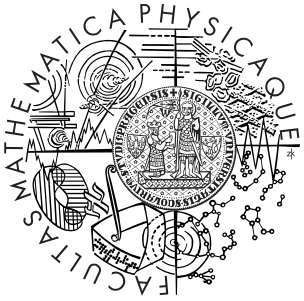
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Dušan Knop

Systém kontroly běhu vzdálených strojů a služeb

Středisko informatické sítě a laboratoří

Vedoucí bakalářské práce: Dan Lukeš
Informatika: Správa počítačových systémů

2009

Na tomto místě bych rád poděkoval vedoucímu práce panu Lukešovi za metodické rady převážně při tvorbě softwaru. Dále pak svým rodičům za podporu při studiu a jazykové korektury této práce.

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

V Praze dne 07.08.2009 Dušan Knop

Obsah

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

Abstrakt: V předložené práci studujeme možnosti vytvoření systému pro monitoring počítačů v sítích a služeb na nich běžících, který musí umožňovat přidávání testů. Pokud tedy nějaký test zatím neexistuje (nebyl implementován) - mělo by být možné jej do systému přidat. Výsledkem pak je i jedna možná konkrétní implementace takového systému (modulární s koordinátorem) v jazyce C pro operační systém FreeBSD. Konkrétní testovací metody jsou implementovány jako samostatně běžící moduly, komunikující s jádrem síťovým protokolem. V případě zjištěné závady jádro odešle informaci daným způsobem stanoveným osobám. Konkrétní metody zasílání informací jsou také implementovány formou modulů.

Klíčová slova: počítačové sítě, testování, kontrola

Title: System for supervision of remote systems and services run
Author: Dušan Knop
Department: Network and Labs Management Center
Supervisor: Dan Lukeš
Supervisor's e-mail address: dan@obluda.cz

Abstract: In the present work we study possibilities of network monitoring system creation. We will audit computers in networks and services running on them. System must allow adding new kinds of tests into it. So if some test is not yet implemented it should be possible to add it into the system. As a result there is a implementation of such a system (modular with coordinator) written in C languague for FreeBSD operating system. Concrete testing methods are implemented as standalone modules which comunicate with coordinator by network protocol. In the case of error location the coordinator will send information by given way to set persons. Concrete methods for sending information are also implemented as modules.

Keywords: computer networks, testing, supervision

Kapitola 1

Účel a návrh

1.1 Zadání projektu

Navrhněte a vytvořte (implementujte) systém, který bude zajišťovat periodicky se opakující testy v počítačové síti pro operační systém FreeBSD. Programovacím jazykem budiž C s možnými objektovými rozšířeními. Za lepší však bude považováno vytvoření čistě procedurální aplikace. Upřesněním zadání jsme poté zjistili, že protože není specifikováno jaké testy se mají provádět konkrétně, musí systém umožňovat doplnění nového druhu testu přímo do naší aplikace. Tím se dosáhne velké univerzálnosti vytvořeného projektu. Čím jednodušší bude doplňování nového druhu testu do aplikace, tím se dosáhne vyšší míry univerzálnosti. Systém by také měl podporovat jisté (opět prozatím neznámé) odesílání informace o nastalé chybě uživateli. Vzhledem k tomu, že postupy pro testování i pro odesílání jsou analogické, nebudu prozatím mezi těmito dvěma částmi programu (testovacího systému) rozlišovat.

1.2 Uvažované varianty druhu aplikace

Na samém počátku jsem se rozhodoval, jaký druh aplikace bude výsledkem práce. Rozhodoval jsem se mezi monolitickou a modulární aplikací. Protože do modulárních aplikací se snáze zasahuje a snáze se též mění jejich části, rozhodl jsem se pro modulární aplikaci. Monolitická aplikace by svou povahou znesnadňovala přidávání nových druhů testů. Poté následovalo další rozhodování, zda bude tato aplikace obsahovat koordinátora či nikoli.

Koordinátor je jeden význačný modul, který udržuje veškeré informace nutné pro běh systému. Pokud systém obsahuje koordinátora plynou z toho jisté výhody i nevýhody. Mezi výhody bych zařadil: snadnější implementaci (alespoň podle mého názoru), jednodušší údržbu (údržba se provádí převážně na koordinátorovi) a jednodušší moduly. Nevýhodou je, že v případě ztráty koordinátora přestává celý systém fungovat.

Pokud bychom uvažovali o systému bez koordinátora, pak jsou všechny aplikace v systému rovnocenné. To znamená, že každá část musí znát nějaké informace o celém systému, je částečně závislá na ostatních a v případě ztráty některých částí by se zbytek měl s touto ztrátou vyrovnat a pokračovat ve své práci. Takovéto aplikace by bylo složitější vytvořit a hlavně poté spravovat. Zejména proto, že systémoví správci, kteří mají být cílovou uživatelskou skupinou, mají v oblibě věci jednoduché na údržbu, jsem volil implementaci s koordinátorem. K této volbě přispěla i skutečnost jednodušší implementace takového systému.

