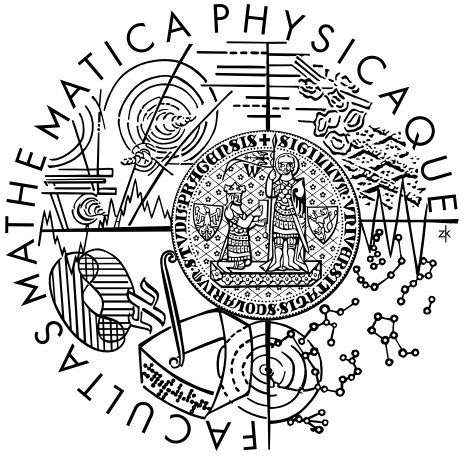
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jiří Setnička

Zálohovací systém

Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, Ph.D.
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2015

Rád bych poděkoval mému vedoucímu práce Mgr. Martinu Marešovi, Ph.D. za nadnesení tématu, které mě zaujalo, a za jeho konzultace a rady k vypracování práce. Těším se na případnou další spolupráci s rozvíjením vzniklého zálohovacího systému. Současně děkuji také bezpočtu dalších, kteří mi nějakou radou, myšlenkou, co by systém mohl umět, nebo jinou podporou pomohli práci dokončit.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....

Název práce: Zálohovací systém

Autor: Jiří Setnička

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, Ph.D., Katedra aplikované matematiky

Abstrakt: Přehled existujících přístupů k zálohování dat a zvážení jejich předností s ohledem na použití. S uvážením předchozího poté popis souběžně s touto prací vzniklého zálohovacího systému FenixBackup jako souborově orientovaného, schopného si držet více historických verzí a pracujícího s rozdílovými zálohami. V závěru je nastíněn možný směr, kterým se může vývoj systému ubírat.

Klíčová slova: zálohování, diff, obnova dat

Title: A backup system

Author: Jiří Setnička

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: An overview of existing approaches to data backup and their strengths with regard to use. After considering the previous describe the resulting backup system FenixBackup as file-oriented differential backup system capable holding more historical versions. In conclusion, outlines the possible directions which the development of the system could follow.

Keywords: backup, diff, data recovery

Obsah

Úvod

Potřeba zálohovat data se pojí s počítačovým světem prakticky od jeho vzniku a vychází z dob ještě před jeho existencí – z potřeby udržovat kopie například důležitých dokumentů a chránit je tak před náhodným nebo cíleným zničením nebo ztrátou.

V dřívějších dobách stačilo pořídit fyzickou kopii důležitého dokumentu a ten uložit ideálně na jiném fyzickém místě, s nástupem počítačového ukládání dat se k tomu však ještě přidal aspekt toho, že data je možná snadno editovat a měnit.

Pokud tedy v současné době mluvíme o zálohování, je potřeba toto brát v potaz a přizpůsobit tomu strategie zálohování. Nestačí pouze pořídit kopii dat ve chvíli jejich vzniku, ale je nutné dívat se na data jako na dynamicky se měnící objekt, u kterého chceme tyto změny sledovat, a mít možnost vrátit se k dřívějšímu stavu, pokud je to zapotřebí.

Pokud si zkusíme rozebrat situace, před kterými by nás zálohovací systém měl chránit, napadne nás pravděpodobně následující:

1. Závada pevného disku či jiného ukládacího média – Před tímto nás zálohování ochrání možností obnovy poslední zálohované verze dat, ale v případě běžného diskového úložiště se zde nabízí ještě jiná lepší alternativa a to použít nějakou variantu RAID[1].

2. Ztráta, odcizení nebo zničení celého počítače (živelná katastrofa) – Zde nám RAID typicky nepomůže, protože jednotlivé disky v něm bývají umístěny na stejném fyzickém místě. Data se dají zachránit z poslední provedené zálohy (zde velmi záleží na politice zálohování, jak je záloha aktuální).

3. Vrácení se do stavu před provedením nějaké akce – Toto se hodí například při neúmyslném smazání některých souborů, nebo při nepovedené úpravě systému (instalace nového software aj.). Zde nám naopak techniky jako RAID nepomohou, protože ty se starají jen o redundanci nejaktuálnější verze uložených dat a nemají historii.

