UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

IMPLEMENTÁCIA DISTRIBUOVANÉHO MONITOROVACIEHO SYSTÉMU

BAKALÁRSKA PRÁCA

2012

Jakub JURSA

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

IMPLEMENTÁCIA DISTRIBUOVANÉHO MONITOROVACIEHO SYSTÉMU

BAKALÁRSKA PRÁCA

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Michal Rjaško

Bratislava 2012 **Jakub JURSA**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jakub Jursa
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Implementácia distribuovaného monitorovacieho systému

Cieľ: Implementovať systém na monitorovanie funkčnosti webových serverov. V rámci siete existuje niekoľko monitorovacích uzlov, kde každý uzol monitoruje skupinu serverov. Výber skupiny serverov, o ktoré sa stará daný uzol prebieha automaticky. V prípade výpadku niektorého z uzlov systém automaticky zabezpečí prenos zodpovednosti za monitorovanie danej skupiny na iný uzol.

Vedúci: RNDr. Michal Rjaško
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 25.10.2011

Dátum schválenia: 25.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent

....
vedúci práce

Poďakovanie

Touto cestou sa chcem poďakovať svojmu vedúcemu bakalárskej práce Mgr. Michalovi Rjaškovi za ochotu, pomoc a podnetné pripomienky, ktoré mi pomohli pri písaní tejto práce. Ďakujem aj Tomášovi Kuzmovi a Michalovi Petruchovi za cenné postrehy a všetkým spoluhráčom AoE za to, že ma vedeli rozptýliť, práve vtedy, keď som to potreboval.

Abstrakt

Autor:	Jakub Jursa
Názov bakalárskej práce:	Implementácia distribuovaného monitorovacieho systému
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci bakalárskej práce:	RNDr. Michal Rjaško
Rozsah práce:	48 strán

Bratislava, jún 2012

Cieľom tejto bakalárskej práce je implementovať jednoduchý distribuovaný monitorovací systém napísaný v Ruby, ktorý je jednoduchý na nastavenie a používanie. Pokúsime sa ho implementovať v súlade s konvenciami Ruby, čo zahŕňa napríklad použitie unit testov a návrhových vzorov.

Kľúčové slová: monitorovanie, voľba koordinátora, ruby, Linux

Abstract

Author:	Jakub Jursa
Thesis title:	Implementácia distribuovaného monitorovacieho systému
University:	Univerzita Komenského v Bratislave
Faculty:	Fakulta matematiky, fyziky a informatiky
Department:	Katedra informatiky
Advisor:	RNDr. Michal Rjaško
Page count:	48 pages

Bratislava, June 2012

The main purpose of this bachelor thesis is to implement a simple distributed monitoring system written in Ruby which will be easy to set up and use. We try to implement it according to Ruby best practices, which contains for example using unit tests and design patterns.

Keywords: monitoring, leader election, ruby, Linux

Obsah

Úvod **1**

1 Špecifikácia a návrh systému **2**

1.1 Úvod do problematiky . . . 2
1.2 Formulácia problému . . . 3
1.3 Návrh riešenia . . . 3
1.3.1 Peer . . . 3
1.3.2 Koordinátor . . . 4
1.3.3 Kontroly . . . 4
1.3.4 Databáza . . . 6
1.3.5 Alerting . . . 6
1.3.6 Fungovanie a správanie sa aplikácie . . . 7
1.3.7 Voľba technológií . . . 8

2 Použité technológie a nástroje **9**

2.1 Ruby . . . 9
2.1.1 Čo je Ruby . . . 9
2.1.2 Zaujímavé ukážky . . . 10
2.2 MongoDB . . . 11

3 Teória **13**

3.1 Voľba koordinátora . . . 13
3.1.1 Bully algorithm . . . 13
3.1.2 Ring algorithm . . . 15
3.2 MapReduce . . . 16
3.3 Použité návrhové vzory . . . 17
3.3.1 Strategy . . . 17
3.3.2 Thread Pool . . . 19

4 Implementácia **21**
4.1 Popis štruktúry aplikácie 21
4.2 Databáza 22
4.2.1 Popis kolekcií 22
4.3 Popis tried 23
4.4 Démonizácia 25
4.5 Ukážky zaujímavých častí kódu 26
4.5.1 Práca s databázovým mapperom 26
4.5.2 Komunikácia medzi peermi 26
4.5.3 Aplikácia návrhového vzoru Strategy 28
4.5.4 Vykonávanie jobov - Thread pool 28
4.5.5 Použitie mutexov 30
4.5.6 Použitie MapReduce 31
4.5.7 Ukážka unit testov 32
4.6 Problémy pri implementácii 33
4.7 Možné zlepšenia do budúcnosti 34
4.8 Návrh implementácie bez centrálnej databázy 34

5 Dokumentácia **37**
5.1 Inštalácia 37
5.1.1 Prerekvizity 37
5.1.2 Aplikácia 37
5.2 Konfigurácia 38
5.3 Spúšťanie 39
5.4 Managovanie taskov 40
5.4.1 Pridanie nového tasku 40
5.4.2 Vypísanie zoznamu taskov 40
5.4.3 Vymazanie tasku 41
5.4.4 Znovunačítanie taskov 41
5.5 Checky 41
5.5.1 Pridanie nového checku 41
5.5.2 Distribúcia checkov medzi peerov 42
5.6 Tutorial 43
5.6.1 Konfigurácia 43
5.6.2 Spustenie 44
5.6.3 Tasky 44

Záver **46**

Zoznam použitej literatúry **47**

Príloha A **48**

Zoznam obrázkov

3.1 Bully algorithm 14
3.2 Ring algorithm 16
3.3 Schéma fungovania thread poolu 20

4.1 Použitie vzoru Strategy - diagram tried 28
4.2 Použitie vzoru Thread pool - diagram tried 30
4.3 Príklad pridania nového tasku bez centrálnej DB, obr. 1 35
4.4 Príklad pridania nového tasku bez centrálnej DB, obr. 2 35
4.5 Príklad pridania nového tasku bez centrálnej DB, obr. 3 36
4.6 Príklad pridania nového tasku bez centrálnej DB, obr. 4 36
4.7 Príklad pridania nového tasku bez centrálnej DB, obr. 5 36

Zoznam listingov

1.1 Ukážka jednoduchej kontroly cez cron . . . 2
2.1 Ukážka code blocku . . . 10
2.2 Ukážka open class . . . 10
2.3 Implementácia Factory v Ruby . . . 11
2.4 Ukážka dokumentov v MongoDB . . . 12
3.1 Ukážka MapReduce v pseudokóde . . . 16
3.2 Strategy pattern s abstraktnou Strategy triedou . . . 17
3.3 Strategy pattern s použitím duck-typingu . . . 18
3.4 Minimalistický thread pool . . . 20
4.1 Ukážka démonizácie procesu v Ruby . . . 25
4.2 Posielanie správ cez sockety . . . 26
4.3 Prijímanie správ cez sockety . . . 27
4.4 Použitie návrhového vzoru strategy (main.rb, riadok 77) . . . 28
4.5 Implementácia thread poolu (thread_pool.rb) . . . 29
4.6 Ukážka použitia mutexov (job_generator.jb) . . . 30
4.7 Použitie MapReduce, funkcia map (database_adapter.rb, riadok 138) . 31
4.8 Použitie MapReduce, funkcia reduce (database_adapter.rb, riadok 145) 32
4.9 Ukážka unit testu (thread_pool_spec.rb, riadok 32) . . . 32
5.1 Vzorová konfigurácia . . . 38
5.2 Ukážka checku kontrolujúceho otvorenosť portu . . . 42

Úvod

Monitorovanie serverov, služieb resp. infraštruktúry je každodennou prácou väčšiny systémových administrátorov. V dnešnom svete cloudov a virtualizácie nie je, najmä pri väčších systémoch, možné všetko monitorovať ručne a preto je potrebné takéto činnosti nejakým spôsobom automatizovať.

Existujúce riešenia (Nagios OpenNMS, Pandora) sú príliš zložité, komplexné, náročné na konfiguráciu, určené skôr pre enterprise prostredie a v neposlednom rade nie sú 'rozumne' distribuované. To znamená, že majú nejaký SPOF (Single Point Of Failure).

Preto sme sa rozhodli implementovať distribuovaný monitorovací software, ktorý nebude trpieť týmito neduhmi (respektíve aspoň nie väčšinou z nich). Ako vhodný programovací jazyk sa javí Ruby, cieľovou platformou je Linux/Unix (kvôli rozšírenosti na serveroch).

Cieľom tejto práce je poskytnúť náhľad do logiky a štruktúry aplikácie a taktiež oboznámiť čitateľa s teoretickými základmi, na ktorých je táto aplikácia postavená.

V prvej kapitole sa budeme venovať špecifikácii a návrhu aplikácie, v druhej kapitole priblížime použité technológie a nástroje. V tretej kapitole vysvetlíme teóriu na pozadí aplikácie.

Štvrtá kapitola tvorí jadro celej práce a zaoberá sa implementáciou aplikácie. Vysvetľuje jej fungovanie a logiku, popisy jednotlivých tried a tiež obsahuje ukážky zaujímavých častí zdrojového kódu. Piata kapitola tvorí dokumentáciu, objasňujúcu inštaláciu a nastavovanie aplikácie.

Kapitola 1

Špecifikácia a návrh systému

V tejto kapitole vyslovíme požiadavky na systém, rozoberieme ich a navrhneme riešenie.

1.1 Úvod do problematiky

Majme niekoľko desiatok až stoviek serverov, na ktorých chceme zaistiť dostupnosť rôznych služieb (odpovedanie na ping, dostupnosť databázy, funkčnosť webservera, ...). Požadujeme, aby boli typy kontrol ľahko rozšíriteľné. Určite potrebujeme aj mechanizmus na hlásenie nedostupnosti. Skúsme načrtnúť jednoduché riešenie cez *cron*.

Vyberieme jeden osobitný monitorovací server, ktorý bude pomocou cronjobov kontrolovať všetky služby na ostatných serveroch a v prípade zlyhania kontroly spustí skript na odoslanie varovania emailom.

```
1  $ cat /etc/cron.d/monitoring
2  10 * * * *    root    /usr/local/bin/check1
3
4  $ cat /usr/local/bin/check1
5  #!/bin/bash
6  ok=0;
7  ping -c 1 172.10.1.1 > /dev/null 2>&1 && ok=1;
8  if [$ok == 0 ]; then
9      /usr/local/bin/send_mail 'ping' '172.10.1.1';
10 fi;
11
12 $ cat /usr/local/bin/send_email
13 #!/bin/bash
14 echo '$1 was unsuccessful on $2' |
```

```
15      /bin/mail admin@company.com "Service not running";
```

Listing 1.1: Ukážka jednoduchej kontroly cez cron

Narážame však na niekoľko problémov. Napríklad - výsledky kontrol sa nikam neukladajú, nevieme teda napríklad zistiť dostupnosť jednotlivých služieb v čase. Takisto, takéto riešenie cez cron je veľmi náročné na spravovanie a nastavovanie. Riešením by bol napríklad samostatný démon s lokálnou databázou. Ďalej, čo ak by tento monitorovací server vypadol, napríklad z dôvodu zlyhania hardwareu ? Tým by, samozrejme, celý monitoring prestal fungovať. Ďalší problém nastáva, ak týchto kontrol bude veľké množstvo - periodicky sa budú vytvárať tie isté kontrolné procesy, čo je zbytočná réžia pre systém.

