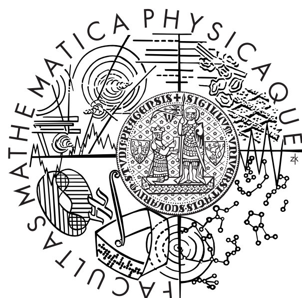
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Dušan Knop

Systém kontroly běhu vzdálených strojů a služeb

Středisko informatické sítě a laboratoří

Vedoucí bakalářské práce: Dan Lukeš
Studijní program: Informatika – správa počítačových systémů

2010

Chtěl bych na tomto místě poděkovat panu Danu Lukešovi za vedení práce a rady při tvorbě softwarového řešení a svým rodičům za podporu při studiu.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Dušan Knop

Obsah

1 Účel a návrh **6**
- 1.1 Zadání projektu . . . 6
- 1.2 Uvažované varianty aplikace . . . 7
- 1.3 Moduly a komunikace . . . 9
- 1.4 Nagios . . . 10

2 Z pohledu koncového uživatele **11**
- 2.1 Co tento systém poskytuje . . . 11
- 2.2 Současné možnosti . . . 11
- 2.3 Instalace . . . 12
- 2.4 Konfigurace . . . 12
- 2.5 Spuštění . . . 14

3 Komunikační protokol **17**
- 3.1 Protokol . . . 17
- 3.2 Architektura . . . 18
- 3.3 Autentizace . . . 18
- 3.4 Navržený komunikační protokol . . . 19
- 3.5 Průběh protokolu . . . 20

4 Program tester **21**
- 4.1 Datové struktury . . . 21
- 4.2 Po spuštění . . . 22
- 4.3 Produkční část . . . 22

5 Moduly systému **24**
- 5.1 Moduly obecně . . . 24
- 5.2 Moduly pro odeslání informace uživateli . . . 26
- 5.3 Moduly pro testování . . . 27

5.4 Funkce pro usnadnění tvorby modulů 27

6 Závěr **29**

Literatura **33**

Název práce: Systém kontroly běhu vzdálených strojů a služeb
Autor: Dušan Knop
Katedra (ústav): Správa informatické sítě a laboratoří
Vedoucí bakalářské práce: Dan Lukeš, SISAL
e-mail vedoucího: dan@obluda.cz

Abstrakt: V předložené práci studujeme možnosti vytvoření systému pro monitorování počítačů a služeb v prostředí počítačových sítí. Je třeba, aby systém umožňoval přidávání nových druhů testů podle potřeby koncového uživatele. Výsledkem práce je také implementace jedné z uvažovaných variant takovéhoto systému v jazyce C pro operační systém FreeBSD. Konkrétní dostupné testovací metody jsou implementovány jako samostatné moduly, komunikující s koordinátorem celého systému. Je také třeba, aby podobným mechanismem bylo možné upozornit uživatele na výpadek ve funkčnosti počítače či nějaké jím poskytované služby.

Klíčová slova: počítačové sítě, monitorování, dohled

Title: Supervision system for network remote systems and services
Author: Dušan Knop
Department: Network and Labs Management Center
Supervisor: Dan Lukeš, SISAL
Supervisor's e-mail address: dan@obluda.cz

Abstract: In the present work we study possibilities of creating a system for monitoring of computers and services in computer networks. For such a system it is required the possibility of adding new species of tests into application on users demand. The work also presents how could such a system be created in C programming languague for FreeBSD operating system. All particular available tests are implemented as a standalone modules. Modules communicate with coordinator through computer network. For this system it is also required to use the same mechanism to warn user about any failure while testing functionality of specified computer or service.

Keywords: computer networks, monitoring, supervision

Kapitola 1

Účel a návrh

1.1 Zadání projektu

Cílem projektu bylo navrhnout a vytvořit (implementovat) systém, který bude zajišťovat periodicky se opakující testy v prostředí počítačové sítě. Systém má být implementován primárně pro operační systém FreeBSD. Programovacím jazykem budiž jazyk C s možnými objektovými rozšířeními. Za vhodnější řešení však bude považováno vytvoření zcela procedurální aplikace.

Upřesněním zadání jsem následně zjistil, že povaha ani účel systémem zajišťovaných testů nebyla specifikována a tedy je třeba, aby mnou vytvořený systém podporoval doplňování nových druhů testů, které budou kompatibilní s dosud vytvořenou částí systému. Tím se dosáhne maximální univerzálnosti celého projektu.

Účelem projektu bude poskytnout jednoduchý nástroj pro monitorování počítačové sítě, který bude určen převážně pro menší počítačové sítě. Z tohoto předpokladu vyplývá, že by se mělo jednat o co možná nejjednodušší systém, který bude lehce pochopitelný i méně zkušeným správcům. Dále pak je možné předpokládat, že v takovýchto sítích se nebudou vyskytovat uživatelé, kteří by se pokoušeli o aktivní útoky proti monitorovacímu systému nebo se snažili z tohoto systému získávat nějaká data. Tomuto faktu budou podřízeny zjednodušené autentizační a bezpečnostní fáze komunikačního protokolu.

Důraz by měl být kladen především na jednoduchost. Čím jednodušší bude doplňování nových druhů testů do projektu, tím větší univerzálnosti bude možné

dosáhnout. A tím jednodušeji se bude moci mnou vytvořený nástroj přizpůsobovat potřebám konkrétní sítě, ve které bude nasazen a provozován.

Je třeba, aby systém podporoval též nástroje pro informování uživatele o nastalé chybě v průběhu testu a o její povaze. Vzhledem k faktu, že postupy pro testování i pro případné odesílání informace o chybě jsou analogické, nebudu prozatím mezi těmito dvěma částmi aplikace (testovacího systému) rozlišovat.

Kromě univerzálnosti aplikací zajišťovaného monitoringu bylo požadováno, aby konfigurace umožňovala hierarchické uspořádání probíhajících testů. To znamená umožnit uživateli definovat závislosti mezi testovanými prostředky a službami, například proto, aby nebyl upozorňován, že na nedostupném počítači neodpovídá žádná z jím poskytovaných služeb a podobně.

1.2 Uvažované varianty aplikace

Na samém počátku jsem se rozhodoval, jaký druh aplikace bude výsledkem mé práce. Uvažované druhy výsledné aplikace byly modulární a monolitická. Jelikož se do modulárních aplikací snáze zasahuje a také dopisování nových částí je jednodušší, zvolil jsem vytvoření modulární aplikace. Monolitická aplikace by svou povahou znesnadňovala přidávání nových druhů testů a některé by možná zcela znemožnila.

Další důležité rozhodování bylo, zda bude tento systém tvořen rovnocennými aplikacemi (označovanými jako peer), zda bude obsahovat dvě aplikační části - koordinátora a výkonné jednotky - popřípadě více koordinátorů v celém systému (jedná se o tzv. hybridní systémy). Všechny přístupy mají svá pro i proti. Z tohoto důvodu je zde (alespoň z části) rozeberu a zdůvodním svou volbu.