4. Odhalování útoků na zálohovaný stroj – Při podezřelé aktivitě na zálohovaném stroji lze porovnáním změn v souborech zjistit, jestli nebyl tento stroj napaden (a nebyly například provedeny změny na systémových souborech) a v jakém časovém období se tak případně stalo.

Pro první a druhý případ nám stačí držet si vždy jen poslední verzi souborů. Pokud ale chceme, aby náš zálohovací systém pokrýval i situace popsané ve třetím a čtvrtém bodě, je potřeba si nějakým způsobem udržovat více historických verzí a umět si vyvolat stav dat libovolné z těchto verzí, případně si mezi libovolnými dvěma zobrazit rozdíly.

Dalším důležitým aspektem zálohovacího systému diskové místo potřebné k zálohování zálohovaného stroje. S nějakým potřebným místem je nutné počítat, ale přílišné plýtvání (například způsobené ukládáním celé nové verze souboru při

[1] Redundant Array of Inexpensive/Independent Disks – systém ukládání dat na více nezávislých disků

jeho drobné změně) není na místě. Stejně tak je většinou potřeba „ředit“ staré verze záloh a držet si jen ty významné.

Neméně důležitou věcí je to, že ke všem datům také nepřistupujeme stejně. Některá data jsou pro nás typicky významnější a je pro nás důležitější zálohovat změny v nich prováděné, jiná data tak významná nejsou a není třeba nutné držet si od těchto dat takové množství historických verzí.

Poslední zásadní věcí je pak pro nás čas provedení zálohy. U strojů stále připojených k rychlé síti to není až tak důležité, ale u přenosných počítačů, které jsou k rychlé síti připojeny jen omezenou dobu nebo u strojů připojených po pomalé síti je toto důležité. Zde přichází ke slovu jednak správné sledování a přenášení jen těch dat, která se změnila, tak prioritizace dat na základě jejich významu pro nás, jak bylo popsáno v předchozím odstavci.

Na všechny tyto aspekty jsem se pokoušel při vývoji svého zálohovacího systému (nazvaného podle bájného mýtického ptáka postávajícího po svém zničení z popela, stejně jako to chceme od zálohovaných dat) brát zřetel a zohlednit je. Vznikl tak zálohovací systém *FenixBackup*.

1. Existující metody a systémy zálohování

Jak již bylo popsáno v úvodu práce, dá se k zálohování přistupovat mnoha způsoby. Pojďme si zkusit tyto způsoby rozebrat, podívat se na ně z několika hledisek efektivity a zhodnotit jejich klady a zápory. Dále se zkusme podívat na několik existujících zálohovacích řešení.

1.1 Jednoduché kopírování

Při zálohování vždy jde o to držet si nějakým způsobem kopii nějakých dat. Výběr dat k zálohování nyní odložíme stranou a budeme si představovat, že máme nějakou dobře definovanou množinu souborů, které chceme zálohovat.

Nejjednodušším způsobem co do implementace je asi **ruční kopírování** – uživatel sám čas od času zkopíruje zálohované soubory na nějaké jiné úložiště. Na tomto úložišti si uživatel může držet buď jen jednu kopii dat (vždy jen nejnovější verzi), což zabere řádově tolik místa, kolik zabírají zálohovaná data na zálohovaném stroji, nebo si může držet všechny historické verze. V takovém případě ale zálohy zaberou místo [počet záloh] × [velikost dat], což je pro časté zálohování velkého objemu dat neudržitelné (objem dat lze sice snížit vhodnou kompresí, ale to oddálí problém jen o kus dále).

Jako metodu kopírování lze zvolit mnoho protokolů, od kopírování po místním souborovém systému (což nebývá moc účinné proti selhání celého počítače nebo jeho ztrátou), přes kopírování přes síť pomocí FTP[1] nebo nad SSH[2] postaveným SCP[3].

Všechny tyto metody kopírování jsou ale „hloupé“, udělají jen přesně to, co od nich žádáme – zkopírují zadané soubory na cílové úložiště bez starosti o to, jestli již zde stejné soubory nebyly kopírovány dříve. Pokud byly provedeny jen malé změny, tak přenášíme zbytečně velké objemy dat, které již v cílovém místě uložená jsou.