1.2 Formulácia problému

Zopakujme teda úvod z predchádzajúcej časti a pridajme ďalšie požiadavky:

- Monitorovanie dostupnosti služieb na niekoľkých desiatkach až stovkách serverov
- Viac monitorovacích serverov (maximálne niekoľko desiatok)
- Self-healing - ak niektorý z peerov 'spadne', funkčnosť celého systému by mala zostať zachovaná
- Jednoduché pridávanie a odoberanie peerov, kontrol a kontrolovaných serverov
- Minimalizovanie nutnosti ručných zásahov do systému (tzn. napríklad aby pri výpadku niektorého peera alebo pridávaní kontroly nového servera nebolo potrebné ručne reštartovať aplikáciu na každom peerovi)
- Paralelné vykonávanie viacerých kontrol jedným peerom
- Schopnosť hlásiť výpadky a robiť nad nimi štatistiky
- Fungovanie na Linuxových serveroch (nie je potrebné GUI)

1.3 Návrh riešenia

1.3.1 Peer

Aplikáciu bude predstavovať démon bežiaci na každom monitorovacom serveri. Tento server nazvime *peer*. Systém bude teda tvorený množinou peerov.

Úloha peera

Úlohou peerov bude periodicky vykonávať dopredu definované kontroly cieľových služieb na monitorovaných serveroch a poslať upozornenie v prípade výpadku.

Komunikácie medzi peermi

Na komunikáciu medzi peermi použijeme sockety, cez ktoré si peeri budú posielať jednoduché správy. Alternatívou pre komunikáciu by bolo buď použitie minimalistického HTTP webservera alebo message queue protokolu (ideálne brokerless typu ako napríklad ZeroMQ[1] alebo ActiveMQ[2]). V našom prípade si kvôli jednoduchosti a relatívne malému počtu správ, ktoré si peeri budú musieť posielať vystačíme so socketmi. Peer sa bude v sieti identifikovať pomocou dvojice $\langle ip, port \rangle$.

1.3.2 Koordinátor

Úloha koordinátora

Koordinátor bude mať na starosti udržiavať konzistenciu siete - tj periodické kontrolovanie dostupnosti všetkých ostatných peerov v sieti, vyhlasovanie iných peerov za *mŕtvych* ak neodpovedajú, prerozdeľovanie kontrol medzi ostatných peerov, posielanie upozornení v prípade výpadku atď.

Voľba koordinátora

Koordinátorom sa stane ten peer, ktorý je zvolený vo voľbe. Každý peer si bude pamätať, kedy ho koordinátor naposledy kontroloval a ak nastane moment, že tento peer nebol už T sekúnd kontrolovaný, tak je vysoká pravdepodobnosť, že koordinátor vypadol. V takomto prípade peer spustí voľbu nového koordinátora. Voľba koordinátora bude implementovaná tak, aby bol algoritmus samotnej voľby vymeniteľný. Konkrétne algoritmy popíšeme v sekcii 3.1 - Voľba koordinátora.

1.3.3 Kontroly

Task

Task bude predpis, ktorý bude predstavovať periodickú kontrolu. Bude obsahovať:

- Adresu cieľového servera

[1] `http://www.zeromq.org/`
[2] `http://activemq.apache.org/`

- *Check*, tj typ kontroly (napríklad *ping* alebo *host*)
- Periodicitu, s akou sa daná kontrola vykonáva
- Referenciu na peera, ktorý má na starosti tento task
- Stručný slovný popis kontroly (napr. 'DNS resolve check for myserver.com')
- Prípadné prídavné argumenty pre samotný skript vykonávajúci kontrolu

Check

Check bude samotné jadro kontroly - skript, ktorý sa periodicky spúšťa. Na vstupe bude mať k dispozícii adresu cieľového servera a prípadné prídavné argumenty. Jeho výstupom bude dvojica - číselný kód (0 v prípade úspešnej kontroly, inak číslo chyby) a dodatočná textová informácia o kontrole (napríklad výstup z konzoly). Chceme, aby kontrolovacie skripty boli užívateľom ľahko doplniteľné o nové. O tom bližšie v kapitole 5.5 - Checky).

Job

Z týchto taskov budú generované jednorázové úlohy - *joby*, každý bude obsahovať:

- Referenciu na task, z ktorého je tento job odvodený
- Referenciu na peera, ktorý ma tento job na starosti
- Flag indikujúci, či tento job už bol naplánovaný na vykonanie
- Flag indikujúci, či tento job už bol dokončený
- Čas dokončenia jobu (*nil* pre ešte nedokončené joby)
- Exitstatus - *return value* kontroly
- Output - text, ktorý kontrolovací skript vypísal do konzoly

Užívateľ teda zadefinuje sadu taskov, ktoré peer-koordinátor rozdelí medzi ostatných peerov a tí z týchto taskov budú generovať joby, ktoré budú vykonávať.

Rozdeľovanie taskov

Rozdeľovanie taskov medzi peerov implementujeme tak, aby bol algoritmus, ktorý sa stará o samotné rozdelenie ľahko vymeniteľný. Pre začiatok implementujeme naivný algoritmus, konkrétne round-robin. Alternatívou by mu mohol byť typ váhovaného algoritmu, ktorý by bral do úvahy napríklad periodicitu jednotlivých taskov. Tým by sa dosiahla istá 'férovosť' rozdelenia taskov medzi peerov.

1.3.4 Databáza

Prečo databáza?

Peerovia potrebujú vedieť rôzne informácie (napr. zoznam taskov, jobov, zoznam ostatných peerov, ...). Pre zjednodušenie použijeme centrálnu databázu, o ktorej budeme predpokladať, že je *vysoko dostupná (high availability)*, tzn. neočakávame, že by mohla vypadnúť. Dôsledkom bude, že 'distribuovanosť' celého systému bude trocha 'falošná', no na ilustráciu použitých konceptov nám to postačí.

Alternatívnym riešením ku *centrálnej* databáze by bolo použitie replikovaných databáz. Každý peer by bol databázovým serverom a mal u seba kompletnú kópiu celej databázy, jeden z peerov by bol master (iba do jeho databázy by bolo možné zapisovať) a všetky ostatné databázy by boli len jeho repliky. V prípade, že by master databáza 'spadla', bolo by nutné zvoliť novú master databázu a všetky zápisy smerovať na ňu. Nevýhodou tohto riešenia je ale zložitosť konfigurácie databáz.

Ak by sme sa rozhodli systém realizovať bez použitia databázy, bolo by potrebné použiť vlastný protokol na výmenu dát medzi peermi. Jeho návrhu sa budeme venovať v kapitole 4.8 - Návrh implementácie bez centrálnej databázy.

Použitie databázy

Databáza bude držať zoznam peerov, taskov a jobov. Rozhodli sme sa použiť NoSQL databázu z dôvodu, že tieto databázy sú vo všeobecnosti rýchlejšie pri pridávaní nových dát. A akurát pridávanie nových jobov do databázy bude najčastejšia operácia.

1.3.5 Alerting

V prípade zlyhania kontroly (exitstatus niektorého jobu bude rôzny od 0) pošle koordinátor email s informáciou o zlyhanej kontrole na vopred definovaný email.

1.3.6 Fungovanie a správanie sa aplikácie

Aplikácia bude fungovať ako démon, nastavenia (ip, port, prístupové údaje k databáze) sa budú nachádzať v súbore *config.yaml*.

Spustenie

Po spustení aplikácie sa peer zaregistruje do systému tak, že uloží do databázy nový dokument obsahujúci informácie o ňom, konkrétne dvojicu $\langle ip, port \rangle$ a spustí voľbu koordinátora. Po skončení každej voľby koordinátor reviduje rozdelenie taskov (či je vyvážené, či má každý task na starosti žijúci peer atď, ak nie, prerozdelí ich). Po voľbe dostane tento novospustený peer pridelené tasky. Z nich bude generovať joby vzhľadom na periodicitu daného tasku a ukladať ich do databázy. Zároveň si bude tento peer vyzdvihovať ešte nenaplánované joby, plánovať ich a vykonávať.

‘Spadnutie‘ peera

Ak peer prestane odpovedať na kontroly koordinátora, tak ten vymaže jeho záznam z databázy a prerozdelí tasky, o ktoré sa staral ostatným peerom.

Pridanie peera

Ak chceme pridať nového peera (ak napríklad systém nestíha vykonávať tasky, alebo prevádzame migráciu), tak jednoducho spustíme aplikáciu. Nový peer iniciuje voľbu koordinátora, po ktorej mu budú pridelené tasky.

Odobratie peera

Pri odobraní peera stačí ukončiť aplikáciu, koordinátor to bude považovať za ‘spadnutie‘ a podľa toho sa zachová.

‘Spadnutie‘ koordinátora

Ako bolo popísané v časti 1.3.2 - Voľba koordinátora, keď niektorý peer zaregistruje, že koordinátor je ‘spadnutý‘, spustí voľbu nového podľa daného algoritmu. Nový koordinátor následne reviduje prerozdelenie taskov, prípadne odoberie nefunkčných peerov zo systému.

Práca s taskmi

Modifikácia taskov je realizovaná ich úpravou v databáze a následným upozornením ľubovoľného peera pomocou POSIXového[3] signálu *USR1*. Po prijatí tohto signálu peer upovedomí koordinátora na zmenu taskov a ten ich následne reviduje. Konkrétne je práca s taskmi popísaná v kapitole 5 - Dokumentácia.

1.3.7 Voľba technológií

Ako programovací jazyk pre túto aplikáciu sme zvolili Ruby. Ako databázu použijeme MongoDB. Prvým dôvodom je jej NoSQL povaha a druhým, že MongoDB je veľmi jednoducho škálovateľné, napríklad pomocou replica-setov (tým vieme zároveň dosiahnuť vysokú dostupnosť). Tieto technológie sú bližšie popísané v nasledujúcej kapitole.

Zdrojový kód bude verzionovaný pomocou programu GIT[4].

[3]Signály sú formou komunikácie medzi procesmi na POSIXových operačných systémoch. Singál je asynchrónna správa poslaná inému procesu (alebo inému threadu v danom procese), ktorá má upozorniť na výskyt istej udalosti.

[4]`http://git-scm.com/`

Kapitola 2

Použité technológie a nástroje

V tejto časti sa bližšie pozrieme na použité technológie, dôvody pre ich voľbu, ich výhody, aj nevýhody.

2.1 Ruby

2.1.1 Čo je Ruby

Ruby je dynamický, reflexívny, multi-paradigmový objektovo-orientovaný programovací jazyk, ktorý vznikol v roku 1995. Syntaxou je podobný Perlu s črtami Smalltalk-u. Jeho autorom je Japonec Yukihiro "Matz"Matsumoto. V posledných rokoch si Ruby získalo veľkú obľubu najmä vo webovom svete kvôli populárnemu frameworku 'Ruby on Rails'. Veľmi vhodný sa javí ale aj ako skriptovací jazyk na písanie rôznych démonov, ako je to aj v našom prípade.

Základné vlastnosti a schopnosti Ruby

- Interpretovanosť
- Automatický memory management (garbage collection)
- Dynamické typovanie
- Plná objektovosť
  - Neexistujú primitívne typy ako napríklad v Jave *int*, *char* alebo *boolean*.
  - Aj *nil* je objekt (konkrétne inštancia triedy *NilClass*)
  - Každá definovaná funkcia je metódou - ak je táto funkcia definovaná na najvyššej úrovni, tak sa stáva metódou triedy *Object*.

- Prvky funkcionálnych jazykov (anonymné funkcie, closures; vlastnosť, že každý *statement* má hodnotu; návratovou hodnotou každej funkcie je hodnota posledného výrazu v nej)
- Metaprogramovanie (úprava existujúceho alebo generovanie nového kódu v runtime)
- Nemá interfaces ani abstraktné triedy a metódy

2.1.2 Zaujímavé ukážky

Blocks

V iných jazykoch nazývané aj *closure*. Je to blok kódu, nachádzať sa môže len pri volaní metódy. Telo bloku nie je vykonávané okamžite, ale je predávané danej metóde ako posledný parameter.

```
1 a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 a.each { |i| puts i }
```

Listing 2.1: Ukážka code blocku

Metóda *each* očakáva ako parameter *block*, ktorý je zavolaný pre každý element poľa. Ten je potom bloku predaný cez parameter *i*. Ukážka vypíše všetkých 10 čísel pod seba.

