.
3. Alice toto snadno dešifruje známým heslem a svým (náhodným) **privátním** klíčem $R = D_A\left(P^{-1}\left(P\big(E_A(R)\big)\right)\right)$. Tím získá R.
4. Alice a Bob nyní sdílí společný klíč pro sezení R.
5. Alice nyní Bobovi pošle $R("OK")$

Přestože EKE využívá asymetrické i symetrické šifrování, jedná se stále o PAKE a nejsou potřeba žádné certifikáty. Útok *Man In the Middle* **není realizovatelný**, protože veřejný klíč je symetricky šifrovaný heslem a je pokaždé nahodilý.

Analýza slovníkového útoku

Pokud bude útočník poslouchat veškerou komunikaci po síti, získá $P(E_A)$, $P\big(E_A(R)\big)$ a $R("OK")$.

1. Zvolím kandidáta na heslo P'
2. Spočtu $E_A{}' = P^{-1}\big(P(E_A)\big)$

[9] Osolení je přidání náhodného řetězce, soli, k heslu. Hesla jsou pak uložena ve formátu {sůl, HASH(heslo+sůl)}. Je to prevence proti zpětnému zjišťování hesel z hashů.

3. Jenže teď nelze zjistit, jestli veřejný klíč $E_A{}'$ je ten správný. Otázka tedy teď zní, zda existuje R′ takové, že platí $E_A{}'(R') = E_A(R)$ a zároveň $R'^{-1}\big(R("OK")\big) = R("OK")$ Musím tedy zvolit náhodné R′ a podmínku ověřit. Pokud vše sedí, uhodl jsem P a zároveň i R.

Z toho plyne zcela zásadní věc. Na to abych uhodl heslo P, je nutné nejdříve uhodnout klíč pro sezení R. Jenže R je náhodné a může být velmi dlouhé. (Např. při použití AES minimálně[10] 128 bitů, což je obrovský prostor pro klíče cca 3.10^{38}). Útok hrubou silou by byl velmi časově náročný.

Důkaz dopředné bezpečnosti

Útočník opět poslouchá veškerou komunikaci ($P(E_A)$, $P\big(E_A(R)\big)$, $R("OK")$. K tomu se mu ale podaří získat heslo P. Útočník se tedy bude snažit rozšifrovat veškerou předchozí komunikaci

1. Zjistíme pravý veřejný klíč Alice $E_A = P^{-1}\big(P(E_A)\big)$.
2. Spočítáme $E_A(R) = P^{-1}\left(P\big(E_A(R)\big)\right)$
3. Jenže z $E_A(R)$už nezjistíme R, protože bychom potřebovali soukromý klíč Alice D_A. (Tento klíč už ani Alice nemá, protože se generuje náhodný pro dané sezení.)

Útočník tedy ze znalosti hesla P a odposlechnuté komunikace nemůže odvodit hesla pro sezení R. Dalším faktorem je, že EKE vyžaduje, aby klient i server znali heslo v otevřené podobě. Při případné krádeži serveru útočník získá rovnou hesla. Tento problém např. řeší Augmentované-EKE, ale za cenu ztráty dopředné bezpečnosti. Hlavní vlastností EKE, které zachovávají všechny varianty je odolnost proti offline slovníkovým útokům. EKE se dá použít s různými asymetrickými šiframi. Originální varianta používá RSA, později se vyvinulo DH-EKE, které používá Diffie-Hellman algoritmus. Každá z těchto variant má určité omezení a vyžaduje další kroky k zajištění bezpečnosti. U RSA se např. naráží na problém, že veřejný klíč není náhodně generované číslo.

4.3.4 Protokol SRP

Protokol SRP (*Secure Remote Password Protocol*) byl poprvé publikován v roce 1992 na univerzitě ve Stanfordu. Jeho poslední verze se nazývá SRP6a a je popsán mj. také v RFC2945. Protokol je odolný proti slovníkovým útokům, zajišťuje dopřednou bezpečnost a ještě navíc je ZKPP (Zero Knowledge Password Proof). *ZKPP je interaktivní metoda ověření identity entity jiné entitě tak, že prokáže, že zná heslo, ale neprozradí žádné další informace.*

[10] AES má délku klíče 128, 192 nebo 256 bitů

V případě SRP je na serveru uložena trojice **{uživatelské jméno**, **sůl**, **verifikátor}**. Server zná pouze otisk hesla. Server je schopen ověřit, že klient zadal správné heslo, ale není již schopen získat původní heslo z hashe jinak než brute-force metodou[11]. Při krádeži serveru lze zkoušením hesel a porovnáváním otisků získat z verifikátorů hesla, nelze však verifikátory použít přímo k autentizaci.

Kromě varianty **{uživatelské jméno**, **sůl**, **verifikátor}** je možné použít kombinaci **{uživatelské jméno**, **sůl**, **x}**, kde $x = H(s, H(username, ":", password))$. Rozdíl je v tom, že verifikátor je vázaný na parametry N a g, kdežto x lze použít po dohodě klienta a serveru s libovolnými bezpečnými parametry N a g.

Proměnné používané v SRP6a

- ***N*** bezpečné prvočíslo
- ***g*** generátor prostoru pro *N*
- ***H*** kryptografická hashovací funkce
- ***k*** = *H(N, g)*, *g* musí být doplněno nulami na délku *N*
- ***a*** soukromý klíč klienta
- ***A*** veřejný klíč klienta
- ***b*** soukromý klíč klienta
- ***B*** veřejný klíč klienta
- ***c*** uživatelské jméno
- ***p*** heslo
- ***s*** sůl
- ***v*** verifikátor
- ***u*** = *H(A, B)*
- ***M1*** ověřovací zpráva; použitá k ověření, že klient zadal správné heslo
- ***M2*** ověřovací zpráva; použitá k ověření, že server zadal správné heslo
- ***K*** výsledný klíč sezení

Výměna zpráv pro vzájemné ověření:

1. $K \rightarrow S: \{c, A\}$
2. $S \rightarrow K: \{s, B\}$
3. $K \rightarrow S: \{M_1\}$ ověření klienta vůči serveru
4. $S \rightarrow K: \{M_2\}$ ověření serveru, posláno pouze pokud souhlasí krok 3

Veškeré matematické operace jsou $modulo\ N$.

První fáze protokolu je nastavení databáze uživatelů:

[11] Za předpokladu použití bezpečné hashovací funkce.

1. $s \coloneqq random$
2. $x = H(s, H(c, ":", p))$
3. $v = g^x$
4. ulož c, s a v do databáze na serveru

Samotná autentizace probíhá takto:

1. Klient
    a. $a \coloneqq random$
    b. $A = g^a$
    c. pošli serveru c a A
2. Server
    a. $b \coloneqq random$
    b. $B = kv + gb$
    c. $u = H(A, B)$
    d. $S_{server} = (Av^u)^b$
    e. pošli klientovi s a B, kde s je sůl daného uživatele z databáze
3. Klient
    a. $S_{klient} = (B - g^x)^{a+ux}$
    b. $M_1 = H(A, B, S_{klient})$
    c. Pošli serveru M_1
4. Server
    a. Spočítej M_1 a porovnej s tím od klienta
    b. Pokud ok, klient je ověřen M_2= H(A, M_1, S_{server}) a pošli klientovi M_2. Jinak nic posílej.
5. Klient
    a. Spočítej M_2 a porovnej s tím od serveru
    b. Pokud ok server je ověřen, jinak ukonči spojení
6. Klient+Server
    a. Společný klíč pro sezení $K = H(S)$

Pokud je autentizace úspěšná, pak $S_{server} = S_{klient}$. Na konci sdílejí obě strany společný klíč K.

4.3.5 Srovnání jednotlivých PAKE protokolů

	Odolnosti proti slovníkovým útokům	Dopředná bezpečnost	Non-plaintext ekvivalence
Klasická výměna	Ne	Ne	Ne
EKE	Ano	Ano	Ne
Augmented-EKE	Ano	Ne	Ano
J-PAKE	Ano	Ano	Ne
SRP	Ano	Ano	Ano

Z jednotlivých protokolů se jeví jako nejvhodnější SRP.

4.4 Implementace SRP

Protože nebyla nalezena knihovna s otevřeným zdrojovým kódem a vhodnou licencí implementující protokol SRP v C++, byla navržena a implementována knihovna DragonSRP v C++. Knihovna modulárně implementuje autentizační protokol SRP a některé jiné kryptografické algoritmy.

Funkčnost knihovny nejlépe znázorní příklad jednoduchého serveru[12], ke kterému se přihlašuje klient. První fází je vytvoření verifikátoru:

```
// hlavičkové soubory byly vynechány
int main(int argc, char **argv)
{
        try {
                OsslSha1 hash; // Jako hashovací funkci volíme SHA1
                OsslRandom random; // Použijeme generátor náhodných čísel z OpenSSL
                Ng ng = Ng::predefined(1024); // 1024 bitový modulus
                OsslMathImpl math(hash, ng);
                // hlavni trida pro autentizaci
                SrpClient srpclient(math, random, false);

                 // Načtení uživatelského jména a hesla z klávesnice
                string strUsername; cout << "username: "; cin >> strUsername; cin.ignore();
                string strPassword; cout << "password: "; cin >> strPassword; cin.ignore();
                bytes username = Conversion::string2bytes(strUsername);
                bytes password = Conversion::string2bytes(strPassword);

                bytes salt;
                if (salt.size() == 0) salt = random.getRandom(SALTLEN);
                bytes verificator = math.calculateVerificator(username, password, salt);

                cout << "salt: "; Conversion::printBytes(salt); // vypíše sůl
                // vypíše verifikátor
                cout << endl << "verificator: "; Conversion::printBytes(verificator) << endl;
        }
        catch (DsrpException e)  {cout << "DsrpException: " << e.what() << endl;}
        catch (...) { cout << "unknown exception occured" << endl; }
        return -1;
}
```

