**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](#_Toc405922246)

[1 Úvod do problematiky 11](#_Toc405922247)

[1 Návrh riešenia 11](#_Toc405922248)

[2 Implementácia 12](#_Toc405922249)

[Záver 10](#_Toc405922250)

[Bibliografia 11](#_Toc405922251)

[Prílohy 12](#_Toc405922252)

# Úvod

# Východiská

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

## Existujúce zálohovacie riešenia

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 | Git |
| Licencia | GPL | GPL | Apple Inc. | BSD | GPL |
| Windows/OS X/ Linux | ✓/✓/✓ | ✓/✓/✓ | 🗶/✓/🗶 | ✓/✓/✓ |  |
| Príkazový riadok | ✓/✓ | ✓/✓ | ✓/✓ | 🗶/✓ |  |
| GUI | ??? |  |  |  |  |
| Prog. jazyk | Python | Python | ? | C, Perl |  |
| Verziovanie | opačné inkrementálne | inkrementálne | inkrementálne | Inkrementálne | opačné inkrementálne |
| Kompresia |  |  |  |  |  |
| Delta kompresia |  |  |  |  |  |
| Jednoduché prezeranie záloh | najnovšia |  | všetky |  | všetky |
| Zmazanie ľubovoľnej zálohy |  |  |  |  |  |

### Rdiff-backup

Rdiff-backup[[1]](#footnote-1) 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 algoritmus rsync (viď kap. 1.3.2) implementovný v knižnici 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 do hlavného adresára novú verziu zmenených súborov a do špeciálneho podadresára aj *opačné rozdiely* (*reverse diff*, teda rozdiely popisujúce rekonštrukciu staršej verzie z novšej) vypočítané algoritmom. Tie môžu byť neskôr využité pri obnove starších verzií súboru. Výhodou takéhoto prístupu je, že 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. Údržba práv umožňuje zálohovanie na ne-Unixové súborové systémy (napr. flash disk vo formáte FAT). …lokálne alebo SSH… …štatistiky…

Open source … možnosť obnovy bez rdiff-backup.

### Duplicity

Duplicity je software pre zálohovanie, ktorý vytvára šifrované (prípadne kryptograficky podpísané) tar archívy. Implementovaný je v jazyku Python a pri svojom behu taktiež využíva algoritmus 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.

Postup zálohovania v Duplicity je nasledovný: Počiatočná záloha obsahuje všetky dáta (full backup). Pri ďalšej zálohe sa ukladajú len rozdiely (diffy) vypočítané algoritmom rsync voči pôvodným súborom na základe signatúr vypočítaných pri predchádzajúcich zálohovaniach, ktoré sa ukladajú spolu zo zálohou ale aj lokálne na zálohovanom počítači. 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.

***Sem pokec o dôraze na bezpečnosť: šifrovaní, podpisovaní a rôznych možnostiach ukladania záloh.***

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***[footnote s URL]*** **[citácia knihy o Amande]** 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 Unixový nástroj vyvinutý pre zálohovanie diskových oddielov na úrovni blokov, nie súborov a adresárov. Kvôli spätnej kompatibilite ukladania záloh na pásky, sú jeho výstupom bloky pevných veľkostí. GNU tar je Unixový nástroj, ktorý slúži na archiváciu dát na úrovni súborov a adresárov. Oba nástroje možno kombinovať s kompresnými programami ako gzip, bzip2, xz.

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***[footnote s URL]*** je vstavaná zálohovacia služba systému Mac OS X, ktorá vytvára zálohy na externý disk alebo na sieťové úložiská dát podporujúce protokol AFP.

Základné vlastnosti:

* Podpora šifrovania dát
* Záloha celého systému, vrátane systémových súborov, aplikácií, používateľských účtov atď.
* Kompletný obraz systému pre daný čas
* Nezmenené súbory i celé adresáre ukladá ako odkazy (hardlinky) k pôvodným súborom alebo adresárom, do zálohy kopíruje len nové a zmenené súbory a adresáre
* 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 podporujúce protokol AFP. Okrem zariadení od Apple tento protokol implementuje napríklad Unixový balík netatalk.***[pridať poznámku pod čiarou s URL]*** 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).