Pokud systém obsahuje jeden řídící prvek - takzvaného koordinátora, hovoří se o systému s koordinátorem. Takovýto význačný prvek je jediný, který obsahuje všechny dostupné informace o celém systému, tyto informace spravuje a rozhoduje o veškerém dění v systému. Toto je jistě přínosem pro administrátora takové aplikace, neboť konfigurace a údržba se provádí centralizovaně a to právě na počítači obsahujícím koordinátorský program. Tento fakt je však nejen výhodou, ale i nevýhodou, jelikož všechna činnost je závislá na jedné jediné části.

Pokud by se však v systému vyskytoval jen jeden druh aplikací, které by tedy byly významově shodné, bylo by třeba, aby měla každá aplikace jistou část informace o celém systému. Každý prvek takového systému je pak částečně závislý na zbylých částech (resp. na těch momentálně dostupných). Tyto prvky v počítačové síti pak jednají do jisté míry samostatně. Buď mohou dělat vše co umějí sami a ostatní využívat až při nedostatku výpočetní síly nebo pokud vyžadovanou operaci sami nepodporují. Je také nutné, aby se navzájem informovaly o své přítomnosti na síti (poskytovaných a požadovaných službách, . . .) a případně i volily koordinátora mezi sebou (toto řešení rozeberu později v této sekci). Tento druh aplikace by s sebou nesl výhodu implementování jen jednoho druhu aplikace (všechny části systému jsou stejné), do jejíhož kódu by se pak doplnila část programu obsluhující daný test. Další výhodou by pak mohlo být oddělené testování v podsítích a v případě výpadku síťového spojení samostatné testování na dostupných segmentech. Mezi nevýhody pak patří zejména obtížnější (rozdistribuovaná) správa celého systému a také větší náročnost na užitou přenosovou kapacitu počítačové sítě.

Hybridní systém, jak už jsem zmínil, obsahuje více koordinátorů. Toto řešení má více mozností. Je možné, aby v systému koexistovalo více nezávislých koordinátorů (každý může obsluhovat svou přidělenou část sítě) a pokud o sobě navzájem netuší, pak nevidím rozdíl mezi takovýmto systémem a více spuštěnými instancemi systému popsaného výše v této části. Je tedy třeba, aby o sobě koordinátoři věděli a rozhodovali, kdo bude provádět testování. Toto je ovšem možné navrhnout i implementovat mnoha různými způsoby. Například tak, že v systému se bude vyskytovat několik koordinátorů a vždy bude právě jeden z nich aktivní a ostatní budou jen záložní. V takovýchto systémech je ale třeba pamatovat na mnoho situací, které mohou nastat. Například co se má stát, když se hlavní koordinátor vráti zpět do systému (obnoví se připojení do jeho části sítě, restartuje se, . . .), Tedy je třeba navrhnout další protokol pro vzájemnou komunikaci koordinátorů nebo jiné obdobné řešení.

Hybridní systém lze vytvořit též na základě peer-to-peer aplikací. Pak se jedná o takzvaný systém s voleným koordinátorem (koordinátory). Pro tyto účely je třeba pro peery v systému definovat nejen protokol, pro zjištění, kdo je dostupný, ale i protokol pro volbu koordinátora. Tyto protokoly bývají velmi komplexní. Pokud by chtěl čtenář zjistit více o těchto systémech či protokolech, může nahlédnout do libovolné knihy o distribuovaných systémech (např. v [6]).

Právě ona jednoduchost ve spravování koordinovaných aplikací na jedné straně a neohrabanost peer-to-peer aplikací na straně druhé rozhodly ve prospěch prvně jmenovaných. K této volbě však přispělo i to, že se domnívám, že vytvoření systému užívajícího koordinátora je jednodušší a snáze se v něm odhalují případné chyby - tedy by měl být z pohledu uživatele bezpečnější. Z podobných důvodů jsem neuvažoval ani řešení pomocí hybridních systémů, které vykazují větší požadavky na prostupnost sítě (což sice dnes nebývá zásadní, ale domnívám se, že není dobré síť zatěžovat, pokud to není nutné a je možné uvolnit kapacitu pro jiné účely) a také by se ze systému vytratila ona zamýšlená jednoduchost nastavení. Hybridní systémy se nejen obtížněji spravují, ale je také o mnoho obtížnější hledání případných chyb. Dle mého názoru je bezchybný (i když jednodušší) systém lepší než systém propracovanější, u kterého ale nikdo nedokáže říci, zda neobsahuje vážné chyby.

1.3 Moduly a komunikace

Jak bylo zmíněno v předcházejícím textu, v době návrhu a vytváření výsledné aplikace byly dostupné pouze obecné informace ohledně budoucích požadavků uživatele na cíle a způsoby testování. Jednou ze základních otázek bylo, zda aplikační moduly, které tyto testy budou zajišťovat, budou spouštěny na počítači s koordinátorskou aplikací či nikoliv. Pokud by oba druhy programů sdílely jeden počítač, bylo by jistě nejvýhodnější užít nejvhodnější z bohaté palety prostředků lokální komunikace(tzv. lokálních komunikačních primitiv). Avšak pokud bude třeba, aby obě strany komunikovaly přes počítačovou síť, použijí se jiné komunikační prostředky(např. BSD sockets).

V průběhu specifikační fáze jsem byl upozorněn, že jelikož je možné testovat mnoho různých zdrojů či služeb, neměla by (pokud možno) žádná má volba v průběhu vývoje ztěžovat nebo dokonce zcela znemožňovat nějaký test. Tomuto se přirozeně nelze zcela vyhnout, ale je třeba mít tuto skutečnost stále na paměti a v případě jakýchkoli rozhodnutí k tomuto faktu přihlížet.

Mohlo by se hodit testovat i systémové prostředky lokálního charakteru (např. dostatek volné kapacity na lokálním disku) a to nejen pro počítač obsahující koordinátorskou část aplikace, ale i pro jiné významné počítače v síti (např. databázový server, . . .). Tento fakt významnou rolí přispěl k rozhodnutí, že je třeba,

aby koordinátor s moduly komunikoval přes počítačovou síť pomocí k tomu určených prostředků meziprocesové komunikace.

1.4 Nagios

Nagios je opensource systém zabývající se podobnou činností jako mnou implementovaný projekt. Projekt Nagios je zdarma dostupný na Internetové adrese `http://www.nagios.org/download/`. Z tohoto softwarového řešení jsem se poučil převážně při tvorbě modulů (konkrétně při návrhu modulu, který bude spouštět lokální programy, což je standardním řešením v systému Nagios).