1.3 Druhy modulů a komunikace

Jak jsem již zmínil v předcházejícím textu, v době návrhu a vytváření systému jsem měl pouze obecné informace ohledně budoucích požadavků uživatele na cíl a způsoby testování. Základní otázkou pak bylo, zda aplikační moduly budou spouštěny na stejném stroji jako koordinátor či nikoli. Pokud by běžely na stejném stroji, pak by byla komunikace mezi koordinátorem a příslušným modulem jednodušší. Lokální komunikace mezi procesy je z pohledu programátora jistě jednodušší. Při běhu na stejném stroji by bylo možné vybírat z mnoha způsobů meziprocesové komunikace, zatímco při spouštění modulů i na jiných strojích, bychom použili komunikaci pomocí socketů[1].

V průběhu specifikace jsem byl upozorněn, že protože testovat lze mnoho věcí, neměla by libovolná má volba ztěžovat nebo neumožňovat nějaký test. Tomuto se přirozeně nelze vyhnout zcela, ale je dobré mít tuto skutečnost

[1] Mezi další možnosti by nejspíše patřilo i využití mechanismu RPC. Ale protože jedním z prvotních cílů práce bylo osvojit si práci se (síťovými) sockety (popř. jinými druhy meziprocesové komunikace), toto řešení jsme neuvažovali.

stále na paměti a v případě důležitých rozhodnutí k tomuto přihlížet. Vzhledem k tomu, že testovat lze například i zda je na lokálním disku dostatek volného prostoru, je zjevné, že pro takové případy je výhodné povolit moduly spouštěné nejen na stroji s koordinátorem. Přestože i toto (v případě lokálních, z pohledu koordinátora, modulů) by bylo možné testovat, vyžadovalo by to ovšem, aby daný modul byl jen prostředníkem mezi naším systémem a aplikací provádějící testování. Takovéto řešení by pak znesnadňovalo užívaní aplikace budoucímu uživateli (respektive programátorovi nových modulů). Z těchto důvodů bylo řešení pouze pomocí lokálně spouštěných modulů zavrženo. Z hlediska jednoduchosti aplikace a pro usnadnění její správy jsem neumožil extra lokální moduly. Moduly mohou být i lokální, ale i tak se pro komunikaci s nimy použije síťový socket.

Kapitola 2

Systém z pohledu koncového uživatele

2.1 Co systém poskytuje

Program „tester“ a jeho moduly se starají o periodický test dostupnosti prostředků v počítačové síti. Prostředek je určen modulem, který se k testování použije. Projekt „tester“ pak působí jako jádro aplikace (koordinátor celého systému), které spravuje okolní moduly a uživatelská data, žádá o testy a ukládá chybová hlášení. Zároveň se „tester“ stará o správu chyb – pokud je zaznamenán jistý (uživatelem definovaný) počet chyb, jsou nashromážděná data odeslána modulu, který má za úkol ohlášení chyby. Způsob nahlášení chyby závisí na konkrétním použitém modulu.

Program je otevřený novým možnostem. Pokud se rozhodnete, že je třeba testovat skutečnost, pro níž zatím neexistuje testovací modul, je možné tento modul relativně jednoduše dopsat. Toto jistě, a snad ještě více, platí i pro moduly odesílající zprávu uživateli. Nespornou výhodou pro koncového uživatele je i to, že program je poskytován zdarma[1].

2.2 Instalace a užívání

Vše se instaluje klasicky ze zdrojových kódů pomocí příkazu make. Instalace všech částí je totožná. Vždy stačí ve složce zadat:

[1] v rámci licence GNU-GPL

$ make
$ make install
$ make clean
Tím se vše potřebné okopíruje do složky /usr/bin/tester (adresář je platný pro instalaci jádra systému, moduly se instalují do svých podadresářů v /usr/bin/). Pokud by si uživatel přál nainstalovat soubory do jiného adresáře v systému, stačí před instalací editovat soubor install_script.sh, kde je třeba změnit proměnnou INSTALL_DIR na adresář instalace.

Nejprve je třeba v adresáři s nainstalovaným projektem editovat tři konfigurační soubory, které systému říkají co se má testovat, čím a pokud se má reagovat na chybu, tak jakým způsobem. Konfigurační soubory jsou tyto: test.mod (moduly pro testování), report.mod (moduly pro odesílání chyb) a services.test (pro služby nebo prostředky, které se mají testovat). Žádný konfigurační soubor nesmí obsahovat národní znaky (obsahuje-li je, může dojít k nepředvídaným chybám). Kurzívou tištěná slova jsou definována uživatelem, zatímco normálně tištěná slova jsou klíčová. Pro oba druhy modulů mají konfigurační soubory stejný tvar:
module *jmeno* (
address *adresa_systemu*;
port *cislo_portu*;
passwd *heslo*;
)

Takováto sekce definuje jeden modul. Omezena je délka jména modulu, a to na 39 znaků. Položka jméno dále nesmí obsahovat znak levé otevírací závorky. Jméno se použije na provázání s testem (odesláním) v konfiguračním modulu testů. Adresa_systemu určuje DNS jméno (IP adresu,...) stroje, na kterém daný modul běží jako daemon (je naistalován a spuštěn). Cislo_portu se musí pohybovat v platném rozsahu čísel portů – tedy 1 - 65535 včetně. Položka heslo může být maximálně 39 znaků dlouhá, nesmí začínat bílým znakem a nesmí obsahovat znak středníku.