Open class

Triedam je možné počas behu programu pridávať metódy - a to aj systémovým, ako sú String, Hash alebo Array. Takýmto úpravám kódu v runtime sa zvykne hovoriť aj ‘monkeypatching‘[1].

```
1  class HelloWorld
2    def greet
3      puts 'Hello World!'
4    end
5  end
6
7  h = HelloWorld.new
8  h.greet               # prints 'Hello World!'
9  h.say_hello           # raises 'NoMethodError'
10
```

[1] http://en.wikipedia.org/wiki/Monkey_patch

```
11 class HelloWorld
12   def say_hello
13     puts 'Hello!'
14   end
15 end
16
17 h.say_hello          # prints 'Hello!'
```

Listing 2.2: Ukážka open class

Factory

Tým, že v Ruby je každá trieda zároveň inštanciou triedy *Class* (tým pádom implementuje metódu *new*) je aplikovanie návrhového vzoru Factory veľmi jednoduché.

```
1 def create_from_factory(factory)
2   factory.new
3 end
4 a = create_from_factory(Array)
5 s = create_from_factory(String)
```

Listing 2.3: Implementácia Factory v Ruby

2.2 MongoDB

MongoDB je ľahko škálovateľná dokumentovo orientovaná NoSQL databáza, prvá jej verzia vyšla v roku 2007. Je bezschémová, dokumenty sú v nej ukladané vo formáte BSON (binary JSON). Je široko použiteľná, hodí sa však najmä na prípady, keď je záznamov v databáze veľmi veľa (real-time štatistiky, archivácia, logging . . .) a keď je najčastejšou operáciou insert. Medzi jej výhody patria aj jednoduchá inštalácia, veľmi dobrá dokumentácia a aktívna komunita.

Hlavné rozdiely oproti klasickým relačným databázam sú:

- Bezschémovosť
  - Miesto tabuliek sa používajú tzv. *collections*, ktoré nemajú pevnú štruktúru
- Memory mapped data
  - Všetky dáta sú v pamäti a pravidelných intervaloch sa synchronizujú na disk.)

- Neexistujú joiny
- Vo všeobecnosti nepodporuje transakcie

Tým, že to nie je relačná databáza, tak *collections* nemajú pevnú štruktúru, každý dokument môže vyzerať úplne inak. V praxi to pridáva databáze veľkú flexibilitu a napríklad odpadá tým nutnosť zapodievať sa migračnými skriptami.

Obsah kolekcie v MongoDB teda môže vyzerať napríklad nasledovne:

```
{
    "_id": ObjectId("4efa8d2b7d284dad101e4bc9"),
    "last name": "Jones",
    "first": "John",
    "age": 47
},
{
    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
    "last name": "Johnas",
    "first name": "Jack",
    "age": 29,
    "address": {
        "street": "42th avenue",
        "city": "New York"
    }
}
```

Listing 2.4: Ukážka dokumentov v MongoDB

Kapitola 3

Teória

V tejto kapitole sa budeme venovať teórii stojacej v pozadí programovanej aplikácie. Vysvetlíme si základné algoritmy na voľbu koordinátora v distribuovanom prostredí, predstavíme si algoritmus MapReduce, ktorý je neoddeliteľnou súčasťou NoSQL databáz a tiež sa pozrieme na použité návrhové vzory.

3.1 Voľba koordinátora

V prvej kapitole sme odôvodnili potrebu existencie akéhosi superpeera (koordinátora) siete, ktorý bude rozdeľovať ostatným úlohy, dohliadať na ich dostupnosť atď. Ak sa však stane, že tento superpeer bude zrazu nedostupný (zlyhanie siete, hardwareu ...), musí ho niekto zastúpiť. Práve na to sa využívajú tzv. 'leader election' algoritmy.

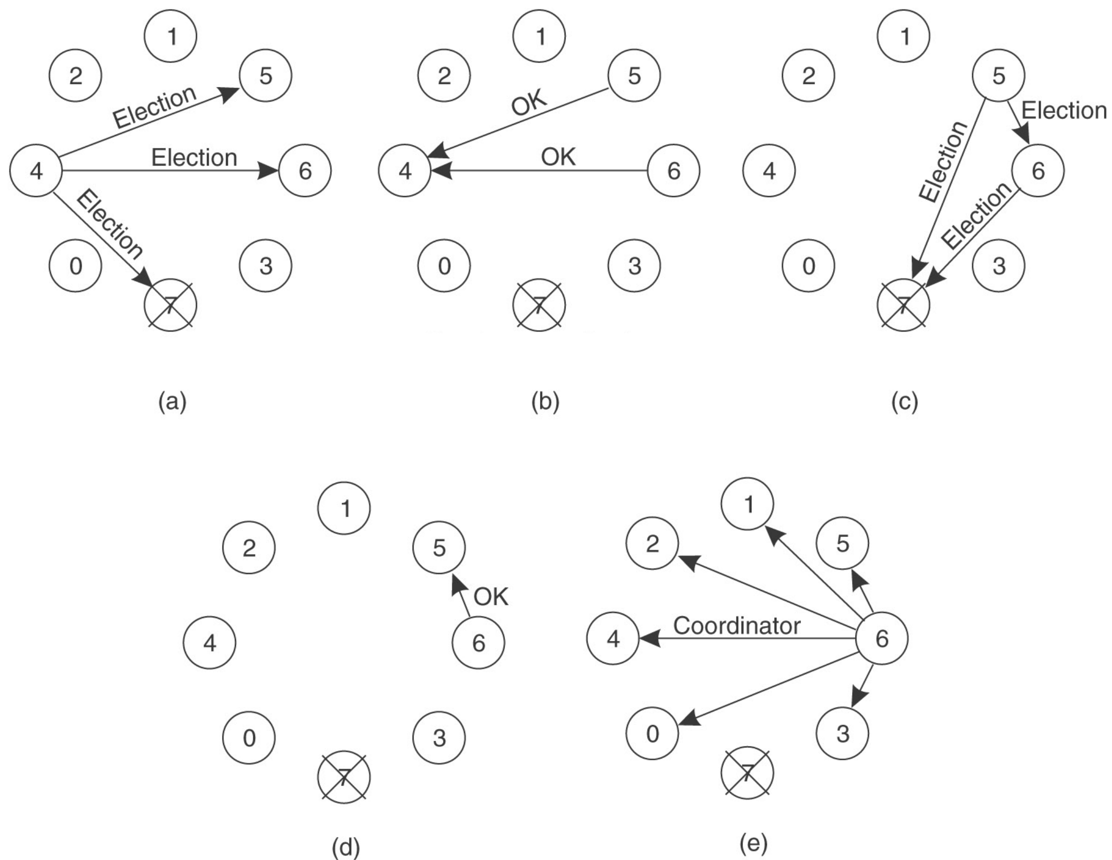
V tejto časti si vysvetlíme bežné algoritmy na voľbu koordinátora v distribuovanom prostredí. Budeme predpokladať, že doručovanie správ je spoľahlivé, že procesy (peerovia) sú unikátne očíslované, každý proces pozná čísla všetkých ostatných procesov a vie s nimi komunikovať (teda kompletnú topológiu siete).

3.1.1 Bully algorithm

Tento algoritmus, ktorého autorom je Garcia-Molina [GM82] bol prvýkrát predstavený v roku 1982. Koordinátorom sa stáva stále proces s najvyšším číslom (prioritou), preto sa volá sa tento algoritmus volá 'bully algorithm', v preklade 'tyranský algoritmus'.

Ak proces P zistí, že koordinátor neodpovedá, tak iniciuje voľbu nového koordinátora nasledovným spôsobom:

1. P pošle správu $ELECTION$ všetkým procesom, ktoré majú vyššie číslo.



Obr. 3.1: Bully algorithm

2. Ak žiaden z ostatných procesov do určitého timeoutu neodpovie, P sa stáva novým koordinátorom.

3. Ak procesu P príde odpoveď, tak čaká, kým sa proces s vyšším číslom vyhlási za koordinátora.

V ľubovoľnom momente môže ktorýkoľvek proces dostať správu $ELECTION$ od niektorého z procesov s nižším číslom. Ak takúto správu dostane, pošle späť správu OK, ktorá znamená, že daný proces ešte žije. Tento proces vzápätí zaháji vlastnú voľbu koordinátora (ak ju už nezačal). Nakoniec teda ostane proces s najvyšším číslom, ktorému sa nevrátia žiadne správy OK od procesov s vyšším číslom a ten sa následne prehlási za koordinátora hromadným poslaním správy $COORDINATOR$ všetkým ostatným procesom.

Ak proces, ktorý ‘spadol‘ opäť ožije, zaháji voľbu koordinátora. Ak to náhodou bol proces s najvyšším číslom, tak ‘vytlačí‘ dočasného koordinátora.

Na obrázku 3.1 [Tan06] vidíme príklad voľby nového koordinátora. Predchádzajúci koordinátor, proces 7 'spadol', ako prvý to zaznamenal proces 4, ktorý poslal správu $ELECTION$ (3.1(a)) procesom s vyšším číslom, konkrétne 5, 6 a 7. Procesy 5 a 6 odpovedali správou OK (3.1(b)), tým pádom proces 4 byť koordinátorom nemôže a čaká na výsledok volieb. Procesy 5 a 6 spustia každý svoju voľbu (3.1(c)), v momente, keď proces 6 posiela OK procesu 5 už vie, že proces 7 je stále 'spadnutý' a teda proces 6 sa stáva novým koordinátorom.(3.1(e)).

V najlepšom prípade je pri voľbe koordinátora vymenených $n - 1$ a v najhoršom prípade $O(n^2)$ správ [Sin96].

3.1.2 Ring algorithm

Ďalším zo základných algoritmov je takzvaný 'ring algorithm' [Tan06]. Predpokladajme, že procesy sú fyzicky alebo logicky usporiadané do kruhu, takže každý proces vie, ktorý proces po ňom nasleduje. Opäť predpokladáme unikátne očíslovanie procesov.

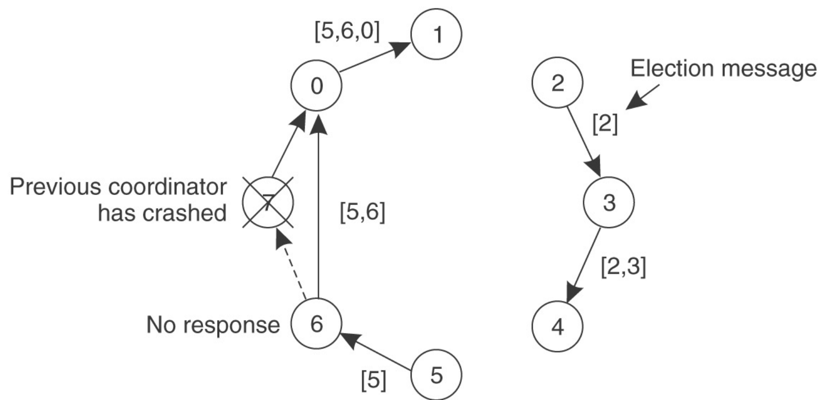
Ak proces P zistí, že koordinátor 'spadol', voľbu nového iniciuje tak, že pošle správu $ELECTION$ spolu so svojím číslom nasledujúcemu procesu. Ak je tento nasledujúci proces 'spadnutý', proces P pošle túto správu najbližšiemu živému nasledujúcemu procesu. Ten k správe pridá svoje číslo a pošle ju ďalej. Časom správa prejde po celom kruhu a vráti sa ku iniciátorovi voľby - ten to zistí tak, že prijatá správa obsahuje jeho číslo. V tomto momente správa obsahuje všetkých 'živých' účastníkov. Proces P, zmení typ správy na $COORDINATOR$ a opäť ju nechá kolovať po celom kruhu. Teraz je už ale jasné, kto je nový koordinátor - je to proces s najvyšším číslom v správe. Po obehnutí celého kruhu je táto správa zahodená.