Výstupem programu je sůl a verifikátor, který spolu s uživatelským jménem předáme serveru, který si ho vloží do databáze uživatelů. Kód serveru pak vypadá následovně:

```
// Hlavickove soubory vynechany
int main(int argc, char **argv)
{
        try {
                OsslSha1 hash; // Pouzijeme SHA1
                OsslRandom random; // Pouzijeme nahodny generator z OpenSSL
                MemoryLookup lookup; // Databaze uzivatelu v pameti
                Ng ng = Ng::predefined(1024); // Nacteme 1024 bitovy modulus
                OsslMathImpl math(hash, ng);
                // Hlavni trida pro server
                SrpServer srpserver(lookup, math, random, false);
```

[12] Serverem je v tomto případě myšlena část protokolu SRP, která autentizuje klienty nikoliv server, který by využíval nějaké síťové prostředky. Protokol SRP je jako takový nesymetrický a rozlišuje klienta, který se přihlašuje a server, který provádí ověření podle verifikátoru. (Ověření je vzájemné.)

```
                // Nyni vytvorime uzivatele na zaklade hodnot z predchoziho programu
                std::string strUsername; cout << "username: ";
                cin >> strUsername; cin.ignore();
                bytes username = Conversion::string2bytes(strUsername);
                bytes verificator = Conversion::readBytesHexForce("verificator");
                bytes salt = Conversion::readBytesHexForce("salt");
                User u(username, verificator, salt);
                lookup.userAdd(u); // pridej uzivatele do databaze
                // Nyni je uzivatel vytvoren (tohle by ve skutecnosti probihalo jen jednou)


                // Zjisti A od klienta
                bytes A = Conversion::readBytesHexForce("A(from client)");
                // ziskej objekt tridy verifikator, ktery se pouziva pro dane sezeni
                SrpVerificator ver = srpserver.getVerificator(username, A);
                // Posli sul a B klientovi
                cout << "salt(send to client): "; Conversion::printBytes(ver.getSalt());
                cout << endl;
                cout << "B(send to client): "; Conversion::printBytes(ver.getB());
                cout << endl;

                // nacti M1 od klienta
                bytes M1_fc = Conversion::readBytesHexForce("M1(from client)");
                bytes M2_to_client; bytes K; // K = tajny klic pro sezeni
                // Pokud M1 souhlasi dostaneme M2 a klic pro sezeni jinak
                // je vyhozena vyjimka
                ver.authenticate(M1_fc, M2_to_client, K);
                // posli M2 klientovi
                cout << "M2 (send to client): "; Conversion::printBytes(M2_to_client);
                cout << endl;
                // zobrazeni klice sifrovaciho klice pro sezenei
                cout << "shared secret session key is: ";
                Conversion::printBytes(K); cout << endl;
                // pokud se dostaneme sem, autentizace je uspesna
                // jinak je vyhozena vyjimka DsrpException
                cout << "authentification successful" << endl;
                return 0;
        }
        catch (UserNotFoundException e) ...  // zkraceno
        catch (DsrpException e) ... // zkraceno, Chyba autentizace, spatne heslo
        return -1;
}
```

Autentizace samotného uživatele pak probíhá:

```
int main(int argc, char **argv)
{
        try {
                OsslSha1 hash; OsslRandom random; Ng ng = Ng::predefined(1024);
                OsslMathImpl math(hash, ng); SrpClient srpclient(math, random, false);

                // pozadej uzivatele o prihlasovaci udaje
                string strUsername; cout << "username: "; cin >> strUsername; cin.ignore();
                string strPassword; cout << "password: "; cin >> strPassword; cin.ignore();
                bytes username = Conversion::string2bytes(strUsername);
                bytes password = Conversion::string2bytes(strPassword);
                SrpClientAuthenticator sca = srpclient.getAuthenticator(username, password);

                // Posli A serveru
                bytes A = sca.getA(); cout << "A (send to server): ";
                Conversion::printBytes(A); cout << endl;
                // prijmi sul a B ze serveru
                bytes salt = Conversion::readBytesHexForce("salt(from server)");
                bytes B = Conversion::readBytesHexForce("B(from server)");
                // posli M1 serveru
                bytes M1 = srpclient.getM1(salt, B, sca); cout << "M1(send to server): ";
                Conversion::printBytes(M1); cout << endl;
                // prijmi M2 ze serveru
                bytes M2 = Conversion::readBytesHexForce("M2(from server)");
```

```
                bytes K = sca.getSessionKey(M2); // pri spatnem hesle vyhodi vyjimku

                cout << "shared secret session key is: "; Conversion::printBytes(K); cout << endl;
                // pokud se dostaneme sem, autentizace je uspesna
                // jinak je vyhozena vyjimka DsrpException
                cout << "authentification successful" << endl;
                return 0;
        }
        catch (UserNotFoundException e) ...  // zkraceno
        catch (DsrpException e) ... // zkraceno, chyba autentizace, spatne heslo
        return -1;
}
```

Z příkladů je patrné, že lze velmi snadno vyměnit použitou hashovací funkci, použitý modulus i generátor náhodných čísel. Samotné matematické operace jsou schovány v implementaci. Při chybě je vyhozena výjimka.

4.4.1 Poznámka k OpenSSL

Samotná knihovna není přímo závislá na OpenSSL. OpenSSL se využívá pouze pro aritmetické operace s velkými čísly (sčítání, násobení, modulární exponenciace), generátor náhodných čísel a hashovací funkce. Samotná funkcionalita SRP není vykonávána v OpenSSL, ale knihovnou DragonSRP.

4.4.2 Kompatibilita s RFC2945

Knihovna byla implementována podle RFC2945 a je s ním plně kompatibilní. Součástí knihovny je plně automatický test, který porovnává výsledky knihovny s testovacími vektory z RFC5054 (Using SRP for TLS Authentication) Appendix B.

4.4.3 Další kryptografická primitiva

Knihovna obsahuje mj. vlastní implementaci HMAC a převzatou implementaci AES (Brian Gladman). HMAC může pracovat s libovolnou hashovací funkcí - HashInterface je stejný jako u SRP. Tyto primitiva se později využijí pro implementaci bezpečného přenosového protokolu.

4.4.4 Shrnutí

Knihovna DragonSRP je umožňuje vytvoření nativního bezpečného přenosového protokolu nad libovolným transportním protokolem. Knihovna plně vyhovuje požadavkům RFC2945.

4.5 Šifrování

Samotná autentizace řeší pouze ověření zúčastněných entit. K zabezpečení zpráv proti odposlechu je třeba zprávy šifrovat. Počítačové šifrovací algoritmy lze rozdělit na blokové a proudové, symetrické a asymetrické. Symetrické šifry používají pro šifrování a dešifrování stejný klíč. Asymetrické šifry mají dva klíče, jeden pro šifrování a druhý pro dešifrování.

Proudové šifry šifrují plaintext (většinou) po jednotlivých bitech a mají mj. uplatnění v telekomunikacích, kde je snaha docílit co nejmenšího zpoždění.[13] Základní princip je kombinace jednotlivých bitů plaintextu s bity pseudo-náhodného generátoru. Kombinace se většinou provádí booleovskou operací XOR.

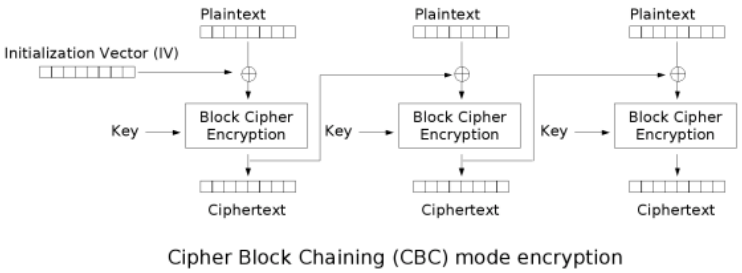
Blokové šifry pracují se zprávou pevné délky. Vstup i výstup mají stejnou délku. Parametrem je klíč, viz obr. 12.

obr. 12: Obecné schéma blokové šifry

Pro zašifrování dat delších nežli jeden blok je třeba zvolit způsob řetězení bloků.

Mód Cipher Block Chaining (CBC)

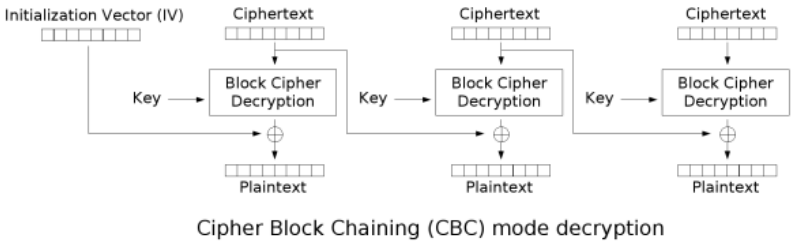
Šifrování se provádí tak, že na vstupu blokové šifry je exkluzivní součet (XOR) otevřeného textu a výsledku předchozího bloku. Na vstupu prvního bloku je tzv. inicializační vektor, který nemusí být tajný.



obr. 13: Šifrování v módu CBC (převzato z http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation)

Dešifrování se provádí obdobně s rozdílem, že XOR se provádí na výstupu. Inicializační vektor musí být shodný.

[13] Např. proudová šifra A5/1 je používána pro šifrování hovorů mezi GSM telefonem a BTS.



obr. 14: Dešifrování v módu CBC (převzato z http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation)

Mód CBC nelze paralelizovat. Pokud není zpráva beze zbytku dělitelná velikostí bloku, je nutné provést zarovnání. CBC musí být implementováno ve všech standardních knihovnách SSL(Secure Sockets Layer) a TLS(Transport Layer Security).

Mód Counter (CTR)

Bloky jsou na sobě nezávislé. Základem je čítač na vstupu bloku, jehož hodnota se nesmí nikdy opakovat. Zašifrovaný text je exkluzivní součin výstupu bloku s otevřeným textem.