Konfigurační soubor definující testy je trochu složitější. Tvar je následující:
service *jmeno* (
address *adresa_systemu*;
test *jmeno_test_modulu*;
time *testovaci_interval*;

depends *jmeno_service*;
report *jmeno_report_modulu pocet_chyb adresa*;
add *nazev=hodnota*;
)

Položka jmeno může být maximálně 39 znaků dlouhá a nesmí obsahovat ani znak středníku, ani znak levé otevírací závorky. Adresa_systemu má podobný význam jako v konfiguraci modulů - doménové jméno stroje, který se má testovat. Jmeno_test_modulu a jmeno_report_modulu musí být platná jména modulů definovaných v konfiguračních souborech příslušných modulů. Testovaci_interval je počet sekund uplynulých mezi testy. Položky depends, report a add nejsou povinné, takže jejich vynechání není chybou. Pokud je položka depends v definici služby uvedena, pak musí odkazovat na platné jméno jiné služby, definované kdekoli v tomto souboru. Pokud služba na kterou depends odkazuje není v pořádku (poslední nebo několik posledních testů neproběhlo správně), pak test závislé služby vůbec neproběhne a jen se odloží do dalšího intervalu. Klíčové slovo report je následováno třemi hodnotami. První z nich již byla popsána, druhá značí počet chyb (tedy přirozené číslo), kdy se má začít komunikovat s modulem a odesílat chyba (chyby) a poslední říká, co se má modulu odeslat jako adresa (toto může znamenat cokoli a je to závislé na modulu – e-mail, telefonní číslo,...). Klíčové slovo add je pak následováno daty specifickými pro testovací modul. Doporučuje se, aby se používala výše uvedená konvence (jméno=hodnota), ale není to nutností. Pro zjištění, co lze napsat do sekce add pro případné testovací moduly je třeba nahlédnout do dokumentace (popřípadě specifikace) příslušného modulu pro testování.

2.3 Současné možnosti systému

V nynější podobě jsou dovednosti systému velmi omezené. Z testovacích modulů je dostupný pouze modul na ping, z odesílacích modulů pak zápis do logu a odesílání e-mailu. Programátorské práci na modulech se věnuji v části 4.

2.4 Obsah přiloženého CD

Na přiloženém CD se z programového vybavení nalézá program „tester“, modul zajišťující ping test, modul zapisující chybový stav do lokálního logu, modul zasílající chybový stav na e-mail(y). Dokumenty k projektu, sestávající z této práce, programotárské a uživatelské dokumentace, jakož i specifikace projektu, se nalézají v adresáři docs. V adresáři report_skelet je umístěn základ pro modul odesílající chybu.

Kapitola 3

Komunikační protokol

3.1 Architektura

Jednoznačná byla volba použité architektury. Je nutné použít klient-server architekturu. Jedná se o síťovou architekturu oddělující klienta a server, kteří spolu komunikují přes počítačovou síť. Server je ta část systému, která poskytuje jistou službu (služby) a klient občas žádá o její zpřístupnění. V našem případě se koordinátor chová jako klient pro moduly. Protikladem k této architektuře je peer-to-peer, kde jsou komunikující strany rovnocenné. Architektura peer-to-peer není v našem případě použitelná, protože ta by se použila v případě návrhu bez koordinátora, kde jsou si všechny části systému rovny.

3.2 Použitý protokol

Způsob, kterým klient žádá server o poskytnutí jeho služby, se označuje jako komunikační protokol. Návrh komunikačního protokolu musí vycházet ze skutečnosti, kde se komunikující strany nacházejí, jaké informace si chtějí vyměnit a jakou dobu přibližně zabere jedno komunikační sezení. Všechny tyto skutečnosti ovlivňují požadavky, které budeme klást na náš komunikační protokol. Protože náš klient bude se servery komunikovat přes počítačovou síť, která je nezabezpečeným médiem, musí umět prokázat svou identitu před serverem. Protože však informace přenášené naším protokolem nejsou nijak kritické, není třeba, abychom celou komunikaci šifrovali. Protože někdy se může testování pozdržet či z povahy testu probíhá delší dobu, je třeba,

aby náš protokol podporoval prázdný signál, který odešle klient serveru v případě dlouhé prodlevy mezi odesláním požadavku a příchodem odpovědi.

3.3 Autentizace

Autentizace je proces ověření proklamované identity komunikujícího subjektu, v našem případě klienta. Toto ověření lze provádět na základě znalosti protistrany(např. zná heslo, PIN), vlastnictví protistrany(např. vlastní privátní klíč), schopností protistrany(např. odpověď na kontrolní výraz),...

