**UNIVERZITA KOMENSKÉHO V BRATISLAVE**

**FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY**

**Zálohovanie súborového systému do obsahom adresovaného úložiska**

**Bratislava 2015  
Michal Molec**

**UNIVERZITA KOMENSKÉHO V BRATISLAVE**

**FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY**

**Zálohovanie súborového systému do obsahom adresovaného úložiska**

**Diplomová práca**

Študijný program: Aplikovaná informatika   
Študijný odbor: 2511 Aplikovaná informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Školiteľ: Mgr. Michal Molec, PhD.

**Bratislava 2015  
Michal Molec**

Naskenované zadanie záverečnej práce

**Čestné vyhlásenie**

Čestne vyhlasujem, že som diplomovú prácu "Zálohovanie súborové systému do obsahom adresovaného úložiska" vypracoval samostatne s použitím uvedenej literatúry a zdrojov dostupných na internete.

V Bratislave dňa 1.6.2015

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Michal Molec

**Poďakovanie**

Chcel by som sa poďakovať vedúcemu diplomovej práce Mgr. Jánovi Kľukovi, PhD. za cenné rady a usmernenie pri vypracovaní tejto práce.

**Abstrakt**

MOLEC, Michal: Zálohovanie súborového systému do obsahom adresovaného úložiska - Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky, Katedra aplikovanej informatiky. Vedúci diplomovej práce: Mgr. Ján Kľuka, PhD. Bratislava, 2015.

Cieľom diplomovej práce je navrhnúť vhodné obsahom adresované úložisko dát (content-addressable storage) a implemetovať nad ním zálohovanie súborového systému unixového OS (napr. Linux), údržbu a prezeranie záloh a obnovu dát. Návrh samotného úložiska môže vychádzať z už existujúceho (napr. git), treba ho však upraviť s ohľadom na požadovaný účel. Očakáva sa kreatívne využitie (a prípadná úprava) vhodne zvolených techník kompresie (klasická bezstratová, rozdielová) a garbage collection, technológie FUSE, prípadne sledovania zmien v súborovom systéme. Implementácia by mala byť objektová, modulárna a rigorózne otestovaná (unit testing pokrývajúci funkcionalitu kritickú pre spoľahlivosť zálohovania). Preferovaným implementačným jazykom je python..

**Kľúčové slová:** Zálohovanie, FUSE, Rsync, Linux, Python

**Abstract**

MOLEC, Michal: File-system backup into content-addressable storage – Comenius University in Bratislava. Faculty of mathematics, physics and informatics, Department of aplied informatics. Dissertation supervisor: Mgr. Ján Kľuka, PhD. Bratislava, 2015.

The goal of this work is to design suitable content-addressable storage (CAS) and implement backup, backup maintenance and browsing, and data recovery of a Unix file system over the CAS. The design of the CAS may be based on existing ones (e.g., Git), it should, however, be adjusted for backup purposes. The student is expected to creatively choose, use, and adjust suitable methods of compression (standard lossless compression, delta compression) and garbage collection, the FUSE technology, and, possibly, the file-system event notification mechanism (inotify). The implementation should be object-oriented, modular, and rigorously tested (unit testing covering components critical to reliability of backup). The preferred implementation language is python.

**Keywords:** Backup, FUSE, Rsync, Linux, Python

**Obsah**

[Úvod 10](#_Toc417984950)

[1 Existujúce riešenia 11](#_Toc417984951)

[1.1 Software 11](#_Toc417984952)

[1.1.1 Rdiff-backup 12](#_Toc417984982)

[1.1.2 Duplicity 12](#_Toc417984983)

[1.1.3 AMANDA (Advanced Maryland Automatic Network Disk Archiver) 13](#_Toc417984984)

[1.1.4 Time Machine 14](#_Toc417984985)

[1.1.5 Git 15](#_Toc417984986)

[1.2 Použitá bakalárska práca 17](#_Toc417984987)

[1.3 Techniky 17](#_Toc417984988)

[1.3.1 Obsahom adresované úložisko (content addressed storage) 17](#_Toc417984989)

[1.3.2 Rsync 18](#_Toc417984990)

[1.3.3 FUSE 19](#_Toc417984991)

[1.3.4 Reference counting garbage collection 21](#_Toc417984992)

[1.3.5 CodernityDB 22](#_Toc417984993)

[1.4 GZIP 22](#_Toc417984994)

[1.5 Rdiff 23](#_Toc417984995)

[2 Návrh riešenia 24](#_Toc417984996)

[2.1 Východiská 24](#_Toc417984997)

[2.2 Obsahom adresované úložisko 25](#_Toc417984998)

[2.3 Prezeranie, údržba a obnova záloh 25](#_Toc417984999)

[2.3.1 FUSE 25](#_Toc417985000)

[2.3.2 Garbage kolekcia 26](#_Toc417985001)

[2.4 Využitie kompresie 26](#_Toc417985002)

[2.5 Žurnálovací systém 27](#_Toc417985003)

[3 Implementácia 28](#_Toc417985004)

[3.1 Obsahom adresované úložisko 28](#_Toc417985005)

[3.2 Backup 29](#_Toc417985006)

[3.2.1 NewBackup 30](#_Toc417985007)

[3.2.2 ExistingBackup 30](#_Toc417985008)

[3.3 Source 31](#_Toc417985009)

[3.3.1 BackupObject 31](#_Toc417985010)

[3.3.2 SourceObject 32](#_Toc417985011)

[3.3.3 SourceDir 33](#_Toc417985012)

[3.3.4 SourceFile 33](#_Toc417985013)

[3.3.5 SourceLnk 34](#_Toc417985014)

[3.4 Store 34](#_Toc417985015)

[3.4.1 Objekty 35](#_Toc417985016)

[3.5 Žurnálovací systém 38](#_Toc417985017)

[3.6 Zálohovanie dát 39](#_Toc417985018)

[3.6.1 Zálohovanie adresárov 40](#_Toc417985019)

[3.6.2 Zálohovanie súborov 40](#_Toc417985020)

[3.6.3 Zálohovanie odkazov 43](#_Toc417985021)

[3.7 Údržba a odstraňovanie dát 44](#_Toc417985022)

[3.7.1 Garbage kolekcia 44](#_Toc417985023)

[3.8 Obnova a zobrazovanie dát 46](#_Toc417985024)

[3.8.1 FUSE 46](#_Toc417985025)

[Záver 51](#_Toc417985026)

[Bibliografia 52](#_Toc417985027)

[Prílohy 53](#_Toc417985028)

# Úvod

# Existujúce riešenia

V tejto kapitole budú uvedené existujúce riešenia zálohovania dát a ich základný princíp fungovania.

## Software

Rdiff-backup: <http://www.nongnu.org/rdiff-backup/>

Duplicity: <http://www.nongnu.org/rdiff-backup/>

Time Machine: <https://support.apple.com/en-us/HT201250>

AMANDA: http://www.amanda.org/

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | rdiff-backup | duplicity | Time Machine | AMANDA |
| Licencia | GPL | GPL | Apple Inc. | BSD |
| Windows / MacOS / Linux | ✓/✓/✓ | ✓/✓/✓ | 🗶/✓/🗶 | ✓/✓/✓ |
| GUI / príkazový riadok | ✓/✓ | ✓/✓ | ✓/✓ | 🗶/✓ |
| Programovací jazyk | Python | Python | ? | C, Perl |
| Verziovanie | opačné inkrementálne | inkrementálne | inkrementálne | Inkrementálne |

### Rdiff-backup

Rdiff-backup je zálohovací software pre plne automatické inkrementálne zálohovanie a spätné zrekonštruovanie zálohovaných dát. Obnovovanie dát môže prebiehať na konkrétne zvolenom súbore alebo celej zálohe. Implementovaný je v jazyku Python a pri svojom behu využíva program rsync (knižnica librsync).

Základné vlastnosti:

* Kopírovanie súborov lokálne alebo po sieti cez protokol SSH
* Zachovanie práv súborov a médií (zálohovanie na flash disk so súborovým systémom FAT)
* Obnovenie zmazaných súborov
* Štatistiky a záznamy o zálohách

Samotné zálohovanie prebieha tak, že na cieľovom adresári je vytvorená presná kópia zálohovaného adresára. Ak nastanú na zálohovaných súboroch zmeny, tak rdiff-backup pri ďalšom zálohovaní ukladá aj reverzné diffy k zmeneným súborom do špeciálneho podadresára. Tie môžu byť neskôr využité pri prípadnej obnove starších verzií súboru. Výhodou takéhoto prístupu je aktuálne verzie zálohovaných dát sú ihneď k dispozícii a stačí ich len prekopírovať.

Spolu s dátami sú k jednotlivým súborom udržiavané aj práva, vlastník a skupina, čas zmeny. Rdiff-backup dokáže zálohovať aj súbory zariadení, pipe, hardlink a symlink.

### Duplicity

Duplicity je software pre zálohovanie dát a funguje na princípe vytvárania kryptovaných (prípadne podpísaných) tar archívov. Implementovaný je v jazyku Python a pri svojom behu taktiež využíva program rsync (knižnica librsync).