Tento systém, pokud má spouštět modul na vzdáleném počítači, použije protokol SSH k připojení se a spustí program lokálně. Toto řešení je velmi bezpečné a pro některá citlivá data jistě zcela adekvátní. Já jsem však uvažoval, že jsou naopak data, jejichž citlivost je nulová (či skoro nulová). U těch mi nepřijde používání protokolu SSH na místě. Jednodušším protokolem si lze ušetřit část síťového provozu. Toto by mělo být patrné i s ohledem na to, že moderní řešení počítačových sítí dělí sítě na VPN podle jejich účelu. VPN je zkratkou pro virtuální privátní síť - virtual private network. Jedná se o počítačovou síť, která využívá fyzické komponenty existujících privátních (vnitropodnikových) a veřejných sítí (Internet) a existuje tedy pouze virtuálně. Jednou z nich pak může být monitorovací VPN, jejíž síťový provoz je pak velmi jednoduše oddělen od ostatního. Dalším možným účelem VPN je připojení uživatelů (z domova či na služební cestě) do prostředí podnikové sítě.

Kapitola 2

Z pohledu koncového uživatele

2.1 Co tento systém poskytuje

Program ”tester” a jeho moduly se starají o periodický test dostupnosti prostředků v počítačové síti. Prostředky, které budou systémem testovány závisejí na dostupných modulech a samozřejmě i konfiguraci. Program ”tester” pak působí jako aplikační jádro (koordinátor), které spravuje moduly a uživatelská data (konfigurace), periodicky žádá moduly o testy a ukládá informace o chybách a nedostupnosti. Zároveň se stará o management (správu a odesílání) chybových zpráv.

Program je otevřen novým možnostem. Pokud se libovolný jeho uživatel rozhodne, že je třeba testovat například novou službu (tedy pro ni dosud neexistuje modul), je možné takovýto modul relativně jednoduše dopsat. Toto jistě platí i pro moduly zajišťující odesílání chybových zpráv koncovému uživateli. Tento systém ocení také uživatelé, kteří chtějí monitorovací nástroj, který (jak jsem již zmínil) využívá nutné minimum síťového provozu. Nespornou výhodou by pro uživatele též mohlo být i to, že je program poskytován zdarma[1].

2.2 Současné možnosti

Z testovacích modulů je dostupný pouze modul pro testování, zda jiný systém odpovídá na zprávy ICMP ECHO a modul sloužící jako mezivrstva pro spouštění lokálních programů a navracení jejich výsledku. Druhý jmenovaný by však spolu

[1] v rámci FreeBSD licence (známé též jako New BSD licence)

s možností shellových skriptů a bohatou paletou dostupných lokálních monitorovacích programů v systému FreeBSD měl poskytnout dostatek nástrojů pro zamýšlené testy síťových či lokálních prostředků. Pokud by přesto budoucí uživatel postrádal potřebný testovací modul, může jej buď sám napsat nebo využít toho modulu a ulehčit si práci - bude tvořit lokálně spouštěnou aplikaci, přičemž nemusí brát ohledy na definovaný komunikační protokol.

Z informačních modulů je prozatím dostupné odesílání e-mailů a zápis do lokálního logu. Také bych chtěl připomenout, že modul pro spouštění lokálních programů lze využít také pro odeslání chyby, kde by opět měl posloužit k odstínění programátora od nutnosti studovat a implementovat stanovený komunikační protokol. Zde bych ale spíše doporučil věnovat pozornost sekci 5, ve které se věnuji implementaci modulů.

2.3 Instalace

Vše se instaluje standardně ze zdrojových kódů za pomoci programu `make`. Instalace všech částí je téměř totožná. Vždy postačí zadat následující posloupnost příkazů:

```
$ make install
$ make clean
```

Tyto příkazy způsobí přeložení zdrojového kódu příslušné aplikace a následné zkopírování takto přeloženého programu na jeho místo v systému. Instalace implicitně použije podadresář `/usr/bin/` pro konkrétní program pod jeho jménem - pro tester tedy úplná instalační cesta bude `/usr/bin/tester`. Pokud by si uživatel přál nainstalovat soubory do jiného adresáře v systému, je třeba, aby před instalací programu upravil soubor `install_script.sh` ve složce se staženými zdrojovými kódy. V tomto souboru je instalační cesta definonována pomocí proměnné `INSTALL_DIR`.

2.4 Konfigurace

Po instalaci je třeba systém nakonfigurovat, aby prováděl uživatelem zamýšlenou činnost. K tomu je potřeb před samotným spuštěním programu ”tester” upravit jeho konfigurační soubory - sdělit mu, kde se nalézají dostupné testovací mo-

duly (test.mod), moduly pro odesílání hlášení (report.mod) a jaké služby má testovat (services.test). Forma obsahu obou souborů pro moduly je shodná

```
module <jmeno> (
  address <adresa_systemu>;
  port <cislo_portu>;
  passwd <heslo>;
)
```

Jedna taková sekvence definuje jeden modul. Jméno modulu je uživatelským pojmenováním tohoto modulu. Maximální délka je 39 znaků a nesmí obsahovat znak levé otevírací závorky. Jméno modulu slouží k provázání tohoto konfiguračního souboru se souborem pro definování testů. Proto je také třeba, aby bylo v rámci celého systému unikátní. Adresa systému je DNS jméno nebo IP adresa počítače, na kterém je daný modul spuštěn (jako deamon). Číslo portu je celé číslo z rozsahu $1 - 65535$ včetně[2]. Heslo, které je sdíleným tajemstvím mezi jádrem systému a daným modulem, bude sloužit k prokázání identity jádra. Heslo nesmí být delší než 39 znaků, nesmí začínat bílými znaky a nesmí obsahovat znak středníku.

Tvar souboru pro definici kontrolovaných služeb, tedy `services.test` je pak následující:

```
service <jmeno> (
  address <adresa_systemu>;
  test <jmeno_test_modulu>;
  time <interval_mezi_testy>;
  depends <jmeno_service>;
  report <jmeno_report modulu pocet_chyb adresa>;
  add nazev=hodnota;
)
```

Položka jméno může být maximálně 39 znaků dlouhá a nesmí obsahovat znak středník ani znak reprezentující levou jednoduchou závorku. Adresa systému, stejně jako u modulů, reprezentuje jméno počítače (identifikátor) v počítačové síti. Jména testovacího a reportovacího modulu musí být platná jména z příslušných

[2] Tak jako DNS jméno určuje počítač v rámci celé sítě, tak port určuje aplikaci. Proto dvojice [DNS, port] plně postačuje k identifikaci modulu v TCP/IP sítích.

konfiguračních souborů. Pokud nejsou, program ohlásí chybu a ukončí se chybou. Testovací interval se vztahuje k testovacímu modulu pro daný síťový zdroj a vyjadřuje kolik sekund má uplynout mezi testy tohoto prostředku (tedy obsahuje kladné celé číslo). Položka *depends* není povinná, ale je-li uvedena, musí se odkazovat na platné jméno jiného testovaného prostředku. Pokud se neodkazuje na jiný service nebo pokud takto definovaný závislostní orientovaný graf obsahuje orientovaný cyklus, je ohlášena chyba a opět program končí. Klíčové slovo *report* je následováno trojicí modul (bylo popsáno), počet chyb a adresa. Počet chyb je kladné celé číslo, které udává po kolika chybných testech bude na zadanou adresu (adresy) odeslána informace o chybě na daném prostředku. Pokud je k dané dvojici (modul, počet chyb) uvedeno v konfiguračním souboru více adres, pak je chyba ohlášena na všechny tyto adresy. Adresa pak může vyjadřovat e-mail, telefonní číslo - tedy cokoli, co umí daný modul interpretovat. Klíčové slovo *add* slouží k rozšíření výše popsané základní konfigurace testů. Předpokládané využití této položky je pro definování hodnot dalších, pro testování nutných, konkrétních dat pro testovací modul. Tato data jádro systému nebude nijak interpretovat, jen je odešle modulu, který jim má jako jediný rozumnět. Z faktu, že se tato položka vztahuje k danému testovacímu modulu také vyplývá, že pro zjištění konkrétních možností je třeba nahlédnout do dokumentace k použitému testovacímu modulu.