Pro naše účely se prokazování otevřeným heslem zcela nehodí, neboť heslo by bylo v počítačové síti snadno odposlechnutelné a poté již není dobré se takovým heslem prokazovat. Toto nebezpečí je třeba eliminovat. Například lze užívat takzvaná jednorázová hesla, kdy je třeba mít seznam hesel, který je třeba bezpečným způsobem doplňovat. To by opět bylo pro naše účely nepřijatelné, neboť některé testy může uživatel vyžadovat velmi často a pak by jen obnova databáze hesel zabírala hodně systémových prostředků. Také lze mít jedno výchozí heslo (i z minulé relace) a přechodovou funkci od jednoho hesla k dalšímu. Zde je ale prostředí nespolehlivé počítačové sítě vyloženě nevhodné pro nasazení takového systému. Uvažme situaci, kdy po úspěšně navázané komunikaci se klient prokáže heslem a následně má stroj, na kterém je klient spuštěn, přerušeno napájení. Klient si po restartu stroje bude myslet, že se komunikace vůbec neodehrála, zatímco server bude přesvědčen, že proběhla a tak přepočítá heslo na další. Pak je třeba ještě řešit synchronizaci a to pro takto jednoduchou aplikaci není potřeba.

Naopak kontrola pomocí odpovědi na kontrolní výraz je celkem běžná a poskytuje jistou míru bezpečnosti. Rozhodl jsem se navrhnout Challenge Handshake systém. Původně se systém CHAP (Challenge Handshake Authentication Protocol) používal pro zařazení autentizace do protokolu PPP. Tento protokol je popsán v [3]. Pro PPP má 4 pracovní fáze:

1. Po ustavení komunikace posílá autentizátor autentizovanému náhodnou výzvu - označuje se jako challenge.

2. Protistrana (autentizovaný) odešle zpět výzvu doplněnou o své heslo transformovanou nějakou jednocestnou hashovací funkcí (MD5, SHA1, ...). Toto se označuje jako odpověď - response.

3. Autentizátor porovná odpověď s výsledkem ke kterému sám dospěl (zná heslo i užitou transformační funkci). Pokud se shodují, pak se autentizace zdařila, v opačném případě nikoli.

4. V náhodných intervalech se kroky 1. - 3. opakují. (Tento bod v našem protokolu vynecháme.)

Tento protokol je navržený jako protokol linkové vrstvy referenčního modelu ISO/OSI. My použijeme jeho principy, ale na aplikační vrstvě. To s sebou nenese prakticky žádná omezení, jen definujeme textový obsah packetu, jak jej získat a jak má být chápán. Tato výměna je pro naše účely definována tak, že modul (autentizátor) odešle 20 znaků dlouhou výzvu ke klientovi (autentizovaný). Výpočet a transformace je pak jednoduchá. Za výzvu se připojuje klientovo heslo a výsledný řetězec se transformuje funkcí MD5. Tento transformovaný text se odešle modulu. Konkrétní podoba této komunikace je popsána v části 3.5

3.4 Příkazy protokolu

Jako v klasické klient-server architektuře, klient navazuje komunikaci se serverem. Protože je třeba odesílat protokolem různé informace a reagovat na ně, definujeme řídící příkazy našeho komunikačního protokolu. Tyto budou textové a budou 4 znaky dlouhé. Následující seznam uvádí vždy příkaz a jeho význam:

- HELO - začátek komunikace
- CHAL - odesílání výzvy
- RESP - odesílání odpovědi
- ACCE - přijetí dat, potvrzení, že data jsou v pořádku
- DATA - informace o dodatečných datech
- ADIT - odesílání dodatečných dat pro test
- RESU - zaslání výsledku
- NOOP - prázdný signál

Možné hodnoty a jejich význam včetně významu příkazu jsou shrnuty v 5.1. Navržená syntaxe má potom tvar:

COMM\tCiselna_hodnotaTextova_hodnota\r\n

kde COMM označuje příkaz protokolu, \t znamená znak tabelátoru, \r znak návrat vozíku(CR - carriage return) a \n znak odřádkování(NL - new line). Ciselna_hodnota je hodnota typu uint16_t v síťovém pořadí bytů (big-endian) a Textova_hodnota je ASCII textový řetězec. Pokud se vyskytuje alespoň jedna hodnota (ať už textová nebo číselná), použije se pro její oddělení od příkazu tabelátor, pokud by se nevyskytovala ani jedna hodnota, pak se odešle jen

COMM\r\n

3.5 Průběh protokolu

Protokol jako takový bych chtěl, pro větší přehlednost, rozdělit na tři významnější části. Ty části pak jsou začátek komunikace, odesílání dat a zakončení komunikace. Začátkem komunikace rozumím komunikaci odeslanou po navázání spojení, v tomto kroku se odehraje autentizace. Odesílá ní dat je fáze, kdy již autentizované jádro odešle všechna data modulu, čímž splní svůj úkol. Zakončení komunikace je pak fáze, kde jádro systému již svou činnost dokončilo a jen čeká na zaslání výsledku práce modulu. Na tyto části dělím protokol také proto, že každá fáze má svou skupinu příkazů, které se v ní mohou vyskytovat. Výjimkou z tohoto pak je příkaz ACCE (který se může vyskytnout v první i druhé fázi) a příkaz RESU (který se může vyskytovat dokonce ve všech fázích komunikace).