Základné vlastnosti:

* Kopírovanie súborov lokálne alebo po sieti cez protokol SSH
* Kryptovanie a podpisovanie vytvorených tar archívov pomocou GnuPG
* Inkrementálne zálohovanie

Princíp zálohovania v Duplicity je nasledovný. Každá záloha je ukladaná do tar archívu. Duplicity poskytuje aj možnosť kryptovania a podpisovania zálohovaných dát pomocou GnuPG. Pre ich využitie je potrebné si najskôr vygenerovať samotný GPG kľúč, prípadne digitálny podpis.

Následne je ako prvá vytvorená počiatočná záloha, ktorá obsahuje všetky dáta (full backup). Pri ďalšej zálohe sa ukladajú len rozdiely (diffy) voči pôvodným súborom. Nevýhodou takéhoto prístupu môže byť čas, potrebný pre obnovenie starších záloh. Napríklad pri prístupe k najnovšej verzii zálohy je potrebné prejsť všetkými inkrementálnymi zálohami, ktoré boli vykonané.

Samotné Duplicity je nástroj ovládateľný cez príkazový riadok. Existuje k nemu však kvalitná grafická nadstavba s názvom Déjà Dup. Medzi jej hlavné výhody patrí napríklad plánovanie záloh, integrácia v Linuxovom grafickom rozhraní GNOME (Nautilus) alebo podpora pre cloudové zálohovanie. Vďaka integrácii v GNOME si v súborovom manažéri Nautilus dokážete jednoducho obnoviť akýkoľvek súbor bez toho, aby ste museli obnovovať celú zálohu.

### AMANDA (Advanced Maryland Automatic Network Disk Archiver)

AMANDA je zálohovací software, ktorý umožňuje zálohovanie dát po sieti LAN. Funguje na princípe jedného zálohovacieho servera, ktorý sa stará o zálohovanie viacerých klientov (Windows, Linux, Unix, BSD, Mac OS-X). Používa k tomu natívne nástroje ako dump a GNU tar. Dump je linuxový nástroj vyvinutý hlavne pre zálohovanie diskových oddielov. Kvôli spätnej kompatibilite ukladania záloh na pásky, sú jeho výstupom bloky pevných veľkostí. GNU tar je taktiež linuxový nástroj ktorý slúži kompresiu dát.

Základné vlastnosti:

* Inkrementálne aj úplne zálohovanie
* Podpora šifrovania dát
* Zálohovanie viacerých platforiem – využíva k tomu buď samba protokol alebo svoj vlastný, natívny klient s názvom amanda-client
* Open Source
* Dáta su obnoviteľné aj bez samotného softwaru Amanda

### Time Machine

Time Machine je vstavaná zálohovacia služba systému Mac OS-X, ktorá vytvára zálohy na externý disk alebo na bezdrôtové zariadenie Time Capsule (Apple), ktoré slúži ako úložisko pre tieto zálohy.

Základné vlastnosti:

* Podpora šifrovania dát
* Záloha celého systému, vrátane systémových súborov, aplikácií, používateľských účtov atď.
* Kompletný obraz systému pre daný čas – možná obnova
* Nezmenené súbory ukladá ako odkazy (hardlinky) k pôvodným súborom a ukladá len nové, prípadne zmenené súbory
* Možnosť vyhľadávania súborov a rýchly náhľad do súboru
* Udržiava hodinové zálohy za posledných 24 hodín, denné zálohy za posledný mesiac a týždenné zálohy

Time Machine má taktiež podporu pre zálohovanie na NAS zariadenia a servery. Záleží však na samotnej verzii softvéru. Kým staršie verzie dokázali spolupracovať s veľkým množstvom NAS serverov, najnovšie verzie vyžadujú server podporujúci AFP (Apple Filling Protocol). AFP je sieťový protokol poskytujúci súborové služby pre Mac OS X a Mac OS.

### Git

Git je decentralizovaný / distribuovaný systém riadenia revízií (DVCS). Neslúži však ako zálohovací software, ale ako software pre správu verzií. Bol vyvinutý pre vývoj Linuxového jadra. Väčšinou je však využívaný vývojármi pri práci na softwarových projektoch, ale dá sa využívať na akékoľvek dáta. Dá sa tak jednoducho zistiť, kedy sa daný súbor zmenil, ako sa zmenil a taktiež kým bol zmenený.

Základné vlastnosti:

* Git je distribuovaný systém, čo znamená, že nepotrebuje žiadny centrálny bod, s ktorým by musel udržiavať spojenie. Všetky základné operácie sú vykonávané lokálne.
* Kompletná história bez možnosti zmeny
* Kompatibilita s existujúcimi systémami a protokolmi: HTTP, FTP, SSH, rsync, alebo komunikácia priamo cez socket
* Garbage kolekcia
* Delta kompresia
* Viaceré vetvy vývoja

Pojem garbage kolekcia bude zadefinovaný a vysvetlený neskôr v kapitole 1.2.4. V Gite je využívaná garbage kolekcia hlavne za účelom exportovania súborov, ktoré už nie sú viac využívané a označí ich ako „stratené“ objekty (loose objects). Pri hľadaní týchto súborov garbage kolekcia mení samotnú štruktúru repozitára na efektívnejšiu a snaží sa tak redukovať diskový priestor a zvýšiť výkonnosť.

Delta kompresia (delta kódovanie) je mechanizmus ukladania prípadne prenášania dát vo forme rozdielov medzi jednotlivými verziami. Rozdiely sú ukladané do súborov, ktoré sú nazývané taktiež delty alebo diffy. Git si vždy ukladá celý obsah najnovších súborov a rozdiely medzi súbormi sú ukladané pomocou delta kompresie len ak sú označené ako „stratené“ súbory (loose objects). Tieto súbory sú následne komprimované do binárnych súborov nazývaných tiež balíčkové súbory (packfile). Vďaka tomuto prístupu dokáže Git šetriť diskový priestor a byť efektívnejší.

## Použitá bakalárska práca

Táto diplomová práca vychádza z pôvodnej bakalárskej práce s názvom Zálohovanie v Linuxe s úsporným ukladaním histórie, ktorej autorom je Patrik Kemény. Jej cieľom bolo navrhnúť a implementovať v operačnom systéme Linux nástroje, ktoré budú vykonávať zálohovanie dát na vzdialený server a následne ich sprístupnenie. Kľúčové vlastnosti týchto nástrojov mali byť: úsporný prenos dát na vzdialený server, úsporné ukladanie záloh s ich históriou (kompresia dát), vymazávanie starších záloh a sprístupnenie záloh formou postupnosti úplných obrazov systému súborov. Pre dosiahnutie úsporného prenosu dát mali byť využité techniky nástroja rsync a git. Následné sprístupňovanie záloh malo byť realizované pomocou FUSE. Ako inšpirácia pre túto prácu mohli byť použité aj existujúce zálohovacie nástroje ako duplicity alebo rdiff-backup.

Autor však splnil len niektoré body zo stanoveného cieľu tejto bakalárskej práce. Vyvinutý zálohovací nástroj dokáže vytvárať a obnovovať plné alebo inkrementálne zálohy. Úsporné ukladanie záloh bolo implementované pomocou inkrementálneho zálohovania. Ak už v danom obsahom adresovanom úložisku existuje súbor s rovnakým obsahom ako sa snažíme zálohovať, tak bude uložený len odkaz (link) na tento súbor. Žiadne ďalšie techniky pre úsporné ukladanie dát neboli implementované.

## Techniky

### Obsahom adresované úložisko (content addressed storage)

Obsahom adresované úložisko dát funguje na nasledovnom princípe. Na strane klienta, je buď aplikačným serverom alebo samotným klientom vytvorený objekt na základe jeho obsahu (content address). Ten je následne odoslaný do úložiska a je pre neho vypočítaná adresa (hash), podľa ktorého je neskôr daný objekt jednoznačne identifikovateľný. Vypočítaná adresa je následne odoslaná späť aplikácií, ktorá pre ďalšie referencie na daný objekt, pracuje už len s ňou.

Výhody a nevýhody:

* Najvhodnejšie na dátach, ktoré sa často nemenia (kvôli častému počítaniu / prepočítavaniu adries pre jednotlivé súbory)
* Rýchle vyhľadávanie pre obsah daného súboru
* Nikdy neexistuje viac ako jedna kópia daného súboru v úložisku (dva také isté súbory majú rovnakú adresu – content address)

### Rsync

Rsync algoritmus bol vyvinutý Andrewem Tridgellom za účelom zefektívnenia prenosu dát za podmienky, že sa na cieľovom počítači už nachádza predchádzajúca verzia daných dát.