Opäť platí, že ak predtým 'padnutý' proces 'ožije', tak iniciuje voľbu. Výhodou tohto algoritmu je menší počet vymenených správ, nevýhodou môže byť rýchlosť.

Obrázok 3.2 [Tan06] znázorňuje situáciu, keď dva procesy, 2 a 5 naraz iniciujú voľbu nového koordinátora po tom, čo pôvodný koordinátor, proces 7 'spadol'. Obe správy $ELECTION$ sa vybudujú a budú cirkulovať nezávisle na sebe a obe budú procesmi 2 resp. 5 premenené na správy typu $COORDINATOR$ s rovnakým obsahom. Voľba koordinátora dopadne korektne, jedinou nevýhodou sú redundantné správy.

V procese voľby koordinátora je pri ring algoritme vymenených stále $2n$ správ za každého iniciátora voľby. (Ak teda k procesov iniciuje voľbu koordinátora v rovnaký moment, tak počet vymenených správ bude dokopy $2kn$.)

Existuje množstvo modifikácií týchto algoritmov, ktoré zlepšujú maximálne množstvo vymenených správ na $O(n. \log n)$, prípadne $O(n)$. Vzhľadom na to, že predpokladáme, že v našom systéme nebude viac ako niekoľko desiatok účastníkov (ako sme to



Obr. 3.2: Ring algorithm

uviedli v časti 1.2 - Formulácia problému) nám postačia aj uvedené algoritmy s horšou zložitosťou.

3.2 MapReduce

MapReduce je programovací model [Dea04] na spracovávanie veľkého množstva dát. Používateľ zadá na vstupe funkciu *map*, ktorá vygeneruje množinu *dočasných* dvojíc kľúč-hodnota a druhú funkciu, *reduce*, ktorá zlúči *dočasné* dvojice s rovnakým kľúčom. Programy používajúce tento model sú ľahko paralelizovateľné. Väčšina NoSQL databáz používa na agregáciu práve MapReduce model.

Príklad: [Dea04] Predstavme si, že potrebujeme zrátať výskyty jednotlivých slov vo veľkom množstve dokumentov. Riešením by bola nasledujúca dvojica funkcií *map* a *reduce*.

```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for each word w in value:
5    EmitIntermediate(w, "1");
6
7  reduce(String key, Iterator values):
8    // key: a word
9    // values: a list of counts
10   int result = 0;
11   for each v in values:
```

```
12    result += ParseInt(v);
13    Emit(AsString(result));
```

Listing 3.1: Ukážka MapReduce v pseudokóde

Funkcia *map emituje* (vráti) pre každé slovo dvojicu $\langle slovo, 1\rangle$ funkcia *reduce* už len zráta jednotlivé výskyty.

Fungovanie funkcie *map* by sa teda dalo zapísať ako

```
1 map(k1,v1) -> list(k2,v2)
```

To znamená, že funkcia je aplikovaná na množinu kľúčov a hodnôt a jej celkovým výstupom je zoznam hodnôt a kľúčov z inej domény. Ten je potom zlúčený podľa rovnakých kľúčov a tento výsledok je predaný funkcii reduce:

```
1 reduce(k2,list(v2)) -> list(v3)
```

3.3 Použité návrhové vzory

3.3.1 Strategy

Strategy pattern spočíva v extrahovaní algoritmu do osobitného objektu. Ideou je mať niekoľko rôznych objektov, *stratégií*, ktoré robia tú istú vec, len iným spôsobom a sú navzájom zameniteľné. Navyše, všetky tieto *stratégie* musia mať rovnaký interface. Štandardným riešením [DP95] je vytvorenie abstraktnej triedy *Strategy*, od ktorej budú konkrétne stratégie dediť, alebo interface, ktorý budu tieto konkrétne stratégie implementovať. Ukážku vidíme na listingu 3.2.

```
1  class AbstractFormatter
2    def format(text)
3      raise NoMethodError, 'This method has to be overridden!'
4    end
5  end
6
7  class PlainFormatter < AbstractFormatter
8    def format(text)
9      text
10   end
11 end
12
13 class HTMLFormatter < AbstractFormatter
```

```
14    def format(text)
15      "<html><p>#{text}</p></html>"
16    end
17  end
18
19  class ConsoleFormatter < AbstractFormatter
20    def format(text)
21      text.gsub("\n", " ").scan(/.{80}|.+/).join("\n")
22    end
23  end
24
25  class Writer
26    def initialize(formatter)
27      @formatter = formatter
28    end
29
30    def write(text)
31      puts @formatter.format(text)
32    end
33  end
34
35  w = Writer.new(PlainFormatter.new)
36  w.write('Hello world!')            # prints 'Hello world!'
```

Listing 3.2: Strategy pattern s abstraktnou Strategy triedou

Napríklad v Jave by bol tento prístup úplne správny. Trieda *AbstractFormatter* by predstavovala spoločného predka všetkých *formatterov* a teda trieda *Writer* by v konštruktore očakávala objekt typu *AbstractFormatter*. Lenže Ruby nie je staticky typovaný jazyk tak, ako Java a preto sa tento návrhový vzor využíva mierne upravený. Prvým dôvodom je, že Ruby nemá abstraktné triedy (a metódy) a druhým dôvodom je konvencia 'duck-typing over inheritance' [Ols07] - tá vychádza práve z dynamickej povahy jazyka. Inštančná premenná *@formatter* triedy *Writer* nemá typ a môžme do nej priradiť objekt ľubovoľného typu pokým bude mať implementovanú metódu *format*. Abstraktnú triedu *AbstractFormatter* teda môžme pokojne odstrániť. Upravený príklad je uvedený v listingu 3.3.

```
1  class PlainFormatter
2    def format(text)
3      text
```

```
4     end
5   end
6
7   class HTMLFormatter
8     def format(text)
9       "<html><p>#{text}</p></html>"
10    end
11  end
12
13  class ConsoleFormatter
14    def format(text)
15      text.gsub("\n", " ").scan(/.{80}|.+/).join("\n")
16    end
17  end
18
19  class Writer
20    def initialize(formatter)
21      @formatter = formatter
22    end
23
24    def write(text)
25      puts @formatter.format(text)
26    end
27  end
28
29  w = Writer.new(PlainFormatter.new)
30  w.write('Hello world!')              # prints 'Hello world!'
```

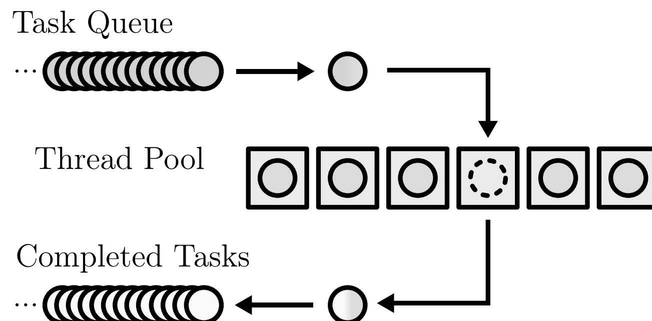
Listing 3.3: Strategy pattern s použitím duck-typingu

3.3.2 Thread Pool

Tento návrhový vzor, nazývaný aj 'work queue', rieši situáciu, keď potrebujeme rýchlo a efektívne vykonávať krátko žijúce úlohy.

Na začiatku sa vytvorí pevný počet threadov, ktoré postupne vykonávajú úlohy z fronty (predpokladá sa, že threadov je menej ako úloh). Výhodou tohto návrhového vzoru je, 'recyklácia threadov', tzn. zamedzenie nutnosti vytvárať (a potom likvidovať) pre každú novú úlohu nový thread - to je totiž zbytočná réžia navyše. Pri implemen-

tovaní tohto návrhové vzoru je potrebné si dať pozor na to, aby práca s frontou úloh bola thread-safe.



Obr. 3.3: Schéma fungovania thread poolu

Na obrázku 3.3[1] je ilustrácia fungovania thread poolu - akonáhle niektorý z threadov dokončí úlohu, tak si z fronty vyzdvihne ďalšiu. Na listingu 3.4 je ukážka minimalistického thread poolu. Jej výsledkom bude, že do konzoly sa vypíšu čísla od 0 po 99, avšak nemusia byť v poradí - závisí to od plánovania threadov.

```
class ThreadPool
  def initialize(size)
    @jobs = Queue.new
    @pool = Array.new(size) do
      Thread.new do
        loop do
          job, args = @jobs.pop
          job.call(*args)
        end
      end
    end
  end

  def schedule(*args, job)
    @jobs << [job, args]
  end
end

tp = ThreadPool.new(10)
100.times { |i| tp.schedule(Proc.new { puts i }) }
```

Listing 3.4: Minimalistický thread pool

[1]zdroj: http://en.wikipedia.org/wiki/Thread_pool_pattern

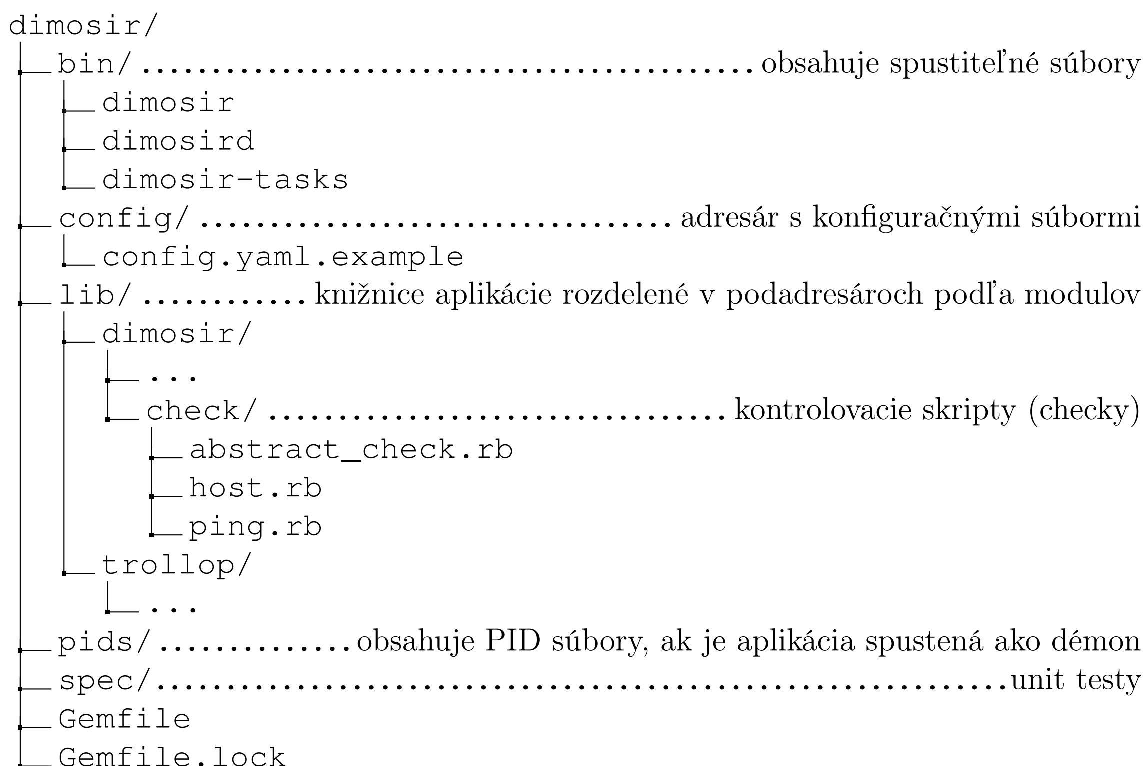
Kapitola 4

Implementácia

Táto kapitola sa venuje implementácii aplikácie. Konkrétne, popíšeme si úlohy jednotlivých tried, detailne vysvetlíme fungovanie a správanie celej aplikácie, pozrieme sa na zaujímavé časti kódu a rozoberieme rôzne problémy, na ktoré sme počas implementácie narazili.