Po navázání spojení klient (koordinátor) odešle modulu příkaz HELO. Tím dává najevo, že bude žádat o test a chce prokázat svou identitu pomocí CHAP podobnému systému - viz 3.3. Na toto modul odpovídá příkazem CHAL (1. fáze CHAP) s textovou hodnotou obsahující 20 znaků dlouhý, náhodně vygenerovaný řetězec. Tento klient přijme provede transformaci a výsledek odešle jako textovou hodnotu v příkazu RESP (2. fáze CHAP). Tento pak modul přijme, porovná s vlastním výsledkem a poté odešle buď příkaz RESU s číselnou i textovou hodnotou, které vyjadřují povahu chyby, anebo příkaz ACCE bez hodnot, který značí přijetí (3. fáze CHAP). Tím je zakončena úvodní fáze. Pokud klient přijme příkaz ACCE, pak pokračuje odesíláním dat, jinak ukončí komunikaci s chybou.

Klient nejprve zašle příkaz DATA, jehož číselná hodnota určuje kolik rozšiřujících dat se bude posílat. Protistrana toto přijme a pokud je počet dat dostatečný, odešle příkaz ACCE, v opačném případě posílá příkaz RESU a komunikace končí. V případě úspěchu klient odešle všechna data jako textové hodnoty příkazu ADIT a vyčká na jejich závěrečné potvrzení příkazem ACCE (popř. RESU v případě chyby - nevyskytla se důležitá proměnná pro test apod.).

Klient čeká na odpověď, kterou dostává ve formě příkazu RESU od modulu, který nese informaci o výsledku proběhlého testu. Vždy pokud tuto neobdrží během 1 sekundy, odesílá prázdný příkaz NOOP. Pokud odeslal příkaz NOOP a během následujícího sekundového intervalu neobdržel odpověď (ať už příkaz NOOP nebo RESU), ukončí klient komunikaci a uloží si chybový výsledek značící nezdařenou komunikaci.

3.6 Nevýhody navrženého protokolu

Předpokládal jsem, že zavedením číselné hodnoty si ušetřím parsování textu. Bohužel testování funkčnosti pak bylo složité, přičemž v případě čistě textového protokolu by bylo možné simulovat část činnosti modulů pomocí programu telnet.

Navržený jednosměrný systém autentizace koordinátora vůči modulu neobsahuje zpětnou autentizaci modulu vůči koordinátoru. Falešný modul může jen vygenerovat výzvu, vyčkat na odpověď a tuto potvrdit jako úspěšnou autentizaci koordinátorovu. Pak může bez vykonávání testů podvrhnout jejich výsledky. Vzhledem k nedůvěryhodnosti sítě a takovýmto skutečnostem soudím, že by bylo vhodné a možné malým zásahem do protokolu toto napravit. Tímto zásahem by se po autentizaci koordinátora vůči modulu prohodily role autentizátora a autentizovaného. Ani programová změna by v takovém případě neměla být nijak razantní, neboť iniciace komunikace je zaobalena do jedné jediné funkce.

Kapitola 4

Moduly z pohledu programátora

4.1 Proč se moduly zabývat

V případě, že je programátorská práce na koordinátoru hotova (nebo alespoň prozatím jsme s ní spokojeni), jediné na čem závisí budoucí činnost a schopnosti celého systému jsou právě moduly. Z tohoto důvodu přikládám velkou důležitost seznámení se se stavbou dosavadních modulů spolu se zkušenostmi s jejich naprogramováním. Právě nyní také přichází okamžik, kdy je třeba začít striktně rozlišovat, zda hovoříme o testovacích modulech nebo o modulech pro odeslání chybového hlášení. Důvodem je to, že nyní se začínáme blíže zabývat jejich činností, jejíž povaha moduly odlišuje. Přesto bych na tomto místě rád připomněl, že oba druhy modulů musí implementovat stejný komunikační protokol pro komunikaci s jádrem systému.

Další společnou vlastností pak je, že oba druhy modulů jsou ve své podstatě servery (viz. 3.1). Proto je nutností, aby komunikovali nad co možná největším počtem síťových protokolů. Aby bylo tohoto (spolu se stále více populární tvorbou IP agnostic aplikací) dosaženo, bylo zakládání síťových socketů pro vytvořené moduly napsáno obecně, takže by měli být otevřeny sockety pro všechny v systému dostupné síťové protokoly. Pak ale vyvstává otázka, jak všechny tyto sockety obhospodařovat. Uvažovali jsme buď obsluhu pomocí vláken nebo pomocí funkce select()[1]. V případě řešení s vlákny

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

by pro každý vytvořený socket bylo založeno vlákno, které by mělo za úkol jeho obsluhu, tj. naslouchat na příslušném socketu pomocí blokovacího volání accept() a po přijetí komunikace by provedlo test. V případě takovéhoto řešení by testovací funkce musela být buď napsána jako reentrandní anebo by bylo třeba ji opatřit mutexovým zámkem, ale potenciálně by to umožnilo více současných testů. Pokud užijeme řešení pomocí select(), pak se můžeme rozhodnout, zda chceme vykonávat jeden či více testů najednou. Protože jsme si uvědomili, že je sice možné, ale vyloženě nepravděpodobné a nepraktické, aby systém obsahoval více (o sobě navzájem neinformovaných) koordinátorů, není třeba reagovat na více podnětů najednou. Právě proto bylo rozhodnuto, že implementuji řešení pomocí select(). Toto řešení, s přihléhnutím k jednoznačnosti koordinátora, by také mělo šetřit systémové prostředky (právě za ušetřená vlákna) stroje, na kterém modul(y) běží.