Algoritmus efektívne počíta rozdiely medzi súbormi pomocou signatúr. V algoritme sú použité dva typy signatúr, silná („strong checksum“) a slabá („weak checksum“ alebo „rolling checksum“) signatúra. Slabá signatúra musí byť veľmi rýchla a lacná na výpočet. Silná signatúra musí mať veľmi nízku pravdepodobnosť vzniku kolízie a je počítaná len vtedy, keď sa dve slabé signatúry zhodujú. Princíp fungovania rsync algoritmu je nasledovný. Predstavme si, že máme dva počítače A a B. V počítači A sa nachádza súbor, ktorý pozostáva ai bajtov a v počítači B sa nachádza súbor, ktorý pozostáva z bi bajtov. B rozdelí týchto bi bajtov na N rovnakých blokov a vypočíta slabú aj silnú signatúru ku každému z nich. Následne sú tieto signatúry odoslané na počítač A. Tu sa vypočítajú pre každý bajtový posun i v ai slabé signatúry pre každý blok začínajúci na pozícii i. Takto získané signatúry sú porovnané s každou slabou signatúrou získanou z počítaču B. Ak nastane prípad, že sa dve signatúry rovnajú, A ďalej vypočíta aj silnú signatúru a porovná ju so silnou signatúrou z B, ktorá patrí danému bloku. Ak sa zhodujú aj tieto dve signatúry, tak A odošle správu B o mieste zhody a ktoré bloky sa zhodujú. Ak sa nezhodujú, tak sú odoslané len samotné bajty. Napokon sú na počítači B použité tieto informácie na rekonštrukciu ai. K výpočtu oboch signatúr je používaný hashovací algoritmus 128 bit MD4.

Samotná rekonštrukcia súboru je jednou z najjednoduchších častí rsync algoritmu. Akonáhle sú na počítač B odoslané všetky informácie o zhodujúcich sa blokoch alebo samotné bajty, tak rekonštrukcia súboru sa začne vykonávať. Pri rekonštrukcii súboru sa postupne zapisujú získané bajty z počítača A alebo jednotlivé zhodujúce bloky bytov v súbore, nachádzajúcom sa na počítači B.

### FUSE

FUSE (Filesystem in Userspace) alebo tiež súborový systém v užívateľskom priestore umožňuje bežným uživateľom (Linux, Unix, FreeBSD, NetBSD, OpenSolaris, Mac OS-X) pristupovať k virtuálnym súborom systému. Taktiež umožňuje vytváranie súborových systémov z akýkoľvek dát a poskytovaných služieb.

Ovládač súborového systému je kernelový modul a je pevnou súčasťou samotného kernelu. Daný súborový systém môže obsluhovať len root a ten prideľuje práva k jednotlivým mountovacím bodom. Samotné použitie súborového systému funguje tak, že kernel dostane požiadavku od aplikácie, napríklad „vypíš adresár“, a ten prenechá ovládaču súborového systému. Ten následne požiadavku obslúži a spracuje odpoveď, ktorú kernel vráti späť aplikácii.

Samotný FUSE v kerneli poskytuje rozhranie ako obyčajný súborový systém. Nezaujíma sa však o obslúženie požiadavky a spracovanie odpovedi a namiesto toho, pomocou knižnice libfuse spustí bežnú používateľskú aplikáciu a tá vykoná všetko potrebné. Akonáhle aplikácia vráti odpoveď v požadovanej forme, FUSE ju prenechá kernelu a kernel ju vráti späť samotnej aplikácii.

Obrázok 1. Tok riadenia FUSE



Výhody:

* Celá logika súborového systému je zapísaná v programe prípadne skripte, ktorý je nezávislý na platforme či operačnom systéme
* Súborové systémy môže obsluhovať aj bežný uživateľ
* Stabilné API pre FUSE – nie je potrebné prepisovať programy

Ako môžeme vidieť na obrázku 1. FUSE a taktiež všetky iné súborové systémy využívajú VFS (virtual file system). VFS poskytuje pre súborové systémy abstraktné rozhranie pre komunikáciu s kernelom. FUSE musí preto pre prácu so súbormi implementovať dôležité metódy ako napríklad open, close, read, write alebo seek. Toto rozhranie je poskytované skupinou štandardou s názvom POSIX (Portable Operating System Interface).

Najznámejšie existujúce FUSE aplikácie sú napríklad sshfs, ntfs 3g, exfat-fuse. Sshfs (secure shell file system) je súborový systém založený na SSH FTP (file transfer protokol) protokole. Vďaka tomu, že väčšina SSH serverov podporuje tento protokol, tak samotné používanie sshfs je veľmi jednoduché. Stačí len vykonať pripojenie (mount) takéhoto súborového systému.

Ntfs 3g a exfat-fuse open source read – write ovládače (drivery). Ntfs 3g je implementáciou súborového systému NTFS od Microsoftu. Medzi jeho hlavné výhody patrí napríklad žurnalovanie, kompresia alebo podpora šifrovania.

Exfat-fuse je implementáciou súborového systému exFAT (Extended File Allocation Table), ktorý je vhodný hlavne pre flash disky. Využíva namiesto NTFS, ktorý je nevhodný pre takéto zariadenia, kvôli množstvu dátových štruktúr, ktoré obshauje.

Využitie FUSE je však skutočne široké a dá sa využiť napríklad aj ako súborový systém pre Wikipédiu (WikipediaFS), Gmail (GmailFS) alebo Dropbox (DropboxFS). WikipediaFS slúži ako súborový systém, pomocou ktorého môžu používatelia Wikipedie priamo editovať články, ako na svojom lokálnom disku. Hlavnou výhodou je teda editácia článkov vo vlastnom textovom editore.

GmailFS je súborový systém, pomocou ktorého dokážete využívať svoje Gmail konto ako úložisko dát. Táto funkcia však nie je priamo podporovaná Googlom.

### Reference counting garbage collection

Garbage kolekcia je mechanizmus, ktorý slúži na automatickú správu zdrojov v počítači. Najčastejšie sa však využíva na správu pamäte. Jeho úlohou je v tomto prípade získavať pamäť, ktorá už nie je využívaná prostriedkami (programy a procesy). Uľahčuje tak aj prácu programátorom, ktorí by vždy museli manuálne definovať, ktoré objekty môžu byť dealokované a uvoľniť tak pamäť systému.

Najdôležitejší problém, ktorý riešia všetky garbage kolektory, je priradenie objektov do dvoch skupín. Na objekty, ktoré sú dosiahnuteľné a na objekty, ktoré nie sú dosiahnuteľné.

Najjednoduchšou implementáciou garbage kolektora je pomocou metódy reference counting. Funguje na nasledovnom princípe. Ku každému objektu je asiciovaný počet referencií na daný objekt. Ak hodnota klesne na 0, objekt je následne označený ako nedosiahnuteľný a je alebo bude uvoľnený z pamäte. Táto metóda však nedokáže pracovať s cyklickými štruktúrami.

### CodernityDB

CodernityDB je opensource databáza typu kľúč-hodnota (key-value), ktorá disponuje viacerými pokročilejšími funkciami. Je naprogramovaná v jazyku Python, a jej kľúčovými vlastnosťami sú napríklad rýchlosť (100 000 vložení alebo výberov za sekundu), multiplatformovosť či podpora viacerých indexov. Samotná databáza si udržiava informácie o indexoch a väčšinu operácií vykonáva na nich. CodernityDB poskytuje používateľom štyri rôzne druhy databáz:

* Database – pre využitie v jedno procesovom / vláknovom prostredí
* DatabaseThreadSafe – pre využitie s viacero vláknami (čítanie neblokuje zápis)
* DatabaseSuperThreadSafe – obmedzenie na jednu databázovú operáciu v danom čase
* CodernityDB-HTTP – HTTP verzia databázy pre v prostrediach s viacerými procesmi / vláknami

## GZIP

GZIP je označením pre typ súboru, ale aj pre konkrétnu implementáciu softvéru slúžiaceho na kompresiu a dekompresiu súborov.

Ako formát súboru, je GZIP založený na DEFLATE algoritme, ktorý je kombináciou algoritmu LZ77 a Huffmanovho kódovania. Jeho autorom je Phil Katz. Samotná štruktúra súboru typu gzip je definovaná nasledovne. Na začiatku sa nachádza 10 bajtová hlavička, ktorá obsahuje „magické“ číslo (magic number), číslo verzie, časový údaj a môže obsahovať aj pôvodné meno súboru. Ďalej nasledujú samotné dáta, ktoré sú kompresované pomocou algoritmu DEFLATE. Na konci sa nachádza 8 bajtová pätička, ktorá obsahuje kontrolný súčet a údaj o dĺžke pôvodných nekompresovaných dát.