### Git

Git[[2]](#footnote-2) ***[citácia knihy]*** je decentralizovaný / distribuovaný systém riadenia revízií (DVCS). Primárne slúži pre správu verzií zdrojových kódov. 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.

***Sem by som dal mostík k zálohovaniu, napríklad:*** Aj keď git nie je určený na zálohovanie, používa viacero techník, ktoré sa stali inšpiráciou pre náš zálohovací softvér: obsahom adresované úložisko dát, kombináciu deflate kompresie a delta kompresie, garbage kolekciu.

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 v obsahom adresovanom úložisku
* Delta kompresia
* Deflate 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.

## Predchádzajúca bakalárska práca

Táto diplomová práca vychádza z bakalárskej práce *Zálohovanie v Linuxe s úsporným ukladaním histórie*, ktorej autorom je Patrik Kemény ***[odkaz do literatúry]***. Jej cieľom bolo navrhnúť a implementovať v operačnom systéme Linux nástroje, ktoré budú vykonávať zálohovanie dát na vzdialený server a následne umožnia 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 a použitý softvér

Východiskami pre náš prístup k zálohovaniu a obnove dát sú techniky a existujúce softvérové nástroje, ktorých prehľad uvádzame v tejto sekcii. Najdôležitejšie techniky sú obsahom adresované úložisko, algoritmus rsync, delta kompresia, garbage kolekcia na základe počítania referencií a virtuálne súborové systémy. Využili sme tiež konkrétne implementácie: na delta kompresiu používame program rdiff implementujúci algoritmus rsync, na bezstratovú kompresiu formát gzip, v ktorom sú dáta komprimované algoritmom deflate, pre virtuálny súborový systém využívame techonológiu FUSE.

### Obsahom adresované úložisko

Obsahom adresované úložisko dát (*content-addressed storage*) ***[zdroj!]*** 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 obsahovú adresu)

### Delta kompresia

*Delta kompresia* (rozdielová kompresia, delta compression) ***[citácia]*** je technika kompresie dátových objektov (napríklad súborov), pri ktorej sa namiesto obsahu objektu ukladá iba *rozdiel* (*delta*, *diff*) voči obsahu iného dátovému objektu. Obyčajne ide o *dopredný* rozdiel (forward delta) voči chronologicky predchádzajúcej verzii príslušného objektu alebo *spätný* rozdiel (reverse delta) voči jeho nasledujúcej verzii.

Pre textové dáta možno rozdiel počítať napríklad algoritmom ***ZISTITE AKÝM A CITUJTE***, ktorý je implementovaný v štandarnom Unixovom nástroji *diff* a knižnici ***DOPLŇTE***. Pre textové a binárne dáta je možné na výpočet rozdielu použiť algoritmus rsync, opísaný v nasledujúcej podsekcii.

### Algoritmus rsync

Algoritmus rsync vyvinul Andrew Tridgell ***[citácia]*** na zefektívnenie prenosu dát, ak sa na cieľovom počítači už nachádza ich predchádzajúca verzia.

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 počítača 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 kryptografický hašovací algoritmus MD4, ktorý generuje 128 bitové signatúry.

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 zmenené bajty získané z počítača A alebo jednotlivé zhodujúce sa bloky bajtov v pôvodnom súbore, nachádzajúcom sa na počítači B.

Ako sa rsync dá využiť na delta kompresiu, čo je *rozdiel/diff/delta*, ktorý spomíname v predchádzajúcej sekcii.

### FUSE

FUSE (Filesystem in Userspace) ***[odkaz na knihu o FUSE, footnote s URL],*** teda súborový systém v užívateľskom priestore, je technológia, ktorá umožňuje bežným používateľom Unixových operačných systémov (Linux, FreeBSD, NetBSD, OpenSolaris, Mac OS X) pripájať do stromu priečinkov a súborov vlastné súborové systémy, implementované programami, ktoré bežia mimo jadra (kernelu) operačného systému. Umožňuje tak okrem iného vytváranie virtuálnych súborových systémov z akýkoľvek dát a poskytovaných služieb, nie len z dát uložených na pamäťových zariadeniach priamo pripojených k počítaču.