4.6 Zajištění integrity zpráv

Samotné šifrování neposkytuje žádné zabezpečení z hlediska pozměnění zpráv během cesty. Proto je nutné šifrovanou zprávu zabezpečit kontrolním kódem. Zprávu lze podepisovat asymetricky např. algoritmy RSA a DSA. Jelikož potřebujeme přenášet zprávy po síti velmi rychle, je asymetrické podepisování nevhodné, protože s porovnání se symetrickým šifrováním je pomalejší. Jedním z faktorů, kterým ovlivňuje rychlost podepisování je délka klíče. Délka klíče u současných asymetrických šifer je několikrát delší než u symetrických. Např. v ČR je minimální povolená délka kvalifikované certifikační autority 2048 bitů. Délka klíče AES je 128, 192 nebo 256 bitů. Další nevýhodou je potřeba důvěryhodné certifikační autority. Vzhledem k použitému autentizačnímu algoritmu SRP, který nevyžaduje certifikační autoritu, se jeví symetrické algoritmy pro SAN jako vhodnější.

Mezi symetrické algoritmy např. patří HMAC a OMAC. HMAC (Keyed-hash Message Authentication Code) je algoritmus založený na kryptografické hash funkci.

Výpočet HMAC:

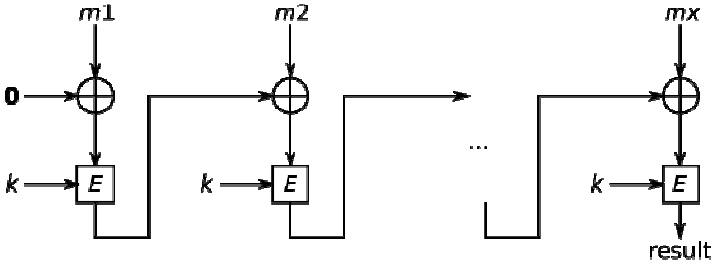
$$HMAC(m) = H\left((K \oplus opad) + H\big((K \oplus ipad) + m\big)\right),$$

kde H je hashovací funkce,
m je zpráva, kterou chceme podepsat,
opad = 0x5c5c5c.....5c5c (vnější zarovnání),
ipad = 0x363636.....3636 (vnitřní zarovnání),
$\oplus$ ... exkluzivní součet

+ ... operace zřetězení

Délka opad, ipad a klíče je závislá na délce bloku hashovací funkce[14]. Klíč se zarovnává zprava nulami na délku bloku. Opad a ipad, jsou 0x5c a 0x36 v délce bloku. Pro zrychlení algoritmu lze pro jeden klíč předpočítat $K \oplus opad$ a $K \oplus ipad$ a tím výpočet zrychlit. Délka kontrolního kódu je rovna délce výstupu hashovací funkce.

Jiné algoritmy jsou založené na blokových šifrách. CBC-MAC zašifruje zprávu blokovou šifrou v módu CBC a kontrolní kód je výstup posledního bloku viz obr. 156.



obr. 15: CBC-MAC (převzato z http://en.wikipedia.org/wiki/CBC-MAC)

Blokové šifry nebyly původně pro účely podepisování zpráv navrženy.

[14] Neplést s délkou výstupu. Např. MD5 má délku bloku 64B a délku výstupu 16B.

5 Meziprocesová komunikace

Meziprocesová komunikace (Inter-Process Communication(IPC)) je komunikace mezi procesy, která může být lokální nebo vzdálená. Výsledná implementace serveru aktivní sítě by měla běžet jak pod OS Linux tak pod OS Windows. Protože způsoby meziprocesové komunikace ve Windows a v Linuxu se poměrně liší, a to především v API, budou tyto rozdíly ve funkčnosti na jednotlivých operačních systémech popsány zvlášť.

5.1 Sdílená paměť

Základním způsobem sdílení dat mezi lokálně běžícími procesy je sdílená paměť. Tu lze úspěšně použít i v malých mikrokontrolérech a i real-time operačních systémech. Sdílená paměť je sice jedním z nejrychlejších způsobů komunikace, ale přináší sebou také nevýhody v podobě složité synchronizace mezi procesy.

5.1.1 Linux

V OS Linux, přesněji řečeno v POSIX, lze mezi procesy vytvořit sdílený prostor paměti voláními:

```
int shmget(key_t key, size_t size, int shmflg);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

Funkce shmget se používá pro získání sdíleného prostoru paměti o dané velikosti. Funkce shmctl mění přístupová práva k paměti. Funkce shmat (at=attach) a shmdt (dt=detach) připojují a odpojují paměť do, respektive z datové sekce procesu. Klíč (`key_t key`) je unikátní číslo, kterým se ostatní procesy připojují k dané sdílené paměti. Po připojení paměti se do ní zapisuje běžnými pointery v jazyce C. Použití funkcí ilustruje následující krátký úsek kódu:

```
char c;
int shmid;
key_t key = 5678; // pojemenovani pameti
char *shm, *s;

// vytvoreni useku sdilene pameti
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) return -1;

// pripojeni pameti do datove sekce procesu
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) return -1;

// ... zapis a cteni pointry ...
```

5.1.2 Windows

Ve Windows se mluví o souboru mapovaném do paměti (*memory mapped file*). Soubor mapovaný v paměti může být buď pojmenovaný, nebo nepojmenovaný. Oba typy mapování se vytváří systémovým voláním CreateFileMapping.

```
HANDLE WINAPI CreateFileMapping(
  _In_      HANDLE hFile,
  _In_opt_  LPSECURITY_ATTRIBUTES lpAttributes,
  _In_      DWORD flProtect,
  _In_      DWORD dwMaximumSizeHigh,
  _In_      DWORD dwMaximumSizeLow,
  _In_opt_  LPCTSTR lpName
);
```

hFile slouží jako handle k otevřenému souboru.[15] Handle hFile lze mj. vytvořit voláním CreateFile. Namapovaný soubor je nutno připojit k procesu voláním MapViewOfFile. Odpojit lze voláním UnmapViewOfFile.[16] Příklad použití funkcí ilustruje následující úsek kódu:

```
HANDLE hFile = CreateFile(szFileName, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ |
FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
assert(hFile!=INVALID_HANDLE_VALUE);
// vytvoření anonymního (nepojmenovaného) mapování
HANDLE hMap=CreateFileMapping(hFile,NULL,PAGE_READWRITE,0,0,NULL);
assert(hMap!=NULL);
DWORD dwFileSize=GetFileSize(hFile,NULL); // zjisteni velikosti
// namapovani pameti do prostoru procesu
char *memory=(char *)MapViewOfFile(hMap,FILE_MAP_WRITE,0,0,dwFileSize);
assert(memory!=NULL);

// ... zapis a cteni pameti pointery ...

UnmapViewOfFile(memory); // unmap file buffer
CloseHandle(hMap); CloseHandle(hFile);
```

5.1.3 Boost::Interprocess

Boost::Interprocess je multiplatformní knihovna s otevřeným kódem pro komunikaci mezi procesy, která se snaží o zjednodušení přístupu k běžným synchronizačním a komunikačním prostředkům různých operačních systémů.

Implementovaná je mj. podpora následujících synchronizačních primitiv:

- sdílená paměť
- soubory mapované do paměti
- semafory, mutexy, podmínkové proměnné
- zamykání souborů

---

[15] Popis ostatních parametrů lze nalézt v dokumentaci na MSDN (http://msdn.microsoft.com/en-us/library/windows/desktop/aa366537(v=vs.85).aspx)

[16] Analogie s POSIX voláním shmat a shmdt

- fronty zpráv

Boost::Interprocess do jisté míry poskytuje unifikované rozhraní pro práci s pamětí nezávislé na operačním systému, ale synchronizaci mezi procesy za nás nevyřeší.

5.1.4 Shrnutí - sdílená paměť

Sdílená paměť je obecným prostředkem pro komunikaci mezi procesy. Její výhodou je vysoká rychlost a dostupnost na většině operačních systémů. Mezi nevýhody patří složitost synchronizace mezi procesy a velká závislost na použitém operačním systému. Pro použití sdílené paměti by v SAN2 by bylo nutné napsat synchronizační mezivrstvu, která by umožňovala přenos dat mezi serverem (uzlem SAN) a více klienty (aplikacemi). Existují ale jiné metody, které toto chování již implementují a které jsou pro SAN2 vhodnější (viz 5.2.3 a 5.2.5), nežli implementovat vlastní multiplatformní synchronizační mezivrstvu.

5.2 Roury

Dalším způsobem komunikace jsou roury. V OS Linuxu existují 2 typy rour, a to nepojmenované a pojmenované.

5.2.1 Nepojmenované roury v OS Linux

Nepojmenované roury se používají hlavně při komunikaci rodiče a potomka tzn. před voláním fork() se vytvoří nepojmenovaná roura, která se po zavolání fork() zkopíruje do potomka.[17] Nepojmenované roury jsou jednosměrné. Použití nepojmenované roury ukazuje následující příklad:

```
int main()
{
    int fd[2]; char buf[30];
    if(pipe(fd) == -1) return -1;

    if(!fork())
    {
        write(fd[1], "Ahoj", 5);
        exit(0);
    else
    {
        read(fd[0], buf, 5);
        printf("RODIC: ctu %s\n", buf);
        wait(NULL);
    }
 return 0;
}
```

[17] Poté je nutné duplicitní file deskriptory uzavřít

Volání pipe() má jediný parametr - adresu pole dvou prvků typu int. Do tohoto pole budou po volání zapsána čísla popisovačů souborů, která odpovídají nově vytvořené rouře. První popisovač je jen pro čtení, druhým se jen zapisuje. Cokoliv je zapsáno do fd[1] přečteme později ve fd[0].

5.2.2 Pojmenované roury v OS Linux

Jméno roury je v OS Linux řetězec určující cestu k souboru. Pojmenovanou rouru vytvoříme voláním `int mkfifo(const char *pathname, mode_t mode)`. Mode jsou standardní linuxová oprávnění. Mkfifo vytvoří v souborovém systému soubor do kterého lze zapisovat a číst. Předtím než lze zapisovat nebo číst musí oba procesy soubor otevřít. Otevření musí probíhat naráz, protože volání open() blokuje do té doby, nežli druhý proces také zavolá open().

Ukázka vytvoření pojmenované roury mezi dvěma procesy:

```
// Zapisovac
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666); // vytvor pojmenovanou rouru
    fd = open(myfifo, O_WRONLY); // otevri rouru jen pro zapis
    write(fd, "Hi", sizeof("ahoj")); // zapis "ahoj"
    close(fd);
    unlink(myfifo); // odstran rouru/soubor
    return 0;
}

// Ctenar
#define MAX_BUF 1024
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
    char buf[MAX_BUF];
    fd = open(myfifo, O_RDONLY); // otevri rouru
    read(fd, buf, MAX_BUF); // precti z roury
    printf("Received: %s\n", buf); // vypis precteny retezec
    close(fd);
    return 0;
}
```

Do roury se zapisuje jako by se jednalo o soubor. Synchronizaci řeší operační systém. Volání read a write jsou ve výchozím nastavení blokující. Při uzavření roury na jedné straně je druhému procesu zaslán signál SIGPIPE - Broken Pipe.

5.2.3 Sockety AF_UNIX

Specifickým způsobem meziprocesové komunikace v OS Linux jsou doménové sockety typu UNIX (AF_UNIX). Existuje server, který přijímá příchozí spojení, podobně jako v TCP. Při přijetí příchozího spojení je vytvořen nový socket, kterým se pak komunikuje způsobem bod-bod. UNIXové sockety jsou obousměrné a mají vždy

spolehlivý přenos. Můžou být jak datagramové, tak proudové (streamované). Za zmínku stojí jistě fakt, že pokud jeden koncový bod spojení ukončí, je tento stav signalizován signálem SIGPIPE - rozbitá roura. Ukázku použití tohoto typu rour lze nalézt v projektu SAN2 ve složce cppl.

5.2.4 Nepojmenované roury ve Windows

Na Windows fungují nepojmenované roury obdobně jako v Linuxu. Funkce

```
BOOL WINAPI CreatePipe(
  _Out_     PHANDLE hReadPipe,
  _Out_     PHANDLE hWritePipe,
  _In_opt_  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  _In_      DWORD nSize
);
```

vrátí 2 handly na soubory. První pro čtení a druhý pro zápis. Do roury lze zapisovat funkcemi ReadFile a WriteFile. Výchozí chování je blokující. Funkce WriteFile čeká, až zapíše celý buffer do roury a pokud je vyrovnávací fronta plná, čeká, až se uvolní místo.

5.2.5 Pojmenované roury ve Windows

Pojmenované roury ve Windows mají chování server-klient, kde klientů může být více. (Chování se podobají socketům typu AF_UNIX.) Ve Windows 7 sockety typu AF_UNIX[18] nejsou. Windows nabízí podobný způsob komunikace, ovšem zcela s jinými API funkcemi a rozhraním. Nazývá ho pojmenované roury ("Named pipes"). Stejně jako v OS Linux, je jméno roury cesta ve filesystému. Nejedná se ale o soubor na disku. Pojmenované roury lze použít i pro komunikaci po síti. Lokální jméno souboru musí zachovávat formát \\.\pipe\jmeno_roury. Vzdálené roury jsou pak ve tvaru \\jmeno_serveru\pipe\jmeno_roury. Důležitým rozdílem je fakt, že oproti OS Linux se roury nechovají jako file deskriptory, i když v některých funkcích např. WriteFile jdou použít, ale jejich chování je odlišné od souborů. [18]

Pojmenovaných rour je ve Windows několik typů:

- podle přístupu
  - příchozí (PIPE_ ACCESS_ INBOUND), kterou lze číst
  - odchozí (PIPE_ ACCESS_ OUTBOUND), do které lze zapisovat
  - obousměrné neboli duplexní (PIPE_ ACCESS_ DUPLEX)
- způsob zápisu
  - proudový (PIPE_ TYPE_ BYTE)
  - datagramový (PIPE_ TYPE_ MESSAGE)
- způsob čtení
  - proudový (PIPE_ READMODE_ BYTE)

[18] V jednom období byly sockety typu AF_UNIX ve Windows podporované, nikdy ale ve Windows 7.

- datagramový (PIPE_ READMODE_ MESSAGE) - lze použít jen u datagramového zápisu

Způsob zápisu je analogický jako u AF_UNIX typů STREAM a DGRAM. Jistou zvláštností je, že zapisovat lze datagramově a přitom číst po bajtech. Příklady práce s pojmenovanými rourami lze nalézt v [19] nebo ve zdrojových kódech SAN2 ve složce cppl.

5.2.6 Srovnání metod meziprocesové komunikace

Tab. 11: Srovnání různých metod meziprocesové komunikace

Metoda	Linux	Win.	Více procesů /klientů	Nutnost synchronizace	Jméno	Pozn.
Sdílená paměť	Ano	Ano	Ano (teoreticky)	Ano	Klíč - číslo	Odlišné API Win/Linux
Memory-mapped file	Ano (mmap)	Ano	Ano (teoreticky)	Ano	Cesta k souboru	Odlišné API Win/Linux
Boost::Interprocess	Ano	Ano	Závisí na použití	Ano	Závisí na použití	Externí knihovna
Nepojmenované routy	Ano	Ano	Ne	Ne	Není	Odlišné API Win/Linux
Pojmenované roury (Linux)	Ano	-	Ne	Ne	Cesta k souboru	POSIX
Pojmenované roury (Windows)	-	Ano	Ano, nativně	Ne	Cesta k souboru	Win32Api
AF_UNIX	Ano	Ne	Ano, nativně	Ne	Cesta k souboru	POSIX

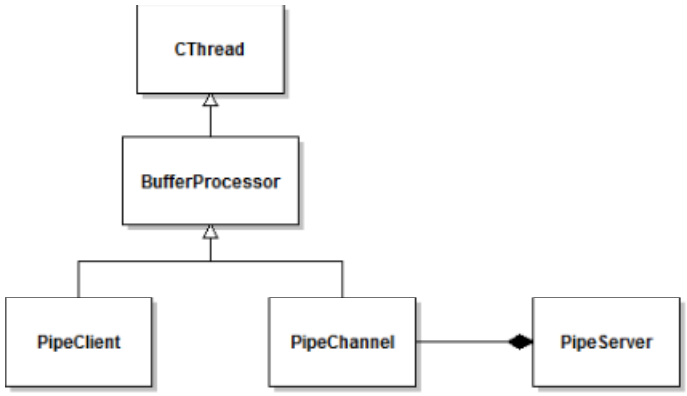
Z tab. 11 je patrné, že univerzální multiplatformní řešení neexistuje. Jediné multiplatformní řešení je použití TCP/IP nebo UDP/IP socketu, které však není považováno za formu lokální meziprocesové komunikace, i když tak může fungovat a dá se tak přes loopback použít. Vzhledem k složité synchronizaci byly metody sdílené paměti a soborů mapovaných do paměti vyřazeny jako zcela nevhodné, včetně knihovny Boost::Interprocess, která také vyžaduje ruční synchronizaci mezi procesy. Nepojmenované roury jsou vzhledem k nutnosti připojit se ke konkrétnímu programu zvenku také pro SAN2 nepoužitelné.

Zbylými metodami jsou pojmenované roury na Windows a Linuxu a doménové sockety AF_UNIX. Vzhledem k tomu že pojmenované roury na Windows podporují nativně více klientů a jejich chování je velmi podobné socketům AF_UNIX používá výsledná implementace SAN2 tyto metody, a to způsobem, že vždy jednu metodu pro daný podporovaný operační systém - sockety AF_UNIX pro OS Linux a pojmenované roury na OS Windows.

Protože jsou velké rozdíly v API funkcích obou metod, byla navržena a implementována knihovna CPPL (Cross-Platform Pipe Library), která tyto platformní závislosti interně řeší a poskytuje jednotné rozhraní pro komunikaci jednoho procesu (serveru) s více procesy (klienty). Implementaci knihovny lze nalézt ve složce cppl projektu SAN2 spolu s příklady použití ve složce examples/cppl.

5.2.7 Knihovna CPPL

Knihovna CPPL se skládá ze tří hlavních tříd a to PipeClient, PipeServer a PipeChannel.



obr. 16: UML class diagram knihovny CPPL

PipeChannel reprezentuje jednoho klienta. Tato třída je automaticky instancována třídou PipeServer po připojení klienta a spuštěna jako nové vlákno. PipeChannel dědí CThread. CThread obsahuje virtuální metodu run(), která musí být implementována třídou, která dědí PipeChannel. PipeServer automaticky vytváří a připojuje (join()) vlákna po připojení klienta, a taktéž po ukončení metody run().

Pro vytvoření serveru je nutné nejdříve vytvořit instanci třídy PipeServer. PipeServer má konstruktor ve tvaru:

```
PipeServer(
        const char *pipeName,
        std::function<PipeChannel* (CPPL_PIPETYPE, unsigned int, unsigned int)> proc,
        unsigned int timCON,
        unsigned int timRX,
        unsigned int timTX
);
```

Parametry timCON, timRX a timTX jsou timeouty v milisekundách pro připojení, příjem a vysílání. Jejich význam je maximální doba, po kterou budou blokovat funkce accept(), recv() a send(). Je to z důvodu, že třída CThread implementující chování vlákna obsahuje metodu terminate(), která nastaví vlajku (flag), že se má vlákno ukončit. Aby bylo možno ukončit vlákno nenásilně v konečném čase, je třeba zajistit, aby žádná blokující funkce neblokovala věčně, a zároveň do kódu na vhodná místa přidat kód, kontrolující, zda nebyl dán požadavek na ukončení vlákna. To, že aktuální vlákno má být ukončeno zjistíme voláním `bool CThread::isTerminated()`.

Význam timeoutu, lze znázornit v pseudokódu takto:

```
while(1) // nekonečná smyčka
{
  // příjem s timeoutem
  if (recv(buffer, len, timRX) == ERROR) break; // pokud chyba na příjmu, ukonči smyčku

  if (isTerminated()) break; // pokud vlákno bylo ukončeno ukonči smyčku

  // zpracuj příchozí data
}
 // ukonči metodu a tím i vlákno
 return 0;
```

Správný slovní význam timeoutu je: Je to maximální doba čekání při volání CThread::terminate() na ukončení blokujícího volání v jiném vlákně nebo jinak řečeno - Je to maximální doba do ukončení činnosti vlákna.[19]

Parametr pipeName je jméno roury. Je to cesta k souboru s jistými omezeními. V OS Linux se může jednat o obyčejný soubor např. /tmp/roura1. V OS Windows musí mít formát \\.\pipe\jmeno_roury. Vzdálené roury ve Windows jsou z bezpečnostních důvodu zakázané v implementaci knihovny CPPL.

Parametr proc je funkce vracející objekt třídy dědící PipeChannel. Jednoduchou metodou je využít lambda funkce. Má to tu výhodu, že do konstruktoru odvozené třídy můžeme vyplnit libovolné parametry z těla volající funkce. Důvod použití lambda funkcí je ten, že se vyhneme šablonám (template) v C++, které mohou působit problémy v tom, že musí být předem známé typy, pro které je šablony vytvářena. Důvodem, proč se nepředává rovnou samotná instance třídy je, že v rámci PipeServer je třeba nové instance vytvářet - jednu pro každého klienta. Zároveň se jedná o vlákna, což by situaci nadále komplikovalo. Použitím lambda funkce oddělíme paralelismus od vlastní implementace serveru. Implementaci rozhraní PipeChannel nezajímá kolik vláken je v systému a v jakém stavu jsou - o to se stará automaticky třída PipeServer.

Vytvoření instance vlastního serveru s vlastními parametry v konstruktoru ukazuje následující příklad:

```
Typ mujParametr;
San2::Cppl::PipeServer ps(
    SRV_PIPENAME,
    [&](CPPL_PIPETYPE handle, unsigned int timRX, unsigned int timTX){return new
MujServerReceiver(handle, timRX, timTX, mujParametr);},
    TIMEOUT_CON,
    TIMEOUT_RX,
    TIMEOUT_TX
);
```

[19] Teoreticky to není zcela korektní, protože tato doba může být ještě delší, jelikož požadavek na ukončení vlákna může přijít i před blokujícím voláním. Aby mechanismus ukončení vlákna fungoval, je nutné na dlouho trvajících voláních implementovat timeout a kontrolu ukončení.

Jak již bylo zmíněno, třída MujServerReceiver musí dědit PipeChannel. Jedinou třídu kromě konstruktoru, kterou je třeba implementovat, je třída receive(). V této třídě můžeme přijímat a odesílat data k čemuž můžeme využít následující metody bázových tříd:

```
ErrorCode send(char *data, int len);
ErrorCode read(char *data, unsigned int dataSize, unsigned int *bytesRead);
ErrorCode readSome(char *buffer, unsigned int bufferSize, unsigned int *bytesRead);
ErrorCode readDelimeter(char *output, unsigned int outputSize, unsigned int *outputLen,
char delimeter);
ErrorCode readLine(char *line, unsigned int lineSize);
ErrorCode send(const char *nullTerminatedString);
ErrorCode send(const std::string &data);
ErrorCode sendLine(const char *nullTerminatedString);
ErrorCode sendLine(void);
```

Pokud bychom chtěli odesílat a přijímat data mimo třídu, je to možné, protože metody jsou public. Je však nutné si uvědomit, že se ale vystavujeme riziku souběhu vláken. Myšlenka automatického vytváření a ukončování vláken jako instancí tříd zajišťuje to, aby nedošlo k souběhu vláken. Každý klient má svůj objekt, a pokud operuje v tomto objektu, nedojde k souběhu. Jakákoliv interakce s ostatními vlákny musí být zabezpečena proti souběhu.

5.2.8 Terminál pro správu uzlu sítě SAN

Protože cílem je mít server SAN běžící na pozadí a spravovat ho z jiného programu byl vytvořen terminál pro správu uzlu SAN, který funguje nad knihovnou CPPL. Ukázka výpisu terminálu:

```
$:san2$ ./terminal /tmp/sanode1
Welcome to SAN node terminal
node1>peers
IFACE status RX:conn   TX:conn
>> iadr: 00000000000000000000000000000000000000000000000000000000000000FF11
>> peer: 00000000000000000000000000000000000000000000000000000000000000FF21

IFACE status RX:conn   TX:conn
>> iadr: 00000000000000000000000000000000000000000000000000000000000000FF12
>> peer: 00000000000000000000000000000000000000000000000000000000000000FF31

node1>help
capsule - get/set capsule fields
send    - send capsule with fileds previously set in the capsule command
peers   - show interfaces addresses
exit    - close terminal connection
node1>
```

Na výpisu je vidět seznam připojených sousedů. V tomto případě jsou sousedící uzly dva. TX a RX určuje stav vysílání a příjmu. Conn znamená connected; diss - disconnected; err - chyba. V případě ukončení sousedního uzlu se stav změní a při znovuspuštění sousedního uzlu se stav změní opět na connected. iadr je adresa lokálního rozhraní a peer je adresa vzdáleného rozhraní.

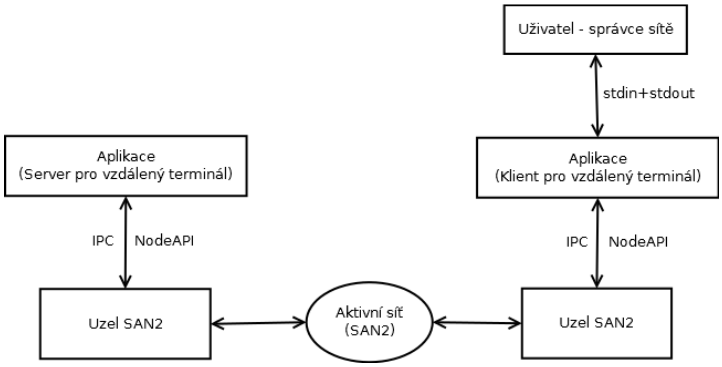
Program terminal má jediný parametr na příkazové řádce, a tím je jméno roury (cesta k rouře v souborovém systému). Jméno roury se dá u uzlu SAN2 nastavit v konfiguračním soboru v klíči ipcAddress.

Terminal umožňuje také zkonstruovat kapsuli a injektovat jí do vstupní fronty serveru.

6 Návrh aplikace pro vzdálenou správu

V této kapitole bude navržena aplikace pro vzdálenou správu uzlu SAN2 s využitím prostředků z předchozích kapitol.

6.1 Schéma vzdálené správy



obr. 17: Schéma vzdálené správy

Na obr. 178 je znázorněno navržené schéma vzdálené správy. Vzdálená správa se skládá ze dvou aplikací - terminálového serveru a terminálového klienta. Vzdálená správa leží mimo uzel serveru SAN2 a je poskytována „jako služba“ nikoliv jako nedílná součást serveru aktivní sítě.

Terminálový server používá API uzlu nejen k příjmu a odesílání kapsulí, ale také pro konfiguraci serveru aktivní sítě. Např. změny adres rozhraní, změny směrovací tabulky apod.

Terminálový klient odesílá kapsule na vzdálený uzel sítě a zpětně z něj kapsule přijímá. Portmapper zajišťuje, že kapsule bude dodána správné aplikaci.

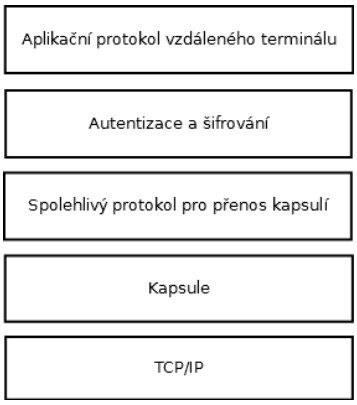
Příkazy poslané terminálovým klientem serveru jsou interpretovány terminálovým serverem a ten zadává požadavky přes NodeAPI uzlu SAN a tím mění jeho nastavení.

6.2 Přenosový protokol

Jak již bylo zmíněno v odstavci 3.3.7 protokol SAN je best-effort protokol. Je nutné zajistit mezivrstvu pro spolehlivý přenos kapsulí. Spolehlivý přenos lze např. zajistit ARQ (Automatic Repeat reQuest) schématem. Známe ARQ schémata jsou

- Stop-and-wait ARQ
- Go-Back-N ARQ
- Selective Repeat ARQ (nebo Selective Reject)

Nad touto vrstvou se nachází vrstva zajišťující autentizaci a šifrování. Poslední vrstvu tvoří aplikační protokol vzdáleného terminálu. Situaci shrnuje obr. 19.



obr. 18: Schéma protokolového zásobníku

6.2.1 Vrstva pro spolehlivý přenos kapsulí

Vrstva implementuje ARQ protokol Stop & Wait, který je pro potřeby vzdálené šifrované správy dostačující a jednoduchý na implementaci. Jak je vidět z obr. 189, protokol je provozován nad kapsulemi, tzn. je přímo v datovém poli kapsule.

Protokol Stop & Wait vyžaduje číslování paketů a vystačí si pouze s 2 čísly - tzn. 1 bitem. Protože nad tímto protokolem bude implementována šifrovací vrstva, která musí zajišťovat, že útočník neprohodil pakety je použit 64 bitový čítač, který je ve vyšší vrstvě podepsán schématem HMAC-SHA1-96.

Fomát paketu S&W ARQ:

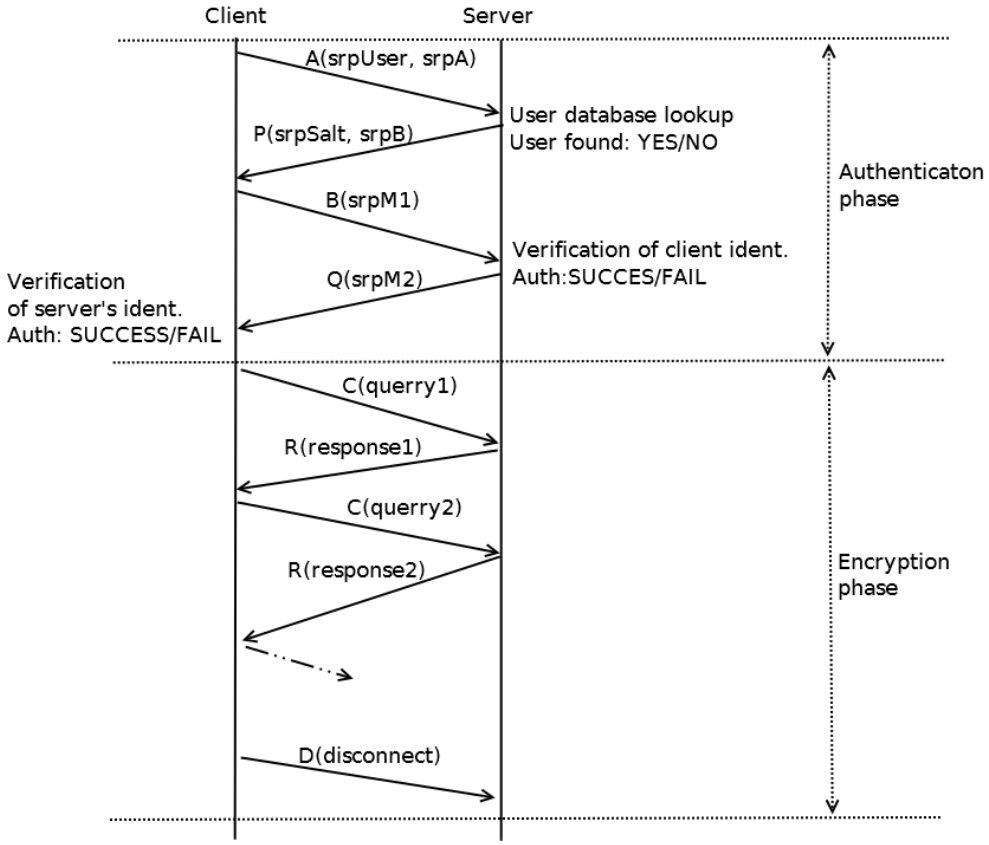
Protocol ID	Sequence Number (uint64_t big endian)	Payload
One byte 0x53	8 Bytes	Variable Length

Délku pole Payload lze vypočítat z celkové délky datového pole kapsule mínus 9 bajtů. První sekvenční číslo je 1. Dotaz klienta i odpověď serveru dodržuje stejný formát. Server odpovídá stejným sekvenčním číslem. Pokud dojde k výpadku paketu cestou ke k serveru, klient po vypršení timeoutu vysílání opakuje. Počet opakování je nastavitelný. Pokud dojde k výpadku paketu směr od serveru ke klientovi, pak si server vždy ukládá poslední odesílaný paket a po opakovaném dotazu klienta jen odpoví původním paketem. Protokol je minimalistický, ale plně dostačuje pro spolehlivý přenos kapsulí.

6.2.2 Šifrovací mezivrstva

Šifrovací mezivrstva je postavena na autentizačním mechanismu SRP6a s 1024 bitovým modulem. Šifrování využívá AES256-CTR a podepisování HMAC-SHA1-96. Před samotným popisem šifrování je vhodné uvést používané typy jednotlivých zpráv.

Typy zpráv jsou označeny písmeny abecedy postupně, jak jdou za sebou. Pro klienta A, B, C, D. Server odpovídá P, Q, R. Tok zpráv je znázorněn na obr. 19.



obr. 19: Schéma autentizace z hlediska typu posílaných zpráv. Např. Q(srpM2) znamená, že zpráva typu Q nese parametr srpM2.

Jednotlivé parametry autentizačního protokolu SRP byly popsány v sekci 4.3.4. Podrobné formátování zpráv lze snadno zjistit ze souboru shell/messageconstructor.cpp.

Šifrovaná část spojení, tj. zprávy C, R a D se šifrují a podepisují následující způsobem.

<table>
<tr><td colspan="8">Kapsule</td></tr>
<tr><td>Hlavička</td><td colspan="7">Data (Stop & Wait ARQ)</td></tr>
<tr><td></td><td>SOF(0x53)</td><td>SEQ#</td><td colspan="5">Data (Šifrovací mezivrstva)</td></tr>
<tr><td></td><td></td><td></td><td>SOF(0x45)</td><td>Typ (1B)
(C, R, D)</td><td>Chybový kód
(1B)</td><td>Data
aplikačního
protokolu</td><td>HMAC
podpis
(12B)</td></tr>
<tr><td colspan="6"></td><td>Zašifrováno</td><td></td></tr>
<tr><td colspan="2"></td><td>Podepsáno</td><td></td><td colspan="3">Podepsáno</td><td></td></tr>
</table>

obr. 20: Enkapsulace protokolu vzdálené šifrované správy

Nejprve se zašifrují data aplikačního protokolu šifrou AES256 v módu CTR. Následně se spolu se sekvenčním číslem zašifrovaná data podepíší algoritmem HMAC-SHA1-96. Čítač se skládá ze 3 částí. Prvních 6 bajtů je tajných a odvozuje se od hodnoty

srpK. Následuje 8 bajtů odpovídajících sekvenčnímu číslu. Poslední 2 bajty tvoří čítač bloků v rámci jednoho datagramu.[20]

Celkem existuje 6 klíčů. Šifrovací klíč, tajná část inicializačního vektoru (čítače) a podepisovací klíč. Server a klient používají odlišné klíče – tzn. zabezpečení směrem z klienta do serveru se provádí jinou trojicí klíčů než v opačném směru. Derivace se provádí hashovací funkcí SHA1 na jejíž vstupu je vždy parametr srpK (klíč pro dané sezení) a předdefinovaný řetězec specifikující typ klíče. Např. klíč pro podepisování klientovo zpráv se odvodí následujícím způsobem:

```
MAC_KEY_CLIENT = SHA1(srpK + “shell client MAC key“)
```

Odvození ostatních klíčů se provádí obdobně.

6.2.3 Aplikační protokol vzdálené správy

Samotný aplikační protokol vzdálené správy je přímočarý. Klient pošle dotaz v terminálu jako prostý text a dostane textovou odpověď od terminálového serveru. Terminálový server komunikuje s uzlem SAN přes NodeApi. Terminálový server obsahuje interpret a parser příkazů.

6.2.4 Shrnutí

Vzdálená šifrovaná správa je ve skutečnosti tvořena několika vrstvami protokolů. Šifrování není přímo závislé na přenosovém protokolu. Může fungovat jak nad kapsulemi, tak případně i na jiném protokolu. Výsledkem kombinace uvedených protokolů je použitelný terminál pro vzdálenou šifrovanou správu serveru SAN.

[20] Systém je obdobný jako u šifrování v IPv6.

7 Ethernet

Místo TCP/IP by teoreticky bylo možné využít přímo Ethernet bez IP k přenosu kapsulí mezi uzly sítě SAN. Tímto krokem by se odstranila závislost na pasivní IP sítí. Aby bylo možné využít Ethernet k přenosu kapsulí, bylo by třeba vybudovat spolehlivý přenosový protokol nad Ethernetem a tento nový protokol s novým ethertypem zaregistrovat do protokolového zásobníku (a packet chainu) příslušného operačního systému. To lze na operačních systémech Windows a Linux realizovat pouze s napsáním ovladačů a zavedením těchto ovladačů do jádra systému. Bylo by nunté napsat minimálně dva (zcela kódem odlišné) ovladače - jeden pro Linux a jeden pro Windows.

Zároveň bychom přišli o veškeré funkce, které protokol IP již implementoval - např. směrování. Přišli bychom i o TCP - řízení toku, prevence proti zahlcení, férové rozdělení kapacity linky mezi jednotlivé spojení.

Výhody:

- odstranění závislost na pasivní IP síti

Nevýhody:

- nutnost implementace nového transportního protokolu
- nutnost do jádra operačního systému implementovat ovladač
- závislost na konkrétní linkové vrstvě

Pro potřeby vzdálené šifrované správy nepřináší přímé využití Ethernetu pro přenos kapsulí vzhledem k časové náročnosti a složitosti implementace takového řešení žádné výhody.

8 Implementace a ověření funkčnosti

8.1 Jádro serveru SAN

Z důvodů výrazných změn v architektuře uzlu SAN bylo rozhodnuto v rámci této práce napsat nový server označovaný jako SAN2. SAN2 se nezakládá na kódu SAN1. SAN2 vychází z poznatků získaných během vývoje SAN1.

Implementace řeší síťová rozhraní, vstupní a výstupní fronty. Rozhraní jsou paralelní, každé má přijímací a vysílací vlákno. Implementace dále řeší komunikaci s aplikacemi přes lokální roury včetně návrhu API funkcí uzlu, které mohou aplikace volat. Popisu SAN2 je věnována třetí kapitola. Implementace je přiložena na CD.

8.1.1 Vyzkoušení komunikace mezi aplikacemi

Součástí projektu jsou dvě testovací aplikace apprx a apptx. Apprx čeká na příchozí kapsuli. Apptx vyšle kapsuli aplikaci apprx běžícím na druhém uzlu.

Komunikace pak probíha AppRx--->Uzel1---->Uzel2---->AppTx.

Pro ověření komunikace mezi aplikacemi lze použít následující postup:

1) Spustit uzly SAN:
    ./sanode node1.cfg
    ./sanode node2.cfg
    ./sanode node3.cfg

2) Spustit aplikaci naslouchající kapsule
    ./apprx

3) Spustit aplikaci, která odešla kapsuli druhé aplikaci
    ./apptx

Aplikace ./apprx by měla vypsat, že přijala kapsuli.