## Rdiff

Rdiff je Unixový nástroj, ktorý slúži na počítanie signatúr a ich následné využitie pre zisťovanie rozdielov medzi dvoma súbormi. Je založený na rovnakom princípe ako Unixový nastroj rsync, ktorý je priamou implementáciou algoritmu rsync, avšak dáva používateľovi priamu kontrolu nad jednotlivými operáciami. Rdiff dokáže vykonávať tieto operácie:

* Výpočet a vytvorenie súboru so signatúrou pre zadaný súbor
* Výpočet a vytvorenie delta súboru na základe dvoch zadaných súborov so signatúrami
* Aplikovanie delta súboru na pôvodný súbor a následné vytvorenie nového súboru (patch)

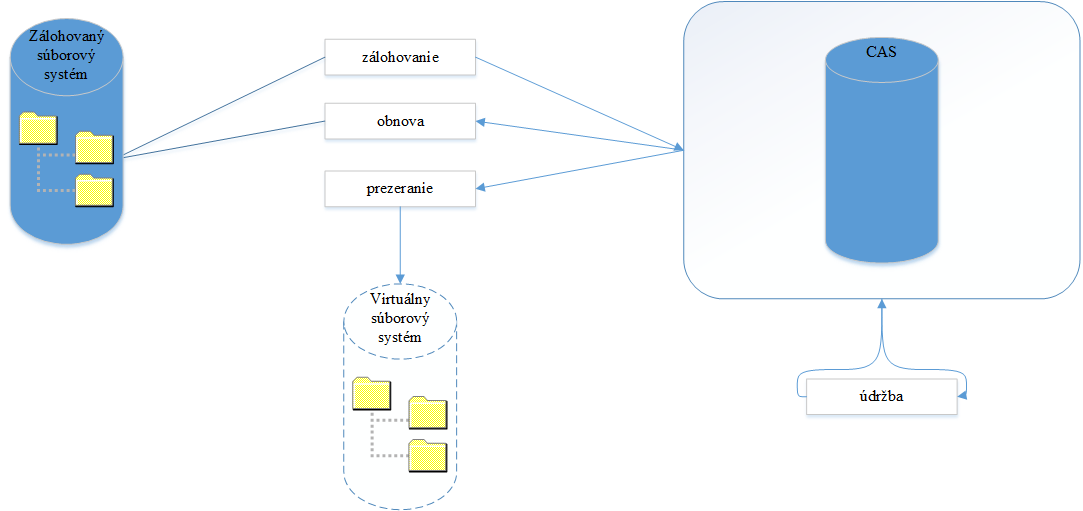
# Návrh riešenia

V tejto kapitole bude popísaný návrh riešenia samotnej aplikácie.

## Východiská

Naším hlavným cieľom je navrhnúť nástroj na zálohovanie dát, ktorý bude charakteristický nasledujúcimi kľúčovými vlastnosťami. Pri zálohovaní by mal čo najefektívnejšie využívať diskový priestor. Sprístupňovanie a prehliadanie existujúcich záloh by malo byť realizované nielen pomocou obnovenia dát, ale aj formou postupnosti úplných obrazov systému súborov. Ďalšou kľúčovou vlastnosťou je odstraňovanie a údržba existujúcich záloh. Používateľ by tak mal byť schopný odstrániť ľubovoľnú zvolenú zálohu a naďalej zachovať konzistenciu ostatných záloh.

Obrázok 2. Diagram popisujúci základnú funkcionalitu



## Obsahom adresované úložisko

Keďže táto práca čiastočne vychádza z pôvodnej bakalárskej práce, využijeme z nej už navrhnuté obsahom adresované úložisko, pre ďalšiu implementáciu a rozvoj.

Pôvodne navrhnuté obsahom adresované úložisko pozostávalo zo samotných zálohovaných objektov a zo súborov, ktoré uchovávali časové údaje jednotlivých záloh. Túto štruktúru sme využili a ďalej rozšírili o novú funkcionalitu. V našej navrhovanej implementácii obsahom adresovaného úložiska pribudne žurnálovací systém a interná databáza, ktorá bude využitá pri implementácii garbage kolekcie. Ako ďalšie pribudnú hlavičkové súbory k jednotlivým zálohovaným objektom, ktoré budú definovať spôsob uloženia daného objektu.

## Prezeranie, údržba a obnova záloh

V pôvodnej implentácii bolo prezeranie záloh možné len pri úplnej obnove dát a chýbala akákoľvek údržba dát. Preto sme sa rozhodli pre tieto účely navrhnúť vlastnú implementáciu FUSE a využiť existujúce techniky garbage kolekcií.

### FUSE

Jedným z hlavných východísk aplikácie je prezeranie záloh. Doposiaľ sme si obsah záloh však mohli prezerať len po ich úplnej obnove. Takýto prístup sa však javí ako veľmi nepraktický z viacerých dôvodov:

* Potreba voľného ukladacie priestoru na obnovu dát (nevýhoda hlavne pri objemných zálohách)
* Čas obnovy pri zálohách s veľkým objemom dát
* Kvôli získaniu jedného súboru je potrebné obnoviť celú zálohu

Preto sme sa v našom riešení rozhodli využiť a implementovať FUSE. Vďaka jeho využitiu dokážeme eliminovať všetky vyššie spomenuté nevýhody. Používateľ si tak bude môcť pripojiť svoju zálohu ako virtuálny disk, prezerať si jeho obsah a následne si z neho vyextrahovať len tie dáta, o ktoré bude mať záujem.

### Garbage kolekcia

Ďalším hlavným východiskom aplikácie je údržba záloh. V pôvodnom riešení však nebol navrhnutý ani implementovaný žiadny spôsob údržby záloh.

V existujúcej implementácii rozlišujeme dva druhy záloh. Plnú zálohu a inkrementálnu. Inkrementálna záloha je vytváraná tak, že ak sa obsah daného súboru voči pôvodnému nijak nezmenil, tak je uložený len odkaz (link) na pôvodný súbor. Inak sa na súbore vykoná záloha. Novšie zálohy sa tak stávajú závislými na starších. Výhodou takéhoto prístupu je šetrenie ukladacím priestorom. Problém však nastáva pri mazaní jednotlivých záloh. Ak by sme sa napríklad rozhodli vymazať najstaršiu zálohu, avšak najnovšia záloha by stále obsahovalo pôvodný nezmenený súbor z najstaršej. A práve pre tieto účely sme sa rozhodli využívať techniky garbage kolekcie. Jej hlavnou úlohou bude udržiavanie záloh, teda mazanie už viac nepotrebných súborov alebo celých záloh.

## Využitie kompresie

Pre využitie kompresie dát sme sa rozhodli využiť dve existujúce techniky, ktoré boli podrobnejšie popísané v predchádzajúcej kapitole. Objem dát sa budeme snažiť najskôr redukovať pomocou algoritmu rsync a jeho Unixovej implementácie rdiff. Ak sa v úložisku už bude nachádzať predchádzajúca verzia daného súboru, tak pomocou nástroje rdiff vytvoríme len rozdielový súbor a ten bude sa uložení. Následne budú všetky súbory uchované v úložisku podliehať kompresii pomocou nástroja GZIP. Týmto spôsobom by sme mali zaručiť rozsiahle šetrenie ukladacím priestorom.

## Žurnálovací systém

Pri zálohovaní dát môže nastať situácia, pri ktorej je zálohovací proces prerušený, či už systémom alebo samotným používateľom. V takomto prípade môže nastať nekonzistencia dát v úložisku a teda aj v samotnej zálohe. Aby sme sa vyhli takýmto situáciám, rozhodli sme sa navrhnúť jednoduchý žurnálovací systém, ktorý by mal fungovať nasledovne. Pri vytváraní záloh sa budú všetky zmeny ukladať a zapisovať do tohoto systému. Ak nastane prerušenie zálohovacieho procesu, v žurnálovacom systéme sa vyhodnotí, či boli všetky dáta úspešne zálohované alebo nie. Ak nie, systém takúto nekonzistentnú zálohu aj s dátami odstráni.

# Implementácia

V tejto kapitole sa budeme zaoberať samotnou implementáciou navrhnutých riešení z predchádzajúcej kapitoly a pre lepšiu predstavivosť a pochopenie budú uvedené aj konkrétne príklady.

## Obsahom adresované úložisko

Navrhnuté a implementované obsahom adresované úložiska z predchádzajúcej práce sme čiastočne využili a po aplikovaní našich navrhovaných riešení sme dosiahli nasledovnú adresárovú štruktúru:

* backups
* objects
* journal
* store.db

Adresár *backups* sa skladá zo súborov, ktorých názvy sú časové údaje vo formáte ISO-8601 a boli vytvorené v časoch vykonávania jednotlivých záloh. Okrem nich sa tu nachádza ešte jeden súbor s názvom *latest*, ktorý je nositeľom časového údaju najnovšej zálohy.

Adresár *objects* obsahuje samotné zálohované objekty (súbory s príponou *.data*) a hlavičkové súbory (súbory s príponou *.meta*) k nim. Všetky súbory sú pomenované pomocou hashu, získaného na základe obsahu jednotlivých súborov.

Hlavičkové súbory slúžia na uchovávanie niektorých dôležitých informácii k jednotlivým súborom a ich formát je nasledovný:

*<typ zálohovaného súboru>* - directory, gz, delta, raw, link

*signature* - čisto informatívny význam (len typy gz, delta, raw)

*číslo* – dĺžka signatúry (len typy gz, delta, raw)

<*signatúra>* – signatúra definujúca daný zálohovaný súbor (len typy gz, delta, raw)

*previous* - čisto informatívny význam (len typ delta)

<*názov súboru*> - predchádzajúca verzia súboru (len typ delta)

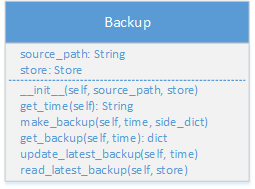
Ďalší adresár má názov *journal* a je využívaný jednoduchým žurnálovacím systémom, ktorému sa budeme venovať neskôr. V princípe sa však skladá z dvoch adresárov s názvom *backups* a *objects* , ktoré majú rovnakú funkciu ako adresáre s rovnakým názvom nachádzajúce sa o úroveň vyššie.

Ďalším a zároveň posledným adresárom našej štruktúry je adresár s názvom *store.db*. Slúži na uchovávanie databázy pre údržbu záloh pomocou garbage kolekcie. Samotnému princípu využívania tejto databázy sa však budeme zaoberať v kapitole popisujúcej garbage kolekciu.