Súborový systém funguje tak, že kernel dostane požiadavku od klientskej aplikácie, napríklad „vypíš adresár“, a tú prenechá ovládaču súborového systému. Ten následne požiadavku obslúži a spracuje odpoveď, ktorú kernel vráti aplikácii.

Ovládač súborového systému (napr. ext3) je kernelový modul, ktorý s kernelom komunikuje cez rozhranie VFS (virtual file system) a je pevnou súčasťou samotného kernelu. Takýto súborový systém môže obsluhovať len privilegovaný používateľ (root) a ten prideľuje práva k jednotlivým pripájacím bodom (mount points).

FUSE vystupuje v kerneli ako obyčajný súborový systém (viď obrázok 1). Nezaujíma sa však o obslúženie požiadavky a spracovanie odpovede, ale namiesto toho, pomocou knižnice libfuse komunikuje s bežnou používateľskou aplikáciou a tá vykoná všetko potrebné. Akonáhle táto obslužná aplikácia vráti odpoveď v požadovanej forme, FUSE ju prenechá kernelu a kernel ju vráti klientskej aplikácii.

FUSE a aplikácie obsluhujúce súborové systémy musia pre prácu so súbormi implementovať dôležité metódy ako napríklad open, close, read, write alebo seek, definované skupinou štandardov POSIX (Portable Operating System Interface).

Obrázok 1. Tok riadenia FUSE



Výhody:

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

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 definované skupinou štandardov 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 SFTP (secure file transfer protokol) protokole, ktorý umožňuje bezpečný prístup k súborom na vzdialenom počítači. 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 sú 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. Existujú zaujímavé virtuálne súborové systémy pre Wikipédiu (WikipediaFS), Gmail (GmailFS) alebo Dropbox (DropboxFS). Pomocou WikipediaFS môžu používatelia Wikipedie priamo editovať články ako lokálne súbory vo vlastnom textovom editore. GmailFS je súborový systém, pomocou ktorého dokážete využívať svoje Gmailové konto ako úložisko dát.

### Garbage kolekcia na základe počítania referencií

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 počítania referencií (reference counting). Funguje na nasledovnom princípe. Ku každému objektu je asociovaný počet referencií na daný objekt z iných objektov. 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.

## Gzip a kompresia Deflate

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 bezstratovom kompresnom algoritme Deflate, ktorý je kombináciou slovníkového 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.

Algoritmus Deflate v porovnaní s alternatívami ako Burrows-Wheeler (bzip2) alebo LZMA (xz) zvyčajne neposkytuje najlepší kompresný pomer, ale jeho výhodou je rýchla kompresia aj dekompresia, malé pamäťové nároky a dostupnosť v rôznych operačných systémoch. Preto sa často používa, napríklad aj v kompresii šifrovanej komunikácie cez SSH alebo pri prenose hypertextových dát protokolom HTTP.

## 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 nového súboru a signatúry jeho pôvodnej verzie
* Aplikovanie delta súboru na pôvodný súbor a následné vytvorenie nového súboru (patch)

### Databáza typu kľúč-hodnota CodernityDB

Databázy kľúč-hodnota patria k najjednoduchším typom databáz. Z pohľadu systému manažujúceho databázu má každý záznam iba dve polia – kľúč, ktorý slúži na adresovanie záznamu a hodnotu, ktorá je s kľúčom asociovaná. Kľúčom i hodnotou často môže byť ľubovoľná postupnosť binárnych dát. Pre efektívne adresovanie sú kľúče indexované v B-strome alebo hašovacej tabuľke. Takéto databázy často nemajú architektúru klient-server, ale sú implementované ako embedované. Celá funkcionalita databázy je teda implementované ako knižnica, ktorá sa prilinkuje ku klientskej aplikácii. K implementáciám patria napríklad BDB (Berkeley DataBase),

CodernityDB je databáza typu kľúč-hodnota (key-value) s otvoreným zdrojovým kódom, 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 jednoprocesovom, jednovláknovom 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

# Návrh riešenia

V tejto kapitole bude popísaný návrh riešenia našej zálohovacej aplikácie.

## Požiadavky

Naším hlavným cieľom je navrhnúť nástroj na zálohovanie dát, ktorý bude charakteristický nasledujúcimi kľúčovými vlastnosťami.