```
awaiting capsule
got: rxcapsule
```

8.2 RPC

V rámci SAN2 byl napsána jednoduchá RPC vrstva umožňující volání funkcí na vzdáleném uzlu včetně přenosu kapsulí mezi uzly. RPC vrstva umožňuje definovat, registrovat a vzdáleně volat funkce. Implementace se nachází ve složkách rpc, comm a stream.

8.2.1 Testování RPC

Pro testování RPC byly vytvořeny čtyři aplikace:

- Použití rour (knihovny CPPL) jako transportního protokolu
    - rcp_server - vykonává požadavky klienta
    - rpc_client - odesílá požadavky na zpracování
- Použití transportního protokolu TCP
- tcprpc_server - vykonává požadavky klienta
- tcprpc_client - odesílá požadavky na zpracování

Pro demonstraci byla použita funkce násobení. Operandy jsou odeslány serveru, který vrátí jejich součin.

Výpis programu rpc_server

```
san2$ ./rpc_server
RpcServer
ServerReceiver::run()
reg success
reg success
Rpc Test function SUCCESS
Multiply: 3 * 5 = 15
Rpc Test function SUCCESS
Multiply: 10 * 20 = 200
Multiply: 7 * 8 = 56
client exit
joined 1 and remains 0
```

Výpis programu rpc_client

```
san2$ ./rpc_client
RpcClient
reg success
reg success
multiplication result is: 15
multiplication result is: 200
multiplication result is: 56
```

Výpis programu tcprpc_server

```
san2$ ./tcprpc_server
RpcServer
ServerReceiver::run()
reg success
reg success
Rpc Test function SUCCESS
Multiply: 3 * 5 = 15
Rpc Test function SUCCESS
Multiply: 10 * 20 = 200
Multiply: 7 * 8 = 56
client exit
joined 1 and remains 0
```

Výpis programu tcprpc_client

```
proj/san2$ ./tcprpc_client
RpcClient
reg success
reg success
```

```
multiplication result is: 15
multiplication result is: 200
multiplication result is: 56
```

Shodnost výpisu klientů a serverů demonstruje nezávislost implementace RPC na transportním protokolu. Server je schopen obsloužit více klientů. Funkčnost RPC byla také ověřena při posílání kapsulí mezi dvěma uzly serveru SAN.

8.3 Autentizace a šifrování

Byla vytvořena knihovna s názvem DragonSRP, která se nachází ve složce crypto. Knihovna implementuje protokol SRP a poskytuje automatický test k ověření plné kompatibility s RFC2945 podle testovacích vektorů v RFC5054 Appendix B.

Knihovna také poskytuje vlastní implementaci symetrického podpisového algoritmu HMAC dle RFC2104. Test je součástí knihovny.

Knihovna je objektová, modulární. Umožňuje použití libovolné hashovací funkce jak s SRP, tak i s HMAC.

8.3.1 Programu dokazující správnost implementace protokolu SRP

Program rfc_test testuje korektnost implementace autentizačního protokolu SRP:

```
san2/crypto/apps$ ./rfc_test
INFO: k matches rfc5054 ok
INFO: calculated k= 7556AA045AEF2CDD07ABAF0F665C3E818913186F
INFO: x(user generation) matches rfc5054 ok
INFO: calculated x(user generation)= 94B7555AABE9127CC58CCF4993DB6CF84D16C124
INFO: verificator matches rfc5054 ok
INFO: calculated verificator=
7E273DE8696FFC4F4E337D05B4B375BEB0DDE1569E8FA00A9886D8129BADA1F1822223CA1A605B530E379BA4
729FDC59F105B4787E5186F5C671085A1447B52A48CF1970B4FB6F8400BBF4CEBFBB168152E08AB5EA53D15C
1AFF87B2B9DA6E04E058AD51CC72BFC9033B564E26480D78E955A5E29E7AB245DB2BE315E2099AFB
INFO: A matches rfc5054 ok
INFO: calculated A=
61D5E490F6F1B79547B0704C436F523DD0E560F0C64115BB72557EC44352E8903211C04692272D8B2D1A5358
A2CF1B6E0BFCF99F921530EC8E39356179EAE45E42BA92AEACED825171E1E8B9AF6D9C03E1327F44BE087EF0
6530E69F66615261EEF54073CA11CF5858F0EDFDFE15EFEAB349EF5D76988A3672FAC47B0769447B
INFO: B matches rfc5054 ok
INFO: calculated B=
BD0C61512C692C0CB6D041FA01BB152D4916A1E77AF46AE105393011BAF38964DC46A0670DD125B95A981652
236F99D9B681CBF87837EC996C6DA04453728610D0C6DDB58B318885D7D82C7F8DEB75CE7BD4FBAA37089E6F
9C6059F388838E7A00030B331EB76840910440B1B27AAEAEEB4012B7D7665238A8E3FB004B117B58
INFO: M1= 3F3BC67169EA71302599CF1B0F5D408B7B65D347
INFO: M2= 9CAB3C575A11DE37D3AC1421A9F009236A48EB55
INFO: u matches rfc5054 ok
INFO: calculated u= CE38B9593487DA98554ED47D70A7AE5F462EF019
INFO: x (client challenge) matches rfc5054 ok
INFO: calculated x(client challenge)= 94B7555AABE9127CC58CCF4993DB6CF84D16C124
INFO: S client matches rfc5054 ok
INFO: calculated S(client)=
B0DC82BABCF30674AE450C0287745E7990A3381F63B387AAF271A10D233861E359B48220F7C4693C9AE12B0A
6F67809F0876E2D013800D6C41BB59B6D5979B5C00A172B4A2A5903A0BDCAF8A709585EB2AFAFA8F3499B200
210DCC1F10EB33943CD67FC88A2F39A4BE5BEC4EC0A3212DC346D7E474B29EDE8A469FFECA686E5A
```

```
INFO: S server matches rfc5054 ok
INFO: calculated S(server)=
B0DC82BABCF30674AE450C0287745E7990A3381F63B387AAF271A10D233861E359B48220F7C4693C9AE12B0A
6F67809F0876E2D013800D6C41BB59B6D5979B5C00A172B4A2A5903A0BDCAF8A709585EB2AFAFA8F3499B200
210DCC1F10EB33943CD67FC88A2F39A4BE5BEC4EC0A3212DC346D7E474B29EDE8A469FFECA686E5A
INFO: master session secret server==client ok
INFO: K(master session secret): 017EEFA1CEFC5C2E626E21598987F31E0F1B11BB
authentification successful
ALL TESTS PASSED OK SUCCESS
```

Test proběhl bez chyb. Vypočtené hodnoty se shodují s testovacími vektory RFC5054.

8.3.2 HMAC Test

Pro ověření, zda jsou podpisy zpráv generovány korektně, byl implementován program hmac_md5_test, který testuje vypočtené hodnoty proti testovacím vektorům v RFC2104. Ukázka běhu programu:

```
crypto/apps$ ./hmac_md5_testvector
calc digest1: 9294727A3638BB1C13F48EF8158BFC9D
real digest1: 9294727A3638BB1C13F48EF8158BFC9D
calc digest2: 750C783E6AB0B503EAA86E310A5DB738
real digest2: 750C783E6AB0B503EAA86E310A5DB738
calc digest3: 56BE34521D144C88DBB8C733F0E8B3F6
real digest3: 56BE34521D144C88DBB8C733F0E8B3F6
Vectors match. Test SUCCESSFUL.
```

Test proběhl bez chyb. Vypočtené hodnoty se shodují s testovacími vektory RFC2104.

8.3.3 Programy client_test, server_test a create_user

Tyto konzolové programy umožňují vyzkoušet autentizaci na lokálním počítači. Server ověřuje uživatele, klient se přihlašuje k serveru. Program create_user je pomocný program pro vytvoření uživatele resp. jeho verifikátoru a soli, které je potom zadáno do databáze serveru.

8.3.4 Program cryptotest

Program cryptotest ověřuje správnou funkčnost šifrování a dešifrování paketů v rámci datagramového programu. Jedná se o test tříd DatagramEncryptor a DatagramDecryptor. Program zašifruje předem daný řetězec metodou DatagramEncryptor::encryptAndAuthenticate a poté dešifruje metodou DatagramDecryptor::decryptAndVerifyMac. Funkci programu demonstruje následující výpis:

```
san2/crypto/apps$ ./cryptotest
data: hello
data: 68656C6C6F00
encpacketLen: 26
encpacket: 0100000000000000A6EF30EA3FB24CF666AE9E5C66372CB4479B
decrypted text: hello
finished
```

```
enc/dec seems ok
```

Paket se podařilo úspěšně zašifrovat a dešifrovat. Šifrovaná zpráva je delší o sekvenční číslo paketu a o HMAC. Celý paket včetně sekvenčního čísla je zašifrován a připojen podpis dle algoritmu HMAC. Následně je dešifrován a zkontrolován MAC kód.

8.4 Knihovna pro meziprocesovou komunikaci (CPPL)

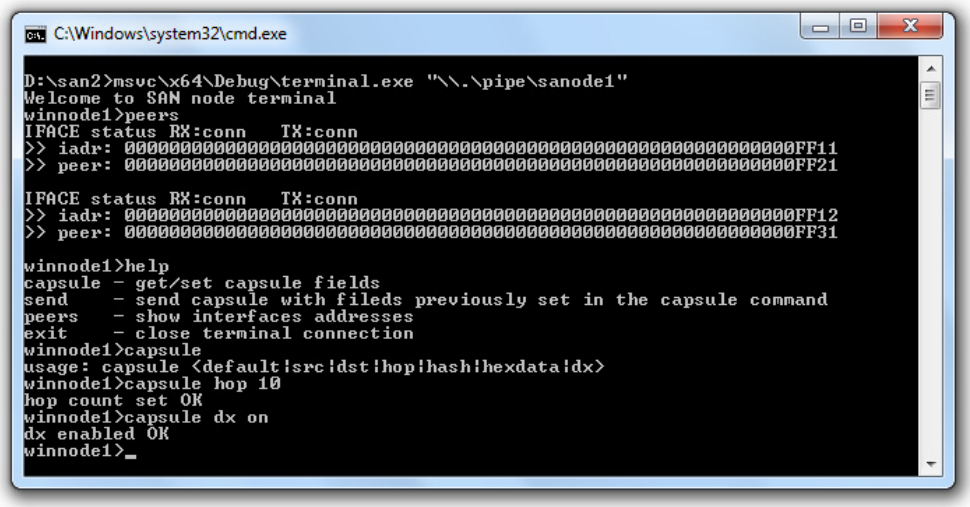
V rámci této práce bylo nutno navrhnout a implementovat multiplatformní knihovnu pro lokální komunikaci mezi procesy. Implementovaná knihovna CPPL (Cross-Platform Pipe Library) funguje jak pod Windows, tak i pod Linuxem. Na OS Windows používá pojmenované roury, na OS Linuxu doménové sokety AF_UNIX. Navenek je rozhraní nezávislé na použitém operačním systému.

Knihovna podporuje více klientů. Každému klientu automaticky alokuje jeho vlastní instanci zděděného objektu. Paralelismus a thread management je oddělený od vlastního výkonného kódu konkrétního serveru.

Knihovnu kromě serveru SAN2 využívají např. výše uvedené programy rpc_server a rpc_klient pro test RPC.

8.5 Terminál pro správu serveru SAN

Pro konfiguraci a správu serveru byl nejprve navržen a implementován terminál pracující nad knihovnou CPPL. Později byl navržen, implementován a otestován server a klient vzdálené šifrované správy pracující jako separátní aplikace používající NodeAPI.



obr. 21: Screenshot terminálu na Windows.

8.6 Terminál vzdálené šifrované správy serveru SAN

Byl implementován terminál pro vzdálenou šifrovanou správu serveru SAN. Jak u serveru, tak u klienta, je třeba specifikovat API adresu uzlu, na kterém poběží. Server načítá jména a hashovaná hesla ze souboru pass.pwd.

Parametry terminálového serveru a klienta:

```
./shell_client <SanNodeApiAddress> <ShellServerSanAddress> <port> <user> <passwrod>
./shell_server <SanNodeApiAddress> [port=2201]
```

Ukázka připojení k serveru (veškerá komunikace je zabezpečena šifrováním):

```
$ ./shell_client /tmp/sanode1api FF21 2201 san smartactivenode
Using source address: 0000000000000000000000000000000000000000000000000000000000000FF11
Authentication SUCCESSFUL.
node2>uptime
0d 0h 0m 57s
node2>version
SAN Node v1.0
Shell server version 1.0
node2>peers
IFACE status RX:conn   TX:conn
>> iadr: 0000000000000000000000000000000000000000000000000000000000000FF21
>> peer: 0000000000000000000000000000000000000000000000000000000000000FF11

IFACE status RX:conn   TX:conn
>> iadr: 0000000000000000000000000000000000000000000000000000000000000FF22
>> peer: 0000000000000000000000000000000000000000000000000000000000000FF32


node>
```

Závěr

Podařilo se implementovat knihovnu zajišťující autentizaci, šifrování a zabezpečení zpráv během přenosu. Knihovna je nezávislá na transportním protokolu.

Byla navržena a implementována multiplatformní knihovna poskytující jednotné API pro komunikaci více procesů na operačních systémech Windows a Linux.

Z důvodů výrazných změn v architektuře uzlu SAN bylo rozhodnuto v rámci této práce napsat nový server označovaný jako SAN2. Návrh a implementace síťové architektury SAN2 tvoří velkou část této práce. Implementace SAN2 byla nutná pro realizaci vzdálené šifrované správy serveru aktivní sítě. Implementace řeší meziprocesovou komunikaci (API) a komunikaci mezi uzly sítě.

Terminály pro správu serveru byly implementovány 2. Nejdříve lokální terminál pracující nad knihovnou pro meziprocesovou komunikaci, který je nedílnou součástí uzlu SAN. Tento terminál byl vytvořen mj. k ověření funkčnosti knihovny pro meziprocesovou komunikaci. Druhý, a mnohem složitější, je terminál zajišťující vzdálenou šifrovanou správy uzlu SAN. Terminál pracuje nad kapsulemi a je zcela nezávislý na nižších transportních protokolech. Na rozdíl od OpenSSL a OpenSSH nevyžaduje použití či tunelování TCP nebo UDP. Terminálový server i klient běží jako separátní aplikace, které se mohou kdykoliv připojit či odpojit od uzlu SAN.

Během práce byl napsán zdrojový kód o více než 40 000 řádkách v C++.

Do budoucna bude třeba implementovat část serveru aktivní sítě, která bude poskytovat bezpečné prostředí pro běh aktivního kódu kapsulí.

Seznam zkratek a vysvětlivek

AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CPPL	Cross-Platform Pipe Library
CTR	Counter
EKE	Encrypted Key Exchange
FAV	Fakulta aplikovaných věd
HMAC	Keyed-hash Message Authentication Code
IP	Internet Protocol
IPC	Inter-process communication
KIV	Katedra informatiky a výpočetní techniky
PAKE	Password Authenticated Key Agreement
PKI	Public key infrastructure
RFC	Request For Comments
RPC	Remote Procedure Call
SAN	Smart Acitve Node - projekt serveru aktivní sítě vyvíjený na Západočeské univerzitě v Plzni
SRP	Secure Remote Password Protocol
SSH	Secure Shell Protocol
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	Universal Datagram Protocol
ZKPP	Zero-Knowledge Password Proof
ZČU	Západočeská univerzita v Plzni

Literatura

1. Douglas Bruey. SNMP: Simple? Network Management Protocol. [Online] Rane Corporation, Prosinec 2005. [Citace: 2. Březen 2013.] http://rane.com/note161.html.

2. RFC1157 - A Simple Network Management Protocol (SNMP). [Online] The Internet Engineering Task Force. http://tools.ietf.org/html/rfc1157.

3. Kozierok, Charles M. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference.* : No Starch Press, 2005.

4. Net-SNMP. [Online] http://www.net-snmp.org/.

5. RFC1445 - Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2). [Online] http://tools.ietf.org/html/rfc1445.

6. RFC1910 - User-based Security Model for SNMPv2. [Online] http://tools.ietf.org/html/rfc1910.

7. RFC2574 - User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3). [Online] http://tools.ietf.org/html/rfc2574.

8. Postel, J. a Reynolds, J. RFC854 - TELNET PROTOCOL SPECIFICATION. [Online] květen 1983. http://tools.ietf.org/html/rfc854.

9. Kantor, B.;. RFC1282 - BSD Rlogin. [Online] prosinec 1991. http://www.ietf.org/rfc/rfc1282.txt.

10. RFC4251 - The Secure Shell (SSH) Protocol Architecture. [Online] http://tools.ietf.org/html/rfc4251.

11. RFC4252 - The Secure Shell (SSH) Authentication Protocol. [Online] leden 2006. http://www.ietf.org/rfc/rfc4252.txt.

12. *Enhancing Performance of Networking Applications by IP Tunneling through Active Networks.* Sýkora, Jakub a Koutný, Tomáš; Networks (ICN), 2010 Ninth International Conference 11-16.duben 2010

13. Rejda, Michal. *Smart Active Node.* KIV, ZČU. 2008.

14. Smart Active Node. [Online] [Citace: 2013. leden 20.] http://www.san.zcu.cz/.

15. Wetherall, David, Guttag, John a Tennenhouse, David. *ANTS: Network Services without the Red Tape.* místo neznámé : IEEE, 1999.

16. *PLANet: An Active Internetwork.* Hiscks, Michael, a další. místo neznámé : University od Pennsylvania, 2. srpen 1998.

17. PLAN - A Packet Language for Active Networks. [Online] [Citace: 6. únor 2013.] http://www.cis.upenn.edu/~dsl/PLAN/.

18. WriteFile function. *MSDN.* [Online] Microsoft. [Citace: 2013. 3 3.] http://msdn.microsoft.com/cs-cz/library/windows/desktop/aa365747(v=vs.85).aspx.

19. Using Pipes. *MSDN.* [Online] Microsoft, 27. 10 2012. [Citace: 2013. 3 5.] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365799(v=vs.85).aspx.

20. American National Project Standard. *Information Technology - AT Attachment with Packet Interface - 7.* 2004.

21. Štěpánek, Petr. *Code Distribution in Active Networks.* KIV, ZČU. 2009.

22. Syrovátka, Jan. *Code Interpreter for Smart Active Node.* KIV, ZČU. 2009.

23. Sýkora, Jakub. *Tunneling IP Applications in Active Networks.* KIV, ZČU. 2009.

24. *Lessons Learned on Enhancing Performance of Networking Applications by IP Tunneling through Active Networks.* Sýkora, Jakub a Tomáš, Koutný. Volume 3, Numbers 3 & 4, 2010, International Journal on Advances in Internet Technology, stránky 233-244. ISSN: 1942-2652.

25. *Simulating Distributed Applications in an Active Network.* Koutný, Tomáš a Šafařík, Jiří. Ljubljana, Slovenia : Proceedings of 6th Eurosim Congress, 2007, str. 379.

26. *Load Redistribution in Heterogeneous Systems.* Koutný, Tomáš a Šafařík, Jiří. Athens, Greece : Proceedings of the Third International Conference on Autonomic and Autonomous Systems, Published in IEEE Digital Library, 2007, stránky 24-32.

27. *Gradient Method with Topology Discovery for Load-Balancing in Active Networks.* Koutný, Tomáš a Šafařík, Jiří. Brno, Czech Republic : Proceedings of 11th Annual IEEE International Conference on the Engineering of Computer Based Systems, 2004, stránky 75-85.

28. *Maintaining Communication Channels for Migrating Processes in the Environment of Active Networks.* Koutný, Tomáš a Šafařík, Jiří. Insbruck, Austria : Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks - PDCN , 2005, stránky 100-105. In cooperation with IEEE Technical Committee on Parallel Processing (TCPP).

29. *RFC2945 - The SRP Authentication and Key Exchange System.* Wu, T. : The Internet Engineering Task Force, Září 2000. Stanford University.

30. RFC5054 - Using the Secure Remote Password (SRP) Protocol for TLS Authentication. [Online] 2007. http://tools.ietf.org/html/rfc5054.

31. RFC1901 - Introduction to Community-based SNMPv2. [Online] http://tools.ietf.org/html/rfc1901.

32. RFC4253 - The Secure Shell (SSH) Transport Layer Protocol. [Online] http://tools.ietf.org/html/rfc4253.

33. Wetherall, David J. *Service Introduction in an Active Network.* : Massachusetts Institute of Technology, 1998.