## Backup

Pomocou triedy *Backup* sú reprezentované jednotlivé zálohy ako jeden celok. Od tejto triedy sú ďalej zdedené ďalšie dve podtriedy s názvom *NewBackup* a *ExistingBackup*.

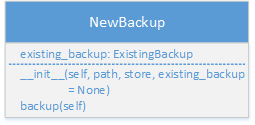
Obrázok 3. UML diagram triedy Backup



### NewBackup

Nové zálohy sú reprezentované pomocou objektov typu *NewBackup*. Samotná trieda implementuje len jednu dôležitú metódu s názvom *backup*, ktorá inicializuje rekurzívne zálohovanie jednotlivých objektov zdedených od triedy *SourceObject*.

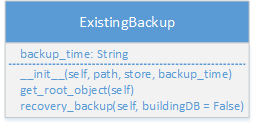
Obrázok 3. UML diagram triedy *NewBackup*



### ExistingBackup

Existujúce zálohy sú reprezentované pomocou objektov typu *ExistingBackup*. Implementuje dve dôležité metódy. Prvou je metóda *get\_root\_object*, pomocou ktorej získame koreňový objekt (adresár) danej zálohy. Druhou je metóda *recovery\_backup*, ktorá slúži na obnovenie danej zálohy. Obnova prebieha rekurzívne na všetkých objektoch zdedených od triedy *StoreObject* a začína od koreňového objektu danej zálohy, ktorý je vždy typu *StoreDir*.

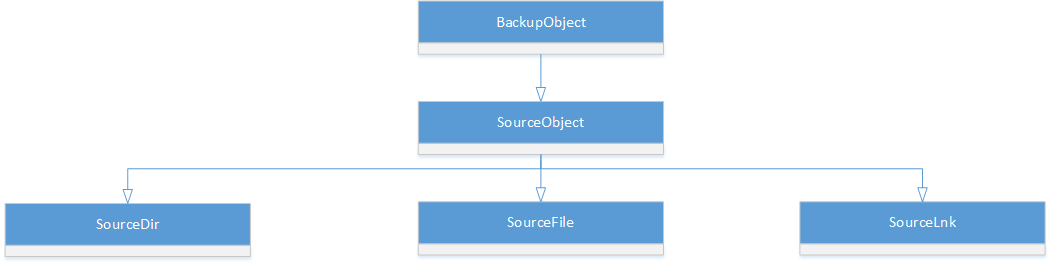
Obrázok 4.UML diagram triedy *ExistingBackup*



## Source

Výrazom *source* (zdroj) označujeme v našej aj predchádzajúcej implementácii všetky zálohované objekty, ktoré sú ešte ďalej rozlišované na tri základné typy. Prvým základným typom je zdroj typu directory (adresár). Pre tento typ je implementovaná trieda *SourceDirectory*. Ďalším typom je file (súbor), ku ktorému je implementovaná trieda *SourceFile*. Posledným základným typom je zdroj typu link (odkaz), pre ktorý je implementovaná trieda *SourceLnk*. Všetky spomínané triedy sú zdedené od objektu *SourceObject*.

Obrázok 3. Diagram dedenia jednotlivých tried

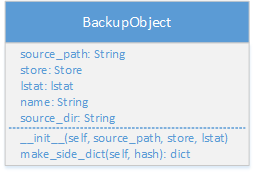


### BackupObject

Trieda *BackupObject* je základom všetkých tried, s ktorými pri procesoch zálohovania, čítania a obnovy pracujeme. Je využívaná pri existujúcich objektoch úložiska, ale aj pri zálohovaných *source* (zdrojových) súboroch. Uchováva všetky dôležité informácie o samotných súboroch ako sú:

* Zdrojovú cestu k danému súboru
* Názov súboru či už na strane úložiska (*store*) alebo zdroja (*source*)
* Názov adresára, v ktorom sa nachádza
* Inštanciu objektu typu *Store*, ktorá reprezentuje samotné úložisko
* Lstat k danému súboru (práva, vlastník, typ atď)

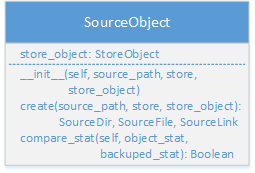
Obrázok 3. UML diagram triedy BackupObject



### SourceObject

Pomocou triedy *SourceObject* a jej metódy *create* sú inicializované všetky objekty zdedené od tejto triedy a reprezentujúce jednotlivé súbory na strane zdroja (source). Keďže samotná trieda *SourceObject* je zdedená od triedy *BackupObject*, stáva sa nositeľom všetkých dôležitých informácií o zálohovanom súbore. Pomocou metódy *compare\_stat* porovnáva informácie (vlastníka, práva, typ, poslednú zmenu a poslednú zmenu metadát) o zálohovanom objekte a objekte, ktorý je už zálohovaný v úložisku. Okrem toho obsahuje ešte triednu premennú *store\_object* typu *StoreObject*, ktorá má reprezentovať predchádzajúcu verziu zálohovaného súboru nachádzajúcu sa v danom úložisku. Na základe tohto objektu a metódy *compare\_stat* sa neskôr pri samotnom zálohovaní rozhoduje, či sa vykonáva inkrementálna alebo plná záloha daného objektu.

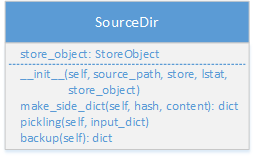
Obrázok 3. UML diagram triedy SourceObject



### SourceDir

Triedou *SourceDir* reprezentujeme zdrojový (zálohovaný) objekt typu adresár. Pomocou implementovanej metódy *backup*, je vykonávané rekurzívne zálohovanie na všetkých objektoch, ktoré adresár obsahuje a nakoniec aj na samotnom adresári.

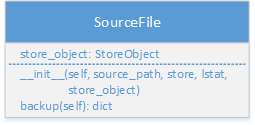
Obrázok 3. UML diagram triedy SourceDir



### SourceFile

Pomocou triedy *SourceFile* reprezentujeme zdrojové (zálohované) súbory.

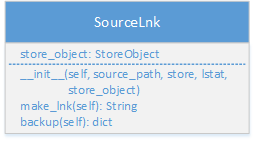
Obrázok 3. UML diagram triedy SourceFile



### SourceLnk

Trieda *SrouceLnk* reprezentuje zdrojový (zálohovaný) objekt, ktorý je typu link.

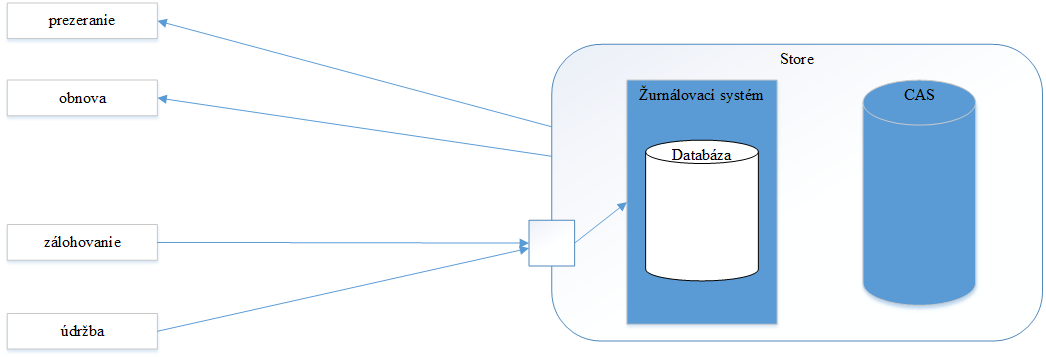
Obrázok 3. UML diagram triedy SourceLnk



## Store

Pojmom *store*, budeme v našej práci označovať implementáciu rozhrania pre samotnú komunikáciu s obsahom adresovaným úložiskom. Toto rozhranie je implementované v triede s názvom *store*.

Obrázok 3. Diagram základnej funkcionality Store

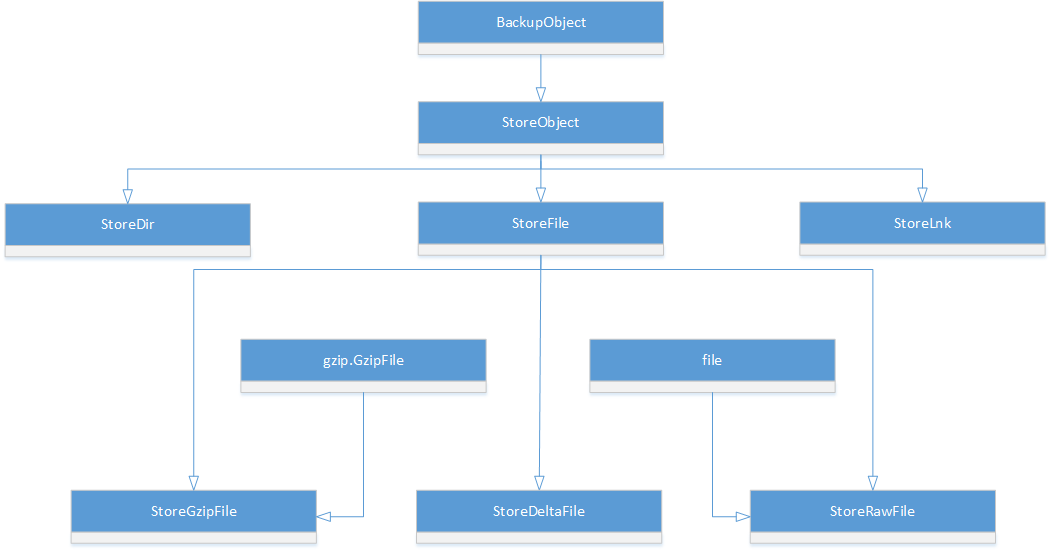


Store implementuje všetky dôležité metódy, potrebné pre zálohovanie a opätovné získavanie dát.