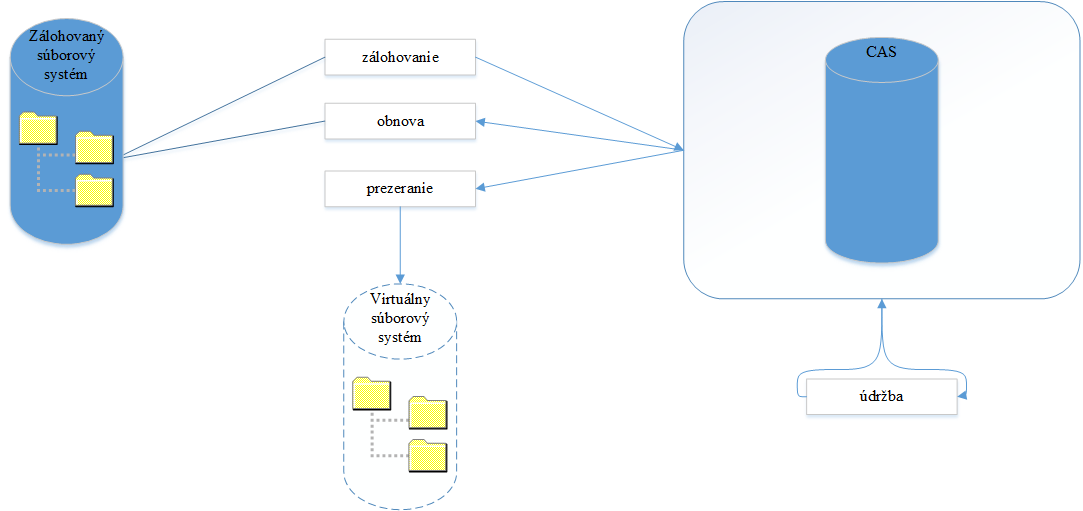
Pri zálohovaní by mal čo najefektívnejšie využívať diskový priestor.

Sprístupňovanie a prehliadanie existujúcich záloh by malo byť realizované nielen pomocou obnovenia dát, ale aj formou postupnosti úplných obrazov systému súborov, ktoré možno prezerať bežným súborovým manažérom a pomocou neho aj prípadne obnoviť vybrané verzie 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. Táto vlastnosť umožní zálohovať v pravidelných krátkych intervaloch, ale staršie zálohy prerieďovať tak, že medzi zálohami sú postupne sa zväčšujúce časové odstupy. Napríklad teda môže používateľ udržiavať 7 záloh s odstupom jeden deň, 4 s odstupom jeden týždeň a 6 záloh s mesačným odstupom.

Zachovanie konzistencie úložiska záloh pri nečakanom prerušení niektorej operácie.

Obrázok 2. Diagram popisujúci základnú funkcionalitu



## Zdieľanie nezmenených dát medzi zálohami

Jedným z prostriedkov, ktoré sme využili na dosiahnutie efektívneho využitia diskového priestoru zálohami, je zdieľanie nezmenených dát medzi viacerými zálohami. Na tento účel sme využili obsahom adresované úložisko dát. To zaručuje, že ten istý dátový objekt (súbor alebo zoznam položiek adresára) nebude uložený dvakrát, aj keď sa zmení jeho meno, umiestnenie v súborovom systéme alebo metadáta (vlastník, prístupové práva, atď).

Navyše objekty adresárov odkazujú na objekty ich položiek obsahovou adresou (obr. 3) bez ohľadu na to, či je súbor v danej zálohe nový, alebo sa vyskytoval už v predchádzajúcej verzii. Tým vytvoríme úplný obraz zálohovaného súborového systému v čase zálohy. Táto vlastnosť často chýba inkrementálnym zálohovacím metódam, ktoré vytvárajú archívy iba zmenených súborov.

Keďže táto práca čiastočne vychádza z pôvodnej bakalárskej práce ***[odkaz do literatúry]***, 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 záloh a selektívna obnova dát

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.

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 virtuálny súborový systém pomocou technológie FUSE. Vďaka nemu 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. Všetky tieto operácie môže vykonať pomocou bežného grafického súborového manažéra alebo štandardnými príkazmi v termináli.

### 2.4 Údržba záloh

GC je iba nástroj pre tento cieľ.