4.1 Popis štruktúry aplikácie

Organizácia súborovej štruktúry sa opiera o Ruby konvencie a vyzerá nasledovne:



4.2 Databáza

Pri práci s databázou využívame knižnicu *MongoMapper*[1], ktorá mapuje dokumenty z databázy na objekty v Ruby.

4.2.1 Popis kolekcií

Databáza obsahuje tri kolekcie, *peers*, *tasks* a *jobs*.
Peers drží zoznam peerov, ktorí sú v systéme. Atribúty každého peera sú:

- *_id* - vnútorné id databázy
- *ip* - IP adresa peera
- *port* - port, na ktorom peer počúva a čaká na správy od ostatných peerov

Tasks je zoznam taskov, tj predpisov pre joby (jednotlivé kontroly). Atribúty tasku:

- *_id* - vnútorné id databázy
- *label* - popis tasku. Musí byť unikátny (tj v databáze nemôžu existovať dva tasky s rovnakým *labelom*).
- *target_host* - adresa alebo hostname cieľového servera, ktorý sa kontroluje
- *check* - kontrola, ktorá sa má vykonávať. K nej prislúcha súbor v adresári *lib/dimosir/check*.
- *periodicity* - periodicita v sekundách v akej sa daný task vykonávať
- *peer_id* - referencia na peera, ktorý má task na starosti (tj generuje z neho joby každých p sekúnd, kde p je periodicita)
- *arguments* - prípadné prídavné argumenty pre check

Atribúty *jobov*:

- *_id* - vnútorné id databázy
- *done* - flag indikujúci, či bol daný job už vykonaný
- *done_time* - čas dokončenia jobu alebo *nil*
- *output* - textový výstup checku (kontrolovacieho skriptu)
- *exitstatus* - návratová hodnota checku
- *task_label* - unikátny popis tasku
- *scheduled* - flag indikujúci, či bol daný job už naplánovaný
- *alerted* - flag indikujúci, či bol prípadný neúspech tohto jobu už ohlásený
- *task_id* - referencia na task, z ktorého je tento job odvodený
- *peer_id* - referencia na peera, ktorý má tento job na starosti

[1] `http://mongomapper.com/`

4.3 Popis tried

Main

Štartovacia trieda, ktorá nakonfiguruje a vytvorí všetky ostatné objekty potrebné pre beh aplikácie.

Cmd

Parser commandline argumentov pre aplikáciu, volá knižnicu *Trollop*. Používa sa v prípade, že je aplikácia spúšťaná cez *bin/dimosir* (teda nie ako démon).

Config

Parser konfiguračných súborov v *yaml* formáte.

SimpleLogger

Jednoduchý logger, umožňuje logovať buď do súboru alebo na *stderr*. Podporuje 4 typy logov - *debug*, *info*, *warning* a *error*. V konštruktore je potrebné špecifikovať od ktorej úrovne majú byť správy logované. Jeho jedinú inštanciu na začiatku vytvorí *Main* a potom predáva v konštruktore všetkým ďalším vytváraným objektom.

DatabaseAdapter

Obaľovacia trieda nad databázou. V konštruktore s ňou iniciuje spojenie.

Peer

Ruby trieda mapujúca dokument z kolekcie *Peers*. Predstavuje peera - účastníka systému.

Task

Ruby trieda mapujúca dokument z kolekcie *Tasks*. Obsahuje metódu na vygenerovanie jobu podľa seba, ako šablóny.

Job

Ruby trieda mapujúca dokument z kolekcie *Jobs*. Obsahuje metódu *run*, ktorá vykoná job.

AbstractCheck

Trieda, od ktorej musia byť odvodené všetky checky.

Host

Predvytvorený check. Pomocou programu *host* kontroluje, či dokáže *target_host* preložiť dané doménové meno na IP adresu.

Ping

Predvytvorený check. Kontroluje, či *target_host* odpovedá na ping.

Loader

Dynamicky načítava súbory s checkmi.

Sender

Zabezpečuje posielanie správ iným peerom pomocou socketov.

Listener

Zabezpečuje prijímanie správ od ostatných peerov. Počúva na definovanom porte na prichádzajúce spojenia.

BullyElection

Implementácia voľby koordinátora pomocou *Bully algoritmu*[2]. Obsahuje metódu *start_ election* na začatie voľby a metódy na spracovanie jednotlivých typov správ, ktoré si peerovia počas voľby posielajú.

RRTaskScheduler

Implementácia rozdelenia taskov medzi peerov *round robin* algoritmom.

JobGenerator

Drží si zoznam taskov, ktoré ma na starosti daný peer a generuje z nich joby.

JobScheduler

V nekonečnej slučke si pýta ešte nenaplánované joby, pošle ich do fronty thread poolu na vykonanie a označí ich ako naplánované.

ThreadPool

Pri inicializácií si vytvorí thread pool danej veľkosti, ktorý potom vykonáva úlohy z fronty. Fungovanie tejto triedy vysvetlíme bližšie v časti 4.5.4 - Vykonávanie jobov - Thread pool.

[2]Viac o *Bully algoritme* v časti 3.1.1

Alerter

Periodicky kontroluje, či sú v databáze joby, ktoré neskončili úspešne. Ak áno, pošle upozornenie emailom a tieto joby označí ako ohlásené.

Kernel

Hlavná trieda. V konštruktore dostáva okrem iného aj:

- Implementáciu voľby koordinátora (napríklad inštanciu triedy *BullyElection*)
- Implementáciu algoritmu na rozdeľovanie taskov medzi peerov (napríklad inštanciu triedy *RRTaskScheduler*)

Jej úlohami sú hlavne:

- Ak je daný peer koordinátorom, tak periodicky kontroluje ostatných peerov, či sú živí.
- Vydanie pokynu *JobScheduleru* na znovunačítanie taskov
- Prerozdelenie taskov pomocou *TaskScheduleru*, ak sa zmenili tasky
- Spracovávanie prichádzajúcich správ od *Listenera*

4.4 Démonizácia

Na démonizáciu bežiaceho procesu pri spustení aplikácie cez *bin/dimosird* je použitá knižnica *Daemons*[3].
Ak by sme chceli proces démonizovať (v Unixovom prostredí) bez použitia knižnice, kód by vyzeral približne nasledovne:

```
1  # forknutie procesu a ukoncenie parenta
2  exit if fork
3  # nastavenie procesu ako session a group leadra, bez terminalu
4  Process.setsid
5  # dalsi fork a ukoncenie rodica. proces bude zaveseny pod init
6  exit if fork
7  # proces stratil terminal, preto je potrebne presmerovat
8  # stdin, stdout a stderr niekam inam (do /dev/null,
9  # pripadne do suboru)
10 STDIN.reopen "/dev/null"
11 STDOUT.reopen "/opt/dimosir/log.out", "a"
12 STDERR.reopen "/opt/dimosir/log.err", "a"
```

Listing 4.1: Ukážka démonizácie procesu v Ruby

[3] http://daemons.rubyforge.org/

Kvôli možnosti komunikácie s démonom je potrebné ešte uložiť číslo procesu, pod ktorým tento démon beží do PID súboru.
Ďalším riešením by bolo použitie linuxového programu *start-stop-daemon*, ten však nie je dostupný vo všetkých distribúciach.

4.5 Ukážky zaujímavých častí kódu

4.5.1 Práca s databázovým mapperom

Databázový mapper umožňuje skutočne jednoduchú a veľmi čitateľnú prácu s databázou.
Napríklad zoznam všetkých peerov (tj dokumentov z kolekcie *Peers*) sa získa volaním

```
Peer.all
```

Zoznam jobov, ktoré sú pridelené danému peerovi a ešte neboli naplánované, zoradené podľa času vytvorenia:

```
Job.all(
  :peer_id => peer.id,
  :scheduled => false,
  :order => :created_at.asc
)
```

4.5.2 Komunikácia medzi peermi

Peerovia komunikujú posielaním si správ cez TCP sockety. To znamená, že doručovanie správ je spoľahlivé a v poradí. Formát správy vyzerá nasledovne:

<peer_ sender><delimiter1><msg_ type><delimiter2><action>

Kde *peer_ sender* je objekt odosielateľa serializovaný do JSONu, *delimiter1* a *delimiter2* sú oddeľovače, *msg_ type* je typ správy (napríklad *election* alebo *kernel*). Takže napríklad pri zvolení nového koordinátora daný peer posiela všetkým ostatným správu

```
{"created_at":"2012-05-23T14:49:22Z",
 "id":"4fbcf8f216b17c327f000001",
 "ip":"192.1.1.1","port":10000,
 "updated_at":"2012-05-23T14:49:22Z"
}|election.leader
```

Odosielanie správ je realizované pomocou triedy *Sender*. Tá využíva na prácu so socketmi triedu *TCPSocket*, ktorá patrí medzi štandardné knižnice Ruby.

```
def send_msg(peer_to, msg)
  no_error = true
```

```
3     begin
4       socket = TCPSocket.new(peer_to.ip.to_s, peer_to.port.to_i)
5       socket.print("#{@peer_sender.to_json}|#{msg}#{DELIMITER}")
6     rescue => e
7       log(ERROR, "Error sending message.\n\terror msg: #{e.message}")
8       no_error = false
9     ensure
10      socket.close unless socket.nil?
11    end
12    no_error
13  end
```

Listing 4.2: Posielanie správ cez sockety

Prijímanie správ zabezpečuje trieda *Listener*, tá využíva štandardnú triedu *TCPserver*. Pre každé nadviazané spojenie je vytvorený nový thread, ktorý spojenie obslúži. Prijatú správu potom posunie ďalej na spracovanie.

```
1   @server = TCPServer.open(@port)
2   loop do
3     Thread.new(@server.accept) do |connection|
4       begin
5         data = connection.gets(DELIMITER)
6         unless data.nil?
7           data.chomp! # removes \n \r ...
8           data.chop! if data[-1] == DELIMITER # removes last char
9         end
10        peer = Peer.new_from_json(data.split("|", 2).first)
11        msg = data.split("|", 2).last
12        @router.consume_message(peer, msg)
13      rescue Exception => e
14        log(ERROR, "Error receiving message\n\tError: #{e.class}\n" +
15          "\tErrormsg:#{e.message}")
16      ensure
17        connection.close unless connection.nil?
18      end
19    end
20  end
```

Listing 4.3: Prijímanie správ cez sockety

4.5.3 Aplikácia návrhového vzoru Strategy

Trieda *Main* predáva konštruktoru triedy *Kernel* implementáciu voľby koordinátora a implementáciu prerozdeľovania taskov peerom.

```
1 scheduler = RRTaskScheduler.new(@logger)
2 election  = BullyElection.new(@logger,db, sender, peer_self)
3 kernel    = Kernel.new(@logger, db, sender, peer_self, election,
4                         scheduler, job_generator, job_executor,
5                         alerter)
```

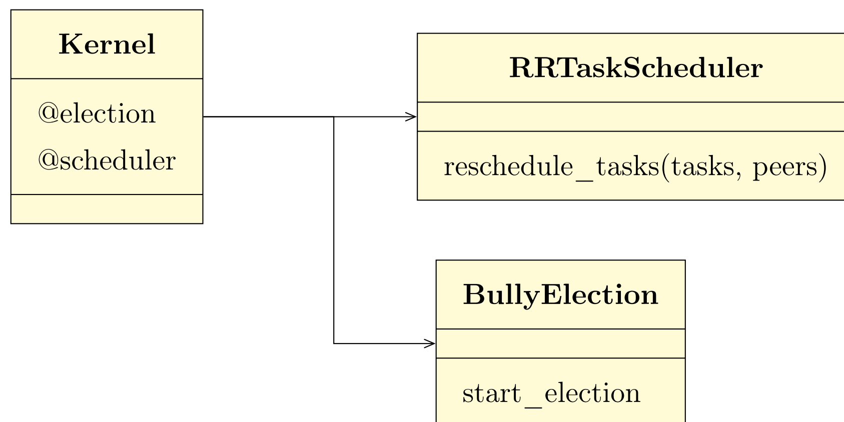
Listing 4.4: Použitie návrhového vzoru strategy (main.rb, riadok 77)

Voľba koordinátora

Trieda *Kernel* v prípade iniciovania voľby volá metódu objektu *election*, *start_ election*. Táto konkrétna implementácia je teda veľmi ľahko zameniteľná za inú implementáciu voľby koordinátora. Akurát na tomto mieste sa uplatňuje konvencia 'duck-typing over inheritance' - rôzne implementácie voľby koordinátora nemajú spoločného predka, ale ak obe majú metódu *start_ election*, tak sú to platné implementácie.