4.2 Moduly pro odeslání informace uživateli

Moduly, které podávají informaci uživateli jsou, aspoň podle mého názoru, jednodušší. K tomuto názoru mě přimělo, že není třeba, aby implementovaly příkaz NOOP spolu s faktem, že samotné hlášení uživateli (jako je zápis do logu, výstup na obrazovku) se snadno naprogramuje, popřípadě lze k tomuto účelu obstojně užít příkazy dostupné na každém operačním systému UNIX (například program mail pro odeslání e-mailu). Také se během implementace ukázalo, že lze pro všechny odesílací moduly navrhnout jednotnou stavbu dat odesílaných po příkazu ADIT (spíše je to nutnost, neboť koordinátor, který informace těmto modulům sestavuje a odesílá je naprogramovaný). Toto jistě neplatí pro testovací moduly, protože u těchto si vše definuje programátor modulu a zadává je uživatel, takže nelze nic předpokládat ani o jejich pořadí.

Protože modul je server, je nutné, aby světu a jeho požadavkům naslouchal na síťovém portu, přes který budou obě strany komunikovat. Pro ověření identity jádra potřebuje znát heslo. Každý testovací modul, který jsem vytvořil funguje v několika fázích - načtení dat (heslo(a) a port(y)), otevření portu, čekání na komunikaci, přijetí dat, odeslání informace. Vždy po odeslání informace se modul vrátí k čekání na komunikaci. Načtení dat může být různé - z příkazové řádky, z konfiguračního souboru, jako vstup od uživatele,...

Práci budoucího programátora se mi snad podařilo (při spokojenosti s mým návrhem) omezit na vytvoření jedné funkce. Pro tyto účely jsem navrhl skelet (základ) pro odesílací modul. Tato funkce je umístěna v souboru main.c v adresáři s tímto projektem a pojmenoval jsem ji error_handler(). Skelet využívá načtení uživatelských dat z příkazové řádky a otevírá jeden port na všech možných adresách.

4.3 Moduly pro testování

Moduly pro testování jsou, vzhledem k charakteru jejich činnosti, složitější entity celého systému. Jejich činnost je jistě svým způsobem podobná. Opět funguje v několika fázích - načtení dat nutných pro běh (těch může být podstatně více než u odesílacích modulů), otevření portu, čekání na komunikaci, přijetí dat, vykonání testu, vyhodnocení výsledku testu a jeho odeslání. Po odeslání výsledku testu modul opět začíná čekat na komunikaci. Tento průběh je samozřejmě ideální, takže některé fáze se mohou přeskočit (například pokud se nebude koordinátorem zaslaná odpověď na výzvu shodovat s hashem ke kterému dospěl modul). Reakce na tuto a podobné situace je popsána v 3.5.

Vytvoření skeletu testovacího modulu mi vzhledem k rozmanitosti přišlo nemožné. Moduly se liší v načtených datech, v přijatých datech od koordinátora (ta upravují povahu testů, práh citlivosti testu, míru chyby,...), v testování a jeho vyhodnocování (pokud se vůbec výsledek testu vyhodnocuje a neodesílá se přímo). Abych však co nejvíce usnadnil tvorbu dalších modulů do tohoto systému, vytvořil jsem sadu funkcí, které mají za účel usnadnit programátorovi práci. Těmto funkcím se věnuji v 4.4.

Pro testovací moduly je také nutností (o povaze testu nic netušíme) implementovat celý komunikační protokol včetně reakce na příkaz NOOP. Reakce na příkaz NOOP je z pohledu modulu reakcí na asynchronní událost. Na takovouto událost lze reagovat několika způsoby. Aktivní čekání je reakce, při které se aktivně testuje, zda na prostředku (v našem případě socketu) nedošlo k události na kterou čekáme (v našem případě příchod příkazu NOOP). Při takovémto čekání bychom museli čas od času přerušit testování, zjistit zda nedošlo k příchodu dat na socket a pokračovat v testu. Druhým

zamýšleným řešením bylo využít blokovacího systémového volání a nového vlákna k testování příchodu příkazu NOOP.

Volba jednoznačně padla na druhý způsob řešení. K této volbě přispělo především to, že toto řešení nezasahuje do funkce provádějící testování. Stejné řešení bych doporučil i budoucím programátorům, proto uvádím více podrobností v 4.4. Implementaci reakčního vlákna usnadnil fakt, že komunikace směrem od koordinátora k modulu může v okamžiku založení vlákna obsahovat pouze příkaz NOOP, protože všechna ostatní data jsou již úspěšně odeslána a koordinátor pouze vyčkává na výsledek.

4.4 Funkce pro usnadnění tvorby modulů

V této sekci pojednám o funkcích, které jsem při psaní modulů vytvořil a použil. Při jejich vytváření jsem se snažil dosáhnout co největší obecnosti a tím zvýšit míru použitelnosti těchto funkcí. Tyto funkce nevyužívají jen testovací moduly, ale používají je i moduly odesílající informace.