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 ďalšie šetrenie diskového priestoru sme sa rozhodli využiť dve existujúce techniky, ktoré boli podrobnejšie popísané v predchádzajúcej kapitole: delta kompresiu a bezstratovú kompresiu. Objem dát sa budeme snažiť najskôr redukovať delta kompresiou 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.

## Zachovanie konzistencie

Pri zálohovaní dát môže nastať situácia, pri ktorej je zálohovací proces prerušený, či už systémom alebo používateľom. V takomto prípade môže vzniknúť 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 a údržbe 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 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 úložisko z predchádzajúcej práce ***[odkaz do literatúry]*** sme čiastočne využili a po aplikovaní našich navrhovaných riešení sme dosiahli nasledovnú adresárovú štruktúru:

* backups
* objects
* journal
  + backups
  + objects
* 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)

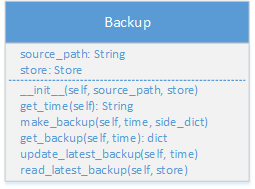
Ďalší adresár má názov *journal* a je využívaný jednoduchým žurnálovacím systémom, ktorému sa budeme venovať neskôr. V princípe sa však skladá z dvoch adresárov s názvom *backups* a *objects*, ktoré majú rovnakú funkciu ako adresáre s rovnakým názvom nachádzajúce sa o úroveň vyššie.

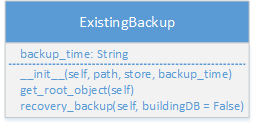
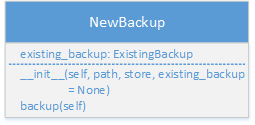
Ďalším a zároveň posledným adresárom našej štruktúry je adresár s názvom *store.db*. Slúži na uchovávanie databázy pre údržbu záloh pomocou garbage kolekcie. Použitiu tejto databázy sa budeme zaoberať v kapitole ***číslo***.

## Trieda Backup a jej podtriedy

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 a jej podtried





### NewBackup

Nové zálohy sú reprezentované pomocou objektov typu *NewBackup*. Táto trieda implementuje len jednu dôležitú metódu s názvom *backup*, ktorá inicializuje rekurzívne zálohovanie jednotlivých objektov zdedených od triedy *SourceObject*. ***Význam dátového atribútu existing\_backup…***

### ExistingBackup

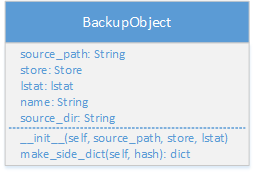
Existujúce zálohy sú reprezentované pomocou objektov typu *ExistingBackup*. Implementuje dve dôležité metódy. Prvou je metóda *get\_root\_object*, pomocou ktorej získame koreňový objekt (adresár) danej zálohy. Druhou je metóda *recovery\_backup*, ktorá slúži na obnovenie danej zálohy. Obnova prebieha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

## Trieda BackupObject

Trieda *BackupObject* je základom všetkých tried reprezentujúcich objekty súborového systému, 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 súboroch:

* 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 úložisko,
* stavové informácie o súbore získané systémovým volaním lstat (typ a práva, vlastník atď).

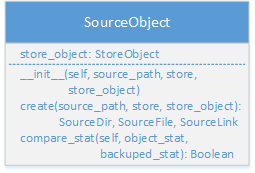
Obrázok 3. UML diagram triedy BackupObject a jej bezprostredných potomkov

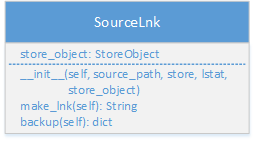
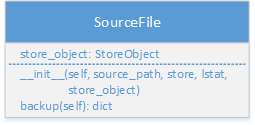
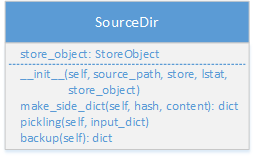


## Modul source

Výrazom *source* (zdroj) označujeme v našej aj predchádzajúcej implementácii všetky zálohované objekty súborového systému, ktoré sú ešte ďalej rozlišované na tri základné typy – súbor, adresár a symbolický link. Triedy reprezentujúce zdrojové objekty sú implementované v module source a sú potomkami triedy SourceObject. Naša implementácia v súčasnosti ignoruje zvyšné typy súborových objektov (socket, pomenovaná rúra, blokové a znakové zariadenie).

