Software pro testování a analýzu výkonnosti síťové aplikace

Testing and analysis software for performance measurements of client-server application

Bc. Aleš Bublík

Diplomová práce
2010

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Aleš BUBLÍK**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Software pro testování a analýzu výkonnosti síťové aplikace**

Zásady pro vypracování:

1. **Vypracujte literární rešerši na dané téma.**
2. **Použijte vhodné a snadno dostupné nástroje a technologie.**
3. **Navrhněte architekturu softwaru tak, aby bylo možno rozložit generování zátěže mezi více počítačů a návrh programově realizujte.**
4. **Proveďte měření výkonnosti HTTP serverů Apache, lighttpd, mochiweb, Tornado, databáze PostgreSQL a SMTP serveru Postfix.**
5. **Vyhodnoťte naměřená data.**

ABSTRAKT

Práce se zabývá návrhem a realizací nástroje pro testování a měření výkonnosti serverové aplikace se schopností využít výkonu připojených počítačů pro generování potřebné zátěže. Nástroj je schopen generovat zátěž pro libovolný HTTP server, SMTP server a databázi PostgreSQL, přičemž výsledky měření promítne do grafů. Aplikace byla napsána v jazyce Erlang, který umožňuje vytvářet robustní řešení toho typu.

Klíčová slova: testování, měření výkonnosti, Erlang, Linux, klient-server aplikace

ABSTRACT

This thesis concerns design and implementation of tool for testing and measuring of server application, with the ability to use the power of connected computers to generate the necessary load.The tool is able to generate load for arbitrary HTTP server, SMTP server and database server PostgreSQL, and measuring results are reflected in the charts. The application was written in Erlang language, which allows you to create robust solutions of that type.

Keywords: testing, performance measurement, Erlang, Linux, client-server application

Rád bych poděkoval svému vedoucímu Ing. Miroslavu Matýskovi, Ph.D. za vedení diplomové práce a čas strávený na konzultacích. Dále bych rád poděkoval Jitce Hlavačkové za četné připomínky a komentáře k mojí práci.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně ....

podpis diplomanta

OBSAH

ÚVOD **7**

I. TEORETICKÁ ČÁST **9**

1 VÝVOJ DISTRIBUOVANÉ APLIKACE **10**

1.1 Požadavky na vybavení 10

1.2 Programovací jazyk Erlang 10

1.2.1 Sekvenční programování 13
1.2.2 Konkurentní programování 16
1.2.3 Knihovna OTP 17
1.3 Návrh aplikace 19

1.3.1 Modul, který zaznamená měření klientských procesů 19
1.3.2 Modul, který zaznamenává chybové stavy klientů 21
1.3.3 Modul, který se stará o tvorbu klientů a jejich provoz 21
1.3.4 Modul, který zastupuje testovací process 22
1.3.5 Implementace HTTP klienta 24
1.3.6 Implementace SMTP klienta 26
1.3.7 Implementace PostgreSQL klienta 28
1.3.8 Supervizor aplikace 30
1.3.9 Aplikační modul 30
1.4 Fungování aplikace 31

1.4.1 Vyhodnocení výsledků 34
II. PRAKTICKÁ ČÁST **37**

2 TESTOVÁNÍ VÝKONNOSTI POMOCÍ ERLANGU **38**

2.1 Příprava 38

2.1.1 Konfigurace hardware 38
2.1.2 Konfigurace software 39
2.2 Testování HTTP serverů 40

2.2.1 Mochiweb 40
2.2.2 Apache 43
2.2.3 Lighttpd 47
2.2.4 Tornado 47
2.3 Testování SMTP serveru Postfix 50

2.4 Testování PostgreSQL serveru 55

ZÁVĚR **58**

CONCLUSION **59**

SEZNAM POUŽITÉ LITERATURY **60**

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK **62**

SEZNAM OBRÁZKŮ **63**

SEZNAM PŘÍLOH **64**

ÚVOD

Internet je už dnes součástí života většiny obyvatel vyspělých zemí. Většina lidí jej využívá především k zábavě, nákupu, komunikaci a také jako obrovský zdroj informací. Služby poskytované internetem nám zajišťují pohodlí ať už v komunikaci s úřady, tak i ve využívání služeb z běžného života. Dnešní firmy poskytující nějaký druh služeb se bez prezentace na internetu neobejdou. Společnosti, které stavějí svou existenci na internetu, se už vůbec neobejdou bez kvalitního technického zázemí a personálu. Pokud se stane jejich služba na internetu nedostupná, znamená to pro ně ztrátu, která může mít dalekosáhlé následky.

Každý se už určitě setkal se situací, kdy potřeboval nutně něco vyřídit na internetu. Může se ale stát, že služba, kterou jste potřebovali využít, z nějakého důvodu přestala fungovat. Uživateli pak jen zbývá trpělivě čekat až se služba zprovozní. Tedy jestli už není pozdě. Pokud ale existuje jiný způsob, i když ne tak pohodlný, jak služby využít, tím lépe. Ale pokud je internet jediným typem přístupu, pak se nedá nic dělat.

Společnosti, které vydělávají díky internetu, si musí zakládat na téměř okamžité odezvě svých služeb, protože nejhorší co můžete chtít po zákazníkovi je, aby si počkal. Dokonce i samotný Google dnes zahrnuje rychlost reakce webových stránek mezi hodnotící kritéria. Proto je nasazení monitorovacího a testovacího nástroje dostupnosti služeb pro ně naprostou nutností pro zajištění kvality služby. Postupem času se služba může stát více populární a uživatelská základna může narůstat až do té míry, kdy běžné nástroje na testování dostupnosti služby přestávají stačit. Nárůst připojených klientů se může snadno projevit v rychlosti odezvy u ostatních připojených klientů a tady se už potýkáme s problémem škálovatelnosti služby. To jak kvalitně je služba naprogramovaná, se nejlépe projeví, když se servery zatíží na maximální možnou zátěž.

Ovšem problém je v tom, jak nasimulovat reálné zátěžové podmínky. Určitě si nebudeme najímat tisíce lidí, aby ve stanovený čas zatěžovali naši službu. Jediná možnost je, že si vytvoříme soustavu několika počítačů, které spojíme do sítě spolu se servery, které poskytují naši službu. Na počítačích, které budou zastupovat připojující klienty pak nainstalujeme základní operační systém, přičemž jeden si zvolíme jako řídící počítač, ze kterého budeme obsluhovat ostatní počítače a řídit testování našich serverů. V našem případě, obsluha si vystačí pouze s jednou osobou.

Ovšem podstatný faktor pro každou společnost je, jaké budou náklady s tím spojené. Pro

toto téma diplomové práce byl záměrně využit běžně dostupný software v podobě operačního systému Linux, již přiložených softwarových balíčků pro vzdálený přístup SSH a balíčků pro programovací jazyk Erlang. Vše bezplatně dostupné z internetu.

Proč právě jazyk Erlang? Vlastnosti tohoto jazyka jsou užitečné právě v síťových aplikacích, a proto jej lze využít pro takový náročný úkol. Jazyk vyniká zejména schopností souběžně zpracovávat několik operací nezávisle na sobě a to díky tomu, že se jedná o funkcionální jazyk, který neklade překážky větší paralelizaci úloh. Je schopen využít všech jader mikroprocesoru a také umí jednoduše spolupracovat distribuovaně s ostatními počítači v síti, na kterých běží také Erlang.

I. TEORETICKÁ ČÁST

1 VÝVOJ DISTRIBUOVANÉ APLIKACE

1.1 Požadavky na vybavení

V úvodu bylo trochu nastíněno jak bude vypadat realizace tohoto projektu. Co se týče požadavků na hardware, nejsou potřeba žádné specifické vlastnosti. Jde zejména o tom, aby byl dostatek výpočetního výkonu na otestování síťové aplikace. Každý pomocný uzel (počítač) by měl mít samozřejmě síťovou kartu.

Dále na každém pomocném uzlu byl nainstalován Linux, konkrétně distribuce Ubuntu Linux. Ale nic nebrání v tom použít distribuci jinou. Jde zejména o to, aby byla na všech uzlech stejná verze softwaru, aby v tom byl pořádek a hlavně aby nebyly problémy. Dalším požadavkem je schopnost aplikace snadno distribuovat generovanou zátěž mezi více počítačů. Tato podmínka byla snadno splněna, protože Erlang je od základů vytvořen pro distribuovaná prostředí.

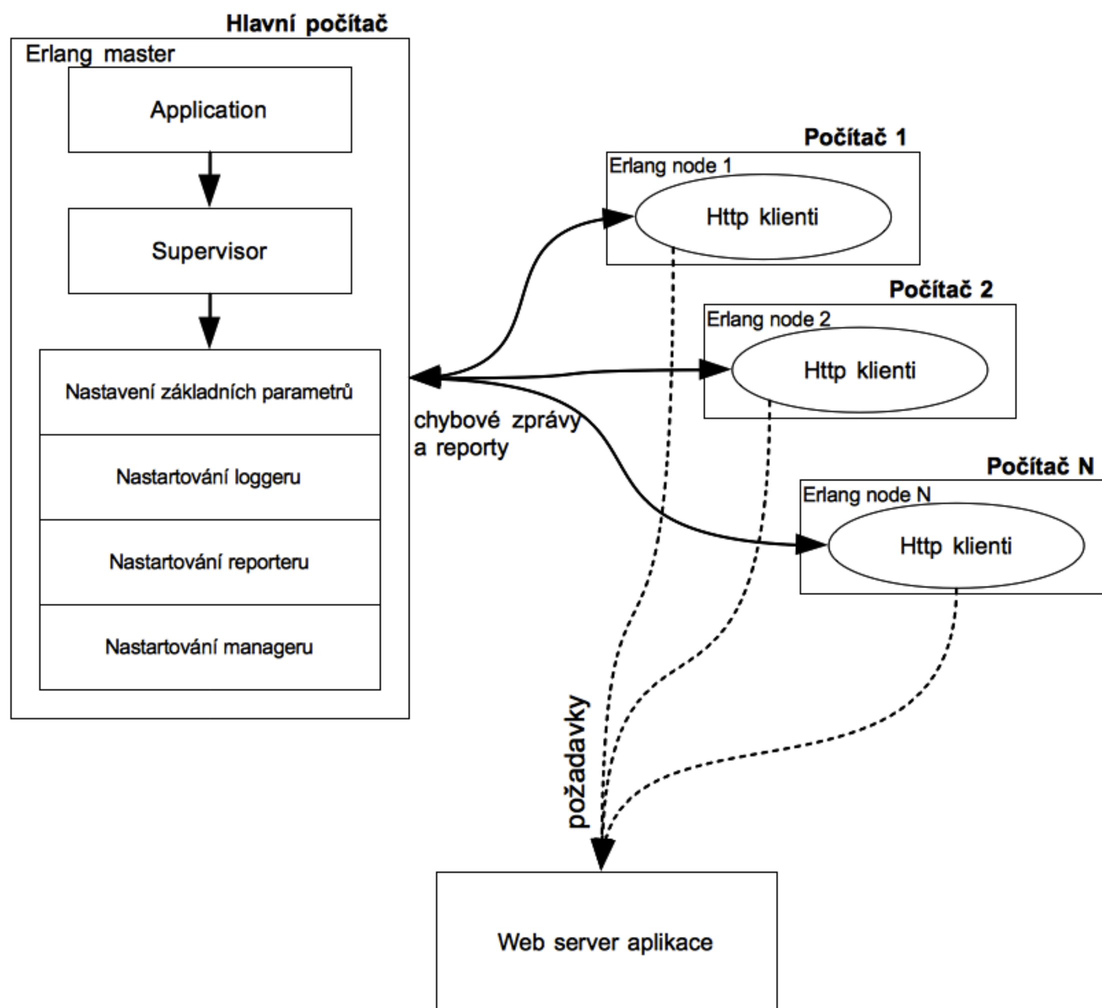
Takže potřebný software byl zvolen a zbývá upřesnit, jak bude vypadat obsluha a provoz. Obsluha nebude nijak náročná. Projekt dokáže zprovoznit kdokoliv kdo zná operační systém Linux a počítačové sítě. Stačí tedy jedna obsluha. Projekt samotný nebude nutné kopírovat na každý počítač, protože díky vlastnostem Erlangu bude tato funkce zajištěna programově.

Následující schéma aplikace popisuje jednotlivé uzly [14] (tzv. Erlang Node), na kterých je spuštěná instance jazyka Erlang. Tyto uzly pak komunikují s hlavním počítačem, který zaznamenává reporty a chybové zprávy a hlavně vytváří a řídí procesy na ostatních uzlech. Procesy pak na povel doslova bombardují aplikační server požadavky. Schéma pak zachycuje i strukturu samotné aplikace na řídícím počítači. Od spuštění aplikace (na obrázku část Application) přejdeme ke spuštění Supervizora, který se postará o načtení konfigurace a nastartování logovacího procesu, dále reportovacího procesu a manažera, který pak obsluhuje procesy na ostatních uzlech.

1.2 Programovací jazyk Erlang

Jazyk Erlang není tak obecně známý a rozšířený. Není to proto, že by toho moc neuměl, ale spíše proto, že se jedná o funkcionální jazyk[1], jež byl vyvinut původně pro aplikace v telekomunikacích ve společnosti Ericsson[2].

1 http://en.wikipedia.org/wiki/Functional_programming
2 http://www.ericsson.com/



Obr. 1. Schéma testovaného serveru spolu s uzly, které vytvářejí síťový provoz.

Spoustě vývojářů se může zdát zbytečné učit se další programovací jazyk, když přece už umí programovat v Javě. Jenže tak jednoznačné to není. Java si své místo už vybojovala, ale také se nehodí na všechno. Dnes společnosti jako Google, Facebook, Nokia a další nasazují tento produkt do svých softwarových řešení. Nejedná se tedy o žádnou věc na hraní, ale o robustní a výkonný jazyk.

V čem tedy Erlang [8] vyniká? Hlavní rysy jazyka jsou následující:

Distribuovatelnost

Erlang byl navrhnut tak, aby byl schopen distribuovat zátěž a funkcionalitu na ostatní uzly, ať už běží jednotlivé instance Erlangu vedle sebe na jednom počítači, nebo každé na svém počítači v rámci počítačové sítě. Tyto instance Erlangu se nazývají nody a ty je možno spojovat díky TCP/IP [7] [15] (Transmission Control Protocol / Internet Protocol) protokolu, přes který jednotlivé procesy mohou komunikovat transparentně a jednoduše tak, jako by běželi na jednom počítači.

Robustnost

Erlang obsahuje konstrukce jak kontrolovat pád aplikace z důvodu nějaké chyby. Je to tzv. „Fault-tolerant“ systém. Tyto konstrukce umožňují reagovat na tyto případy další funkcionalitou, např. restartem části aplikace nebo dokonce celého distribuovaného systému.

Konkurentní jazyk

To je vlastnost, kdy je jazyk schopen ve vlastní režii vytvářet a spravovat procesy, které mohou spolupracovat paralelně. Erlang dokáže vytvořit tisíce procesů (doslova mini procesů) podstatně rychleji a na systémové prostředky nenáročně, tedy na rozdíl od jazyků jako C, C++, Java a Python. Komunikace mezi procesy probíhá buď synchronně [5] a nebo asynchronně a to způsobem nazývaným „Message passing“, k čemuž nepotřebuje sdílenou paměť.

Soft real-time jazyk

Erlang je hodně rychlý co se týče komunikace mezi jednotlivými procesy a odezvy se většinou počítají v řádech milisekund. Umožňuje to zejména tzv. inkrementální „garbage collection“.

„Hot code upgrade“

Neexistuje žádný software bez chyby, tudíž když k této chybě dojde následuje vetšinou pád nebo zatuhnutí celé aplikace. To je pro serverové aplikace krajně nepřípustné a proto Erlang obsahuje konstrukce jak jednoduše změnit (opravit) funkcionalitu programu za jeho běhu. Takže v reálné situaci může dojít i k tomu, že jeden proces běží se starou verzí kódu, zatímco nové procesy běží s již opravenou verzí kódu.

Externí rozhraní

Tato rozhraní umožňují komunikovat s programy napsanými v C, Javě nebo Pythonu podobným způsobem jako by to byl další proces v rámci instance Erlangu. Těchto rozhraní je více, některé jsou rychlejší ale zato mohou shodit celou aplikaci. Jiné jsou pomalejší, ale zato vám neohrozí běh celé aplikace.

Je také potřeba říci, na co se Erlang nehodí. Nepoužívá se v oblasti spracovávání množství

souborů, kde vládnou zejména skriptovací jazyky. Erlang má sice podporu skriptování, ale s jeho rychlostí je to špatné. Koneckonců pro tyto úlohy taky nebyl konstruován. Pokud tedy chcete prohledávat např. dumpy databáze regulárním výrazem, volte raději jiné nástroje. Aplikace psané v Erlangu jsou většinou toho typu, kdy čekají na pozadí na požadavek, aby ho mohli v v zápětí spracovat. Tento jazyk především vyniká v oblastech aplikací typu klient - server.