- int start_module_comm(int fd, const char *passwd) - definována v hlavičkovém souboru net_comm.h
  Funkce zahájí komunikaci (zahájení je popsáno v 3.5) na socketu, který je vstupním parametrem. Druhým vstupním parametrem je heslo, které se má použít pro ověření identity koordinátora(viz 3.3).

- int send_comm(int s, const char *comm, const uint16_t *value, const char *str) - definována v hlavičkovém souboru net_comm.h
  Funkce slouží k odeslání příakzu protokolu do socketu s. Vstupní parametry jsou příkaz, číselná a textová hodnota. Pokud příkaz neodkazuje na textový řetězec, jedná se o chybu. Textová ani číselná hodnota uvedena být nemusí - potom se příslušné ukazatele nastaví na NULL. Funkce odesílá příkaz tak jak předepisuje 3.4.

- int read_comm(int s, char *comm, uint16_t *value, char **str, const size_t len) - definována v hlavičkovém souboru net_comm.h
  Funkce slouží k přijetí příkazu komunikačního protokolu ze socketu s. Případné hodnoty ukládá do proměnných value a str, kde str je ukazatel na pole charů o velikosti len (pokud je len nastaveno na 0, bude alokován dostatečně velký prostor pro uložení hodnoty pomocí

funkce malloc() - při takovémto užití je potřeba dynamiccky alokovanou paměť vrátit pomocí volání funkce free()).

- void * noop_handler(void *in) - definována v hlavičkovém souboru thr_noop.h
  Funkce zajišťuje běh vlákna pro odpovídání na příkaz NOOP. Ukazatel na socket, na kterém má být naslouchání realizováno se předává funkci jako vstupní parametr. Funkce využívá mutexovou proměnnou deklarovanou v tomtéž hlavičkovém souboru. Tuto proměnnou je třeba inicializovat z volajícího kódu pomocí funkce pthread_mutex_init(). Toto vlákno běží v detached módu, nevrací tedy žádnou hodnotu a jakmile nemá co dělat, ukončí se.

Výše uvedené funkce využívají služeb jiných pomocných funkcí - konkrétně passwd_challenge(), my_send() a my_recv() - všechny z hlavičkového souboru net_comm.h. Funkce my_send() a my_recv() jsou variací na funkce ze článku [5], kde jsou podobné funkce vytvořeny jako nadstavba nad elementárními I/O funkcemi read() a write(). Funkce passwd_challenge() ze svých parametrů počítá hodnotu hashe pro ověření identity koordinátora. V hlavičkovém souboru net_comm.h jsou také definovány konstanty pro příkazy komunikačního protokolu; tyto opět doporučuji používat.

Kapitola 5

Životní cyklus programu tester

5.1 Po spuštění

Celý životní cyklus projektu je zaznamenán na obrázku 5.1. Jakmile je program spuštěn, je nutné, aby získal od uživatele data nutná pro běh. Program se spouští bez parametrů příkazové řádky a všechna data získává pouze z konfiguračních souborů (viz. 2.2). Tato data musí načíst do svých datových struktur. Toto se však děje dvakrát, neboť jsem si uvědomil, že některá data načtená z konfiguračního souboru nejsou potřebná pro běh aplikace. Mám konkrétně na mysli jména modulů (jak je pojmenoval uživatel), protože ta slouží jen k provázání konfiguračních souborů a lze je nahradit úspornějšími ID čísly, se kterými se pak i lépe pracuje. Do stejné kategorie padne i jméno služby, na které závisí. Existují ale i opačné případy, tj. části datové struktury, podstatné pro běh, se nenačítají od uživatele (záznam chyb, stav).

Zda modul na který odkazuje service existuje je testováno v okamžiku, kdy se service načítá (vyhledá se modul). Testování zda neexistuje orientovaný cyklus pro service proběhne až po jejich kompletním načtení. Pro tento test byl použit algoritmus topologického třídění, při jehož implementaci byl brán v potaz fakt, že každá část systému má jednoho nebo žádného předchůdce (službu na které závisí). Po úspěšném otestování dat je založena prioritní fronta pro service. Pro prioritní frontu jsem užil datovou strukturu halda, která je uložena v poli. Jakmile máme i frontu, nebrání nic v započetí testování.

5.2 Výkonná část

Celé testování se odehrává v nekonečné smyčce. Získá se prvek pro testování (tj. ten který je na vrcholu prioritní fronty) a vyčká se na čas pro jeho test. V čase testu je nejprve zjištěn stav service, na kterém k testování určený service závisí. Pokud není ve stavu OK, je test přeskočen a následuje reakce na chybu. Toto bylo navrženo kvůli výpadkům v síti - pokud je nedostupný stroj (zjistí se třeba pingem), není třeba testovat, zda na něm běží služby (www,...) a není třeba uživatele informovat (SPAMovat), že nejen není dostupný stroj, ale ani žádná služba která tam má běžet. Pokud však je nadřízený service v pořádku, kontaktuje koordinátor příslušný modul (čímž vyžádá test) a odešle všechna data.