2.5 Spuštění

Program se spouští zadáním příkazu `tester` v adresáři s nainstalovaným programem. Samotný program tester bez modulů neumí provádět žádné testy. Je tedy třeba spustit i moduly, které jsou zaneseny v konfiguračních souborech pro program tester. Uvedu zde malý příklad nakonfigurování celého systému pro jednoduchý test v malé síti. Řekněme, že se v mé síti nachází počítač pro komunikaci s vnějším světem (`gate.heaven.org`) a počítač, který zobrazuje internetové stránky (`www.heaven.org`). Z dohledového stroje se chci přesvědčit, zda je funkční počítač `gate` a pokud ano pak navíc zkontroluji i počítač `www`. K tomuto účelu spustím na stejném počítači jako tester i modul pro ping a testy budu provádět každou minutu. Pokud by nastala chyba jednou, zapíši to pomocí modulu do logu na stejném počítači a pokud zaznamenám 5 výpadků pošlu e-mail správci. Abych se udržel ve střehu zkusím jednou za hodinu pustit na počítači `gate` skript `scan.sh`, který otestuje známky útoku z `www.hell.org`, který se předá jako parametr - dejme tomu.

Soubor `services.test` v adresáři s programem tester obsahuje:

```
service gate (
  address gate.heaven.org;
  test ping;
  time 1000;
  report logger 1 LOG_ALERT;
  report mailman 5 god@heaven.org;
)

service www (
  address www.heaven.org;
  test ping;
  depends gate;
  time 1000;
  report logger 1 LOG_ALERT;
  report mailman 5 god@heaven.org;
)

service hell (
  address www.hell.org;
  test gate_run;
  time 3600000;
  add run=scan.sh;
  add par=www.hell.org;
  report logger 1 LOG_EMERG;
  report mailman 1 everyone@heaven.org;
)
```

Soubor `report.mod` v adresáři s programem tester obsahuje:

```
module logger (
  address localhost;
  port 2020;
  passwd InNominePatris;
)

module mailman (
  address localhost;
  port 2021;
```

```
  passwd EtFilii;
)
```

Soubor `test.mod` v adresáři s programem tester obsahuje:

```
module ping (
  address localhost;
  port 2022;
  passwd EtSpirirtusSanctii;
)

module gate_run (
  address gate.heaven.org;
  port 2025;
  passwd biBle;
)
```

Nyní je třeba jen spustit příslušné programy. Například modul `gate_run` se spustí příkazem `runner -p 2025 -w biBle` v příslušném adresáři na počítači `gate.heaven.org`. Ostatní výše popsané moduly se spustí obdobně.

Kapitola 3

Komunikační protokol

3.1 Protokol

Jak uvádí [2] - **protokol** je v informatice konvence nebo standard, podle kterého probíhá elektronická komunikace a přenos dat mezi dvěma koncovými body (realizované nejčastěji počítači). V nejjednodušší podobě protokol definuje pravidla řídící syntaxi, sémantiku a synchronizaci vzájemné komunikace. Protokoly mohou být realizovány hardwarově, softwarově a nebo kombinací obou.

Při návrhu protokolu (nejen komunikačního) je třeba brát v potaz mnoho hledisek. Při tvorbě mého komunikačního protokolu jsem se zaměřil zejména na tyto skutečnosti:

prostředí ve kterém se komunikující strany nacházejí je počítačová síť - tedy bude třeba prokazovat legalitu žádání o test. Koordinátor musí prokázat svou identitu. K tomuto existuje mnoho známých řešení, kterým se budu věnovat později.

autentizace je proces (mechanismus) zjištění/ověření identity subjektu. V mém protokolu je nutné, aby koordinátor prokázal svou identitu vůči modulu, který žádá o test. Autentizace většinou ověřuje znalost protistrany nebo vlastnictví tokenu (PIN, klíč, . . .).

citlivost informací při testování a v následných odpovědích nebudou (neměla by být) žádná citlivá data. Tedy protokol data nijak nešifruje (putují ve formátu plain textu). Pokud bychom data šifrovali bylo by třeba brát v potaz i dobu, po kterou probíhá jeden test a kdy a jak měnit šifrovací klíče.

test činnosti je třeba, aby protokol umožňoval koordinátorské aplikaci dotázat se modulu, zda stále vykonává test a v případě, že tento neodpovídá ukončil čekání na odpověď, neboť byla činnost modulu nejspíše ukončena a žádná data od něho nedorazí.

3.2 Architektura

Softwarová architektura je, zhruba řečeno, pohled na systém, který zahrnuje hlavní komponenty systému, chování těchto komponent, jak se jeví zbytku systému a způsoby, kterými komponenty interagují a koordinují své chování k dosažení cílů systému. Architektonický pohled je abstraktní pohled přinášející s sebou porozumění vyšší úrovni a potlačení a odsunutí detailů vlastních většině abstrakcí [4]. Po volbě systému s koordinátorem, kde koordinátor žádá moduly o vykonání testu, byla volba architektury jednoznačná, tedy klient-server architektura. Z [3] uvádím definici klient-server architektury (ekvivalentní definici lze nalézt také v [5]): Klient-server je síťová architektura oddělující klienta (často aplikaci využívající grafické uživatelské rozhraní) od serveru. Každá instance klientského softwaru může posílat požadavky na serverovou aplikaci. Existuje mnoho druhů serverů, např. souborový server, terminálový server, mail server apod.

3.3 Autentizace

Vhodná autentizace pro počítače je například systém jednorázových hesel. V podstatě se jedná o systém podobný Challenge-response systému. Já jsem se rozhodl tento systém neupravovat a implementovat jej ve variantě challenge-response systému podobně jako v protokolu CHAP, který se používá pro Point-to-point protokol (PPP).

V challenge-response systému po návázání spojení autentizátor (modul - server) vyšle žádost o autentizaci (tzv. challenge). Challenge je standardně, jako i v mém případě, náhodně vygenerovaný řetězec (popř. číslo, . . .). Autentizovaný (koordinátor - klient) má pak vrátit příslušnou odpověď, kterou se prokáže.