Obrázok 3. UML diagram triedy SourceObject a jej podtried





### Trieda SourceObject a jej podtriedy

Pomocou triedy *SourceObject* a jej statickej metódy *create* sú inicializované všetky objekty zdedené od tejto triedy reprezentujúce jednotlivé súbory na strane zdroja (source). Keďže trieda *SourceObject* je potomkom triedy *BackupObject*, stáva sa nositeľom všetkých dôležitých informácií o zálohovanom súbore. Okrem toho obsahuje dátový atribút *store\_object* typu *StoreObject*, ktorá má reprezentovať predchádzajúcu verziu zálohovaného súboru nachádzajúcu sa v úložisku. Pomocou metódy *compare\_stat* sa porovnávajú vybrané stavové informácie (vlastníka, práva, typ, poslednú zmenu a poslednú zmenu metadát) o zálohovanom objekte a objekte, ktorý je už zálohovaný v úložisku. Trieda SourceObject má tri podtriedy SourceDir, SourceFile a SourceLnk pre jednotlivé typy súborových objektov.

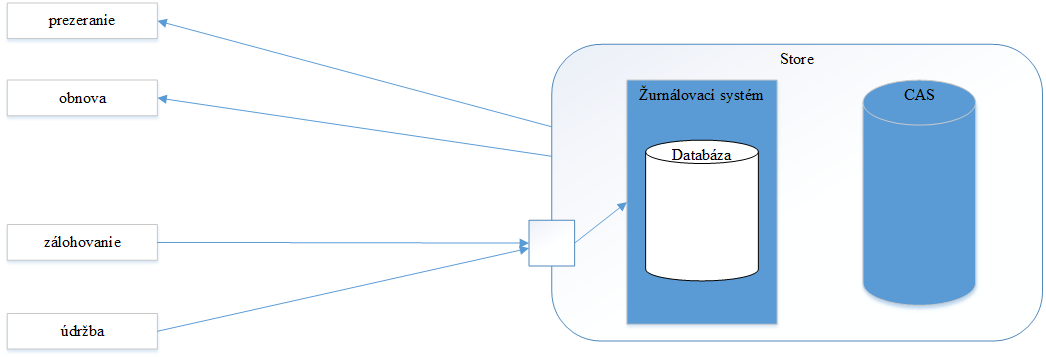
Triedou *SourceDir* reprezentujeme zdrojový (zálohovaný) objekt typu adresár. Jej metóda *backup* vykonáva rekurzívne zálohovanie všetkých objektov, ktoré adresár obsahuje a nakoniec aj samotného adresára.

Pomocou triedy *SourceFile* reprezentujeme zdrojové (zálohované) súbory. Trieda *SourceLnk* reprezentuje zdrojový (zálohovaný) symbolický link. V ich prípade metóda backup na základe store\_object a metódy *compare\_stat* sa pri zálohovaní rozhoduje, či sa vykonáva inkrementálna alebo úplná záloha daného objektu.