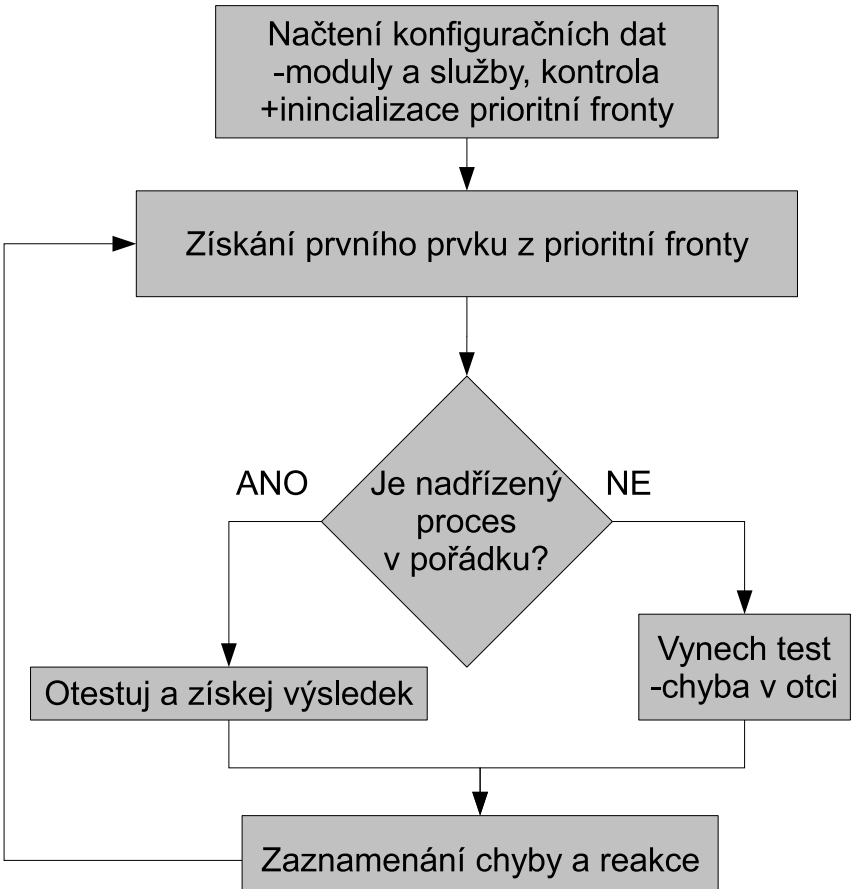
Po získání výsledku testu (za výsledek se považuje i chybou ukončená komunikace s modulem) je třeba na tento reagovat. Reakce závisí na typu výsledku - chybový či nechybový. V případě nechybového je jen nastaven stav service na OK (ať už předtím byl jakýkoli). V případě chybového je třeba zaznamenat chybu do struktury testovaného service, nastavit stav na chybový a pokud bylo dosaženo počtu chyb, pro který je definováno odeslání chybového hlášení pomocí modulu, je třeba kontaktovat tento modul a toto hlášení odeslat. Pak je již jen třeba vypočítat čas následujícího testování pro právě dotestovaný service a vrátit jej do prioritní fronty. Poté následuje nový test.

V případě prioritní fronty jsem se rozhodoval mezi ukládáním vždy všech k testování určených service (jak bylo popsáno výše) a ukládáním jen těch, jejichž testy mají vůbec šanci proběhnout (test prvku na němž jsou závislé proběhl v pořádku). Jak vidno volil jsem první řešení, neboť druhé by vyžadovalo zaznamenání struktury testovaných service do paměti procesu. Toto mi nepřišlo přirozené a proto jsem takovéto řešení zavrhl.

Dodatky

Signál	Číselná hodnota	Textová hodnota	Význam
HELO	Není	Není	Chci komunikovat
CHAL	Není	vygenerovaná výzva – 20 znaků	Posílám výzvu
RESP	Není	Md5 hash výzvy s heslem	Odpověď na výzvu
ACCE	Není	Není	Potvrzení přijetí minulých dat
DATA	Počet dat pro modul $< 1, 256 >$	Není	Kolik ADIT dat pošlu
ADIT	Není	Add data z konfiguračího souboru	Dodatečná data od jádra
NOOP	Není	Není	Ověření možnosti komunikovat
RESU	Návratový kód testu	Krátký textový popis (max. 255 znaků)	Návratový kód, jeho popis a konec komunikace

Tabulka 5.1: Shrnutí příkazů komunikačního protokolu



Obrázek 5.1: Diagram životního cyklu programu tester

Literatura

[1] Pachanec J.: *Slidy k Programování v UNIXu*, dostupné on-line na http://www.devnull.cz./mff/pvu/slides/programovani_v_unixu.pdf.

[2] Mitchel M., Oldham J., Samuel A.: *Pokročilé programování v operačním systému Linux*, Softpress. (2002) 75–108, 130–140.

[3] RFC-1994: *PPP Challenge Handshake Authentication Protocol (CHAP).*

[4] Dostálek L., Kabelová A.: *Velký průvodce protokoly TCP/IP a systémem DNS*, Computer press. (2002) 78–100, 135–138.

[5] Rudolf A. M., Fenner B., Stevens W. R.: *readn, writen, and readline Functions*, on-line článek o základních I/O funkcích - http://www.informit.com/articles/article.aspx?p=169505&seqNum=9. (2004)