CHAP pro protokol PPP je popsán v dokumentu [8] a já jej zde pro úplnost uvedu. V návrhu RFC má tento protokol čtyři fáze:

1. Po ustanovení komunikace mezi koncovými uzly zašle autentizátor autentizovanému náhodnou výzvu.

2. Protistrana (autentizovaný) utvoří řetězec z identifikátoru (který je součástí příchozího packetu), sdíleného tajemství (hesla) a příchozí výzvy. Takto vzniklý řetězec je transformován jednosměrnou hašovací funkcí. Výsledek odešle autentizátorovi.

3. Autentizátor ověří odpověď vůči svému (očekávanému) výsledku. Pokud se shodují, je autentizace přijata, jinak by spojení mělo být ukončeno.

4. V náhodných intervalech autentizátor zasílá nové výzvy protistraně a opakuje kroky 1. - 3.

Ve mnou zvolené variantě vypočítá autentizovaná pomocí jednosměrné hašovací funkce (konkrétně MD5) z challenge (v mnou implementované variantě 20 znaků) a hesla digest, který vrátí protistraně. Jelikož ta také zná sdílené heslo, může provést totožnou kalkulaci a po doručení odpovědi jen porovná výsledky, Pokud se oba řetězce shodují, pak autentizovaný prokázal znalost hesla a tedy prokázal svou totožnost. Má transformace se heslo připojuje za challenge což, jak jsem si později uvědomil, není zcela dobré z hlediska bezpečnosti. Případný útočník by mohl využít blokové povahy hašaovací funkce MD5 a útok by byl efektivnější tak, že by útočník předpočítal hašh alespoň pro začátek challenge. Naštěstí má MD5 bloky velikosti $512b$, což přesahuje délku mé challenge ($160b$), takže toto nebezpečí nehrozí. Lepší řešení je však na straně výše zmíněného PPP.

Pokud by to můj protokol vyžadoval, mohl bych se inspirovat u PPP, kde se na obranu proti útoku na challenge-handshake systém po uplynutí náhodně zvoleného času autentizační proces může zopakovat. Jelikož jsem nepředpokládal, že by systém vykonával časově náročné testy, tuto možnost jsem do protokolu nezahrnul.

3.4 Navržený komunikační protokol

Jako každý řízený protokol definuje i můj protokol takzvané řídící signály. Jejich popis je následující (v pořadí užitém během komunikačního sezení):

HELO začátek komunikace (žádost serveru o komunikaci)

CHAL odeslání výzvy

RESP vrácení odpovědi na předchozí výzvu

ACCE potvrzení výzvy (úspěšné prokázání koordinátorovy identity), přijetí dat

DATA informace (počet) o dodatečných datech

ADIT odeslání dodatečných dat pro test (adresa, ...)

RESU výsledek testu (následuje ukončení komunikace)

NOOP prázdný signál - pro zjištění zda modul stále vykonává test nebo zda neodpovídá. Pokud koordinátor neobdrží odpověď na tento signál do 1 sekundy, ukončí komunikaci a test je ukončen chybou.

Příkazy protokolem putují ve tvaru:

```
COMM\t<ciselna_hodnota><textova_hodnota>\r\n
```

kde `COMM` označuje jeden z výše uvedených příkazů, *t* tabelátor, *r* návrat vozíku a *n* nový řádek. Pokud se v příkazu žádná z hodnot nevyskytuje, pak se znak tabelátoru vynechává.

3.5 Průběh protokolu

Komunikační protokol bych pro přehlednost rozdělil do následujících částí (protokol popíši z pohledu koordinátora):

iniciace koordinátor zahájí spojení s modulem. Bude jej tedy žádat o test.

autentizace modul poskytne své služby jen tomu, kdo zná sdílené tajemství (heslo). Za tímto účelem provede challenge-handshake ověření identity, kterou koordinátor prokáže. Po úspěšné autentizaci je možno pokračovat.

odeslání dat potřebných pro vykonání požadovaného testu. Tato data odesílá koordinátor modulu ve dvou fázích. Nejprve dá navědomí kolik rozšiřujících dat bude odesílat a po schválení tato odešle.

vrácení výsledku po úspěšném odeslání dat vyčká koordinátor na výsledek testu o který požádal. Jakmile jej obdrží, komunikace je ukončena.

Tento průběh komunikačního protokolu je ideálním průběhem. Protokol může být zakončen i mimo takto naznačený průběh. Diagram znázorňující kompletní průběh komunikačního protokolu je na obrázku 7.4.

Kapitola 4

Program tester

4.1 Datové struktury

Po spuštění je třeba načíst uživatelská data pro veškerou činnost do paměti (datových struktur) programu. Tomuto procesu se věnuji v sekci 4.2. Načtená data se pak ještě upraví do podoby v jaké jsou používána po běh programu. To proto, že ne všechna data získaná od uživatele jsou důležitá po celou dobu běhu aplikace. Mezi tato data bych zařadil například jména testovacích či odesílacích modulů, která jsou důležitá v první fázi na provázání konfiguračních souborů, ale pro další práci je nahrazuji úspornějšími ID čísly, která se také jednodušeji a rychleji používají. Existenci dvou různých datových struktur bych přiřadil i opačnému faktu, tedy existují data, například chybová hlášení pro service, která se od uživatele nenačítají, ale naopak existují jen za běhu a to jen tehdy, je-li to nutné.

V případě prioritní fronty jsem se rozhodl pro datovou strukturu halda, která garantuje mnou požadované operace výběru minima a přidávání prvku v čase $O(log(n))$. Použil jsem haldu reprezentovanou v poli tak, jak ji popisuje [7]. Pro tuto strukturu jsem se rozhodl právě kvůli časové náročnosti operací, která je lepší, než prosté uložení položek do pole, kdy by měly obě operace lineární čas (tedy $O(n)$) nebo použití spojového seznamu, kde by výběr minima byl sice v konstantním čase, ale lineární přidání prvku by zůstalo.

4.2 Po spuštění

U dat pocházejících od uživatele je třeba ověřit jejich konzistenci a (alespoň částečnou) správnost. Tento postup se provede v několika krocích:

1. Nejprve načte konfigurační soubory definující v systému dostupné moduly a ty si zaznamená. Testovací moduly ze souboru test.mod a odesílací moduly z report.mod.

2. Načítají se data pro testování ze souboru services.test, u kterých je v době načítání prováděn test, zda moduly na které se odkazuje existují (je pro ně záznam v příslušném konfiguračním soboru, resp. je takový modul uložen).

3. Načtená data se upraví do podoby užité v programu.

4. Graf definovaný nad strukturou services je otestovaný na acykličnost. K tomuto se používá algoritmus topologického třídění (popsaný v [7]), který jsem upravil tak, že při testování navíc využívá fakt, že každý vrchol má nejvýše jednoho předchůdce.

5. Je založena prioritní fronta a program pokračuje na produkční část.

4.3 Produkční část

Zbytek práce programu tester je vykonáván v nekonečné smyčce:

1. Prvek (service) aktuálně určený k otestování se získá z prioritní fronty (v mé implementaci se prvek z vrcholku haldy odebere a halda se přestaví). Ná-sledně je pak proces uspán až do doby, kdy má být tento test proveden.