### Objekty

K službám, ktoré trieda store implementuje je využívaných ďalších päť rôznych tried. Každá z nich je zdedená od triedy *StoreObject*, ktorej konštruktor následne vytvára podľa príslušného typu zálohovaného súboru jednotlivé inštancie týchto tried. Názvy týchto piatich tried sú: *StoreDir*, *StoreRawFile*, *StoreGzipFile*, *StoreDeltaFile*, *StoreLnk*. Na nasledujúcom diagrame je znázornená hierarchia dedenia jednotlivých objektov.

Obrázok 4. Diagram dedenia jednotlivých tried



Kvôli neskoršiemu využitiu objektov typu *StoreRawFile*, *StoreGzipFile* a *StoreDeltaFile* v implementácii FUSE sa snažíme docieliť, aby sa jednotlivé objekty správali ako reálne súbory. Preto je trieda *StoreRawFile* zdedená od Pythonovskej triedy *file*. Trieda *StoreGzipFile* dedí od triedy *gzip.GzipFile* z knižnice *gzip*, určenej na kompresiu dát. A napokon trieda *StoreDeltaFile* je len wrapperom, ktorý implementuje niektoré metódy potrebné pre samotný FUSE.

#### StoreObject

Nasledujúci zdrojový kód znázorňuje inicializáciu jednotlivých objektov podľa ich typu v metóde *create* triedy *StoreObject*.

if S\_ISDIR(mode) and object\_type == "directory":

return StoreDir(source\_path, store, lstat, side\_dict)

elif S\_ISREG(mode) and object\_type == "gz":

return StoreGzipFile(source\_path, store, lstat, side\_dict,

store.get\_object\_path(side\_dict['hash']))

elif S\_ISREG(mode) and object\_type == "raw":

return StoreRawFile(source\_path, store, lstat, side\_dict,

store.get\_object\_path(side\_dict['hash']))

elif S\_ISREG(mode) and object\_type == "delta":

return StoreDeltaFile(source\_path, store, lstat, side\_dict,

store.get\_object\_path(side\_dict['hash']))

elif S\_ISLNK(mode) and object\_type == "link":

return StoreLnk(source\_path, store, lstat, side\_dict)

#### StoreDir

#### StoreGzipFile

#### StoreRawFile

#### StoreDeltaFile

Táto trieda reprezentuje objekt zálohy, ktorý uchováva len deltu k predchádzajúcej verzii súboru. Samotná trieda je len wrapper, ktorý sa ma správať ako objekt typu *file*. Implementuje vlastné metódy *open*, *read*, *seek*, *close* z nasledovného dôvodu. Ak by sme chceli pristupovať a čítať dáta tohto zálohovaného objektu, potrebujeme ho najskôr zrekonštruovať. Rekonštrukcia vyžaduje prechod všetkými verziami daného súboru od najstaršej až po požadovanú verzie. Na najstaršiu verziu zálohovaného súboru sa tak aplikujú dané delty a tak vzniká požadovaná verzia súború. Celý tento postup je implementovaný v metóde *get\_patched\_file*.

while (len(list) > 0 or not first == 0):

first = 0

if not base\_file\_hash == None:

patchProcess = subprocess.Popen(['rdiff', 'patch',

self.store.get\_unzipped\_tempFile(base\_file\_hash, tempFile2).name,self.store.get\_unzipped\_tempFile(list.pop(), tempFile3).name, '-'], stdout=subprocess.PIPE)

base\_file\_hash = None

else:

patchProcess = subprocess.Popen(['rdiff', 'patch',

temp.name, self.store.get\_unzipped\_tempFile(list.pop(), tempFile2).name, '-'], stdout=subprocess.PIPE, stdin=subprocess.PIPE)

patch, patchErr = patchProcess.communicate()

V priloženom zdrojovom kóde, je zobrazený proces získavania požadovanej verzie súboru pomocou temporárnych súborov a modulu *subprocess*. Tento modul nám poskytuje rozhranie pre vytváranie a prácu so systémovými procesmi. Pomocou Unixového nástroja *rdiff* aplikujeme danú deltu na predchádzajúcu verziu súboru a pomocou výstupnej pipe získame požadovanú verziu, ktorá je uložená do temporárneho súboru.

#### StoreLnk

## Žurnálovací systém

Naša implementácia žurnálovacieho systému je veľmi jednoduchá, avšak pre zachovanie konzistentnosti dát je postačujúca.

Princíp fungovania je nasledovný. Všetky objekty, ktoré sú zálohované budú najskôr uložené do žurnálovacieho systému (adresár journal). Adresár journal obsahuje okrem dvoch ďalších adresárov aj súbor *journal\_incomplete* (pri nedokončenej zálohe) alebo *journal\_complete* (pri dokončenej zálohe), ktorý nám indikuje aktuálny stav systému. Do tohto súboru sa ďalej zapisujú príkazy, ktoré má žurnálovací systém vykonať za podmienky úspešného ukončenia zálohovacie procesu. V prípade zálohovacieho procesu je to len príkaz *move <cesta> <cesta>*, pomocou ktorého sú presunuté uložené súbory do zložiek *objects* a *backups*. Po úspešnom dokončení zálohovacieho procesu sa postupne začnú vykonávať jednotlivé príkazy zo súboru *journal\_complete*. Ak by došlo k prípadnému prerušeniu tohto procesu, budú zvyšné príkazy vykonané pri najbližšom vykonávaní zálohovacieho procesu, kde sa vždy v úvode kontroluje stav žurnálovacieho systému. Ten sa môže nachádzať v troch rôznych stavoch: úplný, neúplný a prázdny. To, že sa nachádza v stave úplný nám indikuje už vyššie spomínaný súbor s názvom *journal\_complete*. Po úspešnom vykonaní všetkých príkazov z tohto súboru sa systém dostáva do stavu prázdny. Žurnál nachádzajúci sa v stave neúplný nám indikuje súbor *journal\_incomplete*. V tomto prípade sú všetky dáta zo systému odstránené, čím sa mení jeho stav na prázdny.

Žurnálovací systém je taktiež využívaný pri odstraňovaní záloh. V tomto prípade sa do žurnálovacieho súboru zapisujú príkazy v tvare *remove <cesta>*, pre odstránenie jednotlivých zálohovaných objektov.

## Zálohovanie dát

Ako sme už spomenuli v predchádzajúcej kapitole, dáta sú najskôr zálohované do žurnálovacieho systému. Za týmto účelom sú v triede *store* implementované štyri rôzne metódy, ktoré sú rozlišované podľa typu ukladaného objektu:

* metóda *save\_directory*, volaná metódou *pickling* pri zálohovaní objektu typu *SourceDirectory*
* metóda *save\_link*, volaná metódou *make\_lnk* pri zálohovaní objektu typu *SourceLink*
* metóda *save\_file*, volaná metódou *save\_file* pri zálohovaní objektu typu *SourceFile*
* metóda *save\_data*, volaná metódami *make\_backup* a *update\_latest\_backup* triedy *Backup* pri vytváraní súborov nachádzajúcich sa v adresári *backups*

Všetky menované metódy vo svojich implementáciách vždy po úspešnom uložení súboru prípadne hlavičkového súboru volajú metódu *write\_to\_journal*. Pomocou nej sa do žurnálovacieho systému zapíšu príslušné príkazy pre premiestnenie vytvorených súborov. V nasledujúcom zdrojovom kóde je uvedený konkrétny príklad využitia metódy *write\_to\_journal*.

self.write\_to\_journal("move " + self.get\_journal\_object\_path(

file\_hash.hexdigest()) + " " + os.path.join(self.store\_path, "objects", file\_hash.hexdigest() + ".data"))

self.write\_to\_journal("move " + self.get\_journal\_object\_header\_path(

file\_hash.hexdigest()) + " " + os.path.join(self.store\_path, "objects", file\_hash.hexdigest() + ".meta"))

### Zálohovanie adresárov

Zdrojové adresáre sú reprezentované v našej implementácii pomocou triedy *SourceDir*. Pre samotné zálohovanie je implementovaná metóda *backup*. Tá prechádza celým obsahom adresára a ku každému objektu sa na základe jeho názvu snaží z úložiska získať predchádzajúcu verziu. Následne je pomocou metódy *create* triedy *SourceObject* inicializovaný príslušný objekt a na ňom zavolaná metóda *backup*. Každý typ objektu má na výstupe metódy *backup* slovník (dictionary), ktorý sa skladá z jeho hashu a lstat. Objekt typu *SourceDir* si vytvára svoj vlastný slovník, v ktorom si uchováva jednotlivé slovníky každého objektu, ktoré daný adresár obsahuje. S takto vytvoreným slovníkom sa zavolá triedna metóda *pickling*, ktorej cieľom je uložiť daný slovník do úložiska. K tomu je využitá knižnica *pickle*, ktorá v Pythone slúži pre serializovanie a deserializovanie objektov. Pomocou nej sa serializuje daný slovník a následne je podľa toho vypočítaný hash. Na záver, ak sa ešte v úložisku nenachádza objekt s daným hashom, je zavolaná metóda *save\_directory* triedy *Store*, ktorá uloží serializovaný slovník.