Kdybychom dokázali dostatečně bezpečně poznat, jaká data se nezměnila, mohli bychom objem přenášených dat výrazně snížit a kopírovat jen rozdílná data. Když zálohování provádí uživatel sám a ručně, může posloužit jako rozhodovací veličina v tom, která data se podle jeho mínění změnila a která je tedy potřeba zálohovat. To je ale jednak nespolehlivé a může to vést k chybám, ale hlavně se tento postup nedá použít při zautomatizování zálohování (což by mělo být jedním z hlavních cílů zálohovacích systémů).

[1] File Transfer Protocol – jednoduchý síťový protokol pro přenos souborů
[2] Secure shell – zabezpečený komunikační síťový protokol
[3] Secure copy – zabezpečený protokol pro přenos souborů na bázi SSH

1.2 Kopírování rozdílů

Když si vezmeme za cíl kopírovat jen ta data, která se oproti poslednímu zálohovanému stavu změnila, dostáváme se k otázce, jak dostatečně bezpečně poznat, co se změnilo.

Vzhledem k vlastnostem v současnosti používaných souborových systémů je nejsnadnější dívat se na změny soubor po souboru a v případě změny ho přenést celý. Dá se zavést i drobnější granularita a sledovat data po menších blocích, než po celých souborech, ale to je již výrazně obtížnější.

Současné souborové systémy si u každého souboru pamatují několik parametrů (vezmeme si za příklad souborové systémy vycházející z operačního systému UNIX[4]): vlastník a skupina, vlastnická, skupinová a ostatní práva, čas poslední změny, velikost souboru a několik dalších parametrů.

Pokud uvažujeme soubory s parametry, je potřeba odlišit změny pouze v nich (například změna vlastníka či skupiny) od změn na datech souboru. Parametry souboru je nutné si přečíst vždy a tak změny na nich poznáme jednoduše, možností jak poznat změnu na datech souboru je pak vícero:

- Kompletní porovnání obsahu souboru po bytech – zabere čas lineární s velikostí souboru a stejnou přenosovou kapacitu, je ale neomylné.
- Kontrolní součet obsahu souboru – je nutné stále spočítat kontrolní součet (třeba pomocí MD5[5] nebo SHA1[6]) souboru v lineárním čase k jeho velikosti, ale přenést mezi zálohovaným a zálohovacím strojem je potřeba pouze tento kontrolní součet (a porovnat ho s uloženým). Při volbě dobré hashovací funkce se věří, že nalezení *kolize* (nalezení více různých souborů se stejným hashem) je extrémně nepravděpodobné a tedy spolehlivé.
- Porovnání jen parametrů (času poslední modifikace a velikosti) – je nejrychlejší (jde jen o několik porovnání v konstantním čase) a pro většinu situací je dostačující.

Důvodem, proč je poslední možnost považována za dostačující je ta, že je při běžném provozu jen velmi málo situací, kdy se změní obsah souboru, ale nezmění se čas jeho poslední modifikace a jeho velikost. Většinou to vyžaduje přímou a vědomou akci, kterou se čas modifikace souboru nastaví zpět na původní hodnotu.

Situace, ve které je porovnání jen podle velikosti a času modifikace nedostačující, je již nastíněné odhalování útoků na zálohovaný stroj. Pokud se již útočníkovi povede získat kontrolu nad strojem, může snadno modifikovat systémové soubory a potom změnit čas jejich modifikace nazpět. Pokud se před takovou situací chceme chránit, je vhodné občas nechat zkontrolovat alespoň kontrolní součty všech souborů oproti verzi uložené v zálohovacím systému.

[4]Operační systém vzniklý v roce 1969, který dal vzniknout celé rodině odvozených systémů – Linux, BSD, Mac OS X, aj.

[5]Kryptografická hashovací funkce vytvářející z libovolného vstupu výstup o velikosti 128 bitů

[6]Secure Hash Algorithm – pokročilejší kryptografická hashovací funkce, vytváří otisk o velikosti 160 bitů

V současnosti jedním z nejvíce využívaných protokolů a současně i programů implementující nějakým způsobem kopírování jen rozdílných souborů je **rsync** (viz [2]). Ten potřebuje na jedné straně běžícího klienta a na druhé straně běžící server. Ty si mezi sebou vyměňují různé parametry a kontrolní součty souborů a jejich částí. Hlavním cílem protokolu je snížit objem síťové komunikace na minimum a přenášet jen nezbytné množství dat (přenos navíc probíhá komprimovaně).