2. V čase testu je zjištěn stav service, na které testovaný závisí. Pokud není ve stavu OK, je aktuální test ukončen a je indikována chyba na otci. Pokud toto nastane, přeskakuje další krok a pokračuje na 4. Tento mechanismus je v systému kvůli možným výpadkům v síti, kdy při celkové nedostupnosti počítače není třeba uživatele informovat o tom, že na nedostupném počítači neběží žádná služba. Pokud je otec aktuální service, pokračuje se v testu.

3. Tester kontaktuje příslušný modul a prokáže se mu heslem z konfiguračního souboru. V případě úspěšného ověření identity odešle zbylá data určená pro testovací modul a vyčká na konec testu. V případě, že během 1 sekundy

nedostane odpověď, odešle navíc signál NOOP, aby se ujistil, že testovací modul stále odpovídá. Pokud obdrží výsledek nebo nezíská ve stanoveném časovém limitu odpověď na signál NOOP, pokračuje na další krok.

4. Zpracuje výsledek testu a popřípadě zapíše chybu, změní stav. Dále může kontaktovat modul(y) určené k odeslání, pokud bylo dosaženo uživate-lem definovaného počtu chyb.

5. Vypočte se čas následujícího testu a aktuální prvek je umístěn zpět do prioritní fronty. Znovu se pokračuje dalším prvkem na bodu 1.

Kapitola 5

Moduly systému

5.1 Moduly obecně

Jelikož je programátorská práce na koordinátorské části aplikace ukončena, závisí veškeré budoucí možnosti vzniklé aplikace na modulech. Z tohoto důvodu považuji seznámení se s těmito entitami aplikace za významné. Právě nyní přichází také okamžik, kdy je nutné začít oddělovat, zda pojednávám o testovacích modulech či o modulech pro odesílání chybových hlášení. Důvodem jsou rozdíly v činnosti těchto modulů. I přes tento fakt však zůstávají jisté části podobné či dokonce stejné. Rád bych zde ještě zdůraznil, že oba druhy modulů musí implementovat stejný komunikační protokol.

Další významnou společnou vlastností je, jak jsem již zmiňoval při návrhu, že oba druhy modulů jsou servery, tedy aplikace naslouchající na síťovém rozhraní (socketu). Aby výsledné aplikace byly co nejpoužitelnější, je třeba aby podporovaly co možná nejvíce síťových protokolů, především IP agnostic aplikace - tedy taková, která je použitelná jak nad IPv4 tak nad IPv6. Moduly prozatím v systému dostupné se pokusí otevřít socket pro každý v systému dostupný protokol. Nyní je třeba si uvědomit, že tímto může vzniknout velké množství otevřených socketů, které je třeba obhospodařovat. Zabýval jsem se dvěmi, dle mého názoru nejpoužívanějšími, možnostmi obsluhy. A to obsluhou pomocí vláken nebo pomocí funkce select()[1].

[1] Alternativou by bylo použití funkce poll(). Poskytují však prakticky stejné služby

V případě řešení s použitím vláken jsem uvažoval vždy jedno vlákno na obsluhu jednoho socketu. Toto řešení by ale mohlo hodně plýtvat systémovými prostředky. Navíc libovolné řešení za pomoci vláken by vyžadovalo, aby funkce činnosti modulu (testování či odeslání informace) byly realizovány jako reentrandní, což je termín označující funkci, kterou smí volat více vláken najednou[1] (tedy např. nemá žádné statické struktury apod.). Další možností by mohlo být nechat se inspirovat daemonem Apache a jeho tzv. "worker modem", kde se pro obsluhu více událostí na jednom socketu nejprve vytvoří více synovských pocesů, které pak při příchozí komunikaci předávají obsluhu na vlákna. To jsem si představoval tak, že množinu socketů bych rozdělil na (menší) množiny a jejich obsluhu přenechal vždy jednomu procesu (který by případně mohl zakládat vlákna).

Pokud bych použil řešení využívající služeb funkce select(), mohu se rozhodnout, zda obsloužím více požadavků najednou (asynchronně) nebo zda je budu obsluhovat sekvenčně. Vzhledem k faktu, že typické nasazení mnou implementovaného řešení využívá jednoho (sekvenčního) koordinátora a nevyužilo by tedy souběžného testování (navíc by bylo opět nutné, aby příslušné funkce byly reentrandní), rozhodl jsem se pro sekvenční testování a tedy také využil funkci select(). Důležitým faktem ale zůstává, že je možné v případě zájmu (nebo změny koordinátorské části aplikace) toto řešení relativně jednoduše změnit tak, že se využije služby vláken a pro každé (resp. konstantně mnoho) příchozí spojení bude založeno nové vlákno.

Podobným problémem jako je výše popsaný, je reakce modulu na přijetí signálu NOOP (což je z pohledu modulu asynchronní událost). Na takovouto událost je opět možné reagovat více způsoby. Při aktivním čekání by byla práce modulu vždy po uplynutí jistého časového kvanta přerušena a na prostředku (v mém případě socket) je otestováno, zda k očekávané události nedošlo. Tento způsob čekání používá například operační systém pro kontrolu paralelního portu. Tento způsob řešení by vyžadoval (opakované) přerušování testu a následné pokračování. To, jak se domnívám, nemusí být vždy možné. Proto jsem uvážil jiné řešení, které využívá vlákno, které je možno zablokovat, dokud na socketu nenastane událost - příchozí komunikace.

Druhé zamýšlené řešení je výhodnější a navíc jej lze snadno implementovat, neboť v době, kdy modul již pracuje (testuje) nemůže, podle definovaného protokolu (viz sekci 3.4), příchozí komunikace obsahovat jiný signál než je právě

NOOP. Shodné řešení bych doporučil i budoucím programátorům modulů, proto jsem funkci, která tuto reakci obhospodařuje v mých modulech umístil do sekce 5.4, kde podám více informací.

5.2 Moduly pro odeslání informace uživateli

Moduly, které podávají informaci uživateli jsou, alespoň podle mého názoru, jednodušší. K tomuto názoru mě přiměl fakt, že není třeba, aby tyto moduly implementovaly příkaz NOOP (z definice komunikačního protokolu, viz sekci 3.4). Tyto mohou po přijetí dat od klienta mohou komunikaci ukončit, neboť se nečeká na výsledek odesílání. Podání hlášení uživateli (jako je odeslání e-mailu, zápis do systémového logu, . . .) lze snadno realizovat, ať už naprogramováním, nebo využitím na systému FreeBSD dostupných programů. Během fáze implementace se také ukázalo, že je třeba navrhnout pevnou strukturu odesílání chybových informací od jádra (příkazem ADIT komunikačního protokolu). Této pevně dané struktury lze při tvorbě těchto modulů využít. Toto zajisté pro testovací moduly neplatí, neboť u takových definuje možné příkazy programátor modulu a systému je zadává uživatel v konfiguračním souboru. Nelze tedy předpokládat, zda budou přítomné (některé příkazy mohou být nepovinné, uživatel se mohl přepsat, zapomenout, . . .) ani v jakém pořadí budou přijaty.