### Zálohovanie súborov

Triedou *SourceFile* reprezentujeme zálohovaný súbor. Tak ako pri zálohovaní adresárov, aj tu voláme triednu metódu *backup*, ktorá slúži na zálohovanie daného súboru. Výstupom tejto metódy je opäť slovník (dictionary), vytvorený pomocou metódy *make\_side\_dict* triedy *BackupObject*. Implementácia metódy *backup* rozlišuje nasledovné situácie, ktoré môžu pri zálohovaní súboru nastať:

1. K zálohovanému súboru neexistuje žiadna predchádzajúca verzia, ktorá by sa nachádzala v úložisku. V tomto prípade je volaná metóda *save\_file* triedy *Store* bez parametra *previous\_hash*, ktorá vráti na výstupe hash zálohovaného súboru.
2. K zálohovanému súboru existuje predchádzajúca verzia nachádzajúca sa v úložisku. V tomto prípade môžu nastať ďalšie tri situácie:
   1. Zálohovaná verzia sa voči predchádzajúcej verzii súboru nijak nezmenila. V tomto prípade je na výstupe vrátený slovník predchádzajúcej verzie súboru.
   2. Zálohovaná verzia sa voči predchádzajúcej verzii súboru líši len na základe získaných informácií o súbore (lstat). Obsah súborov je ale rovnaký. V tomto prípade je na výstupe vrátený slovník s aktuálnym lstat a s hashom predchádzajúcej verzie súboru.
   3. Zálohovaná verzia sa voči predchádzajúcej verzii líši obsahom. V tomto prípade je volaná metóda *save\_file* triedy *Store* s hashom predchádzajúcej verzie súboru, ktorá vráti na výstupe hash aktuálnej verzie zálohovaného súboru.

#### Ukladanie súborov v úložisku

Ako už bolo spomenuté, ukladanie súborov do úložiska je vykonávané pomocou metódy *save\_file*. Táto metóda je z pohľadu efektívneho využívania diskového priestoru najzaujímavejšia.

Pri zálohovaní je do úložiska vždy ukladaný komprimovaný súbor. Kompresia je vykonávaná pomocou knižnice *gzip*. Princíp samotného ukladania súborov je nasledovný. Metóda *save\_file* obsahuje nepovinný parameter *previous\_hash*, ktorý definuje či sa bude vykonávať celá alebo inkrementálna záloha súboru. Pri oboch typoch sa zapisuje do hlavičkového súboru signatúra zálohovaného objektu. Tá je vypočítaná pomocou Unixového nástroja *rdiff*, ktorý opäť voláme pomocou modulu *subprocess*. V nasledujúcom zdrojovom kóde je znázornený samotný proces získavania signatúry pre daný zdrojový súbor a jej následný zápis do hlavičkového súboru.

with open(store\_file\_header, "wb") as THF:

...

THF.write("signature\n")

sigProcess = subprocess.Popen(['rdiff', 'signature',

source\_path], stdout=subprocess.PIPE)

signature, signatureErr = sigProcess.communicate()

if (signatureErr is None):

THF.write(str(len(signature)))

THF.write("\n")

THF.write(signature)

else:

THF.write(str(0))

...

Pri vykonávaní celej zálohy súboru je čítaný zdrojový súbor blok po bloku a pomocou knižnice *gzip* je zapisovaný do daného súboru v úložisku. Na základe prečítaného bloku je aktualizovaný aj hash pre daný súbor. Pomocou výsledného hashu je zálohovaný súbor jednoznačne definovaný v obsahom adresovanom úložisku.

with open(source\_path, "rb") as SF:

with gzip.open(store\_file, "wb") as TF:

while True:

block = SF.read(block\_size)

file\_hash.update(block)

TF.write(block)

if not block:

self.file\_rename(target\_file, file\_hash.hexdigest() + ".data")

...

break

TF.close()

...

SF.close()

Podmienkou vykonávania inkrementálnej zálohy na danom zdrojovom súbore je vyplnenie nepovinného parametra *previous\_hash*. Na základe tohto hashu je zistený typ predchádzajúceho súboru. Môže sa jednať o objekt typu *StoreGzipFile* alebo *StoreDeltaFile*. Následné je k danému objektu získaná signatúra z jeho hlavičkového súboru. Daná signatúra je pomocou vstupnej pipe odoslaná nástroju *rdiff*, kde je vypočítaná delta medzi predchádzajúcou a aktuálnou verziou súboru. Delta je prečítaná z výstupnej pipe tohto procesu a zapísaná opäť pomocou knižnice *gzip* do daného súboru v úložisku.

...

previous\_file = self.get\_object\_file\_header(previous\_hash, "rb")

...

sig\_size = previous\_file.readline()

sig\_data = previous\_file.read(int(sig\_size))

deltaProcess = subprocess.Popen(['rdiff', 'delta', '-', source\_path],

stdout=subprocess.PIPE, stdin=subprocess.PIPE)

deltaProcess.stdin.write(sig\_data)

deltaProcess.stdin.close()

with gzip.open(store\_file, "wb") as TF:

while True:

deltaData = deltaProcess.stdout.read(16)

if deltaData:

file\_hash.update(deltaData)

TF.write(deltaData)

...

### Zálohovanie odkazov

Pomocou triedy *SourceLnk* definujeme zálohovaný odkaz. Pre samotné zálohovanie odkazu, je tak ako pri predchádzajúcich zálohovaných objektoch volaná triedna metóda *backup*, ktorej výstupom je opäť slovník vytvorený pomocou metódy *make\_side\_dict* triedy *BackupObject*. Metóda *backup* je implementovaná tým istým princípom ako v prípade zálohovania súborov, avšak namiesto metódy *save\_file* je volaná triedna metóda *make\_lnk*. Jej úlohou je vypočítať hash na základe prečítaného odkazu a následne zavolať metódu *save\_link* triedy *Store*.

## Údržba a odstraňovanie dát

Pri údržbe a odstraňovaní dát je dôležité, aby v úložisku nenastala nekonzistencia dát. Táto situácia môže nastať napríklad pri odstraňovaní ľubovoľnej zálohy. Ak bol vymazaný niektorý objekt, na ktorý sa ešte odkazujú niektoré zvyšné zálohy, spôsobilo by to nekonzistenciu celého úložiska. Preto sme pre účely údržby a odstraňovania dát bola implementovali garbage kolekciu využívajúcu metódu reference countingu (počítanie referencií na jednotlivé objekty).

### Garbage kolekcia

Ako už bolo spomenuté, naša implementácia garbage kolecie využíva metódu reference countingu. Preto sme sa rozhodli pri jej implementácii využiť externý databázový systém typu key-value (kľúč-hodnota), ktorá bude slúžiť na uchovávanie počtu referencií na jednotlivé objekty. Názov použitého databázového systému je CodernityDB.

Princíp fungovanie garbage kolekcie je nasledovný. Pri prvej, inicializačnej zálohe sa v úložisku vytvorí prázdna databáza, ktorá je uchovávaná v adresári *store.db*. Pri jej vytváraní si vytvoríme dva vlastné indexy, ktoré budeme následne využívať pre získavanie dát.

class WithHashIndex(HashIndex):

def \_\_init\_\_(self, \*args, \*\*kwargs):

kwargs['key\_format'] = '16s'

super(WithHashIndex, self).\_\_init\_\_(\*args, \*\*kwargs)

def make\_key\_value(self, data):

value = data['hash']

return md5(value).digest(), None

def make\_key(self, key):

return md5(key).digest()

V priloženom zdrojovom kóde je znázornená trieda *WithHashIndex*, pomocou ktorej definujeme nový index v databáze. Vďaka tomuto indexu budeme do databázy vkladať záznamy definované podľa hashu. V našom prípade to je názov súboru uloženého v úložisku, ktorý je vypočítaný podľa obsahu daného súboru. Podobným spôsobom je implementovaná aj trieda *WithPointerIndex*, ktorá definuje index na uchovávanie jednotlivých referencií na daný objekt. Jednotlivé záznamy v databáze vyzerajú nasledovne.