Prerozdelenie taskov

Trieda *Kernel* takisto dostáva v konštruktore implementáciu prerozdeľovania taskov medzi peerov. Táto trieda je opäť ľahko zameniteľná. Stačí na to trieda, ktorá bude implementovať metódu *reschedule(tasks, peers)*.



Obr. 4.1: Použitie vzoru Strategy - diagram tried

4.5.4 Vykonávanie jobov - Thread pool

V nasledujúcej ukážke je implementácia thread poolu.

```ruby
module Dimosir
  class ThreadPool
    include Loggable
    def initialize(l, size)
      set_logger(l)
      @size = size
      @jobs = Queue.new
      @pool = Array.new(@size) do |i|
        Thread.new do
          Thread.current[:id] = i
          catch(:exit) do
            loop do
              job, args = @jobs.pop
              job.call(*args)
            end
          end
        end
      end
      log(DEBUG, "Thread pool initialized with size #@size")
    end

    def schedule(*args, job)
      log(DEBUG, "New proc scheduled. #{@jobs.size} jobs in queue")
      @jobs << [job, args]
    end

    def shutdown
      @size.times do
        schedule Proc.new { throw :exit }
      end
      @pool.map(&:join)
    end
  end
end
```

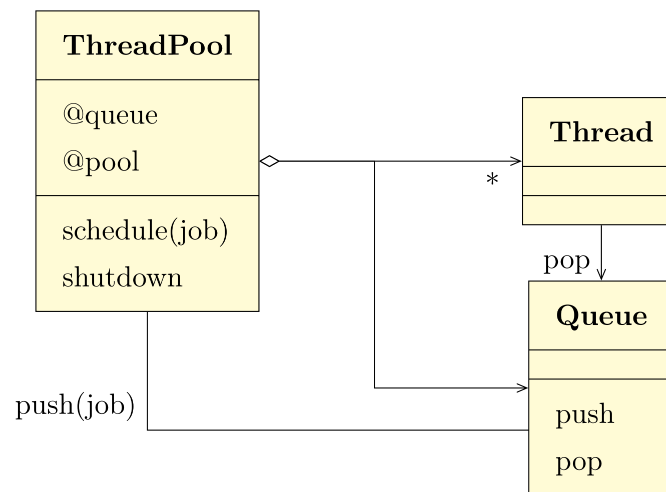
Listing 4.5: Implementácia thread poolu (thread_pool.rb)

Trieda *Main* vytvorí inštanciu triedy *ThreadPool*, ktorú potom predá cez konštruktor inštancii triedy *JobExecutor*. Tá potom plánuje inštancii triedy *ThreadPool* úlohy

pomocou

```
threadpool.schedule Proc.new { ... }
```

Podstatným prvkom tejto implementácie je *@queue*, čo je inštancia triedy *Queue*. Tá je v Ruby implementovaná ako synchronizovaná fronta. Volanie *@queue.pop* je blokujúce, to znamená, že ak *@queue* - fronta úloh je prázdna, tak thready čakajú na tomto volaní a zbytočne nemíňajú systémové prostriedky.
Metóda *shutdown* naplánuje 'vypnutie' thread poolu. A to tak, že naplánuje *@size* úloh na ukončenie threadov. Tie sa ale dostanú na rad až keď sa spracujú všetky predchádzajúce úlohy.



Obr. 4.2: Použitie vzoru Thread pool - diagram tried

4.5.5 Použitie mutexov

V ukážke je časť triedy *JobGenerator*. Po vytvorení inštancie je zvonku zavolaná metóda *start*, ktorá v osobitnom threade pustí *start_generating* tj generovanie jobov z taskov. Ak však 'zvonku' príde pokyn na znovunačítanie zoznamu taskov (cez metódu *reload_tasks*, riadok 25), tak bez použitia mutexov by hrozilo, že počas vykonávania *each* loopu (riadok 14) by sa obsah premennej *@tasks* zmenil, čo by mohlo mať nepredvídateľné následky.

```
def initialize(...)
  ...
  @tasks_mutex   = Mutex.new
end

def start
  reload_tasks
  Thread.new { start_generating }
end
```

```
10
11 def start_generating
12   loop do
13     @tasks_mutex.synchronize do
14       @tasks.each do |task, last_run|
15         next if Time.now.to_i - last_run < task.periodicity
16         log(DEBUG, "Generating job for task #{task.label}")
17         task.generate_job(@peer_self)
18         @tasks[task] = Time.now.to_i
19       end
20     end
21     sleep(SLEEP_TIME)
22   end
23 end
24
25 def reload_tasks
26   tasks_new = {}
27   ...
28   @tasks_mutex.synchronize { @tasks = tasks_new }
29 end
```

Listing 4.6: Ukážka použitia mutexov (job_generator.jb)

4.5.6 Použitie MapReduce

Trieda *DatabaseAdapter* obsahuje metódu na výpočet percentuálnej úspešnosti vykonávania jednotlivých taskov cez MapReduce[4]. Podstata spočíva v tom, že pre každý task chceme vyjadriť pomer úspešných vykonaných jobov oproti všetkým vykonaným jobom (úspešné joby sú také, ktorých *exitstatus* je rovný 0). Na listingu 4.7 vidíme funkciu *map*:

```
1 function() {
2   if (this.done) {
3     emit(this.task_label, {successful: (this.exitstatus == 0)});
4   }
5 }
```

Listing 4.7: Použitie MapReduce, funkcia map (database_adapter.rb, riadok 138)

Tá každému vykonanému jobu priradí dvojicu popis tasku (od ktorého je job odvodený) a *successful: true*, ak bol job úspešný. Ak nebol úspešný, tak mu priradí *successful: false*.

[4] V MongoDB sa funkcie *map* a *reduce* píšu v jazyku Javascript a predávajú serveru na vykonanie

Nevykonané joby budú ignorované. Funkcia reduce dostáva na vstupe *key*, čo je v našom prípade popis tasku a *values* - pole asociatívnych polí, z ktorých každé obsahuje len jedinú hodnotu. For cyklom prejdeme všetky hodnoty a pre danú úlohu zrátame počet hodnôt *true* a *false* pod kľúčom *successful*.

```
function(key, values) {
  var successful = 0;
  var unsuccessful = 0;
  for (var i = 0; i < values.length; i++) {
    if (values[i]["successful"]) {
      successful++;
    } else {
      unsuccessful++;
    }
  }
  return {
    task_label: key,
    successful: NumberInt(successful),
    unsuccessful: NumberInt(unsuccessful),
    success: (successful/(successful+unsuccessful)*100)
  }
}
```

Listing 4.8: Použitie MapReduce, funkcia reduce (database_adapter.rb, riadok 145)

Výsledkom celého MapReduce je teda pole asociatívnych polí, pre každý task jedno. Každé obsahuje popis tasku, počet úspešných a počet neúspešných vykonaní jobov a nakoniec percentuálnu úspešnosť vykonávania jobov.

4.5.7 Ukážka unit testov

Poznámka na úvod - Ruby síce má priamo v štandardnej knižnici podporu pre unit testing cez triedy modulu *Test::Unit*, no my používame knižnicu *RSpec*[5]. Príklad unit testov ilustrujeme na triede *ThreadPool*, ktorej zdrojový kód sme uvideli v listingu 4.5. Pozrieme sa na konkrétny test testujúci metódu *schedule*.

```
describe Dimosir::ThreadPool do
  describe "#schedule" do
    it "schedules single job and executes it" do
      job = double("job")
      job.should_receive(:run).once
      p = Proc.new { |j| j.run }
```

[5] http://rspec.info/

```
7        @pool.schedule(job, p)
8        @pool.shutdown
9      end
10   end
11 end
```

Listing 4.9: Ukážka unit testu (thread_pool_spec.rb, riadok 32)

Knižnica *RSpec* využíva pri písaní testov vlastný DSL[6] - preto sa kľúčové slová *describe*, *before* a *it* správajú ako metódy berúce ako parametere názov testu a blok kódu. Každý *it* blok predstavuje jednotlivý test, *describe* bloky *opisujú* súbory testov (podľa metódy, triedy, atď). *It* blok začína slovným popisom, ako by sa mala trieda v danom prípade správať. Potom nasleduje samotný blok kódu. Predpokladajme, že v premennej *@pool* máme uloženú inštanciu testovanej triedy. Chceme testovať to, že ak pomocou metódy *schedule* naplánujeme nejakú úlohu, tak očakávame, že niekedy bude zavolaná metóda *run* tejto úlohy. Úlohu si teda *napodobníme* - vytvoríme takzvaný *mock object*, (pomocou metódy *double* testovacej knižnice) a tejto *napodobenine* povieme, že má očakávať, že na nej bude práve raz zavolaná metóda *run*.

4.6 Problémy pri implementácii

Pri implementácií sme sa nestretli s vážnejšími problémami. Jednou z nevýhod Ruby ale je, že je to relatívne mladý programovací jazyk a mnoho knižníc nemá úplne kompletnú dokumentáciu - v tom prípade je dokumentáciou samotný zdrojový kód knižnice.
Malá nepríjemnosť nastala, keď kvôli chybe v aplikácii prestalo spoľahlivo fungovať posielanie správ medzi peermi - niekoľkí peerovia sa naraz vyhlasovali za koordinátorov. Chyba bola nakoniec v istom threade, ktorý vo svojej nekonečnej slučke nemal príkaz *sleep* (ani žiadne blokujúce volanie), tým pádom tento thread vyťažoval procesor na 100%. Kvôli tomu thread spracovávajúci prichádzajúce správy dostával oveľa menej procesorového času a z toho vyplynulo, že správa buď vôbec nedošla, alebo došla s meškaním. Preto si niektorí peerovia mysleli, že ostatní sú 'spadnutí' a vyhlásili sa za koordinátorov.
Ďalšia chyba, ktorej odhalenie zabralo nemalý čas bolo, keď aplikácia pustená vo forme démona odrazu skončila bez chybovej hlášky v logu. Keď bola pustená ako normálny proces, tak skončila tiež s chybou, ale na konzolu sa vypísala výnimka a jej stack trace. Problémom bolo, že *stdout* procesu ako démona bol síce presmerovaný do súboru a teda logovanie fungovalo, no nechytené výnimky sa implicitne vypisujú na *stderr*. A *stderr* smeroval do */dev/null*, keďže pri démonizácii proces príde o terminál. Riešenie bolo teda jednoduché - presmerovať aj *stderr* do súboru.