Existují hodně aplikací využívajících Erlang. Tak tedy vzpomeňme replikační databázi Mnesia, XMPP (The Extensible Messaging and Presence Protocol) server ejabberd, HTTP (Hypertext Transfer Protocol) server mochiweb, dokumentačně orientovanou databázi CouchDB, OpenGL (Open Graphics Library) modelovací nástroj Wings3D, AMQP (Advanced Message Queuing Protocol) server RabbitMQ, map-reduce framework Disco od společnosti Nokia.

Další oblastí, kde se nasazení Erlangu přímo nabízí, je umělá inteligence[3] a neuronové sítě[4]. V rámci Erlangu lze provozovat až několik set tisíc procesů souběžně, které mohou všechny reprezentovat neuronovou síť. Když k tomu přidáme vlastnost Erlangu připojit další výpočetní uzly do jednotné sítě, tak se nám nabízí opravdu velké možnosti.

V následujících řádcích bude popsán jazyk Erlang, ale ne moc podrobně, protože o tom tato diplomová práce není.

1.2.1 Sekvenční programování

Když se řekne sekvenční [8] programování, znamená to, že program je psám tak, aby se postupně vykonával v jednom směru [3]. Opakem je paralelní programování.

Jako každý programovací jazyk má Erlang vlastní datové typy, které jsou v ostatních jazycích obvyklé, ale má i jednu neobvyklou vlastnost. Anglicky se vyjadřuje pojmem jako „single asignment language“ [2].

Jedná se o to, že jakmile přiřadíte proměnné hodnotu, nelze již této proměnné přiřadit jinou. Další důležitá věc je, že proměnná musí začínat vždy velkým písmenem. Následuje ukázka kódu z příkazové řádky.:

```
1> State = 3.14159.
2> State = 1.
** exception error: no match of right hand side value 1
```

3 http://trapexit.org/Artificial_Intelligence_with_Erlang:_the_Domain_of_Relatives
4 http://trapexit.org/Erlang_and_Neural_Networks

Takže když došlo k pokusu o změnu hodnoty proměnné, následovala chybová hláška, že se pravá strana neshoduje s hodnotou proměnné. To je další vlastnost Erlangu a nazývá se „pattern matching“ a hodně se využívá. Více o tom bude pojednáno dále.

Erlang využívá jeden zvláštní datový typ a to tzv. *atom*, který lze využít ke strukturování abstraktních datových typů nebo jako zprávu atd. Příklad takového atomu je následující.:

```
1> A = cleanup.
2> B = 'Work is done.'.
3> C = 'lower_case'.
```

Jedná se tedy o řetězce znaků, které jsou užitečné pro již zmiňovaný „pattern matching“. Ještě zde nepadla zmínka o tom, že se jednotlivé příkazy vyhodnotí až poté, co je ukončena tečkou. Pokud byla vložena sekvence příkazů, pak je potřeba je oddělit čárkou a pro ukončení příkazů dosadit tečku.

Celá čísla

např. 20, -315, 2#10101, 16#AB10F

Reální čísla

např. -3.14159

tuple

je neměnná struktura ohraničená složenými závorkami, která může obsahovat jakýkoliv další datový typ např. `{person, "Aleš", "Bublík"}`, `{age, 21}`

list

je klasický seznam v hranatých závorkách, který lze měnit např. `[1,2,4, {age, 21}, 'State']`

řetězec

je ohraničen dvojitými uvozovkami a je to vlastně také *list*, protože jde o posloupnost znaků, které jsou reprezentovány číselnými kódy např.: `"Aleš Bublík"` je vlastně seznam (*list*) těchto čísel `[65,108,101,353,32,66,117,98,108,237,107]`

Datových typů má Erlang více, ale tohle jsou ty nejvíce využívané. Podrobnější informace lze nalézt na stránkách http://www.erlang.org.

V čem se tento jazyk podstatně liší od ostatních, je to, jak přistupuje k hodnotám proměnných. Už tady padla zmínka o mechanismu „pattern matching“, který se často využívá. Například tuple `{person, "Aleš", "Bublík"}`, ze kterého je potřeba dostat jméno a příjmení bude vypadat takto. `{person, Jmeno, Prijmeni} = {person, "Aleš", "Bublík"}`. Tím se dosadí do proměnných *Jmeno* a *Prijmeni* hodnoty z pravé strany. Další konstrukce jsou např.:

```
1> X = 10.
2> [A, B, C] = [1, 2, 3]
3> [D, D, F] = [ales, ales, iks]
4> [G, H | T] = [1,2,3,4,5,6,7] % G = 1, H = 2, T = [3,4,5,6,7]
```

Další možnosti této funkcionality lze využít na úrovni funkce. Následující kód ukazuje funkci *cena*, které jako parametr dáme dvouprvkový tuple, kde na první pozici je atom jablka nebo hrusky a na druhé pozici je proměnná, která obsahuje číslo. Pokud tuple nesplňuje uvedený předpis, spustí se třetí varianta funkce, kde lze takový chybový stav ošetřit.:

```
-module(cenik).
-export([cena/1]).

cena({jablka, Pocet}) ->
 55.0 * Pocet;
cena({hrusky, Pocet}) ->
 83.0 * Pocet;
cena(Jine) ->
 invalid_object.
```

Každý program v Erlangu se skládá z tzv. modulů, což jsou soubory, které obsahují definované funkce a data. Např.:

```
-module(demo).
-export([double/1]).

double(X) ->
 times(X, 2).

times(X, N) ->
 X * N.
```

Jak je vidět na předchozím zdrojovém kódu, každý modul má své jméno (direktiva module), seznam funkcí, které budou přístupné vně modulu (direktiva export) a specifikace jednotlivých funkce (double, times). Volání funkce z konkrétního modulu se provádí např. takhle `demo:double(9)`. Popřípadě pokud se volá funkce, která se nachází v tom samém modulu, pak stačí pouze její název `time(9, 2)`.

Prostředí jazyka Erlang lze spustit příkazem `erl`. Předtím než se modul *demo* spustí, je

potřeba jej zkompilovat. A to buď příkazem `erlc demo.erl` nebo rovnou z prostředí Erlangu příkazem `c(demo)`. Erlang tak automaticky zajistí jeho inicializaci. Pokud je potřeba příkazovou řádku Erlangu ukončit, pak stačí zadat příkaz `q()`.

Některé funkce (tzv. BIFs) jsou obtížné natolik, že jsou v Erlangu raději implementovány v jazyce C. Jsou to například funkce: `date()`, `time()`, `length([1,2,3])`, `size({a,b,c})`, `atom_to_list(an_atom)` atd. Vyskytují se v modulu *erlang*, který obsahuje funkce spojené zejména se správou běžícího prostředí.

1.2.2 Konkurentní programování

Základní jednotkou konkurentního [8] programování je proces. V případě jazyka Erlang jsou tyto procesy plně v režii jazyka a ne operačního systému, tak jak to bývá u ostatních jazyků. Procesy jsou nenáročné na systémové prostředky a vynikají rychlou odezvou [2]. Každý proces je veden pod jedinečnou kombinací čísel např. `<0.24.0>`, kterému se říká Pid (Process identification). Dále každému procesu lze přiřadit jméno příkazem `register(Alias, Pid)` tak, aby jej bylo možno v systému volat jednodušeji.

Dalším znakem konkurentního programování je zabudovaný systém zpráv (tzv. Message passing). Což je způsob jakým jednotlivé procesy mezi sebou komunikují. Erlang také zavádí konstrukci, která umožňuje procesu čekat na zprávu pouze po určitý časový úsek a po uplynutí reagovat jinou funkcionalitou.

Vytváření procesu je snadné. Slouží k tomu příkaz `spawn(Mod, Func, Args)`. Parametry jsou tedy postupně název modulu, dále funkce v tomto modulu a seznam argumentů, který se funkci předá. Funkce pak vrací jedinečné číslo nového procesu. Vytvořit takový proces lze i na vzdáleném stroji.

Komunikace pak může vypadat následovně. Strana odesílatele zprávy příkazem `Pid ! {self(), start}` pošle procesu *Pid* zprávu (tuple), která obsahuje identifikaci odesílatele a atom *start*. Pokud chce odesílatel dostat odpověď, musí ve zprávě uvést svou identifikaci. Aby mohl příjemce přijímat zprávy musí obsahovat následující konstrukci.:

```
receive
 {Odesilatel, Message} ->
 Odesilatel ! {self(), ok},
 loop(Message)
 Other ->
 error
after 10000 ->
 timeout()
end
```

Kód jde vysvětlit následovně. Konstrukce `after 10000` říká, že po uplynutí 10000 milisekund, kdy nedorazila žádná zpráva, vykonej funkci `timeout()`. Je to sice nepovinná ale zato velmi užitečná věc. Pokud tedy do té doby dorazí nějaká zpráva, tak se porovná s tuplem `{Odesilatel, Message}` a vykoná se kód na následujících řádcích, tj. zpět odesílateli se odešle zpráva, že je vše v pořádku a vykoná se funkce `loop(Message)`.

Může ale nastat situace, kdy proces dostane jinou zprávu, a to v jiném tvaru, než má přijímat. Pro tyto případy je dobré přidat mezi `receive` a `after` proměnnou třeba s názvem `Other`, která odchytí všechny ostatní zprávy, které nesouhlasí s uvedeným předpisem (tj. dvouprvkový tuple).

1.2.3 Knihovna OTP

Knihovna OTP (Open Telecom Platform) řeší jednu zásadní věc. Sjednocuje způsob jak psát klient server aplikace srozumitelně a přehledně [16]. Vývojáři v laboratořích Erlangu vyřešili běžný problém u ostatních jazyků. Ať už programujete v jazyce C, C++, Javě nebo Pythonu máte vícero způsobů jak aplikaci napsat a jak navrhnout architekturu. Neexistuje tedy jeden univerzální a jednotný způsob tvorby takových aplikací a tudíž musí často vývojář věnovat značné úsilí, aby pochopil jak vůbec aplikace pracuje.

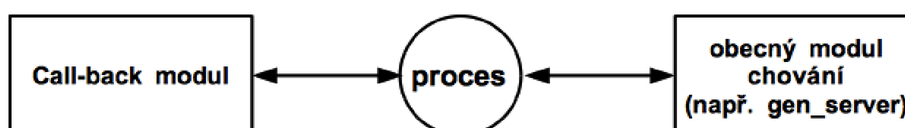
Je dobré vědět, že v Erlangu je také více způsobů, jak napsat klient server aplikaci. Bez knihovny OTP je ale vývoj funkcionalit jako je supervizor, politika restartování a reakce na chyby aplikace o něco náročnější.

Knihovna tedy standardizuje způsob zápisu a chování aplikace a vytváří tak vzorové návrhy, které lze opakovaně používat. Nejvíce takovým používaným návrhem je *gen_server*, který je v tomto projektu využívám. Používá se v situacích, kdy je potřeba aby se proces choval jako centrální server, který obsluhuje požadavky vyslané jednotlivými klienty.

Hlavním úkolem takových modulů jsou následující činnosti [16]:

- vytvářet proces s definovaný modulem a případně jej registrovat pod určitým jménem
- odesílat a přijímat zprávy od jiných procesů (klientů) buď jako synchronní nebo asynchronní zprávy, včetně definice protokolu zasílání zpráv
- starat se o samotný proces a uchovávat aktuální data
- zastavení procesu

Základní myšlenkou je tedy použít standartní modul knihovny OTP tzv. *behaviour modul*, který definuje způsob chování, a v jiném modulu definovat tzv. callback funkce, které už vykonávají konkrétní funkcionalitu. V tomto modulu je ale potřeba toto chování deklarovat direktivou *behaviour*. Takže pro případ modulu *gen_server* to vypadá takhle: `-behaviour(gen_server).`. Jenže to není všechno. Je nutné doplnit modul o funkce jako *handle_call* , *handle_cast*, *handle_info*, *terminate* a *code_change*. Bez nich nepůjde přeložit tento modul.



Obr. 2. Rozdělení funkcionality procesu na obecnou a specifickou část.

Následující zdrojový kód představuje minimální formu, která je nutná pro definici takového modulu.:

```
-module(muj_server).
%% gen_server_mini_template
-behaviour(gen_server).
-export([start_link/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
 terminate/2, code_change/3]).

start_link() ->
 gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
 {ok, State}.

handle_call(_Request, _From, State) -> {reply, Reply, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

Nejdůležitjší funkcí je *start_link*, která vytvoří nový proces. Poté vykoná funkci *init*, které také předá parametry. V této funkci lze nadefinovat stavové proměnné, nebo zde lze uchovávat odkaz na datové uložiště. Pro tyto informace slouží část *State*, která zastupuje interní proměnné, se kterými se pracuje.

Prostřednictvím funkcí *handle_call* a *handle_cast* se definuje protokol, tj. zprávy, na které proces reaguje funkcionalitou a to buď synchronně nebo asynchronně. Další funkce *handle_info* slouží pro informování supervizora o stavu procesu. Funkce *terminate* může obsahovat funkcionalitu, která se vykoná v případě chyby, která donutí proces ukončit. A poslední funkce *code_change* je volána v případě, že je potřeba za běhu nahradit modul

novým.

Kromě výše uvedeného modulu chování *gen_server* obsahuje Erlang další modul *gen_fsm*, který implementuje chování konečného automatu. Využívá se hlavně v případě, kdy je potřeba, aby se funkcionalita vykonávala podle předem určeného scénáře a uchovávala tak svůj definovaný stav. Což se docela hodí při implementaci síťových protokolů. Další modul *gen_event* slouží pro zpracovávaní a správu tzv. notifikací a reakcí na ně. Díky tomuto modulu lze implementovat třeba logovací proces.

1.3 Návrh aplikace

Samotná aplikace využívá původní knihovnu OTP. Jejím úkolem je usnadnit vývoj aplikací do té míry, aby se programátor nemusel zaobírat věcmi jako „hot-code upgrade“, restartovací politiky, distribuovatelnost atd., a mohl se rovnou zaměřit na požadovanou funkcionalitu [17].

V následujícím textu budou nejprve popsány všechny části (moduly) projektu a pak samotný běh aplikace od začátku až do konce.

1.3.1 Modul, který zaznamená měření klientských procesů

Stěžejní modul tohoto projektu byl nazván reporter. Jak název napovídá, jeho úkolem je zaznamenávat výsledky měření jednotlivých procesů, které bombardují testovaný server požadavky (requesty). Jelikož může být těchto procesů stovky, dá se tedy předpokládat, že tento modul bude hodně vytížen.

Zde využijeme direktivu behaviour s parametrem *gen_server* knihovny OTP. Ta způsobí to, že se modul reporter bude chovat jako server, který přijímá požadavky od jednotlivých klientů. Dále bude nutné měření průběžně ukládat do souboru.

Moment, kdy se jednotlivé reporty ukládají, je kritický. Je nepředstavitelné, aby každý příchozí report musel být okamžitě uložen do souboru. Jednak z toho důvodu, že zápis do souborů je náročný na systémové prostředky a také z toho důvodu, že takové operace dlouho trvají.

Obr. 3. Návrh aplikace a komunikace mezi jednotlivými procesy.

Naštěstí je k dispozici modul *ets*, což je paměťové uložiště, které se hodí pro rychlé operace. Data, které do něj vložíte jsou uloženy v operační paměti, přičemž operace ukládání a vyhledávání jsou podstatně rychlejší než operace s daty v souboru. Takže řešení spočívá v tom, že příchozí reporty si se budou průběžně ukládat v operační paměti a v pravidelných intervalech je se budou uklízet do konkrétního souboru.

Následuje ukázka jednoho reportu:

```
1271860801475295|<0.73.0>|1062|CONNECTION created.
1271860801475365|<0.73.0>|37|GET request sent.
1271860801476146|<0.73.0>|750|RECEIVED data.
```

Report se skládá standartně ze tří řádků. Řádek je dále rozdělen do čtyř polí.

- První pole obsahuje absolutní čas zaznamenání události.
- Druhé pole obsahuje v lomených závorkách číslo procesu, potažmo číslo HTTP klienta.
- Třetí pole je čas strávený danou operací.
- Čtvrté pole obsahuje popis operace.

Všechny tyto záznamy budou pomocí skriptu analyzovány a výstupem bude i graf. Více o tom bude psáno v následujícím textu.

1.3.2 Modul, který zaznamenává chybové stavy klientů

Neméně důležitý modul, který zaznamenává chybové události běžících klientů. Pokud se tedy klient dostane do nepatřičného stavu, např. odmítnutí spojení se serverem, pak o tom dá vědět odesláním zprávy tomuto tzv. loggeru chybových stavů.

Princip, na kterém pracuje tento logger je podobný reporteru, ale nelze zde předpokládat takové vytížení. Opět ukládá došlé zprávy do interní datové struktury, která se uchovává v operační paměti, ale k uklízení zpráv na disk zde nedochází tak často.

Následující odstavec ukazuje řádky, které jsou uvozeny časovým údajem, po kterém následuje stručný popis události.:

```
Wed, 21 Apr 2010 14:30:17 GMT: Socket error in: <0.75.0> with: timeout.
Wed, 21 Apr 2010 14:40:21 GMT: Client <0.71.0> exits loop.
```

První řádek říká, že klient nebyl v daném časovém limitu obsloužen. A druhý řádek zas, že klient ukončil generování požadavků na server.

1.3.3 Modul, který se stará o tvorbu klientů a jejich provoz

Další podstatný modul, díky kterému se řídí vytváření a spouštění generování požadavků se jmenuje manager. Stejně jako předchozí moduly, i tento využívá pomocí direktivy behaviour modul *gen_server*, který modulu přidá vlastnost obecného serveru.

Tento modul si udržuje v proměnných seznam vytvořených procesů, které budou generovat požadavky a další proměnné, které předá vytvořeným procesům nutným pro jejich běh.

Nejdůležitějšími parametry je seznam adres na které se budou generovat požadavky, délka trvání generování a maximální prodleva mezi generováním požadavků každého procesu.

Modul obsahuje následující direktivy.

start_clients(Node)

procesy vytvořené na pomocném uzlu (proměnná Node) začnou generovat požadavky

stop_clients(Node)

procesy běžící na danému uzlu dokončí požadavek a vyčkávají na další pokyn

is_stopped(Node)

příkaz poskytne informaci o tom, zda jsou procesy na daném uzlu stopnuty

create_clients(Node, Amount, Start={start, Boolean})

příkaz vytvoří na konkrétním uzlu požadovaný počet procesů (Amount), pokud je nutné procesy odstartovat okamžitě, pak proměnná Boolean musí obsahovat příkaz true

create_clients_on(On, Node, Amount, Start={start, Boolean})

příkaz vytvořil na daném uzlu (On) požadovaný počet procesů (Amount), pokud je nutné procesy odstartovat okamžitě, pak proměnná Boolean musí obsahovat příkaz true

create_infinitely_on(On, Node, Amount, Seconds)

příkaz bude vytvářet na daném uzlu (On) požadovaný počet procesů (Amount) po zadaných sekundách (Seconds) a tyto procesy hned odstartuje. Příkaz ve výsledků bude kontinuálně navyšovat počet procesů generujících požadavky

1.3.4 Modul, který zastupuje testovací process

Jak už bylo v předchozím textu zmíněno, modul manager poskytuje příkazy, kterými lze na jakémkoliv uzlu vytvořit určitý počet procesů. Tyto procesy zastupuje modul *projekt_client*, který je docela jednoduchý a definuje chování procesu po jeho vytvoření, zastavení a generování požadavků.

Chování aplikace je následující. Prostřednictvím manažera bude vytvořen požadovaný počet procesů na každém pomocném uzlu. Procesy budou zatím čekat na pokyn od manažera. Až bude vše připraveno lze prostřednictvím manažera všechny procesy naráz odstartovat. V ten moment začnou procesy bombardovat server requesty. Jakmile každý testovací proces skončí a vyhodnotí dotaz, zastaví se a vyčkává určitý čas. Jak dlouho bude čekat záleží na funkci `random:uniform(Delay)`, která vrací pseudonáhodnou hodnotu, která ale může být maximálně taková, jaká je uvedena v proměnné `Delay`. Po této pauze se proces znovu uvede do činnosti a vyšle znovu dotaz na testovaný server.

Následující část kódu řeší odstartování procesu a jeho zastavení.:

```
init(Conf, Delay) ->
 receive
 {start, From} ->
 From ! {ok, self()},
 loop(Conf, Delay);
 {stop, _From} ->
 logger:log(?LOGGER,"Client ~p exits.", [self()]),
 exit(normal);
 Other ->
 logger:log(?LOGGER,"Error!!! Client ~p receives: ~s.", [self(), Other])
 end.
```

Vstupní bod této funkce tvoří první řádek *init* s konfigurací pro klienta (Conf), která odkazuje na testovaný server, a maximální prodlevou (Delay) v milisekundách. Další řádek začínající direktivou *receive* a končící direktivou *end* uvozuje oblast kódu, která přijímá zprávy z jiného procesu. V tomto případě jsou to zprávy `{start, From}`, `{stop, _From}` a `Other`.

Takže pokud proces dostane zprávu uvozenou direktivou *start*, odpoví procesu, který zprávu poslal, zprávou `{ok, self()}` a začne vykonávat kód ve funkci *loop*, která bude uvedena později.

Dále pokud proces přijme zprávu uvozenou direktivou *stop*, pak tuto událost uloží do logu a proces skončí direktivou `exit(normal)`.

Všechny ostatní příchozí zprávy, které nejsou explicitně uvedeny mezi direktivami *receive* a *end* se automaticky zařadí do proměnné *Other*, kde se taková zpráva zaznamená (`logger:log`).

Takže jakmile je proces vytvořen, čeká až dostane nějakou zprávu. Pokud dostane zprávu `{start, From}`, potvrdí že zprávu zachytil a odskočí do funkce `loop({http, Hosts}, Delay)`.

```
loop({http, Hosts}, Delay) ->
 Host = lists:nth(random:uniform(length(Hosts)), Hosts),
 http_client:conn(Host),
 receive
 {stop, _From} ->
 logger:log(?LOGGER,"HTTP client ~p exits loop.", [self()]),
 init({http, Hosts}, Delay)
 after 0 ->
 timer:sleep(random:uniform(Delay)),
 loop({http, Hosts}, Delay)
 end;
```

V této smyčce si proces vybere náhodně HTTP adresu ze seznamu a vykoná jeden request. Následuje `receive after end` direktiva, která si ověří zda mezitím nedostal proces pokyn k zastavení a pokud ne, tak na nic nečeká, vypočítá si čas jak dlouho má pauzu a pak pokračuje v generování dalšího requestu.

Kromě toho modul obsahuje další dvě smyčky tj. `loop({smtp, Par}, Dealy)` pro spouštění SMTP (Simple Mail Transfer Protocol) klienta a `loop({pgsql, Par}, Delay)` pro spouštění PostgreSQL klienta. Tyto mají téměř totožnou strukturu jako HTTP klient.

1.3.5 Implementace HTTP klienta

Protokol HTTP byl vyvinut v roce 1990 ve Švýcarsku v CERNu (European Organization for Nuclear Research) jistým Timem Berners-Lee na počítači od společnosti NeXT [12]. Někoho možná překvapí, že tuto společnost založil samotný Steve Jobs, který dnes opět řídí společnost Apple. Nicméně díky tomuto protokolu se prosadila celosvětová počítačová síť internet tak jak si známe dnes. Jedná se o tzv. *request-response* protokol pro klient-server architekturu. Protokol definuje způsob jak se dotazovat na určité zdroje, které jsou v definovaném formátu URI (Uniform Resource Identifiers). Zdrojem [12] zde může být jakýkoliv objekt (tj. binární soubor, textový soubor), ale nejčastěji jím je soubor s HTML (HyperText Markup Language) značkami. Právě specifikace značkovacího jazyka HTML je nedílnou součástí internetu.

Původní verze protokolu je HTTP/1.0 se ještě využívá, ale trpí takovým nedostatkem, kdy je pro každý odkazovaný zdroj vytvořeno extra spojení, načež je pak uzavřeno. Když si uvědomíme, že prostřednictvím prohlížeče navštívíme zhruba 5 stránek za minutu, kdy se musí pro každý objekt na stránce vytvořit zvláštní spojení [11], pak se jedná o docela velké plýtvání zdroji serveru.

Protokol verze 1.1 [12] tohle již řeší lépe a pro požadovanou stránku z konkrétního serveru vytvoří pouze jedno spojení, přes které stahuje všechny objekty definované v HTML

dokumentu. Navíc takového spojení ihned neukončuje, ale ponechává jej otevřené určitou dobu pro případné další požadavky.

Cílem této práce ale není vytvořit plnohodnotného HTTP klienta, a pro tyto účely lze vystačit s jednoduchým klientem, který ovládá verzi 1.0. Erlang sice obsahuje HTTP klienta, ale pro tento projekt je nevyhovující, protože jej nelze jednoduše spouštět jako samostatný proces. Proto bylo nutné přistoupit k vlastní implementaci jednoduchého HTTP klienta, který podporuje metodu *GET*.

Implementace byla rozdělena na tři funkční části.

- vytvoření spojení se serverem
- poslání požadavku (*GET request*)
- přijetí odpovědi

Každý takový krok byl obalen funkcí, která změří jak dlouho trvala daná operace. Funkce není nijak složitá a vejde se na tři řádky. Přijímá pouze tři parametry a to modul, ve kterém se funkce vyskytuje, vlastní název funkce a její parametry. Získáme tak časový interval, po který daná funkce pracovala. Viz. následující zdrojový kód.:

```
tc(M, F, A) ->
 Before = os:timestamp(),
 Val = (catch apply(M, F, A)),
 {timer:now_diff(os:timestamp(), Before), Val}.
```

První krok, tj. vytvoření spojení, spočívá v navázání TCP [6] spojení se serverem. Tahle část bývá časově nejdelší. Pak následuje vytvoření hlavičky dotazu, která může vypadat takhle.:

```
GET / HTTP/1.1
Host: http://www.seznam.cz
User-Agent: UTBClient/1.00
From: UTBProjekt
\r\n\r\n
```

Proces tedy hlavičku dotazu pošle serveru a čeká na odpověď. Jakmile data dorazí, všechna přijme, uzavře spojení a zaznamená časy do reportu. Hlavní tělo funkce vypadá následovně a lze z něj pěkně vyčíst funkcionalitu, která zde byla popsána.:

```
{ok, Socket} = Res,
Acc1 = [?REPORT_LOG(Time1, "CONNECTION created.")],
{Time2, _Res2} = tc(?MODULE, request, [get, Socket, Host, Path]),
Acc2 = [?REPORT_LOG(Time2, "GET request sent.")|Acc1],
{Time3, Bin} = tc(?MODULE, rec_response, [get, Socket]),
Acc3 = [?REPORT_LOG(Time3, "RECEIVED data.")|Acc2],
result_log(Acc3),
```

Před samotným spuštěním testu je nutné vyplnit základní parametry do konfiguračního souboru `http_client.conf`. Obsah souboru vypadá následovně.:

```
{http_hosts, ["http://192.168.1.102:8000",
"http://192.168.1.102:8000/shop"]}.
```

Je nutné dodržet uvedený tvar. Tj. první znak na jediném řádku musí být složená závorka, kterou následuje direktiva *http_hosts* a po čárce v hranatých závorkách musí být uvedeno celé URL (Uniform Resource Locator) zdroje. URL adres může být libovolné množství, ze kterého si pak klientské procesy náhodně vybírají.

1.3.6 Implementace SMTP klienta

Protokol SMTP se používá pro odesílání a doručování elektronické pošty a jeho první specifikace se datuje do roku 1982 [10]. Dnes je to hodně využívaný protokol, kterým lze díky standardu MIME (Multipurpose Internet Mail Extensions) přenášet libovolný ať textový nebo binární soubor [1].

Je potřeba zdůraznit, že tato aplikace testuje pouze část protokolu, kdy se jedná o doručení mailu prostřednictvím mailového agenta MUA (Message User Agent) konkrétnímu serveru MSA (Message Submission Agent), který se už postará o jeho doručení.

Jelikož nativní implementace SMTP klienta v Erlangu neexistuje, bylo rozhodnuto přistoupit k vlastní implementaci jednoduchého klienta. Na rozdíl od protokolu HTTP, má SMTP protokol obsáhlejší zásobu příkazů, které je nutno vyslat, aby byl mail odeslán.

Nejdříve je ale potřeba si definovat konfigurační soubor, který musí obsahovat následující nutné informace. Předem je tedy nutné znát jméno testovaného SMTP serveru, uživatelský účet a heslo, dále je potřeba definovat mailovou adresu odesílatele a předmět mailu, samotný text zprávy a seznam mailových adres, na které se má email odeslat.

`smtp_server`

  název SMTP serveru, který přijímá maily

`smtp_login`

  jméno uživatele SMTP serveru, pod kterým odesílá mail

`smtp_passwd`

  heslo k mailové schránce

`smtp_from`

lze očekávat, že to je ta samá mailová adresa jako v případě `smtp_login`, ale specifikace protokolu to nevyžaduje

`smtp_to_mail_addresses`

seznam mailových adres, na které se má mail odeslat

`smtp_subject`

předmět zprávy

`smtp_msg`

samotný text zprávy

Konfigurační soubor se jmenuje `smtp_client.conf` a celý vypadá následovně.:

```
{smtp_server, "alex.xx"}.
{smtp_login, "ales.bublik@alex.xx"}.
{smtp_passwd, "heslo"}.
{smtp_from, "<ales.bublik@alex.xx>"}.
{smtp_to_mail_addresses, ["ales@alex.xx", "fantom@alex.xx"]}.
{smtp_subject, "pokus"}.
{smtp_msg, "Pozor! Tohle je testovaci zprava"}.
```

Příkazy protokolu lze rozdělit na několik částí, tak jak jdou po sobě. První část tvoří logovací údaje, dále adresy odesílatele a příjemce a poslední částí je samotné tělo mailu. Nejdřív ale popořádku. Samozřejmé je úvodní navázání spojení následované uvítacím příkazem `HELLO alex`, dále logovacími údaji následované po příkazu `AUTH LOGIN` tj.:

```
send(Mod, Socket, "AUTH LOGIN"),
send(Mod, Socket, binary_to_list(base64:encode("ales@alex.xx"))),
send(Mod, Socket, binary_to_list(base64:encode("heslo"))),
```

Po zalogování je potřeba uvést mailovou adresu odesílatele příkazem `MAIL FROM: ales.bublik@seznam.cz` dále mailovou adresu příjemce `RCPT TO: ales@gmail.com` a nakonec obsah mailu. Samotný text mailu začíná tedy příkazem `DATA`, po kterém následuje parametr `From: ales.bublik@seznam.cz`, dále `To: ales.bublik@gmail.com`, datum odeslání a předmět mailu `Subject: test`. Po odeslání znaků konce řádku je nutno poslat text mailu ukončený znovu znaky konce řádku. Nakonec je potřeba zaslat serveru znak tečky následovaný příkazem `QUIT`.

1.3.7 Implementace PostgreSQL klienta

Relační databázový server PostgreSQL[5] je otevřená alternativa k databázovému serveru [9] od společnosti Oracle. Sice nemusí mít všechny jeho vlastnosti, ale přesto dokáže poskytnout robustní řešení. Nejznámějším příkladem nasazení PostgreSQL je společnost Skype nebo Yahoo, která díky němu spravuje petabyty dat.

Implementace PostgreSQL klienta není v jazyce přítomna, ale jsou k dispozici volně dostupné balíčky. Je možné si vybrat s více alternativ, ale zřejmě nejpoužitelnější se jeví ta od Willa Glozera[6]. Ta vyniká jednoduchým návrhem a netrpí některými nedostatky jako v případě pgsql klienta (http://github.com/noss/pgsql).

Zakomponování tohoto modulu není obtížné. Přiložená dokumentace hovoří o základních funkcích modulu, tj.

- `pgsql:connect`
- `pgsql:squery`
- `pgsql:equery`
- `pgsql:close`

Scénář klientského modulu, který potřebuji vytvořit je následující. Pro samotné testování výkonnosti databáze je nutné uvést v souboru `postgresql.conf` parametry pro přihlášení. Tj.

`pg_host`

název serveru, který chceme testovat

`pg_name`

jméno uživatele, pod kterým se do databáze přihlašujeme

`pg_pwd`

heslo účtu

`pg_database`

název databáze, na kterou se budou vykonávat tzv. *selecty*

`pg_queries`

je vlastně seznam SQL (Structured Query Language) dotazů, které se budou vykonávat na testovaném serveru a to v náhodném pořadí zvlášť u každého

5 http://www.postgresql.org
6 http://glozer.net/src/epgsql/

simulovaného klienta, např. `["select now()", "select 1 + 1"]`

Takže celý konfigurační soubor `postgresql.conf` může vypadat následovně.:

```
{pg_host, "localhost"}.
{pg_name, "ales"}.
{pg_pwd, "alesbublik"}.
{pg_database, "test"}.
{pg_queries, ["select now()", "select 1 + 1"]}.
```

Dále modul `pgsql_client.erl` musí obsahovat, stejně jako ostatní klientské moduly, logovací funkce pro zaznamenávání činností tj. `REPORT_LOG` a `result_log`. Poslední a nejdůležitější částí je funkce `connect`. Ta přijímá výše uvedené parametry z konfiguračního souboru. Funkce nejdříve vyhodnotí čas potřebný pro přihlášení se do databáze a pokud proběhne úspěšně, zaznamená čas, dále si vybere náhodně jeden SQL dotaz, vykoná jej, změří dobu, zaznamená časy, uzavře spojení s databází a odešle výsledky reportovacímu procesu.

V případě, že se nepodaří připojit k databázi, zaznamenáme to do chybového logu. Celý funkce `connect` je na následujících řádcích.:

```
connect(Host, Name, Pwd, Database, Queries) ->
 Acc = [],
 {Time1, Res} = tc(pgsql, connect,
 [Host, Name, Pwd, [{database, Database}]]),
 case Res of
 {ok, C} ->
 Acc1 = [?REPORT_LOG(Time1, "CONNECTION created.")|Acc],
 Query = lists:nth(random:uniform(length(Queries)), Queries),
 {Time2, _Val} = tc(pgsql, squery, [C, Query]),
 Acc2 = [?REPORT_LOG(Time2, "QUERY done.")|Acc1],
 pgsql:close(C),
 result_log(Acc2);
 {error, Error} ->
 logger:log(?LOGGER, "Socket error in: ~p with: ~p.",
 [self(), Error]),
 exit(kill)
 end.
```

Modul `projekt_client`, který zastupuje klientské procesy, obsahuje vlastní část pro testování PostgreSQL, která je zase viditelná z pohledu manažera klientů. Vedle předaných parametrů pro přihlášení přijímá smyčka `loop` i parametr o maximální prodlevě mezi jednotlivými SQL dotazy. Takže poté co úspěšně provede daný dotaz a ukončí spojení, podívá se, zda nedostal ukončovací zprávu `stop`. Pokud ne, pokračuje ve smyčce `loop` a vykonávání dotazů. Ale v případě, že dostane zprávu `stop`, zaznamená to do logovacího procesu a vyčkává ve funkci `init`. V podobě zdrojových kódů to lze vyčíst zde.:

```
loop({pgsql, Par}, Delay) ->
 pgsql_client:connect(Par#pgsql_conf.pg_host,
 Par#pgsql_conf.pg_name,
 Par#pgsql_conf.pg_pwd,
 Par#pgsql_conf.pg_database,
 Par#pgsql_conf.pg_queries),
 receive
 {stop, _From} ->
 logger:log(?LOGGER, "PGSQL client ~p exits loop.", [self()]),
 init({pgsql, Par}, Delay)
 after 0 ->
 timer:sleep(random:uniform(Delay)),
 loop({pgsql, Par}, Delay)
 end.
```

1.3.8 Supervizor aplikace

Tento modul má za úkol spustit ostatní moduly a dohlížet na jejich správný běh [8]. Tato funkcionalita je především určena pro moduly, které implementují některou z vlastností např. gen_server, gen_fsm, gen_event atd. Umožňuje nadefinovat chování restartování jednotlivých modulů ve správném pořadí tak, aby byla zajištěna patřičná funkcionalita, tzv. strategii restartování.

Pro definování modulu jako supervizora je potřeba zadat parametr *supervisor* do direktivy *behaviour*. Bez ní by to nešlo. Pro běh aplikace jsou stěžejní tři části. Jsou to:

- logger (logger.erl)
- reporter (reporter.erl)
- manager (projekt_manager.erl)

Pro můj projekt není důležité v jakém pořadí se moduly spustí, ale je nutné, aby byly ihned znovu restartovány, pokud tedy nastane chyba.

1.3.9 Aplikační modul

Modul *projekt_app.erl* je vstupním bodem aplikace. Využívá direktivy *behaviour* s parametrem *application*, což ji přisuzuje vlastnosti standartní erlangovské aplikace, kterou lze takto administrovat, tj. nastartovat, stopnout atd [8].

Dále je potřeba vytvořit soubor s popisem aplikace, kde se uvede seznam modulů, seznam registrovaných procesů a seznam aplikací potřebných pro samotný běh aplikace.:

```
{application, projekt,
 [{description, "UTB Projekt"},
 {vsn, "1.0"},
 {modules, [projekt_manager, report, logger, projekt_sup,
projekt_driver,
 projekt_client, http_client, projekt_utils]},
```

```
 {registered, [report, logger, projekt_driver]},
 {applications, [ssl, inets, kernel, stdlib]},
 {mod, {projekt_app, []}}
 ]
}.
```

1.4 Fungování aplikace

Aplikace se řídí z příkazové řádky v prostředí Erlangu. Před samotným testováním, je nutné znát adresy uzlů. Tyto uzly musí splňovat následující požadavky

Jednotná distribuce Linuxu

Všechny uzly musí být instalovány ze stejné distribuce Linuxu, aby byly všechny balíčky stejné verze. To platí zejména pro balíčky Erlangu

SSH přístup

Na všechny uzly musí být SSH (Secure Shell) přístup a to pod stejným uživatelským účtem. Tzn. že je nutno na všech počítačích založit uživatelský účet např. *utb*. Na tento účet je dobré rozdistribuovat svůj SSH klíč, aby nebylo nutné pokaždé zadávat heslo. Erlang právě přistupuje k ostatním počítačům přes SSH protokol.

Prostředí Erlangu

Je nutné, aby všechny počítače měli nainstalované balíčky jazyka Erlang ve stejné verzi.

Jmenný překlad

Aby na sebe všechny uzly viděli, je nutné v souboru `/etc/hosts` každého počítače jména a IP (Internet Protocol) adresy okolních počítačů. Pokud je v dané síti překlad adres řešen DNS serverem, pak je o starost méně.

projekt.conf

Tento konfigurační soubor obsahuje ve složených závorkách direktivu *nodes* a v hranatých závorkách je potřeba uvést seznam pomocných uzlů. Další direktiva je *delay*, která udává maximální prodlevu mezi jednotlivými dotazy v milisekundách. Direktiva *client_type* určuje jaká služba se bude testovat, tj. HTTP, SMTP nebo pgsql. A poslední direktiva ve složených závorkách *duration* udává v sekundách dobu, po kterou bude aplikace generovat požadavky.

http_client.conf, smtp_client.conf, postgresql.conf

Tyto soubory obsahují konfigurační parametry pro testovanou službu. Více podrobností o konfiguracích jednotlivých služeb budou uvedeny v následujících kapitolách.

Zajímavou vlastností Erlangu je také to, že každou instanci běžícího prostředí si lze pojmenovat. Takže když spustíme Erlang s parametrem `-sname master`, lze se na něj odkazovat v síti jako na *master@localhost*. Aby byla správa pomocných uzlů co nejjednodušší umí aplikace automaticky přidělovat názvy uzlům podle písmen abecedy (např. *c@node1*).

Další zajímavostí je, že není nutno řešit distribuci zdrojového kódu aplikace standartní cestou, tj. rozkopírování aplikace na každý uzel zvlášť. To není v případě Erlangu vůbec potřeba. Erlang má v sobě zabudovaný tzv. *code server*, který dokáže na ostatní připojení Erlangovské počítače rozeslat požadované moduly nebo s trochou námahy i celé aplikace. Odpadají nám tak další administrativní starosti.

Pro spojení všech uzlů na kterých poběží Erlang, je potřeba, aby tyto uzly byly spuštěny s parametrem *setcookie*, který definuje jakoby heslo pro komunikaci. Heslo bylo nastaveno na slovo *tea*, takže výsledný parametr bude vypadat takhle: `-setcookie tea`.

Aplikaci (tj. Erlang) lze z příkazové řádky spustit následujícím příkazem.:

```
erl -sname emaster -rsh ssh -pa ebin -setcookie tea
 -boot start_sasl -config config/projekt
 -run projekt_utils start_projekt -run projekt_utils try_connect_slaves
```

Parametr `-pa ebin` popisuje cestu ke zkompilovaným zdrojovým kódům aplikace, což bývá standartně adresář *ebin*. Parametry `-boot start_sasl` a `-config config/projekt` mají na starosti spuštění interního logovacího systému. Další parametry `-run projekt_utils start_projekt` a `-run projekt_utils try_connect_slaves` spouští danou funkci v uvedeném modulu *projekt_utils*.

Modul *projekt_utils.erl* obsahuje pomocné funkce a nástroje, které se nehodí mít v ostatních modulech. Již zmiňovaná funkce z předchozího odstavce *start_projekt*, nastartuje aplikace nutné pro běh mého projektu jako jsou *inets* a *ssl*. Nakonec funkce odstartuje můj projekt příkazem `application:start(projekt).`

Poté dojde ke spuštění supervizora, který se postará o načtení konfiguračních parametrů

aplikace spolu se seznamem HTTP adres, které budou doslova bombardovat *GET* requesty. Dále dojde ke spuštění loggeru, reporteru a manageru.

V tento moment je aplikace spuštěná a teď přijde na řadu napojení na pomocné uzly. Ke spojení s pomocným uzlem pomůže modul *slave*, který je standartní součástí knihoven Erlangu. Příkazem `slave:start(Node, Name, Cookie)` lze na vzdáleném uzlu *Node* vytvořit pod jménem *Name* běžící instanci Erlangu, se kterou lze komunikovat Proměnná *Cookie* musí obsahovat slovo *tea*, které bylo zvoleno na začátku.

Tímto způsobem se spojí se všemi ostatními uzly, které byly uvedeny v konfiguračním souboru `projekt.conf` v direktivě nodes. Dále bude potřeba na tyto uzly rozdistribuovat moduly, které jsou potřeba pro běh aplikace.

Předtím, než přijde na ukázku kódu, který zajišťuje nahraní libovolného modulu na vzdálený server, bude pojednáno o funkcionalitě vzdáleného volání funkce, neboli RPC (Remote procedure call). V podání Erlangu může vypadat vzdálené volání funkce tak, že lze použít standartní zaslání zprávy např. takhle `Pid ! {self(), cleanup}` a výsledek lze odchytit zase tímto příkazem: `receive Msg after 10000 -> timeout end.`. Nebo lze využít modul *rpc* z knihovny Erlangu. V projektu byla dána přednost knihovní funkci `rpc:call`, která zjednodušuje tento proces komunikace.

Takže za pomoci modulu rpc bude funkcionalita pro distribuci modulu na libovolný uzel vypadat následovně.:

```
upload_module(Node, Module) ->
 {_Module, Binary, Filename} = code:get_object_code(Module),
 rpc:call(Node, code, load_binary, [Module, Filename, Binary]).
```

Proměnné *Node* a *Module* jsou jasné. Druhý řádek využívá standartního modulu *code*, který poskytuje na základě názvu modulu jeho binární obraz, který lze dále přenášet. O následný transport se zaslouží funkce *call* již zmiňovaného modulu *rpc*. Funkce *call* zavolá na vzdáleném uzlu *Node* funkci *load_binary* z modulu *code*, která nahraje binární obraz modulu *Module* pod jménem *Filename* do paměti.

Moduly jsou rozdistribuované na pomocné uzly a zbývá je uvést v činnost. Pro začátek je potřeba spustit na pomocných uzlech *logger* a *reporter*. Takže pomocí klasické funkce *spawn* budou spuštěny tyto procesy a to následovně.:

```
Pid = spawn(Node, ModuleServer, start, [{global, As}])
```

Proměnná *Node* je název uzlu, na kterém se spouští daný proces, *ModuleServer* je název

modulu a *start* je klasická funkce, které jsou předány parametry *{global, As}*. Funkce *spawn* vrací číslo procesu.

1.4.1 Vyhodnocení výsledků

V předchozím textu bylo vysvětleno, jak aplikace funguje. Procesy *logger* a *reporter* mají za úkol zaznamenávat chybové a testovací zprávy od ostatních procesů, které vysílají požadavky. V průběhu testování se záznamy postupně ukládají do adresáře *logs* ve následujícím tvaru.:

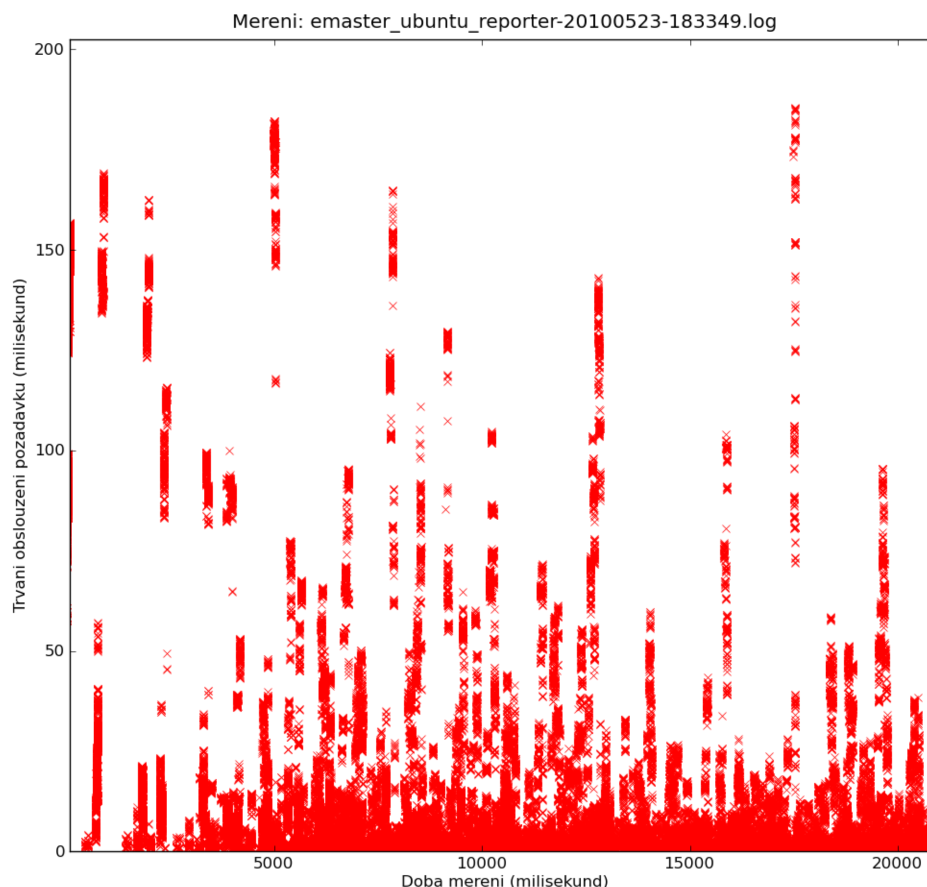
```
emaster_bubliks_reporter-20100522-201326.log
```

Název se automaticky vytvoří z názvu našeho řídícího uzlu tj. *emaster*, který běží na počítači s názvem *bubliks*. Dále se v názvu rozlišuje, zda se jedná o záznamy reportu nebo loggeru a nakonec se připojí časové a datumové razítko, kdy došlo ke spuštění aplikace. Ale pouze jen v případě záznamů reporteru se vytváří pro každé testování nový soubor. Pro záznamy loggeru to nemá moc smysl. V případě, že se bude distribuovat zátěž na více počítačů, pak se pro každý takový pomocný uzel vytvoří na řídícím počítači vlastní logovací a reportovací soubor.

K vyhodnocení výsledků měření byla zvolena knihovna *matplotlib*[7], která je napsána v jazyce Python a jedná se vlastně o silný nástroj pro vytváření různých dvourozměrných grafů [13]. Pro naše účely je plně vyhovuje, i když stále neumí vytvářet popisky v jiném kódování než ASCII (American Standard Code for Information Interchange).

Pro vyhodnocení byl zvolen obyčejný graf, kde na osu *x* vynese čas, po kterou probíhalo měření v jednotkách milisekund a na osu *y* zas vynese doba, kterou strávil proces vykonáním jednoho požadavku, který byl uspokojen, taktéž v milisekundách. Graf může vypadat následovně.

7 http://matplotlib.sourceforge.net/



Obr. 4. Příklad grafu, který zobrazuje výsledky měření

Pro snadnou obsluhu byl tedy vytvořen skript *gen_graphs.py*, který se nalézá v adresáři *logs* a který přijímá jako parametr reportovací soubor. Skript tento soubor analyzuje, tj. spočítá trvání každého požadavku, a výsledné časy vynese do grafu. Skript současně vyhodnotí maximální, minimální a průměrný čas, který byl stráven požadavkem. Kromě toho jsem přidal i medián, abychom měli lepší představu o výkonu aplikace. Protože se běžně stává, že extrémní hodnoty bývají na počátku měření nebo na konci, takže medián nám v tomto dost poslouží a eliminuje tyto extrémní časy.

Funkce pro výpočet mediánu je následující.:

```
def getMedian(numericValues):
 sortedValues = sorted(numericValues)
 if len(sortedValues) % 2 == 1:
 return sortedValues[(len(sortedValues)+1)/2-1]
 else:
 lower = sortedValues[len(sortedValues)/2-1]
 upper = sortedValues[len(sortedValues)/2]
 return (float(lower + upper)) / 2
```

Dále skript těchto pět hodnot vypíše a vytvoří graf, na který vykreslí čas každého dotazu. Na ose x je vynesen čas měření, neboli doba trvání testu v milisekundách. Na ose y je zase vynesen čas trvání obsloužení požadavku, jež je také v milisekundách.

II. PRAKTICKÁ ČÁST

2 TESTOVÁNÍ VÝKONNOSTI POMOCÍ ERLANGU

Na začátku je potřeba zdůraznit, že tu nejde o porovnávání výkonnosti jednotlivých síťových aplikací, ale o představu jak se aplikace chová, když je vystavena zátěži. Každý webový server se chová jinak a záleží zejména na tom, jak se zkonfiguruje. V praxi je často potřeba vyladit aplikaci tak, aby uspokojila aktuální potřeby. Což zejména u webových serverů může znamenat hodně věcí. Tj. např. je požadavek, aby server udržoval trvalá spojení (HTTP Keepalive [4]), která se využívají pro interaktivní webového aplikace jako např. Gmail. Nebo aby byl schopen poskytovat statický obsah, kde trvalá spojení jsou zbytečná.

2.1 Příprava

Obsluha všech pomocných uzlů by měla být rychlá a nenáročná. Operační systém Linux je pro tuto situaci nejlepší volbou, protože jej lze snadno získat a přizpůsobit požadavkům. Linux vyniká zvláště ve výkonu v síťovém provozu. Pro účely testování byla vytvořena na paměťovém USB (Universal Serial Bus) disku spustitelná verze Linuxu Ubuntu verze 10.04[8], která bude potřeba, protože je k dispozici pouze počítač, na kterém je nainstalovaný operační systém Windows XP a jiný operační systém by se na jeho disk už nevešel. Z tohoto disku bude spuštěn operační systém aniž by byl narušen interní disk počítače.

2.1.1 Konfigurace hardware

Pro testování je k dispozici jeden počítač, který bude brán jako server a jeho konfigurace je následující.

- dvou jádrový procesor Core2Duo s frekvencí 2.5 GHz
- 2514 MiB paměti
- 100 Mbps integrovaná síťová karta
- grafická karta Nvidia GeForce 9400
- pevný disk Seagate 80 GB

Na počítači je nainstalován Linux Ubuntu také ve verzi 10.04. Na něm bude provozována síťové aplikace, které bude testována.

Další počítač, který bude sloužit jako řídící pracoviště má následující konfiguraci.

- dvou jádrový procesor Core2Duo s frekvencí 2.1 GHz
- 2007 MiB paměti

8 http://www.ubuntu.com

- 100 Mbps integrovaná síťová karta
- SATA disk 250 GB
- ATI/AMD Mobility Radeon 4300

Na něm bude provozován Linux z USB disku, kde bude i samotná aplikace spouštěna.

Poslední počítač je postarší MacBook Pro s operační systémem Mac OS X Leopard verze 10.6.3. Bude sloužit jako pomocí uzel, na kterém je nainstalováno prostředí jazyka Erlang. Jeho konfigurace je následující.

- dvou jádrový procesor Core2Duo 2.2 GHz
- 2 GB paměti
- 1000 Mbps integrovaná síťová karta
- SATA disk 120 GB
- Nvidia GeForce 8600 GT

2.1.2 Konfigurace software

Takže hardware byl představen a zbývá nastavit správně síť na všech počítačích. Jak bylo popisováno v kapitole *Fungování aplikace*, je potřeba se ujistit, že:

- všechny počítače používají stejnou verzi Erlangu, jinak se mohou vyskytnout potíže
- všechny počítače mají uvedeny názvy ostatních počítačů v souboru `/etc/hosts`, tj. aby šlo jménem odkazovat na ostatní počítače
- existuje přístup ke všem pomocným počítačům přes SSH protokol pod stejným uživatelský jménem, např. účet *bublik*
- jsou pro pohodlnější správu pomocných uzlů rozdistribuované veřejné klíče, aby nebylo zadávat heslo pro každé připojení uzlu
- jsou správně vyplněny konfigurační parametry testované služby, viz. soubory `http_client.conf`, `smtp_client.conf`, `postgresql.conf` nebo v kapitole o fungování aplikace
- jsou v souboru `projekt.conf` definovány pomocné uzly a další parametry jako doba testování a doba zpoždění za každým následujícím požadavkem daného procesu a typ služby, která se má testovat

Testovat se budou následující síťové aplikace, které poskytují HTTP rozhraní.

- mochiweb
- tornado

- lighttpd
- Apache

Dále se bude testovat SMTP server Postfix a databázový server PostgreSQL

2.2 Testování HTTP serverů

2.2.1 Mochiweb

Jako první na řadě je http server *mochiweb*, který je vyvíjen v Erlangu. Pro účely testování byl vytvořena následující HTML stránka, která se klientským procesům ukáže když vyšlou požadavek.:

```
<html>
 <head>
 <title>It Worked</title>
 </head>
 <body>
 Hello World.
 </body>
 </html>

 .. [9] http://code.google.com/p/mochiweb/
```

Je potřeba upozornit také na fakt, že síťové aplikace potřebují zvláštní nastavení, tj. je potřeba jim přidělit potřebné zdroje. Co se týče síťových aplikací a obecně i procesů, pak je potřeba navýšit počet otevřených souborových deskriptorů, aby tak nebyli omezováni. V tomto případě bylo nastaveno pomocí příkazu `ulimit -Hn 200000` a `ulimit -Sn 200000` na maximální počet 200000 otevřených deskriptorů.

Zbývá ještě nastavit testovací parametry http klienta, tj. `{http_hosts, ["http://192.168.1.101:8000"]}.` a hlavně parametry aplikace, které jsou následující:

```
% test duration in seconds
{duration, 20}.
% max delay between each request of a process in milliseconds
{delay, 1000}.
% list of slave nodes
{nodes, ['ubuntu']}.
% type of the client: http, smtp, pgsql (postgresql)
{client_type, http}.
```

Teď je potřeba spustit samotný test příkazem `start_dist.sh`, což způsobí to, že se aplikace pokusí spojit s definovanými pomocnými uzly. Zobrazí se na příkazová řádka Erlangu, kde si lze příkazem `nodes()` ověřit zda došlo ke spojeni s ostatními počítači. Pokud ne, pak tato funkce vrátí prázdný seznam. Předtím, než dojde k vytváření samotných procesů je nutno na tyto vzdálené počítače přenést potřebné moduly. Lze to zajistit

příkazem `projekt_utils:prepare_node(a@ubuntu)`, kde *a@ubuntu* je název uzlu, který vrátil příkaz `nodes()`.

Dále je potřeba zvolit potřebný počet procesů, které budou testovat serverovou službu a to příkazem `projekt_utils:create_clients(900)`, který vytvořil na řídícím počítači 900 čekajících procesů. Dále je potřeba na ostatních pomocných uzlech udělat to samé, ale pomocí příkazu `projekt_utils:create_clients_on(a@ubuntu, 100)` z řídícího počítače. Tento počet, dohromady 1000 klientů na obou uzlech, byl zjištěn postupným navyšováním zátěže a je to zhruba maximum, co dokáže server obsloužit.

Teď už nic nebrání v tom, aby se test spustil příkazem `projekt_utils:run()`. Poté si lze zkontrolovat logovací záznamy a díky skriptu *gen_graphs.py* lze z nich vygenerovat následující grafy.

Na obrázku 5 není sice graf moc vypovídající, ale to je pouze tím, že program *matplotlib* zobrazil všechny časy. Takže křížek na horním grafu úplně nahoře je prakticky ten nejpomalejší dotaz vedený od řídícího počítače, kde si dal mochiweb dost načas. Detailnější pohled poskytuje dolní graf, kde bylo zvětšeno rozlišení hodnot osy *y*. Skript také vypsal maximální, minimální časy a také počet dotazů, které byly úspěšně obslouženy.

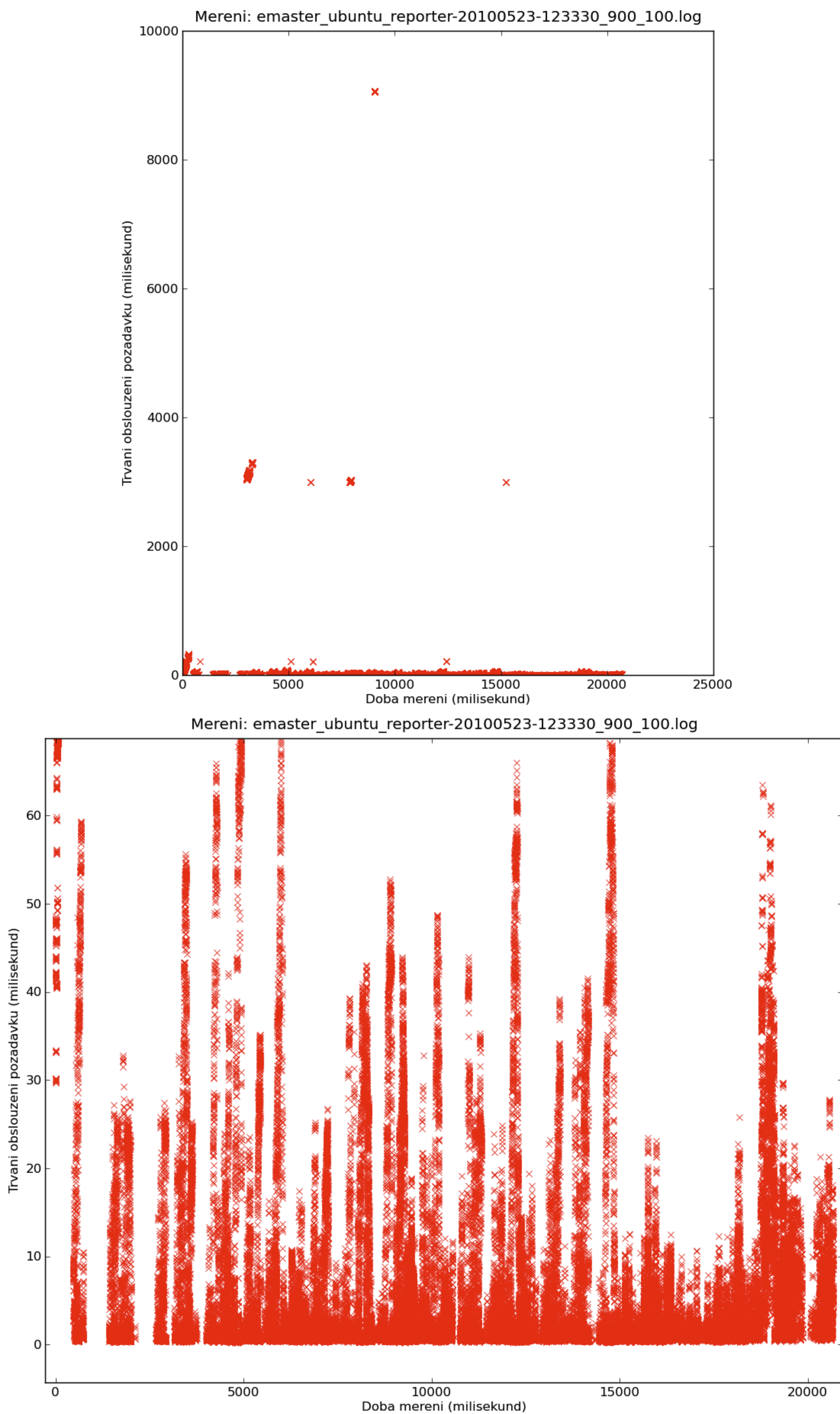
Výsledky měření při zatížení 900 procesy z řídícího počítače jsou::

```
Odezva (milisekundy).
_________________________
Počet obsloužených dotazů: 92595.
Max: 9071.781000.
Min: 0.232000.
Průměr: 69.606100.
Medián: 1.443000.
```

Výkonnosti severu mochiweb dokazuje i to, že zvládl obsloužit 92595 dotazů za 20 sekund. To je dost slušný výkon na to, že je server mochiweb napsat ve vyšším programovacích jazyce než je jazyk C. Ve většině případů právě implementace v jazyku C bývá v těchto případech rychlejší alternativou.

Horní graf na obrázku 6 ukazuje jak server mochiweb obsluhoval požadavky generované z pomocného počítače a spodní graf zobrazuje větší detail při ose x.

Výsledky měření při zatížení 100 procesy z pomocného počítače jsou:



Obr. 5. Vytížení serveru mochiweb z řídícího počítače 900 procesy

```
Odezva (milisekundy).
_______________________
Počet obsloužených dotazů: 6774.
Max: 1050.924000.
Min: 0.302000.
Průměr: 39.050419.
Medián: 1.087000.
```

Mochiweb je určitě nejvýkonnější z testované sestavy, ale jeho konfigurační možnosti se nemohou serveru Apache vyrovnat. Je určen spíše pro interaktivní aplikace než na poskytování statického obsahu a neposkytuje rozhraní pro aplikace v PHP natož v Pythonu. Tudíž se hodí speciálně pro řešení v Erlangu.

Nasadit tento server lze jako základ pro herní internetové aplikace nebo komunikační a sociální weby. Tento povedený kousek softwaru vyvíjí společnost Mochimedia[9], která se specializuje právě na internetový zábavní průmysl a poskytla jej volně k používání i se zdrojovými kódy. Další oblastí, kde se dá použít, jsou inzertní servery, které potřebují velmi krátkou reakční dobu pro vyplnění inzertních ploch webových stránek.

Mochiweb se určitě nehodí pro poskytování statického obsahu, kde spíše dominuje řešení v podobě serveru nginx[10].

2.2.2 Apache

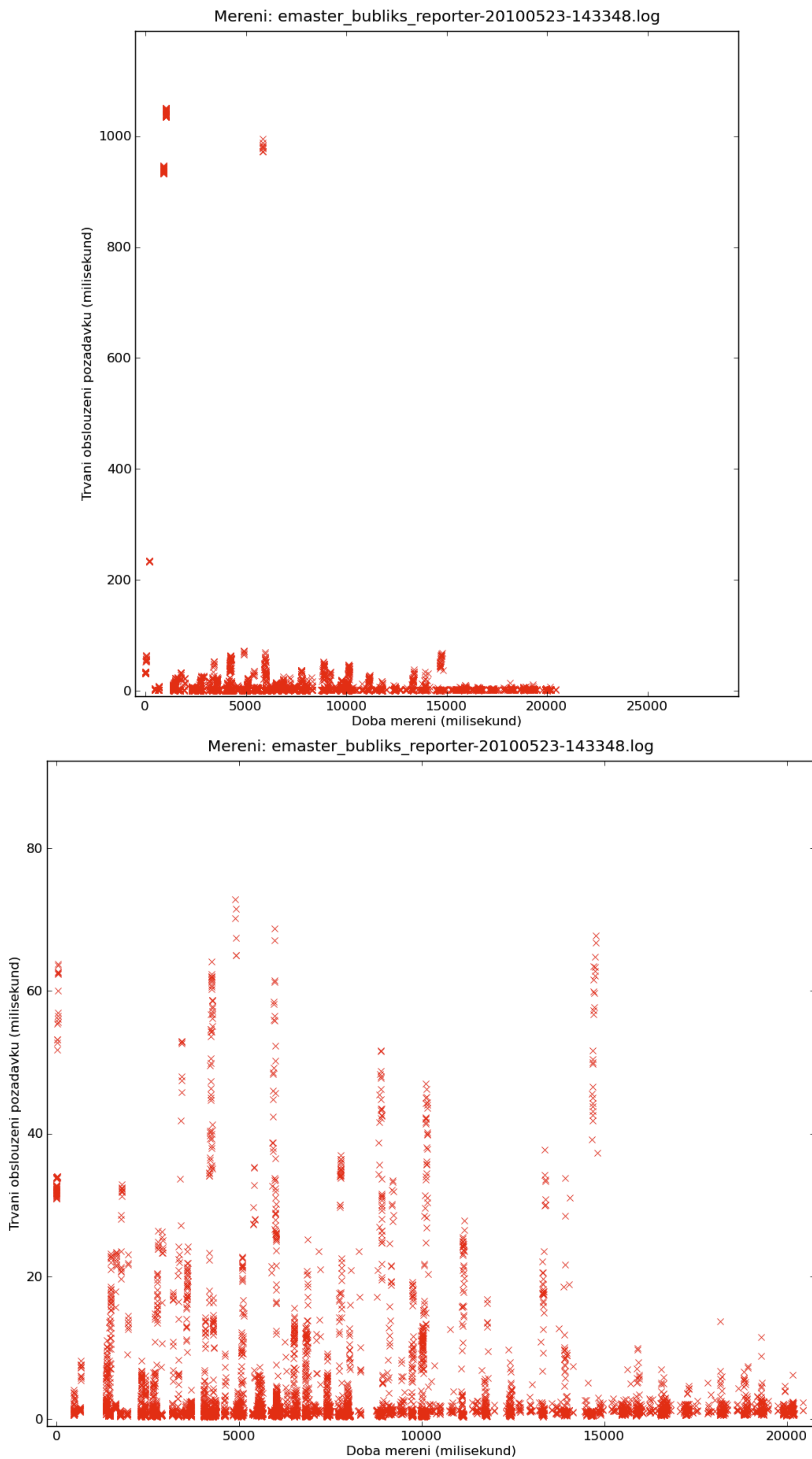
Testování webového serveru Apache probíhalo stejným způsobem jako v případě serveru mochiweb a ani v ostatních případech tomu nebude jinak. Na grafech lze vidět, že se Apache moc nezadýchal, ale to je také tím, jakou má nastavenou konfiguraci. Určitě by šlo z něj vytěžit více, kdyby se vyladila jeho konfigurace.

Apache byl spuštěn s touto konfigurací.:

```
# worker MPM

<IfModule mpm_worker_module>
 StartServers 2
 MinSpareThreads 25
 MaxSpareThreads 75
 ThreadLimit 64
 ThreadsPerChild 25
 MaxClients 150
 MaxRequestsPerChild 0
</IfModule>
```

9 http://www.mochimedia.com
10 http://nginx.org/



Obr. 6. Vytížení serveru mochiweb z pomocného počítače 100 procesy

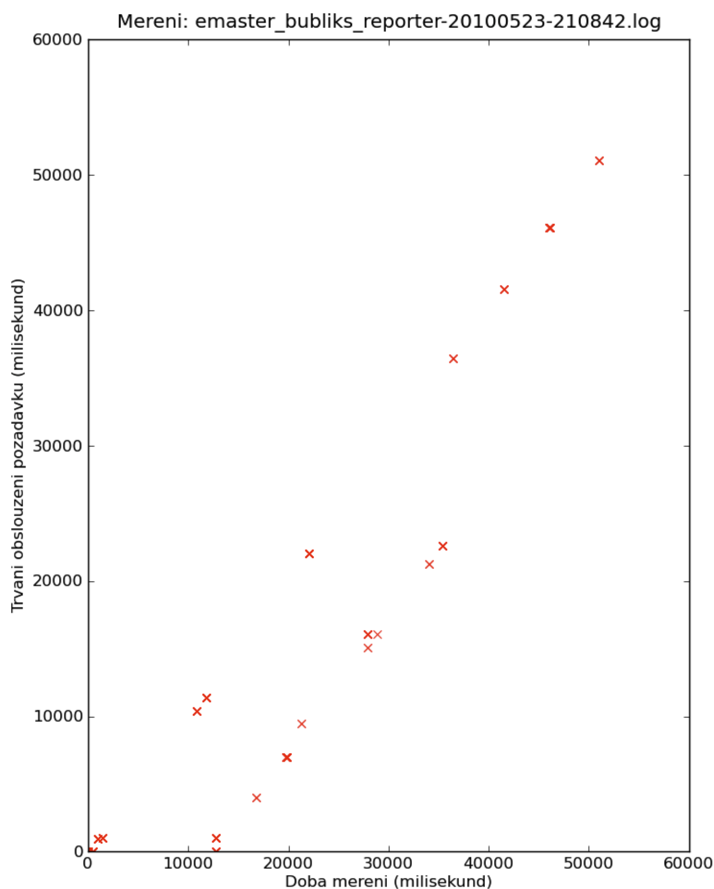
Výsledky měření při zatížení 900 procesy z řídícího počítače.:

```
Odezva (milisekundy).
________________________
Počet obsloužených dotazů: 1446.
Max: 52524.568000.
Min: 0.574000.
Průměr: 6895.330270.
Medián: 123.722500.
```

Výsledky měření při zatížení 100 procesy z pomocného počítače.:

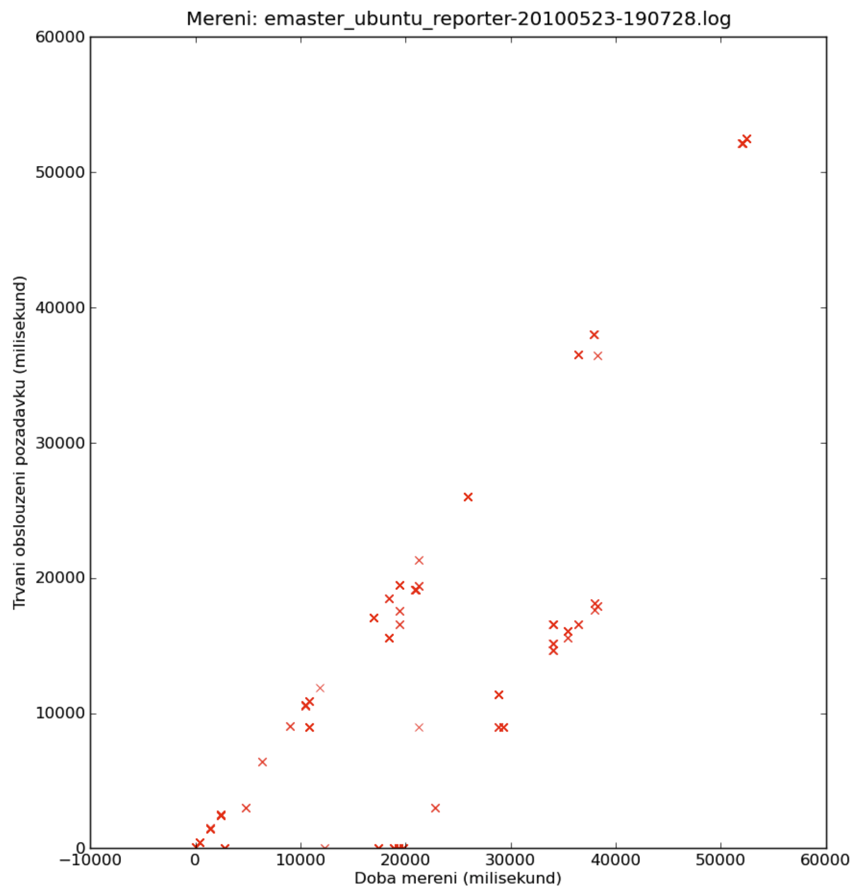
```
Odezva (milisekundy).
________________________
Počet obsloužených dotazů: 498.
Max: 51059.428000.
Min: 0.647000.
Průměr: 7339.251540.
Medián: 958.944500.
```

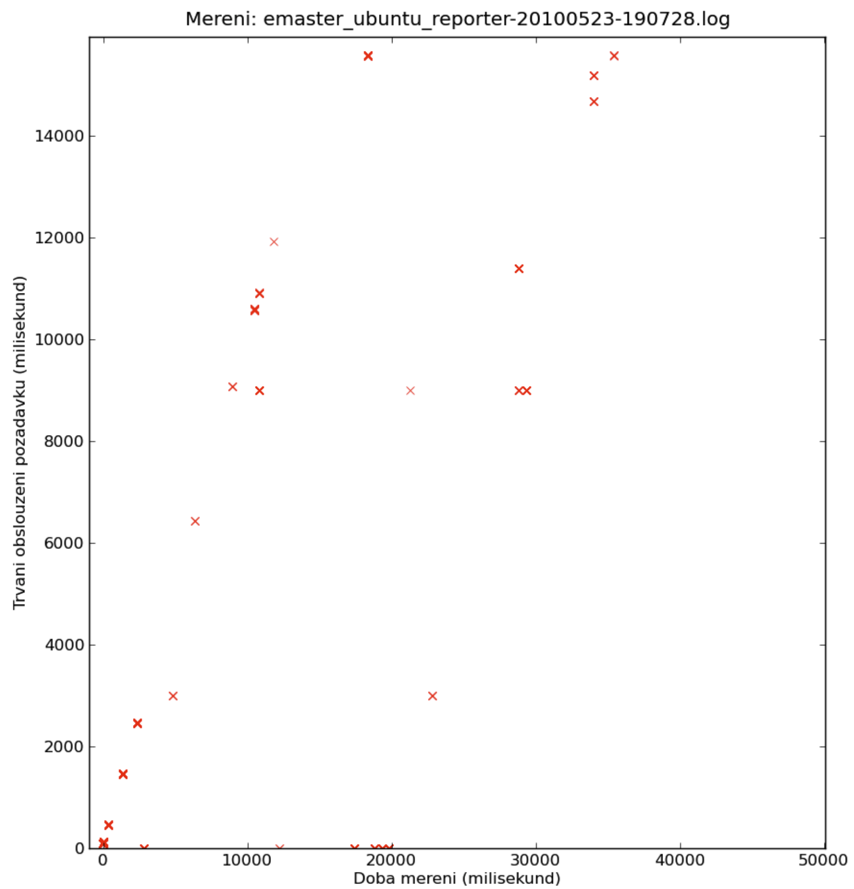
Tento server je často nasazován pro weby a vyniká zejména řízenou správou výkonu a silnou podporou modulů. Z výsledků nelze usuzovat na to, že se jedná o nevýkonný server, protože díky široké škále nastavení jej lze uzpůsobit aktuálním požadavkům.



Obr. 7. Vytížení serveru Apache z pomocného počítače

100 procesy





Obr. 8. Vytížení serveru Apache z řídícího počítače 900 procesy

2.2.3 Lighttpd

Lighttpd se dnes hojně využívá, protože má malé systémové nároky a přitom vyšší výkon než Apache. Nasadit do ostrého provozu se jej odvážili takové společnosti jako Wikipedia[11] nebo YouTube[12]. A ani s konfiguračními schopnostmi nezaostává.

Instalace byla provedena z balíku z distribuce a do konfigurace nebylo zasahováno. Došlo pouze k nastavení stejné úvodní stránky jako u ostatních serverů.

Výsledky měření při zatížení 900 procesy z řídícího počítače.:

```
Odezva (milisekundy).
________________________
Počet obsloužených dotazů: 6030.
Max: 15733.242000.
Min: 0.359000.
Průměr: 378.286816.
Medián: 16.030500.
```

Výsledky měření při zatížení 100 procesy z pomocného počítače.:

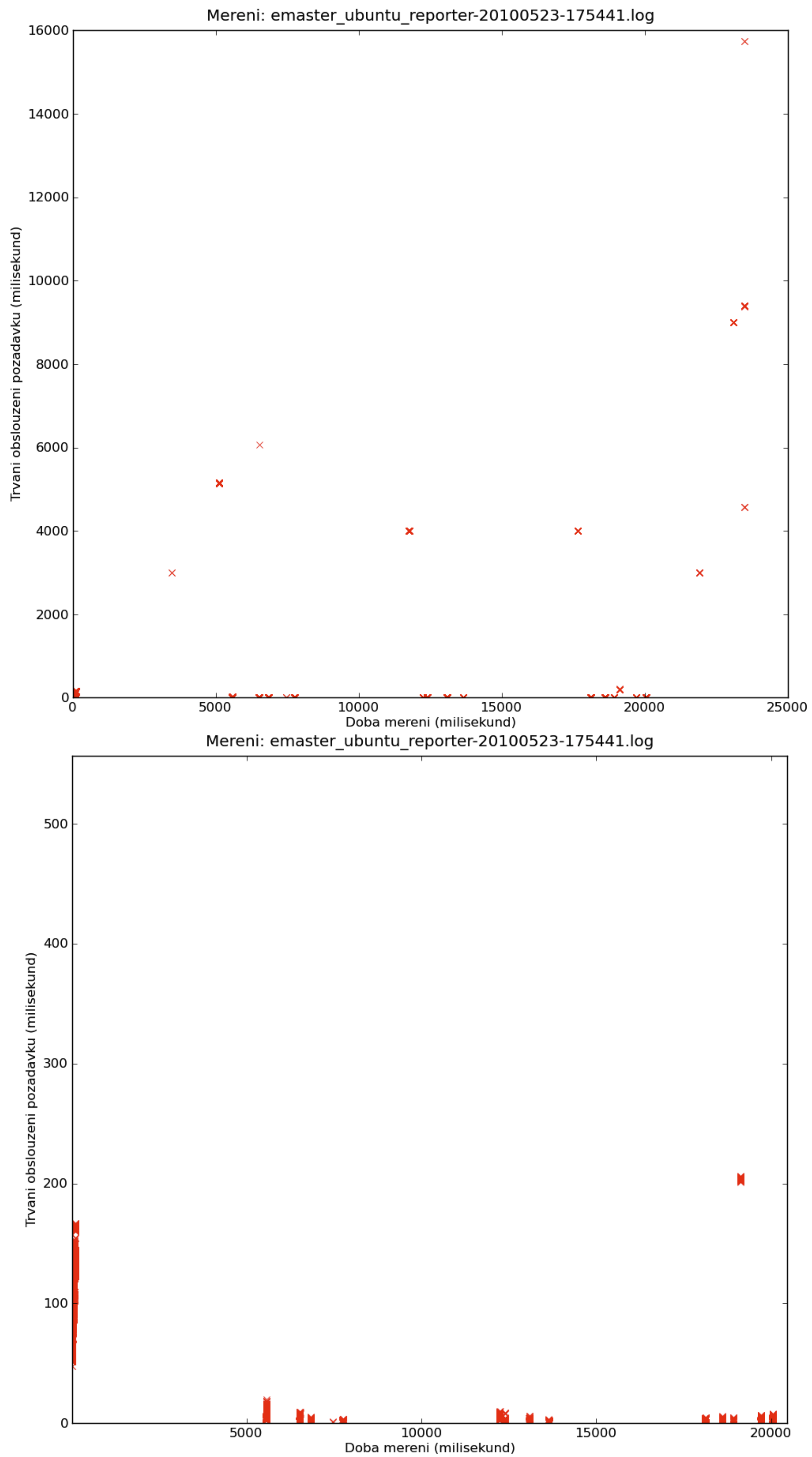
```
Odezva (milisekundy).
________________________
Počet obsloužených dotazů: 4098.
Max: 6473.794000.
Min: 0.315000.
Průměr: 713.343028.
Medián: 3.481000.
```

Stejně jako Apache Lighttpd nedosahuje v základní konfiguraci vysokého výkonu, ale to je způsobeno konfigurací a přidělenými zdroji. Tak jako u předešlých případů lze v záznamech vyčíst, že mnoho požadavků bylo odmítnuto a stejně tak i mnohým vypršel čas připojení (timeout).

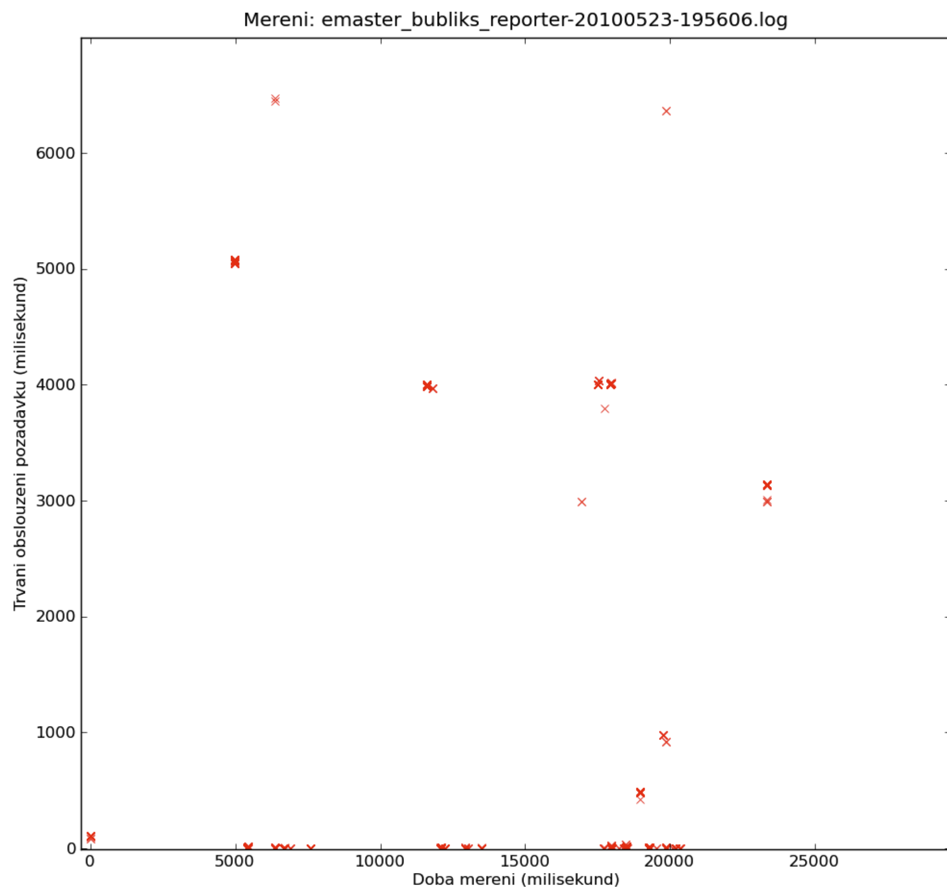
2.2.4 Tornado

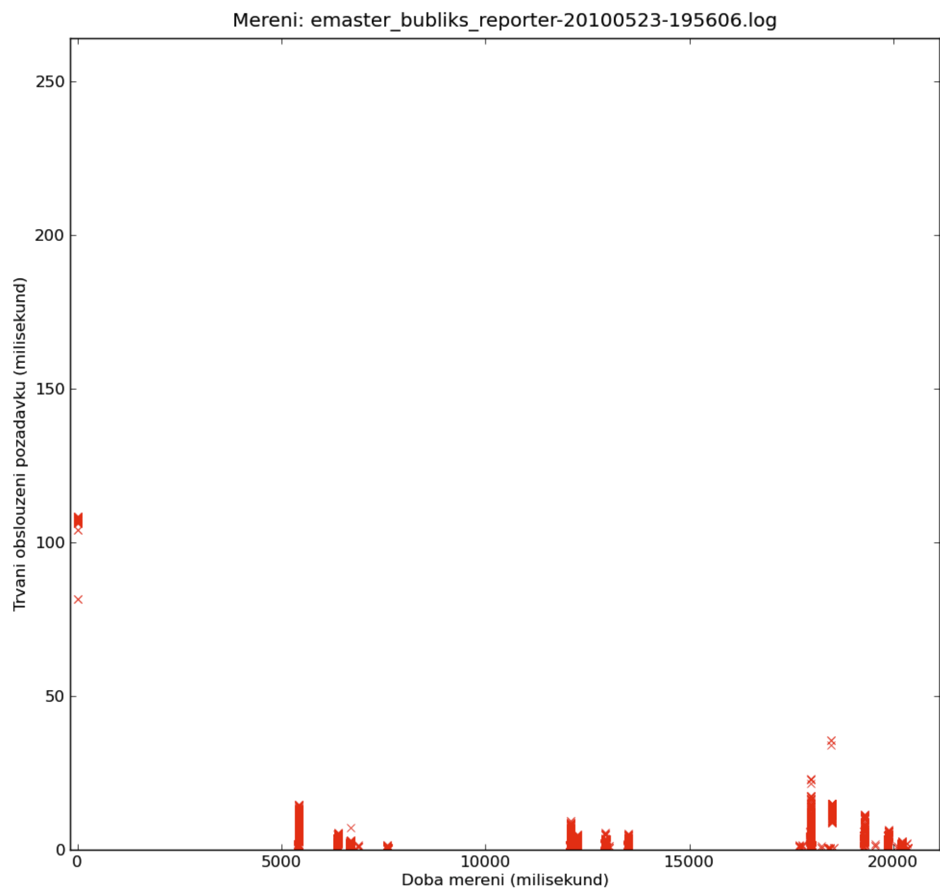
Webových server Tornado je sympatický kousek, který je napsán v jazyce Python a překvapuje svým vysokým výkonem. Je určen zejména pro interaktivní webové aplikace než pro statický obsah a naměřené hodnoty v grafech jsou vypovídající o jeho schopnostech. Využívá jej sociální internetová služba FriendFeed[13], která také stojí za jejím vývojem. Nedílnou součástí tohoto softwaru je i jednoduché databázové rozhraní, které usnadňuje připojení k databází MySQL a nechybí mu ani vlastní šablonovací nástroj a v neposlední řadě i lokalizační podporu webových aplikací.

11 http://www.wikipedia.org
12 http://www.youtube.com
13 http://www.friendfeed.com



Obr. 9. Vytížení serveru lighttpd z řídícího počítače 900 procesy





Obr. 10. Vytížení serveru lighttpd z pomocného počítače 100 procesy

Výsledky měření při zatížení 900 procesy z řídícího počítače.:

```
Odezva (milisekundy).
_______________________
Počet obsloužených dotazů: 83103.
Max: 9074.146000.
Min: 0.264000.
Průměr: 135.860143.
Medián: 3.993000.
```

Výsledky měření při zatížení 100 procesy z pomocného počítače.:

```
Odezva (milisekundy).
_______________________
Počet obsloužených dotazů: 6780.
Max: 981.497000.
Min: 0.308000.
Průměr: 24.937385.
Medián: 1.806500.
```

Jak lze vidět na grafech, tak požadavky jsou obsluhovány velmi rychle. Jak sami autoři píší na stránkách, tak je lepší jej nasadit spolu s programem nginx. Postrádá množství konfiguračních voleb, ale to už vyplívá z toho, že je určen vyloženě pro aplikace psané v Pythonu.

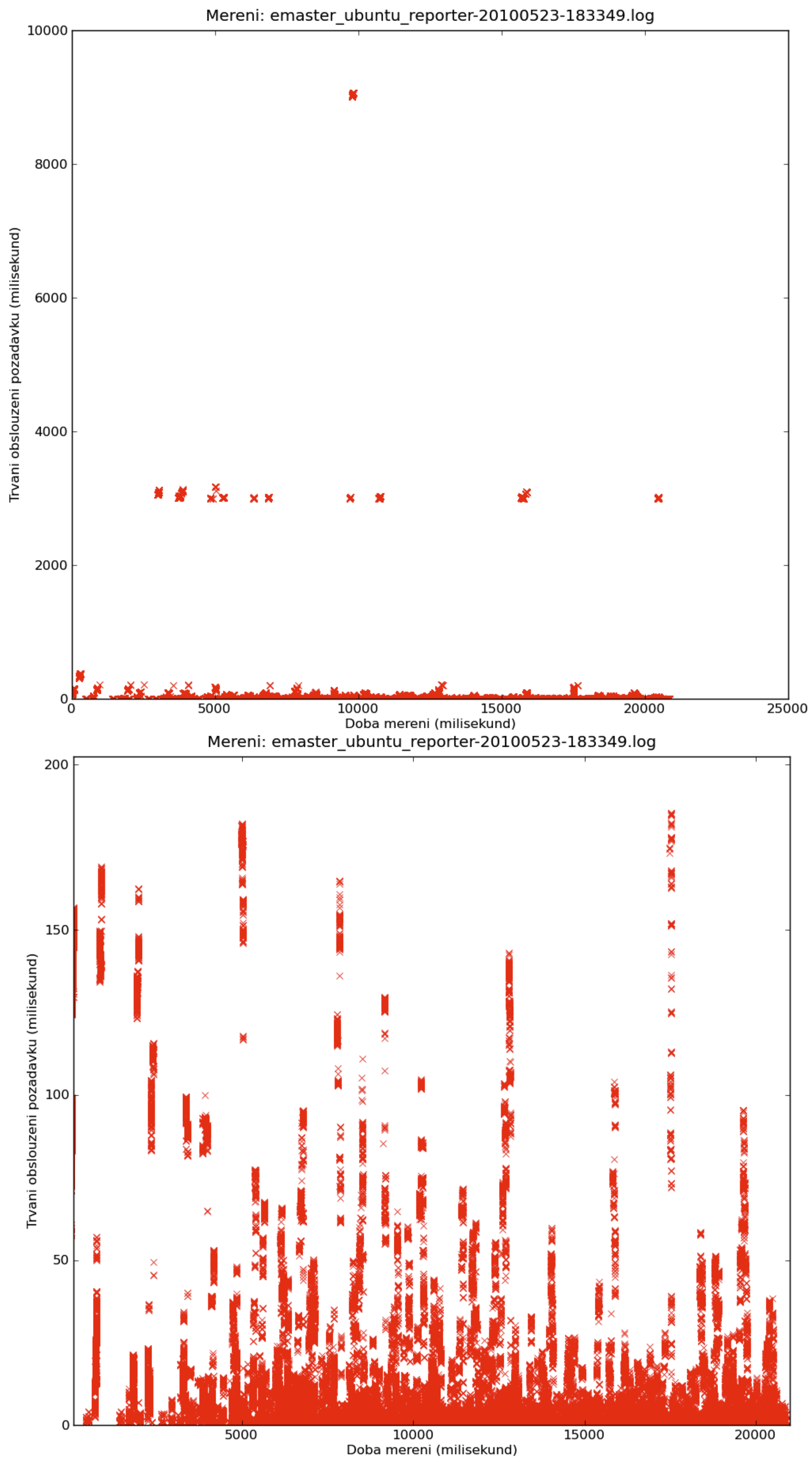
2.3 Testování SMTP serveru Postfix

Stejně jako v předešlých testováních byl program spuštěn na těch samých sestavách, jediné co je bylo potřeba nastavit, bylo v konfiguračním souboru `projekt.conf` parametr *client_type* na hodnotu *smtp* a v souboru `smtp_client.conf` následující hodnoty.:

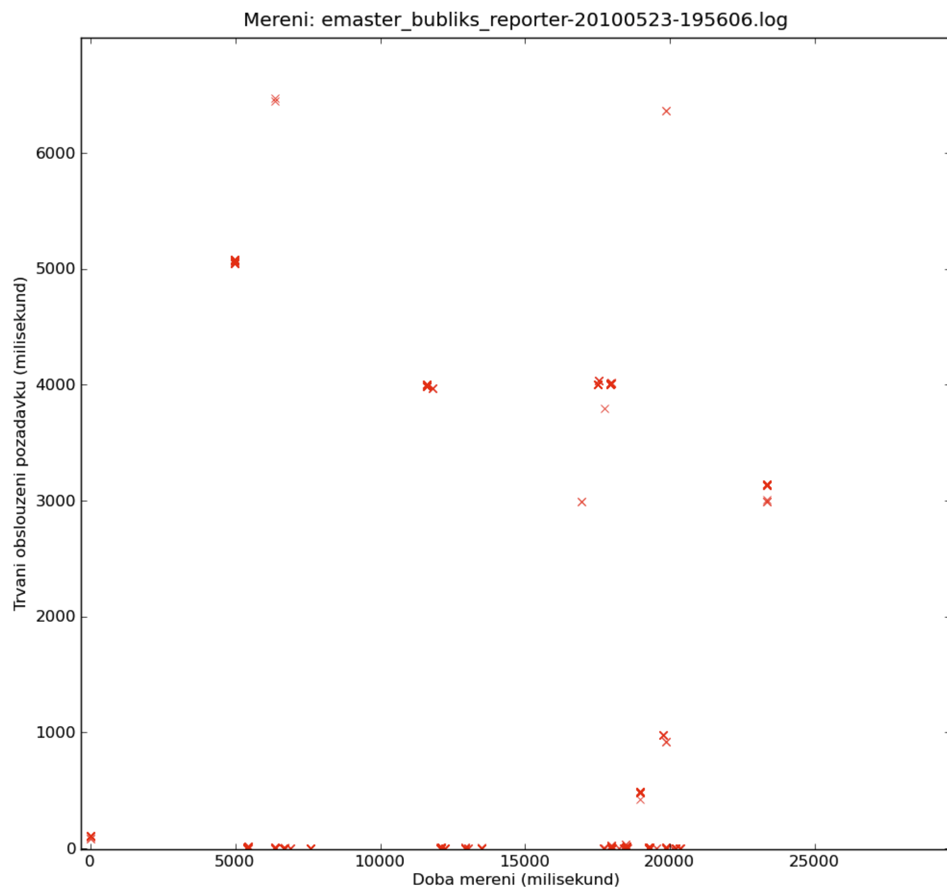
```
{smtp_server, "alex"}.
{smtp_login, "ales.bublik@gmail.com"}.
{smtp_passwd, "heslo"}.
{smtp_from, "<ales.bublik@gmail.com>"}.
{smtp_to_mail_addresses, ["ales@alex"]}.
{smtp_subject, "pokus"}.
{smtp_msg, "Pozor! Tohle je testovaci zprava"}.
```

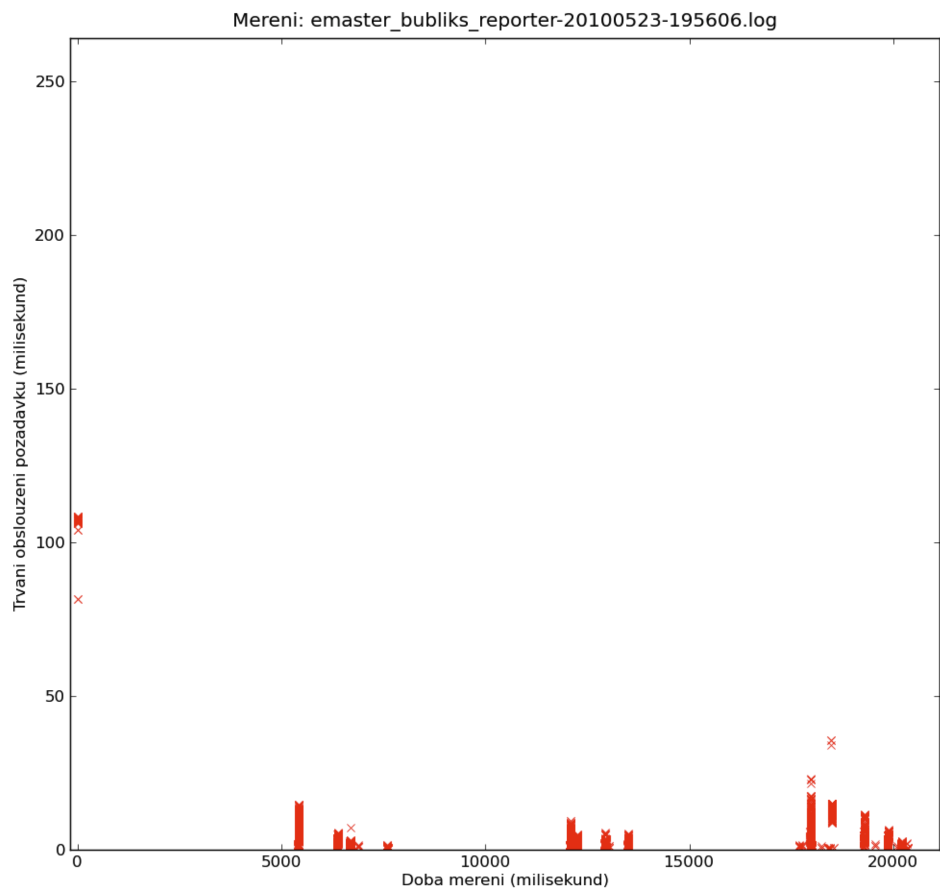
Tady je potřeba říct, že mailové servery jako služba jsou určené pro odesílání zpráv příjemci, který ji ale nemusí obdržet okamžitě jakmile ji odesílatel odeslal. Takové doručení zprávy může být pozdrženo až na několik hodin aniž by o tom odesílatel rozhodoval.

Zpráva může být jak na přijímacím SMTP serveru, tak u odesílacího SMTP serveru kontrolována, zda neobsahuje viry nebo zda se nejedná o nevyžádanou poštu. Navíc správci takových serverů mohou nastavit pozdržení přijímaní pošty na několik hodin aniž by tak ohrozili přijetí důležitého mailu.



Obr. 11. Vytížení serveru tornado z řídícího počítače 900 procesy





Obr. 12. Vytížení serveru tornado z pomocného počítače 100 procesy

Spousta uživatelů se domnívá, že jim mail musí přijít téměř okamžitě, což se ve většině případů děje. Ale mohou nastat situace, kdy tomu tak není. Běžně se stává, že odesílatel informoval příjemce, že mu má dorazit důležitý mail. Načež příjemce po pár minutách čekání bývá už nervózní, protože mu mail stále nedorazil. Informuje tak odesílatele a ten pro jistotu mail odešle znovu. Nic tomu nepomůže, protože mail se jen zařadí do stejné čekací fronty. Pokud je tedy vyžadována okamžitá reakce na příchozí mail, pak takové služby poskytují jiné nástroje.

Program tedy z této konfigurace generuje tuto mailovou zprávu.:

```
From ales.bublik@gmail.com Mon May 24 14:49:13 2010
X-Original-To: ales@alex
From: <ales.bublik@gmail.com>
To: ales@alex

Mon, 24 May 2010 12:49:12 GMT
Subject: pokus

Pozor! Tohle je testovaci zprava
```

Takže na řídícím počítači bylo vytvořeno 900 procesů a na pomocném 100 procesů. Po 20ti sekundách od startu aplikace vygenerovala následující grafy. Na grafu je zajímavé pozorovat jak byly požadavky vyřizovány ve vlnách v intervalu 2500 milisekund a zhruba 8000 milisekund.

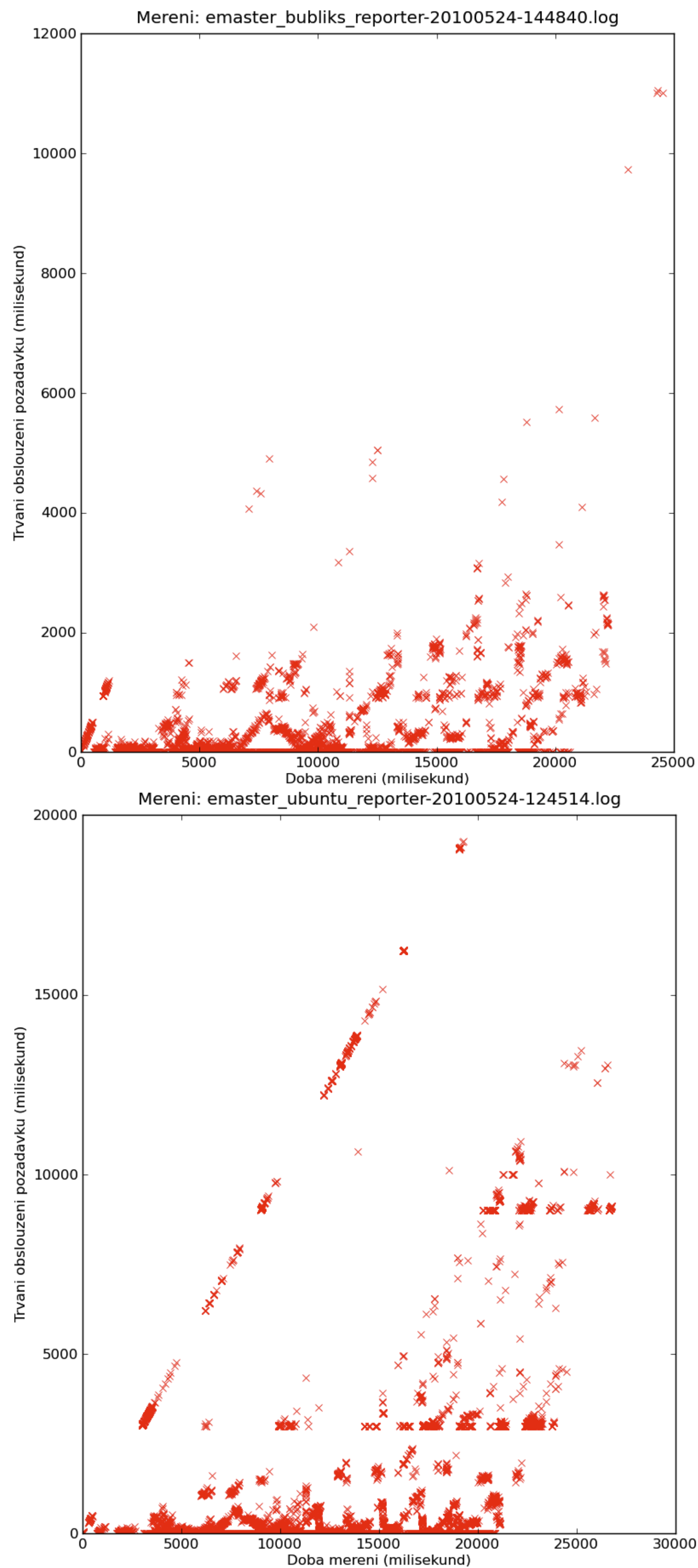
Graf, který byl získán z dat na pomocném počítači vypadá líp, ale to je tím, že jsme z něj negenerovali takovou zátěž.

Výsledky měření při zatížení 900 procesy z řídícího počítače.:

```
Odezva (milisekundy).
_______________________
Počet obsloužených dotazů: 12994.
Max: 19276.532000.
Min: 0.426000.
Průměr: 1822.276703.
Medián: 76.298500.
```

Výsledky měření při zatížení 100 procesy z pomocného počítače.:

```
Odezva (milisekundy).
_______________________
Počet obsloužených dotazů: 4262.
Max: 11060.652000.
Min: 0.287000.
Průměr: 317.446219.
Medián: 56.063500.
```



Obr. 13. Vytížení serveru Postfix z pomocného a řídícího poč. 100 resp. 900 procesy

SMTP server Postfix tedy přijat dohromady 17256 mailů. Postfix nebyl nijak zvlášť konfigurován a byl ponechán ve standartní konfiguraci, ale i tak dokázal přijmout za 20 sekund hodně mailů.

2.4 Testování PostgreSQL serveru

Pro testování databázového serveru PostgreSQL je potřeba zase nastavit v konfiguračním souboru `projekt.conf` parametru *client_type* na hodnotu *pgsql* a v konfiguračním souboru *postgresql.conf* bylo nastaveno tohle.:

```
{pg_host, "alex"}.
{pg_name, "ales"}.
{pg_pwd, "alesbublik"}.
{pg_database, "data"}.
{pg_queries, ["select now()", "select 3.14 + 2.4"]}.
```

Server byl ponechán ve standartním nastavení, ale bylo možné jej vyladit pro větší výkon, zejména přidělení více operační paměti pro data. Pro účely testování byly zvoleny tyto jednoduché SQL dotazy, ze kterých si procesy vyberou jen jeden.:

```
select now();
select 3.14 + 2.4;
```

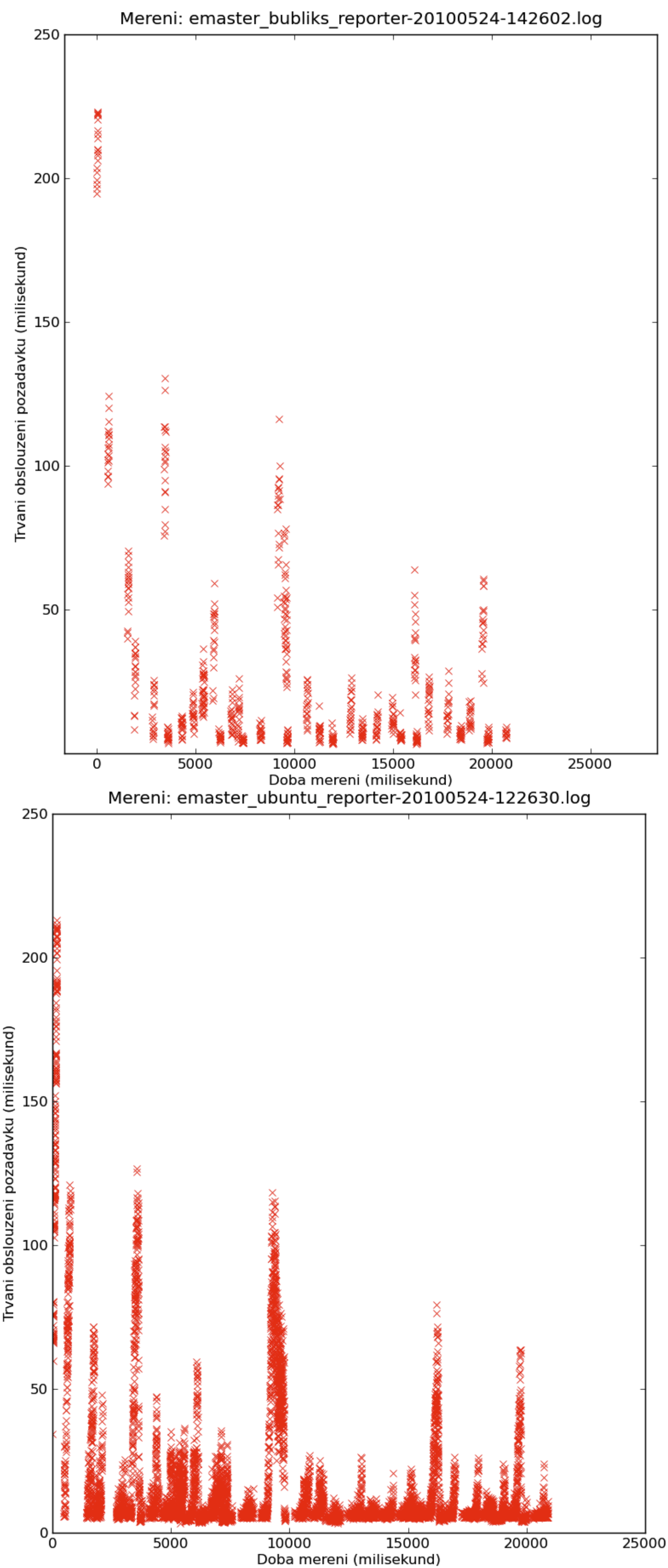
Tyto dotazy neoperují s žádnými daty a proto moc server nezatíží. V základním nastavení může server obsloužit zároveň 100 připojených uživatelů. Proto byl nastaven na řídícím počítači stav 90 testovacích procesů a na pomocném 10 procesů.

Na grafech jsou vidět výrazné sloupcovité shluky, které jsou způsobeny buď nevyladěností serveru nebo Linuxu. Jak bylo výše popsáno, tak aplikace generuje požadavky v náhodných intervalech, takže příčina těchto sloupců bude nejspíš někde v PostgreSQL.

V záznamech měření nebylo nalezeno žádné spojení, které by bylo odmítnuto, což by mělo souhlasit při nastavení maximálně 100 souběžných uživatelů.

Výsledky měření při zatížení 90 procesy z řídícího počítače.:

```
Odezva (milisekundy).
________________________
Počet obsloužených dotazů: 6828.
Max: 213.030000.
Min: 3.396000.
Průměr: 22.612000.
Medián: 8.951000.
```



Obr. 14. Vytížení serveru PostgreSQL z pomocného a řídícího poč. 10 resp. 90 procesy.

Výsledky měření při zatížení 10 procesy z pomocného počítače.:

```
Odezva (milisekundy).
_______________________
Počet obsloužených dotazů: 750.
Max: 223.175000.
Min: 3.388000.
Průměr: 29.158031.
Medián: 13.037500.
```

Databáze celkem obsloužila 7578 velmi jednoduchých dotazů za 20 sekund. Z výsledků je také vidět, že časy dotazů jsou z obou testovacích počítačů srovnatelné.

ZÁVĚR

Stále více a více lidí se spoléhají na internet jako nedílnou součást jejich života. Stále více lidí také na něm uchovává svá data, tj. fotografie, dokumenty, emaily a kontakty. Časem budou lidé muset komunikovat s úřady prostřednictvím svých datových schránek, které by měli usnadnit a zrychlit proces doručování písemností. Všechny tyto služby nám mohou usnadnit život, ale také docela znepříjemnit, pokud dojde k jejich výpadku.

Cílem této diplomové práce bylo vytvořit software, který je schopen využít celého výkonu počítače (všech výpočetních jader) a ve spolupráci s ostatními počítači (jako otroky) vytvořit dostatečný síťový provoz na otestování odolnosti síťového serveru, potažmo síťové aplikace. Tento síťový provoz by měl přibližně odrážet reálnou zátěž. Byla otestována výkonnost různých HTTP serverů a SMTP serveru Postfix a databázového serveru PostgreSQL.

Vygenerované grafy vypovídají o chování síťové aplikace v průběhu testování. Všechny testované aplikace v prvních sekundách obsluhovali požadavky pomaleji. Zajímavé byly výsledky měření Postfixu, který vyřizoval požadavky v pravidelných vlnách. Příčinou mohly být časté operace zápisu na disk, které ostatní služby nemusely vykonávat. Z webových serverů byl nejrychlejší mochiweb, ale ten je určen na specifické případy. Na výsledcích je také vidět, že aplikace napsaná ve vyšším jazyce Erlang může být i rychlejší než aplikace psaná v jazyce C.

Bylo pamatováno i na obsluhu při testování. Takže kromě toho, že za nástroje jako Erlang a operační systém Linux nemusíte platit, nepotřebujete také k takovému testování žádnou obsluhu. Díky vlastnostem jazyka Erlang lze takovou aplikaci rozdistribuovat na další výpočetní uzly pouze pomocí SSH přístupu. Jakmile je testování u konce, aplikace se na ostatních uzlech ukončí. Reporty se během testu průběžně ukládají na řídící počítač, ze kterého byla aplikace spuštěna, takže není potřeba je kopírovat.

CONCLUSION

More and more people rely on the Internet as an integral part of their lives. More and more people save their data, fotos, documents, emails and contacts on the Internet too. Eventually people will have to communicate with authorities through its data box, which should facilitate and accelerate the process of document delivery. All of these services can make your life easier, but also quite annoying, if there is failure.

The aim of this thesis was to create software that is able to utilize the performance of your computer (all cores) and in collaboration with other computers (slaves) to create a sufficient network traffic to test the resilience of a network server, network applications, respectively. This network traffic should approximately reflect the real burden. Was tested performance of various HTTP servers, SMTP server Postfix and PostgreSQL database server.

Generated graphs show the behavior of network applications during testing. All tested applications in the first seconds served the requests slowly. Interestingly, the results of measurements of Postfix, which served the requests in regular waves.The reason could be frequent write operations to disk, which may not perform other services. The fastest was Web server mochiweb, but it is designed to specific cases. The results show that applications written in higher language Erlang can be even faster than applications written in C language.

It was also remembered at service for testing. So in addition to the software such as Erlang and the Linux operating system you do not pay, you do not need a test service. Due to the characteristics of Erlang language, such an application can be uploaded to other computer nodes using only SSH access. Once testing is complete, applications on other nodes are closed. Reports are during the test continuously stored on computer control, from which the application is running, so there is no need to copy them.

SEZNAM POUŽITÉ LITERATURY

[1] DOSTÁLEK, Libor, KABELOVÁ, Alena. *Velký průvodce protokoly TCP/IP a systémem DNS*. 2. aktualiz. vyd. Praha: Computer Press, 2000. 435 s. ISBN 80-7226-323-4.

[2] ARMSTRONG, Joe. *Programming Erlang: Software for a Concurrent World*. first edition. United States of America : The Pragmatic Programmers, 2007. 519 s. ISBN 1-9343560-0-X.

[3] DONALD, Erwin Knuth. *Umění programování* : Základní algoritmy. 2008. vyd. Praha : Computer Press, 2008. 672 s. ISBN 978-80-251-2025-5.

[4] OREBAUGH, Angela, et al. *Wireshark a Ethereal*. 1. vyd. Praha : Computer Press, 2008. 448 s. ISBN 987-80-251-2048-4.

[5] HLAVENKA, Jiří, et al. *Výkladový slovník výpočetní techniky a komunikací*. 3. vyd. Praha : Computer Press, 1997. 456 s. ISBN 80-7226-023-5.

[6] PETERKA, Jiří. eArchiv.cz : *Rodina protokolů TCP/IP* [online]. Verze 2.5. Peterka, 2008 , 2008 [cit. 2009-12-24]. Dostupný z WWW: <http://www.earchiv.cz/l219/index.php3>.

[7] PETERKA, Jiří. eArchiv.cz : *Počítačové sítě* [online]. Verze 3.4. Peterka, 2009 , 2009 [cit. 2009-12-26]. Dostupný z WWW: <http://www.earchiv.cz/l220/index.php3>.

[8] Ericsson AB. *Erlang/OTP R13B documentation* [online]. Verze R13B03. Ericsson AB, c1999-2009 [cit. 2009-12-02]. Text v angličtině. Dostupný z WWW: <http://erlang.org/doc.html>.

[9] The PostgreSQL Global Development Group. *PostgreSQL 8.4.2 Documentation* [online]. Verze 8.4.2. The PostgreSQL Global Development Group, c1996-2009 [cit. 2010-01-25]. Dostupný z WWW: <http://www.postgresql.org/docs/>.

[10] Wikimedia Foundation, Inc.. *Simple Mail Transfer Protocol* [online]. Wikimedia Foundation, Inc., 2001, last modified on 24 January 2010 at 13:27 [cit. 2010-01-24]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol>.

[11] Wikimedia Foundation, Inc.. *Transport Layer Security* [online]. Wikimedia Foundation, Inc., 2001, last modified on 22 January 2010 at 16:26 [cit. 2010-01-24]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Transport_Layer_Security>.

[12] Wikimedia Foundation, Inc.. *Hypertext Transfer Protocol* [online]. Wikimedia Foundation, Inc., 2001, last modified on 27 January 2010 at 09:48 [cit. 2010-01-23].

Dostupný z WWW: <http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol>.

[13] JOHN, Hunter, DARREN, Dale, MICHAEL, Droettboom. *Matplotlib : User's Guide* [online]. verze 0.99.1.1. 2008, Last updated on Jan 15, 2010 [cit. 2009-12-02]. Dostupný z WWW: <http://matplotlib.sourceforge.net/users/index.html>.

[14] HAVLÍČEK, Miroslav. *Osobní počítače a základy elektroniky anglicky : Z praxe pro praxi čtenářů, studentů a překladatelů*. 1. vyd. Praha : BEN, 2001. 478 s. ISBN 80-86056-17-1.

[15] SATRAPA, Pavel. *Internetový protokol IPv6*. 1. vyd. Praha : CZ.NIC, z. s. p. o., 2008. 359 s. ISBN 978-80-904248-0-7.

[16] CESARINI, Francesco; THOMPSON, Simon. *Erlang Programming*. First Edition. United States of America: O'Reilly Media, Inc., 2009. 496 s. ISBN 978-0-596-51818-9.

[17] KEGEL, Dan. *Dan Kegel's Web Hostel* [online]. verze 1.212. Florida, USA : Kegel, 2006-09-02 [cit. 2010-05-21]. The C10K problem. Dostupné z WWW: <http://www.kegel.com/c10k.html>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AMQP	Advanced Message Queuing Protocol
ASCII	American Standard Code for Information Interchange
CERN	European Organization for Nuclear Research
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
MIME	Multipurpose Internet Mail Extensions
MUA	Message User Agent
OTP	Open Telecom Platform
OpenGL	Open Graphics Library
Pid	Process identification
RPC	Remote procedure call
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language
SSH	Secure Shell
TCP	Transmission Control Protocol
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
USB	Universal Serial Bus
XMPP	The Extensible Messaging and Presence Protocol

SEZNAM OBRÁZKŮ

Obr. 1. Schéma testovaného serveru spolu s uzly, které vytvářejí síťový provoz................11

Obr. 2. Rozdělení funkcionality procesu na obecnou a specifickou část.............................18

Obr. 3. Návrh aplikace a komunikace mezi jednotlivými procesy.......................................20

Obr. 4. Příklad grafu, který zobrazuje výsledky měření..35

Obr. 5. Vytížení serveru mochiweb z řídícího počítače 900 procesy...................................42

Obr. 6. Vytížení serveru mochiweb z pomocného počítače 100 procesy.............................44

Obr. 7. Vytížení serveru Apache z pomocného počítače 100 procesy.................................45

Obr. 8. Vytížení serveru Apache z řídícího počítače 900 procesy....................................... 46

Obr. 9. Vytížení serveru lighttpd z řídícího počítače 900 procesy.......................................48

Obr. 10. Vytížení serveru lighttpd z pomocného počítače 100 procesy...............................49

Obr. 11. Vytížení serveru tornado z řídícího počítače 900 procesy......................................51

Obr. 12. Vytížení serveru tornado z pomocného počítače 100 procesy...............................52

Obr. 13. Vytížení serveru Postfix z pomocného a řídícího poč. 100 resp. 900 procesy........54

Obr. 14. Vytížení serveru PostgreSQL z pomocného a řídícího poč. 10 resp. 90 procesy.. .56

SEZNAM PŘÍLOH

P I Datový nosič CD

PŘÍLOHA P I:

Datový nosič CD

Elektronická verze diplomová práce: Bublik_DP.pdf

Zdrojový kód aplikace: Bublik_projekt.zip

Záznamy z meření: mereni.zip