## 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

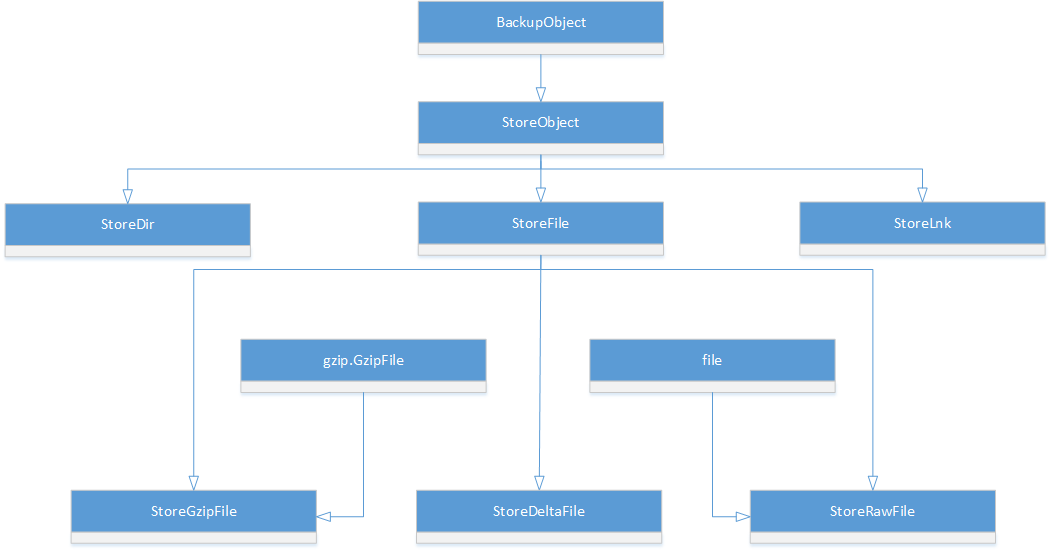


Store implementuje všetky dôležité metódy, potrebné pre zálohovanie a opätovné získavanie dát.

### 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

Trieda *StoreDir* reprezentuje zálohovaný adresár na strane úložiska. Uchováva v sebe jeho pôvodnú štruktúru a taktiež všetky informácie (haš, lstat) o vnorených objektoch.

Obrázok 3. UML diagram triedy StoreDir



Pri jej inicializácii je volaná metóda *unpickling*, pomocou ktorej načítavame knižnicou *pickle* serializovaný slovník uložený v danom súbore v úložisku.

def unpickling(self, store\_path):

with open(store\_path, "rb") as UPF:

pi = UPF.read()

UPF.close()

return pickle.loads(pi)

#### StoreGzipFile

Pomocou triedy *StoreGzipFile* reprezentujeme zálohované súbory v úložisku, ktoré pri samotnom zálohovaní podliehali kompresii pomocou knižnice *gzip*.

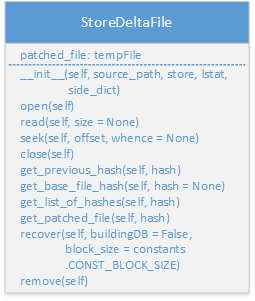
#### StoreRawFile

Trieda *StoreRawFile* reprezentuje objekty v úložisku, ktoré nepodliehajú kompresii pomocou knižnice *gzip* a boli zálohované ako pôvodné dáta.

#### StoreDeltaFile

Táto trieda reprezentuje objekt zálohy, ktorý uchováva len deltu k predchádzajúcej verzii súboru.

Obrázok 3. UML diagram triedy StoreDeltaFile



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:

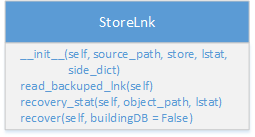
patchProcess = subprocess.Popen(['rdiff', 'patch',

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).

Samotné odstraňovanie záloh prebieha rekurzívne od koreňového objektu zálohy pomocou metódy *remove*, ktorú implementuje každý objekt zdedený od triedy *StoreObject*.

### 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 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. Obnova každého objektu prebieha inak. Pri objektoch typu *StoreDir* postupujeme tak, že najskôr vytvoríme zdrojovú cestu k danému adresáru a následne spustíme obnovu na ďalších objektoch, ktoré sú načítané v slovníku *loaded\_dict* (štruktúra daného adresára).

Obnova objektov typu *StoreLink* je z hľadiska implementácie veľmi jednoduchá. Stačí len vytvoriť daný odkaz pomocou Pythonovskej metódy *os.symlink*.

Posledným typom, ktorý obnovujeme sú všetky objekty zdedené od triedy

*StoreFile*. Pre tieto objekty je implementácia metódy *recover* jednotná. *Z* úložiska získame konkrétny objekt, ktorý je čítaný po blokoch a následne zapisovaný do daného súboru. Implementácia sa líši len pri objektoch typu *StoreDeltaFile*, kde sú dáta čítané z dočasného súboru získaného pomocou metódy *get\_patched\_file*.

Záverečným krokom, ktorý musí byť vykonaný na každom type objektu je obnova jeho lstat (práva, vlastník atď). Za týmto účelom bola implementovaná metóda *recovery\_stat*.

### 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

Implementácie metód *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 synchrónne.

# 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

1. <http://www.nongnu.org/rdiff-backup/> [↑](#footnote-ref-1)
2. http://git-scm.com [↑](#footnote-ref-2)