[6]domain specific language

4.7 Možné zlepšenia do budúcnosti

Dokumentácia

Plne zdokumentovať kód podľa existujúcich štandardov, aby bolo možné vygenerovať pomocou *RDoc*, resp. *YARD* HTML dokumentáciu pre aplikáciu.

Testovanie

Pokryť unit testami čo najväčšie percento kódu aplikácie, aby bolo možné ľahko odhaľovať prípadné chyby pri úpravách kódu. Takisto by bolo dobré napísať komplexnejšie integration (application) testy pokrývajúce celú funkcionalitu aplikácie.

Init.d/Upstart skripty

Napísať *init.d* a *upstart* skripty s využitím programu *start-stop-daemon* určené na jednoduchšiu prácu s démonizovanou aplikáciu - tzn démon by sa ovládal pomocou */etc/init.d/dimosir <command>* resp. *service dimosir <command>*, tak, ako klasické linuxové démony.

Administračná utilita

Ďalším možným zlepšením by bolo naprogramovanie administračnej utility s grafickým prostredím, ktorá by vedela spravovať tasky, komunikovať s démonmi, vedela by vizualizovať rôzne štatistiky pomocou grafov, napríklad dostupnosť jednotlivých monitorovaných serverov, úspešnosť jobov daných taskov atď.

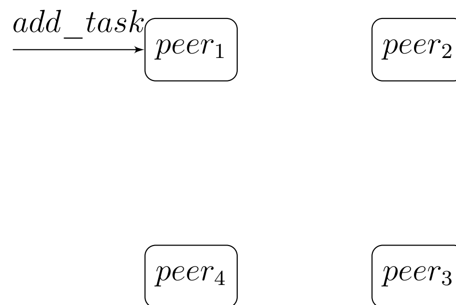
Zlepšený alerting

Vhodné by bolo aj zlepšiť spôsob upozorňovania na výpadky. Alternatívnym spôsobom ku posielaniu emailov by bolo posielanie sms správ (napríklad pomocou Skype API). Administrátor by dostal sms správu o výpadku a v prípade, že by na ňu odpovedal textom napríklad *ACK* (z angl. *acknowledged*), prestali by mu chodiť sms správy upozorňujúce na zlyhanie danej kontroly (alebo nedostupnosti daného servera).

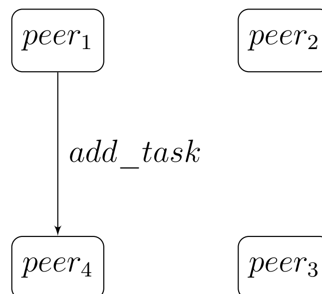
4.8 Návrh implementácie bez centrálnej databázy

Ak by mal celý systém fungovať bez databázy, tak by si musel každý peer držať všetky informácie - zoznam ostatných peerov a zoznam všetkých taskov (tu teda predpokladáme, že každý peer má nejakú lokálnu databázu, ktorá prežije jeho výpadky. Môže to byť pokojne aj súbor). Tieto dáta môže výhradne spravovať koordinátor a ostatným ich len distribuovať. To znamená, že napr. pridávanie tasku by bolo realizované poslaním daného tasku koordinátorovi a ten by (po uložení tasku do svojej lokálnej databázy) tento task ďalej rozposlal ostatným peerom (obdobne by fungovalo pripojenie nového peera do systému). Generovanie jobov by fungovalo podobne ako pri riešení s centrálnou databázou - každý peer by si lokálne generoval joby z jemu pridelených taskov.

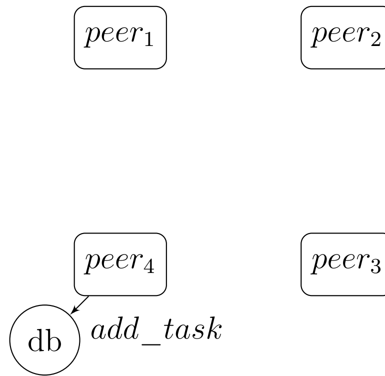
Joby by ale po vykonaní neukladal do databázy, ale posielal koordinátorovi, ktorý by ich potom rozposlal ostatným.
Zjavným problémom je eventuálna inkonzistencia údajov v prípade, že peer pošle koordinátorovi informácie o svojich vykonaných joboch a ten spadne skôr, ako ich stihne rozdistribuovať všetkým ostatným peerom. Povinnosť novozvoleného koordinátora by teda bola hneď po štarte vypýtať si od každého peera zoznam *ním* vykonaných jobov. Tie by si koordinátor lokálne uložil a následne rozposlal ostatným.
Ďalším problémom by bolo, keby koordinátor vypadol akurát v momente, keby upravoval ostatným zoznam peerov alebo zoznam taskov. V tomto prípade potrebujeme vedieť, ktorý peer ma novšiu verziu, preto potrebujeme zaviesť časové pečiatky. Tie by fungovali tak, že stále, keď koordinátor posiela nový zoznam peerov alebo taskov ostatným, tak tento záznam dostane časovú pečiatku.
Uveďme príklad. Majme systém so štyrmi peermi, A, B, C a D. Nech A je koordinátor. Nastala situácia, že peer B požiadal o pridanie nového tasku. V tom prípade si koordinátor najprv lokálne uloží tento task a následne nový zoznam pošle ostatným. Ak však koordinátor 'spadne' v momente, keď peerovi B už poslal nový zoznam a peerom C a D ešte nie, tak nastáva konflikt. Ak však nový koordinátor vidí, že zoznam taskov u peera B má novšiu časovú pečiatku než zoznam u peerov C a D, tak vie jednoznačne povedať, ktorý je správny. Zvyšok by fungoval rovnako, ako pri riešení s centrálnou databázou.



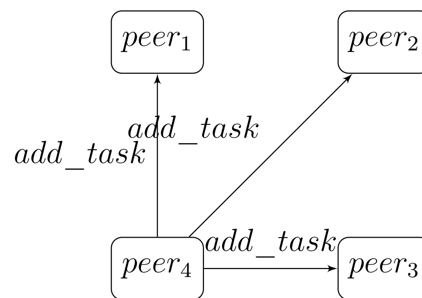
Obr. 4.3: Zvonku prichádza požiadavka na pridanie nového tasku $peerovi_1$.



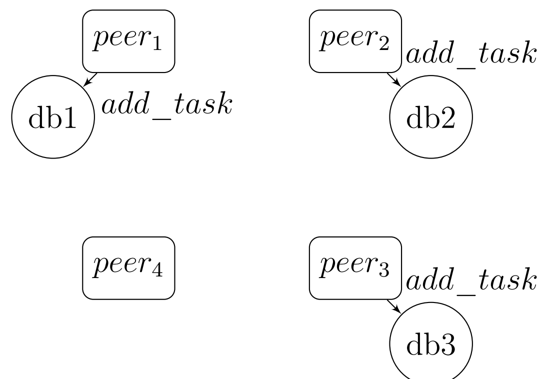
Obr. 4.4: $Peer_1$ preposiela túto požiadavku aktuálnemu koordinátorovi, tj $peerovi_4$.



Obr. 4.5: $Peer_4$ si task uloží do lokálnej databázy.



Obr. 4.6: Po uložení do lokálnej databázy rozpošle nový task $peer_4$ ostatným peerom



Obr. 4.7: Ostatní peerovia si po prijatí uložia task do lokálnej databázy.

Kapitola 5

Dokumentácia

5.1 Inštalácia

5.1.1 Prerekvizity

Aplikácia potrebuje pre svoj beh Ruby od verzie 1.9.3 a MongoDB od verzie 2.0.5. Návod na inštaláciu Ruby je napríklad na stránke `http://www.ruby-lang.org/en/downloads/`, návod na inštaláciu MongoDB je možné nájsť na `http://www.mongodb.org/display/DOCS/Quickstart+Unix`. Predpokladajme teda, že máme funkčnú inštaláciu Ruby a MongoDB bežiace na preddefinovanom porte 27017.

5.1.2 Aplikácia

Aplikáciu je možné stiahnuť na stránke `https://github.com/kuboj/dimosir`. Odporúčaný spôsob je naklonovať si GIT repozirár z URL `git://github.com/kuboj/dimosir.git`.

```
1  $ git clone git://github.com/kuboj/dimosir.git
2  Cloning into 'dimosir'...
3  remote: Counting objects: 569, done.
4  remote: Compressing objects: 100% (218/218), done.
5  remote: Total 569 (delta 326), reused 568 (delta 325)
6  Receiving objects: 100% (569/569), 85.52 KiB, done.
7  Resolving deltas: 100% (326/326), done.
8  $ cd dimosir && ls -1F
9  bin/
10 config/
11 Gemfile
12 Gemfile.lock
13 lib/
```

```
LICENSE
Rakefile
README
spec/
```

Naklonovanie GIT repozitára aplikácie

Ďalším krokom je inštalácia chýbajúcich Gemov

```
$ bundle install
```

Tým je samotná inštalácia dokončená.

5.2 Konfigurácia

Ďalším krokom je vytvorenie konfiguračného súboru podľa vzorového súboru *config/config.yaml.example*.

```
$ cp config/config.yaml.example config/config.yaml
$ cat config/config.yaml
---
database:
  host: localhost
  port: 27017
  db_name: dimosir
  user: dimosir
  password: eh554h4u3yhe
peer:
  ip: 127.0.0.1
  port: 10000
logging:
  log_file: /tmp/dimosir/dimosir.log
  log_level: 1
performance:
  thread_pool_size: 5
mail:
  from: peer1
  to: bubo47@gmail.com
  via: smtp
  via_options:
    address: smtp.gmail.com
    port: 587
```

```
25      enable_starttls_auto: true
26      user_name: dimosir1@gmail.com
27      password: rh36sdg4
28      authentication: plain
29      domain: localhost.localdomain
```

Listing 5.1: Vzorová konfigurácia

Popis parametrov v konfiguračnom súbore:

- *database*
  - *host* - adresa databázového servera
  - *port* - port, na ktorom je databáza pustená
  - *db_name* - meno databázy
  - *user* - užívateľ platný pre túto databázu
  - *password* - heslo
- *peer*
  - *ip* - IP adresa, pod ktorou má aplikácia vystupovať navonok
  - *port* - port, na ktorom bude aplikácia počúvať a prijímať správy od ostatných peerov
- *logging*
  - *log_file* - cesta k súboru, kam sa bude zapisovať log
  - *log_level* - hranica vážnosti správ, ktoré majú byť logované. 1 znamená, že budú logované všetky správy vrátane debuggovacích, hodnota 2 neloguje debuggovacie výpisy, iba správy typu info, warn a error, 3 - warn, error, 4 - iba error hlášky.
- *performance*
  - *thread_pool_size* - počet threadov v thread poole, ktorý vykonáva joby.
- *mail* - obsahuje konfiguráciu na emailové posielanie upozornení. Táto konfigurácia je predávaná knižnici *Pony*, detaily konfigurácie je možné nájsť na `https://github.com/benprew/pony`. V ukážke je nastavenie pre posielanie emailov cez server Gmailu.