{'hash': '4fd195d33795c81f2ba839f058b0ecc0a108f0de', '\_rev': '0001eb00',

'\_id': '9b65ea0d4bac4622938e417f390949ac', 'pointer': 1}

{'hash': '3e11df51f0f06e76ae1030f8a4dd931c9f584d01', '\_rev': '0003fb16',

'\_id': 'e9e3c9d9784f45ebb6e03a85e15f1ae9', 'pointer': 3}

Obsluha a riadenie databázy je vykonávané pomocou triedy *store*, v ktorej sú implementované dve metódy pre vkladanie, upravovanie a vymazávanie záznamov z databázy. Prvou z nich je metóda *incIndex* , ktorá je volaná len pri vytváraní záloh. Ak sa zálohovaný objekt ešte nenachádza v obsahom adresovanom úložisku, tak je podľa jeho hashu vytvorený záznam, kde hodnota indexu *pointer* bude 1. Inak sa existujúcemu záznamu zvýši hodnota indexu *pointer* o 1.

def incIndex(self, hash):

try:

element = self.db.get('hash', hash, with\_doc=True)

element = element['doc']

element['pointer'] = element['pointer'] + 1

self.db.update(element)

except RecordNotFound:

self.db.insert({'hash':hash, 'pointer':1})

return 1

return element['pointer']

Podobne funguje aj metóda *decIndex*, ktorá je volaná len počas odstraňovania záloh. V tomto prípade sa pri odstraňovaní zálohovaného objektu znižuje hodnota indexu *pointer* o 1. Ak po tejto operácií nadobúda hodnotu 0, tak sa do žurnálovacieho systému zapíše príkaz k odstráneniu tohto súboru z úložiska a zároveň je odstránení aj daný záznam z databázy.

Aby sme zamedzili prípadnej nekonzistencii dát v databáze, implementovali sme metódu *rebuildDB*, pomocou ktorej je možná rekonštrukcia celej databázy na základe uložených záloh v úložisku. Metóda prechádza všetkými vytvorenými zálohami a následne ich stromovými štruktúrami a vkladá jednotlivé údaje do databázy.

## Obnova a zobrazovanie dát

Pri obnove jednotlivých záloh sú využívané len jednotlivé objekty na strane úložiska. Samotná obnova je inicializovaná pomocou triedy *ExistingBackup*, pomocou ktorej je načítaná zvolená záloha z úložiska. Následne metódou *get\_root\_object* inicializujeme koreňový objekt danej zálohy, ktorý je vždy typu *StoreDir*. Na tomto objekte je volaná rekurzívna metóda *recover*, vďaka ktorej sú jednotlivé objekty obnovené.

### FUSE

Pri implementácii vlastného súborového systému sme využili modul *fusepy*, ktorý nám poskytuje jednoduché rozhranie pre FUSE. Keďže hlavnou požiadavkou je prezeranie jednotlivých záloh a prípadne kopírovanie zálohovaných objektov, rozhodli sme tento súborový systém implementovať len ako *read-only*. Pri prípadnom zapisovaní alebo editovaní súborov by nastala nekonzistencia dát v celom úložisku.

Samotný FUSE implementuje trieda *BackupFS*. Tá je zdedená od triedy *fuse.operations*, ktorá definuje všetky operácie potrebné k behu súborového systému. Keďže sme sa rozhodli pre obmedzený súborový systém typu *read-only*, stačilo implementovať len niektoré metódy.

Princíp fungovania je nasledovný. Pri inicializácii triedy *BackupFS* je pomocou vstupného parametru *allbackups* odovzdaný slovník, ktorý sa skladá z objektov typu *StoreDir*. Obsahuje teda všetky koreňové adresáre existujúcich záloh vytvorených v danom úložisku. Ak je súborovým systémom vykonávaná metóda, ktorá v svojej implementácii vyžaduje daný objekt, je volaná rekurzívna metóda *get\_object\_by\_path* na konkrétnom objekte typu *StoreDir*. Jej výstupom je požadovaný objekt, ktorý je získaný na základe danej cesty. V nasledujúcich riadkoch bude uvedený úryvok zdrojového kódu a princíp fungovania tejto metódy.

...

if name in self.loaded\_dict:

if ('object\_' + name) in self.loaded\_obj:

if (name == file\_name and size == 0):

return self.loaded\_obj['object\_' + name]

else:

return self.loaded\_obj['object\_' + name]

.get\_object\_by\_path(folders, file\_name)

else:

new\_store\_object = StoreObject.create(os.path.join(

self.source\_path, name), self.store,

self.loaded\_dict[name])

self.loaded\_obj['object\_' + name] = new\_store\_object

if (name == file\_name and size == 0):

return self.loaded\_obj['object\_' + name]

else:

return self.loaded\_obj['object\_' + name]

.get\_object\_by\_path(folders, file\_name)

else:

return None

V úvode je vykonávaná kontrola, či sa v načítanej štruktúre adresára nachádza objekt s daným názvom. Ak áno, tak sa opäť skontroluje či sa v slovníku *loaded\_obj* nachádza načítaný objekt podľa jeho názvu. Slovník *loaded\_obj* je v tomto prípade využívaný pre cachovanie a zamedzenie opakovanému načítavaniu jednotlivých objektov pri viacnásobnom prístupe zo súborového systému. V prípade nenačítaného objektu sa daný objekt inicializuje pomocou metódy *create* triedy *StoreObject*. Ak sa už v danom slovníku nachádza načítaný objekt a nachádzame sa v konečnom adresári, tak je vrátený na výstupe metódy. V prípade, že sa ešte nenachádzame v konečnom adresári, zavolá sa metóda *get\_object\_by\_path* na tomto objekte (objekt typu *StoreDir*).

#### Metódy súborového systému

Nasledovné metódy sú nevyhnutné pre samotný beh súborového systému:

* access – skontroluje oprávnenia pre prístup k súboru
* getattr – získa atribúty (vlastník, typ, čas vytvorenia atď) pre daný súbor
* readdir – prečíta obsah daného adresára
* readlink – prečíta daný odkaz (link)
* statfs – získa štatistiky súborového systému

Metódy *getattr*, *readdir*, *readlink* pracujú priamo s objektami úložiska (všetky triedy zdedené od triedy *StoreObject*).

#### Metódy pre prácu so súbormi

Pri našej implementácii súborového systému sú potrebné len niektoré metódy pre prácu so súbormi respektíve objektami úložiska:

* open – otvorí daný súbor (v našom prípade objekt úložiska) na čítanie
* release – zatvorí a uvoľní daný súbor (objekt úložiska)
* read – čítanie dát zo súboru (objektu úložiska)

Ako už bolo spomenuté v kapitole 3.4.1, jednotlivé objekty v úložisku reprezentujúce zálohované súbory sme sa snažili implementovať tak, aby sa správali ako objekty typu *file*. To znamená, že musia implementovať niektoré dôležité metódy pre prácu so súborom, ako napríklad *open, read, close*. Tieto metódy následne využívame pri implementácii vyššie spomenutých metód súborového systému. Jedinou podmienkou je získať správny objekt, opäť pomocou metódy *get\_object\_by\_path* a na ňom vykonávať jednotlivé operácie.

Pri implementácii metód *open* a *read* sme sa stretli s problémom viacnásobného pristupovaniu k danému objektu. Rozhodli sme sa preto využiť Pythonovský modul *threading* a jeho funkciu *Lock*.

...

self.fileslock = Lock()

...

def open(self, path, flags):

object = self.\_get\_object(path)

if object is not None:

with self.fileslock:

global counter

counter += 1

self.gzipFiles[counter] = object

fh = counter

return fh

else:

raise IOError(errno.EINVAL, 'Invalid file path')

def read(self, path, length, offset, fh):

with self.fileslock:

if self.gzipFiles[fh].closed:

self.gzipFiles[fh].open()

self.gzipFiles[fh].seek(offset)

return self.gzipFiles[fh].read(length)

V priloženom zdrojovom kóde je znázornená metóda *open* a *read*. Pomocou metódy *open* získavame jednotlivé objekty na základe ich cesty (path) a následne ich ukladáme do slovníka *gzipFiles*. Pomocou premennej *counter* určujeme jednotlivým objektom ich unikátny identifikátor (filehandle). Pôvodne toto dočasné číslo prideľuje operačný systém otvorenému súboru a následne ho využíva pre prístup k nemu.

V metóde *read* je pred samotným čítaním dát z daného objektu overované, či sa nenachádza v stave *closed*. V tomto prípade daný objekt najskôr otvorí pomocou metódy *open*.

Zdrojový kód, ktorý sa nachádza v ohraničenom bloku *with self.fileslock:* je vykonávaný konkurentne.

# Záver

# Bibliografia

[1] ANDREW, Tridgell. Efficient Algorithms for Sorting and Synchronization [Dátum Júl 1999] http://gan.anu.edu.au/~brent/pd/Tridgell-thesis.pdf

[2] SCOTT, Chacon. Pro Git [Dátum 8. Február 2010] <http://labs.kernelconcepts.de/downloads/books/Pro%20Git%20-%20Scott%20Chacon.pdf>

[3] CURTIS, Preston. Backup & Recovery (4. kapitola)

[Dátum Január 2007]

[4] <https://www.kernel.org/doc/Documentation/filesystems/fuse.txt>

[5] L. PETER Deutsch, DANIEL G. Bobrow. An efficient, incremental, automatic garbage

collector

[Dátum 9. September 1976]

<http://dl.acm.org/citation.cfm?id=360345>

[6] HENRY Lieberman, CARL Hewitt. A real-time garbage collector based on lifetimes of

objects

[Dátum 6. Jún 1983]

http://dl.acm.org/citation.cfm?id=358147

# Prílohy

1. CD so zdrojovými kódmi aplikácie a bakalárskou prácou vo formáte PDF