Protože modul, jak jsem již obecně zmínil, je server, je nutné aby naslouchal na síťovém portu, přes který bude s klientem komunikovat. Navíc bude ověřovat identitu jádra žádajícího modul o test a tedy musí znát heslo. Každý modul pro odeslání dat, který jsem doposud vytvořil, pracuje v několika fázích. Načte uživatelská data (heslo a port), otevře port a poté v nekonečné smyčce opakuje vyčkání na příchozí komunikaci, přijme data (zde pevně stanovená), ukončí komunikaci a na závěr provede s přijatými daty činnost, ke které byl stvořen (např. zápis do logu). Obecné schéma odesílacích modulů zachycuje obrázek 7.2.

Práci budoucího programátora odesílacího modulu se mi podařilo (v případě, že bude spokojený s mým návrhem aplikace) omezit na naprogramování jedné funkce. Pro tyto účely jsem navrhl skelet (základ) pro odesílací modul. Funkce, která vykoná hlavní činnost nového modulu, je umístěna v souboru main.c tohoto skeletu a je pojmenovaná error_handler(). Skelet načítá konfigurační data (port a heslo) z příkazové řádky, otevírá jeden port pro všechny, v systému dostupné síťové protokoly a příchozí požadavky zpracovává sekvenčně.

5.3 Moduly pro testování

Moduly pro testování jsou, vzhledem k charakteru jejich činnosti, složitější entity celého systému. Jejich činnost je jistě svým způsobem podobná modulům pro odesílání. Opět lze činnost těchto modulů rozdělit do několika fází. Modul načte data nutná pro správný běh, otevře port pro komunikaci, v nekonečné smyčce vyčká na příchozí spojení, ověří identitu protistrany, přijme data nutná pro vykonání testu, otestuje a vyhodnotí výsledek testu, vrátí výsledek testu protistraně a přejde zpět na začátek nekonečné smyčky. Obecné schéma testovacích modulů zachycuje obrázek 7.3.

Testovací moduly se mohou vzájemně velmi lišit například v načtených datech, v přijatých datech od koordinátora systému (která mohou upravovat vykonávané testy), ve vyhodnocování výsledku testu, ... Vytvoření skeletu testovacího modulu mi, vzhledem k tomuto faktu, přišlo nemožné. Abych však co nejvíce usnadnil tvorbu dalších modulů do tohoto systému, vytvořil jsem knihovny, jejichž účelem je usnadnění programátorské práce. Těmto funkcím se podrobně věnuji v sekci 5.4.

5.4 Funkce pro usnadnění tvorby modulů

V této sekci pojednávám o funkcích, které jsem vytvořil a použil při programování modulů. Účelem těchto funkcí je také odstínit budoucího programátora od nutnosti studovat podrobně některé fáze komunikačního protokolu (například zahájení komunikace, přesný tvar odesílaných/přijímaných dat, ...).

int start_module_comm(fd, passwd) (net_comm.h) Funkce zahájí komunikaci modulu s jádrem na socketu fd, vygeneruje a odešle náhodnou výzvu, vyčká na přijetí odpovědi a tu porovná s vlastním výpočtem. Pokud se MD5 hashe shodují vrací 0, jinak ukončí komunikaci a vrací -1. Volající by po obdržení -1 měl socket uzavřít.

int send_comm(s, comm, value, str) (net_comm.h) Funkce slouží k odeslání jednoho příkazu komunikačního protokolu (dle tvaru definovaném v 3.4) socketem s a využívá ji i koordinátor. Jako comm se uvede příslušný příkaz protokolu (doporučuji pro tyto účely používat konstanty definované taktéž v hlavičkovém souboru net_comm.h). Parametr value je číselná hodnota odesílaná příkazem a str je hodnota textová. Pokud nemá některá z těchto

být odeslána, předá se funkci ukazatel NULL. Funkce vrací 0 v případě, že se odeslání podařilo, -1 pokud nastala chyba.

int read_comm(s, comm, value, str, len) (net_comm.h) Funkce je do jisté míry podobná předchozí funkci, ale slouží k přijetí dat od protistrany ze socketu s. Případné hodnoty ukládá do proměnných value a str. Parametr funkce len předává maximální možnou délku řetězce, který se bude ukládat do str. Pokud je tento nastaven na 0 a ukazatel str není NULL, pak bude potřebné místo alokováno dynamicky pomocí funkce malloc() a mělo by být po použití dealokováno. Funkce vrací 0, pokud je vše v pořádku a -1 nastane-li chyba (socketu nebo při alokaci paměti).

void * noop_handler(in) (thr_noop.h) Tato funkce realizuje běh vlákna, které odpovídá na signál NOOP. Jejím parametrem je ukazatel na socket na kterém probíhá komunikace s koordinátorem. Funkce využívá mutexovou proměnnou[1] (proměnnou pro vzájemné vyloučení přístupu více vláken ke sdílenému prostředku) ze stejného hlavičkového souboru. Tuto proměnnou je třeba inicializovat z volajícího kódu pomocí pthread_mutex_init(&socket_mx, NULL). Toto vlákno je spouštěno v takzvaném detached modu[1], tedy nevrací žádnou hodnotu a není-li již nadále potřeba okamžitě se samo ukončí (nečeká se na hlavní vlákno a volání funkce pthread_join()).

Výše uvedené funkce využívají pro komunikaci po socketu funkcí my_send() a my_recv(), které jsou variacemi na funkce popsané v článku[9]. V tomto článku autor popisuje správné čtení/zapisování při použití elementárních I/O funkcí read() a write(), které se velmi podobají funkcím send() a recv() pro sockety.