5.3 Spúšťanie

V adresári *bin* sa nachádzajú 3 spustiteľné súbory:

- *dimosir* - používa sa na spustenie aplikácie v popredí, logovanie nie je presmerované do súboru, ale priamo do konzoly. Nepovinným parametrom *-c* je možné špecifikovať konfiguračný súbor, ktorý sa má použiť - implicitne je braný súbor *config/config.yaml*.
- *dimosird* - má na starosti spúšťanie, zastavovanie a reštartovanie aplikácie ako démona. Cesta ku konfiguračnému súboru je v tomto prípade napevno *config/config.yaml*.
- *dimosir-tasks* - predstavuje jednoduchú administráciu taskov. Umožňuje vypísať zoznam taskov, pridávať, odoberať tasky, poslať signál systému na ich znovunačítanie a jednoduché zobrazenie štatistík. Prístupové údaje k databáze číta zo súboru *config/config.yaml*.

5.4 Managovanie taskov

5.4.1 Pridanie nového tasku

Realizuje sa pomocou príkazu

```
dimosir-tasks add <label> <check> <periodicity> <target> <args>
```

- *label* je unikátny popis tasku
- *check* je názov kontrolovacieho skriptu, ktorý sa má vykonávať
- *periodicity* predstavuje periodicitu vykonávania tasku
- *target* je adresa alebo hostname cieľového servera, ktorý sa má kontrolovať
- *args* je JSON predstavujúci prídavné argumenty pre kontrolovací skript

Pridanie nového tasku, ktorý by zabezpečoval pingnutie servera s IP 172.17.21. každých 60 sekúnd by vyzeralo nasledovne:

```
$ ./bin/dimosir-tasks add testing_task ping 60 172.17.21.1 {}
-> Task added.
```

5.4.2 Vypísanie zoznamu taskov

Na to slúži príkaz

```
dimosir-tasks list
```

Ukážka:

```
$ ./bin/dimosir-tasks list
testing_task - ping, 60, 172.17.21.1
testing_task2 - host, 600, myserver.com
-> Total 2 tasks.
```

5.4.3 Vymazanie tasku

Task sa maže príkazom

```
dimosir-tasks remove <label>
```

Kde *label* je unikátny popis tasku.
Príklad:

```
$ ./bin/dimosir-tasks remove testing_task2
-> Task 'testing_task2' removed.
```

5.4.4 Znovunačítanie taskov

To je možné úspešne vykonať iba ak lokálne beží démon aplikácie. Znovunačítanie sa púšťa pomocou príkazu

```
dimosir-tasks reload
```

Lokálnemu peerovi je poslaná informácia o zmene taskov a ten ju prepošle koordinátorovi, ktorý nanovo prerozdelí tasky. Tento príkaz by mal byť pustený po každom pridaní respektíve odobraní tasku.

5.5 Checky

Checky (kontrolovacie skripty) sa nachádzajú v zložke *lib/dimosir/check*. Pole *check* ľubovoľného tasku referuje na práve na jeden zo súborov v tejto zložke. V aplikácii sú predvytvorené dva checky, konkrétne *ping* a *host*.

5.5.1 Pridanie nového checku

Pre pridanie nového checku je potrebné vytvoriť novú triedu, ktorá bude dediť od triedy *AbstractCheck* a uložiť do súboru s názvom identickým ako meno triedy (samozrejme, s príponou .rb). V tejto novovytvorenej triede je potrebné implementovať metódu *perform_check*. Výstupom musí byť pole s dvomi hodnotami - textovým výstupom a návratovou hodnotou checku. Návratová hodnota 0 sa vyhodnocuje ako úspešných check, nenulové hodnoty predstavujú chybový kód. Trieda ma k dispozícii premenné *@target_host* a *@arguments*, ktoré obsahuje daný task, ktorý check iniciuje.

Check, ktorý by zisťoval, či je na danom serveri otvorený daný port by vyzeral nasledovne:

```
module Dimosir
  module Check
    class Portcheck < AbstractCheck
      def perform_check
        host = @target_host
        port = @arguments["port"]
        timeout = @arguments["timeout"]
        output = `nc -zv -w #{timeout} #{host} #{port} 2>&1`
        retval = $?.exitstatus

        [output, retval]
      end
    end
  end
end
```

Listing 5.2: Ukážka checku kontrolujúceho otvorenosť portu

5.5.2 Distribúcia checkov medzi peerov

Po vytvorení checku (kontrolovacieho skriptu) je ho potrebné ešte rozdistribuovať medzi všetkých peerov. To je, samozrejme, možné cez *scp* alebo *ftp*, no odporúčaným (a lepšie spravovateľným a automatizovateľným) spôsobom je vytvoriť si osobitný GIT repozitár pre tieto *check* súbory a naklonovať ho do *lib/dimosir/check*. Po pridaní nového *check* súboru už len stačí pustiť *git pull* u každého peera (to sa dá zautomatizovať napríklad pomocou cronu). Workflow pridávania nového checku by teda vyzeral nasledovne:

```
$ cd /opt/dimosir/lib/dimosir/check
$ vim mynewcheck.rb
$ git add mynewcheck.rb
$ git commit -m 'new check added'
$ git push
```

A potom u každého peera:

```
$ cd /opt/dimosir/lib/dimosir/check && git pull
```

Navyše peera nie je potrebné reštartovať, keďže checky sú načítavané dynamicky.

5.6 Tutorial

V tejto časti si krok po kroku ukážeme na konkrétnom príklade, ako aplikáciu nastaviť na jednoduché monitorovanie niekoľkých serverov. Monitorovacími peermi budú servre s IP adresami 172.1.100.1 a 172.1.100.2. Predpokladajme, že na oboch serveroch máme nainštalované Ruby aj samotnú aplikáciu a nakonfigurovaný program *sendmail*. MongoDB nám beží na adrese 172.1.100.3.
Cieľom bude monitorovať funkčnosť dvoch serverov:

- server č.1 je databázový MySQL server. Jeho IP je 172.1.1.1. Chceme kontrolovať jednak, či je server dostupný (tj či odpovedá na *ping* a tiež, či má otvorený port 3306, na ktorom by mala bežať MySQL databáza.
- server č.2 bude DNS server s IP 172.1.1.2. Potrebujeme kontrolovať odpovedanie na *ping* a tiež či vie prekladať hostnames na IP adresy.

V prípade, že niektorý z checkov zlyhá chceme, aby nám došiel email s informáciou na *admin@admin.com*.

5.6.1 Konfigurácia

```
1 $ ssh 172.1.100.1
2 $ cd /opt/dimosir/
3 $ cp config/config.yaml.example config/config.yml
4 $ vim config/config.yaml
```

Konfiguračný súbor upravíme nasledovne:

```
1  ---
2  database:
3    host: 172.1.100.3
4    port: 27017
5    db_name: dimosir
6    user: dimosir
7    password: eh554h4u3yhe
8  peer:
9    ip: 172.1.100.1
10   port: 10000
11 logging:
12   log_file: /tmp/dimosir/dimosir.log
13   log_level: 1
14 performance:
15   thread_pool_size: 10
16 mail:
```

```
17    from: peer1
18    to: admin@admin.com
19    via: sendmail
```

Podobne nakonfigurujeme druhého peera (172.1.100.1), potrebujeme zmeniť jedine riadky *peer:ip* a *mail:from*.

5.6.2 Spustenie

Aplikáciu ako démona spustíme príkazom:

```
1  $ /opt/dimosir/bin/dimosird start
```

5.6.3 Tasky

Dostupnosť oboch serverov budeme kontrolovať pomocou checku *ping*, prekladanie hostnames pomocou *host*. Oba tieto checky sú už v aplikácii preddefinované. Otvorenosť portu budeme kontrolovať pomocou nového checku, nazvime ho *portcheck*. Návod na pridanie sme uviedli v 5.5.1 - Pridanie nového checku. Pripojíme sa na jedného z peerov a pridáme tasky:

```
1  $ ssh 172.1.100.1
2  $ cd /opt/dimosir/
3  $ ./bin/dimosir-tasks add server1_ping ping 60 172.1.1.1 {}
4  -> Task added.
5  $ ./bin/dimosir-tasks add server1_port portcheck 60 172.1.1.1
6  {'timeout':5,'port':3306}
7  -> Task added.
8  $ ./bin/dimosir-tasks add server2_ping ping 60 172.1.1.2 {}
9  -> Task added.
10 $ ./bin/dimosir-tasks add server2_host1 host 300 172.1.1.2
11 '{"lookup":"myserver.com"}'
12 -> Task added.
13 $ ./bin/dimosir-tasks add server2_host2 host 300 172.1.1.2
14 '{"lookup":"google.com"}'
15 -> Task added.
```

Vysvetlenie taskov:

- server1_ping - každých 60 sekúnd pustí check (kontrolovací skript) *ping* na server 172.1.1.1.
- server1_port - každých 60 sekúnd skontroluje otvorenosť portu 3306 pomocou checku *portcheck* na serveri 172.1.1.1, s timeoutom 5 sekúnd.

- server2_ping - každých 60 sekúnd pingne server 172.1.1.2.
- server2_host1 - každých 5 minút sa spýta servera 172.1.1.2 na IP adresu hosta *myserver.com*.
- server2_host2 - každých 5 minút sa spýta servera 172.1.1.2 na IP adresu hosta *google.com*.

Skontrolujeme pridané tasky:

```
$ ./bin/dimosir-tasks list
server1_ping - ping, 60, 172.1.1.1, {}
server1_port - portcheck, 60, 172.1.1.1,{"timeout"=>5,"port"=>3306}
server2_ping - ping, 60, 172.1.1.2, {}
server2_host1 - host, 300, 172.1.1.2, {"lookup"=>"myserver.com"}
server2_host2 - host, 300, 172.1.1.2, {"lookup"=>"google.com"}
-> Total 5 tasks.
```

Pošleme informáciu lokálnemu démonovi o zmene taskov:

```
$ ./bin/dimosir-tasks reload
-> Sending USR1 signal to process 9682 ...
```

Tým sa nové tasky rozdelia medzi peerov a začne sa monitorovanie.

Záver

Úspešne sa nám podarilo naprogramovať jednoduchý distribuovaný monitorovací systém napísaný v jazyku Ruby a zverejniť ho ako opensource na stránke `https://github.com/kuboj/dimosir`. Pri implementácii systému sme sa pokúšali držať sa konvencií Ruby.
Implementácia používa na voľbu koordinátora *bully algorithm*, ktorý sme popísali v časti 3.1.1 a niekoľko návrhových vzorov popísaných v časti 3.3.
Jednou z jeho nevýhod je ale akási 'falošná' distribuovanosť z dôvodu použitia centrálnej databázy. Ako by bolo potrebné implementáciu upraviť ak by sme sa chceli centrálnej databáze vyhnúť sme popísali v časti 4.8.
Do budúcnosti by sme chceli do aplikácie doimplementovať čo najviac zo zlepšení prezentovaných v časti 4.7, aby bola aplikácia čo najpoužiteľnejšia v praxi.

Literatúra

[Sin96] Sinha, Pradeep K., *Distributed Operating Systems: Concepts and Design*, Wiley-IEEE Press, 1996

[GM82] Garcia-Molina, H., *Elections in a Distributed Computing System*, IEEE Transactions on Computers, Vol. C-31 1982.

[Tan06] Tanenbaum, Andrew S. and Steen, Maarten van, *Distributed Systems: Principles and Paradigms (2nd Edition)*, Prentice-Hall, Inc., 2006.

[Dea04] Dean, Jeffrey and Ghemawat, Sanjay, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI, 2004.

[Ols07] Olsen, Russ, *Design Patterns in Ruby*, Addison-Wesley, 2007.

[DP95] Gamma, Erich and Johnson, Ralph and Helm, Richard and Vlissides, John, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[PR09] Thomas, Dave and Fowler, Chad and Hunt, Andy, *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, Pragmatic Bookshelf, 2009.

[Bro10] Brown, Gregory T. *Ruby Best Practices*, O'Reilly Media, 2010.

Príloha A

K práci je priložené CD so zdrojovým kódom aplikácie.