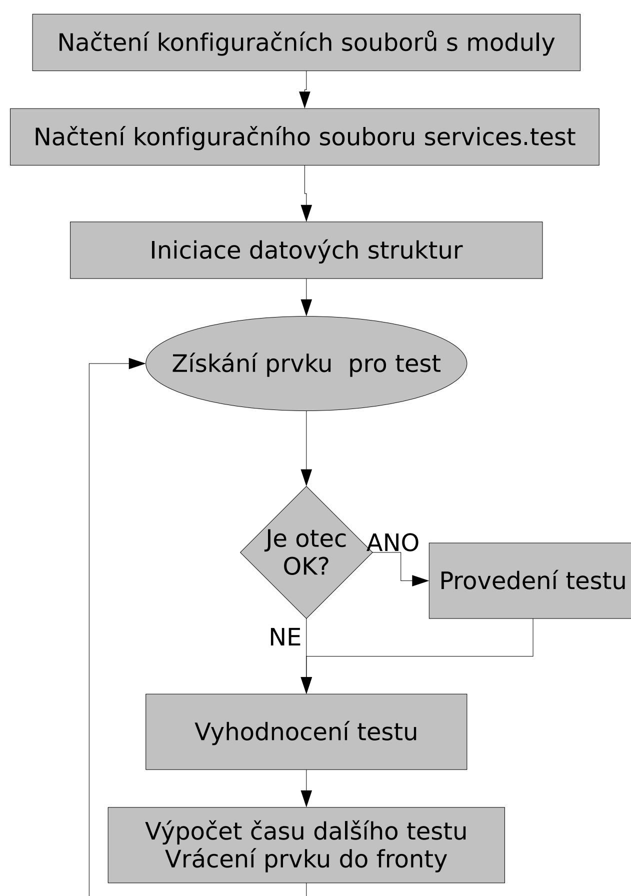
Kapitola 6

Závěr

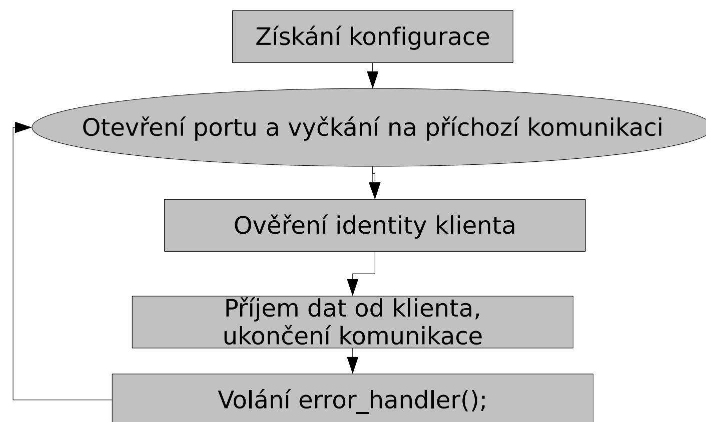
Mnou navržené a naprogramované řešení se prokázalo jako použitelné v domácích sítích, ve kterých jsme já a někteří mí spolužáci provedli testování. V takových, popřípadě v menších podnikových sítích bych také viděl jeho případnou budoucnost, přestože si nemyslím, že by správci zvyklí na stávající software tento měnili. V případě, že jsou tito lidé zvyklí například na mnou zmiňovaný Nagios, nebudou jej nejspíše měnit. Pokud by přesto požadovali (jiný) program, jehož kódu je jednoduché porozumnět (nebo jeho části jednodušeji verifikovat), mohl by se tento nástroj hodit.

Pokud by však vznikala nová sít' menšího či středního rozsahu, pak by mnou stvořený nástroj mohl být využit. Nebo by měl být alespoň vzat v potaz. Zejména pokud by byla tato sít' velmi využita (s velkým sít'ovým provozem), pak by bylo vhodnější užít výše zmíněný program, který sít'ový provoz zatěžuje minimálně. Mnou vytvořený software by také měl nalézt uplatnění tam, kde je třeba univerzáního, nenáročného, ale také snadno přizpůsobitelného nástroje pro dohled nad počítačovou infrastrukturou.

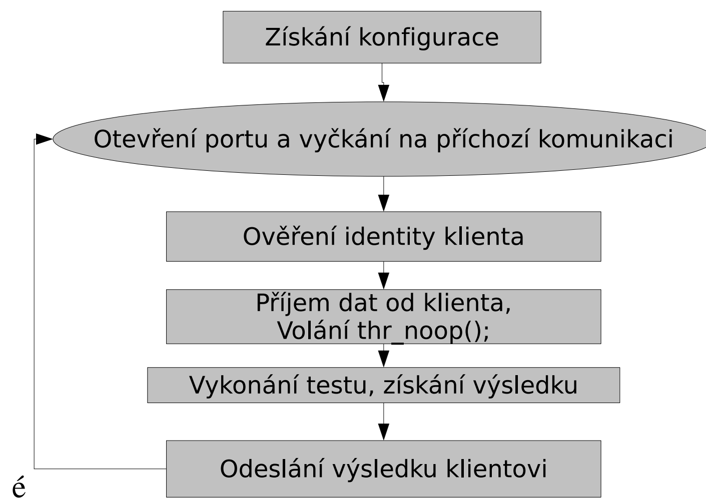
Dodatky



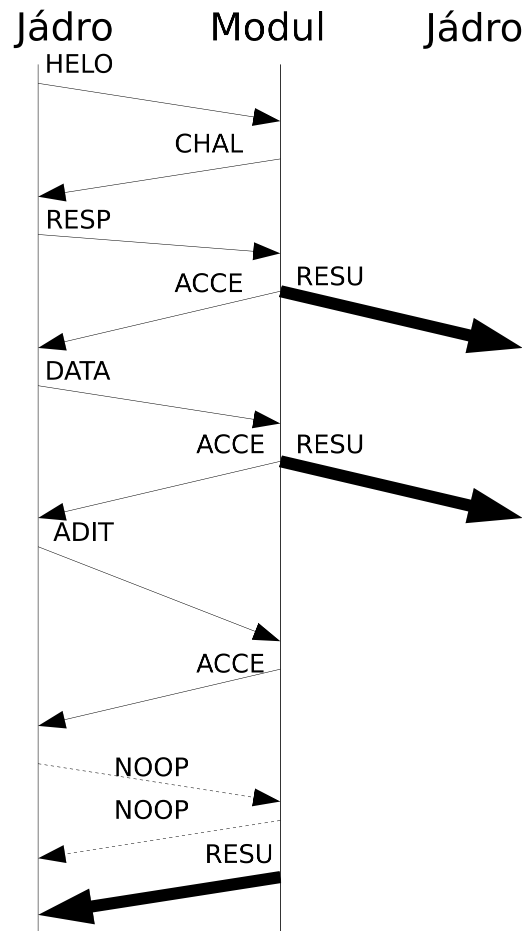
Obrázek 7.1: Životní cyklus programu tester



Obrázek 7.2: Schéma odesílacích modulů



Obrázek 7.3: Schéma testovacích modulů



Obrázek 7.4: Znázornění průběhu protokolu

Literatura

[1] Martin Beran, Jan Pechanec: *Programování v UNIXu*, http://www.devnull.cz/mff/pvu/slides/programovani_v_unixu.pdf

[2] Wikipedia: *Komunikační protokol*, http://cs.wikipedia.org/wiki/Komunikační_protokol

[3] Wikipedia: *Architektura klient-server*, http://cs.wikipedia.org/wiki/Klient-server

[4] Clements, P.: *Comming Attractions in software architecture*, Technical Report, CMU/SEI-96-TR-008

[5] Douglas Comer: *Computer Networks and Internets*, ISBN 978-0136061274, Prentice Hall

[6] Andrew S. Tanenbaum, Maarten van Steen: *Distributed Systems: Principles and Paradigms*, ISBN 978-0130888938, Prentice Hall

[7] Pavel Töpfer: *Algoritmy a programovací techniky*, ISBN 978-80-7196-350-9

[8] RFC-1994: *PPP Challenge Handshake Authentication Protocol (CHAP)*, http://www.ietf.org/rfc/rfc1994.txt

[9] Rudolf A. M., Fenner B., Stevens W. R.: *readn, writen, and readline Functions*, http://www.informit.com/articles/article.aspx?p=169505&seqNum=9. (2004)