Bohužel primárním cílem rsyncu je *synchronizace* dvou složek a není tak úplně vhodný k tomu držet si více historických verzí záloh – to by pravděpodobně znamenalo mít synchronizovanou složku pro každou historickou verzi a to rsync neumí dělat efektivním způsobem. Neumí v nových složkách synchronizovat jen změny oproti úplně jiné složce, bylo by potřeba si vždy na zálohovacím stroji zkopírovat celou starší zálohu do nové složky a pak synchronizovat až vůči ní.

Toto je obecně problém většiny synchronizačních protokolů a programů, takže se většinou nedají samostatně použít k účinnému zálohování. Je však možné je využít jako efektivní přenosovou část zálohovacího systému.

1.3 Rozdílové zálohování

Pokud už přenášíme pouze rozdíly, nešel by udělat další krok a místo vždy celých kopií zálohovaných dat pro každou historickou verzi si ukládat jen tyto rozdíly? Tím bychom odstranili významný problém nastíněný výše, a to přesněji místo potřebné k uložení záloh (což by bez rozdílového zálohování znamenalo ukládat velký datový objem: [počet záloh] × [velikost dat]).

Pokud se na problém podíváme v prvním přiblížení, stačí nám si jen na počátku uložit plnou kopii dat a všechny další historické verze si ukládat jen jako rozdíly vždy oproti té předchozí. Pro získání konkrétní verze dat nám pak stačí jen na původní data aplikovat postupně všechny zapamatované rozdíly až do nějaké verze.

Pro získání rozdílu textových dat (takový rozdíl se typicky nazývá anglickým výrazem *diff* nebo u binárních dat někdy *delta*) je možné použít mnoho nástrojů porovnávajících zadané textová data řádek po řádku. Produkují pak takzvaný *patch*, který obsahuje řádky, které ubyly, a řádky, které naopak přibyly.

U binárních dat je situace složitější a není tak jednoduché stanovit, čeho by se měl binární diff držet a co by mělo být jeho výsledkem. Pro sjednocení mnoha implementací vznikl formát VCDIFF[7]. Ten specifikuje tři různé instrukce: pro přidání nové sekvence, pro zkopírování úseku ze staré sekvence a pro opakování určité sekvence dat.

Dá se použít i pro kompresi (i když pro tyto účely existují lepší algoritmy a formáty), ale hlavně se dá za sebe zřetězit stará a nová verze binárních dat, ale kóduje se jen úsek odpovídající nové verzi dat. Díky tomu, že se ale může odkazovat na sekvence v původní verzi dat, vznikne tak vlastně jen požadovaný binární diff.

[7]Formát a algoritmus definovaný v RFC 3284 založený na článku *Data Compression Using Long Common String*

Problémy reálné implementace

V reálné implementaci rozdílové zálohování musíme vyřešit ještě několik drobných problémů. Prvním z nich je to, že postupně se nabalující rozdíly způsobí, že při delším provozu zálohovacího systému bude získání nejnovější verze dat trvat neúměrně dlouho. Bude totiž nutné začít u první verze a postupně aplikovat všechny rozdíly – to u několika málo verzí ještě nepředstavuje problém, ale u několika desítek až stovek verzí to již začne velice zdržovat.

Řešením je omezit maximální hloubku odkazování a pokud by měla být nová verze v již moc velké hloubce, uložit ji buď jako kompletní verzi (tedy by se z ní stala nová „nultá hladina“), nebo jako diff oproti vhodné verzi v menší hloubce. Zde se dá uplatnit heuristika vybírající nejvhodnější předchozí verzi (například s nejmenším diffem).

Druhým problémem, před který se musí reálná implementace postavit, je možnost mazat historické zálohy. Dokud byla každá historická verze nezávislá na ostatních, tak nás její smazání nic nestálo a nijak neovlivnilo okolní verze. U na sebe navazujících rozdílů si takové prosté smazání nemůžeme dovolit, protože bychom tím znehodnotili všechny verze navazující v posloupnosti rozdílů na mazanou verzi.

Pokud tedy chceme mazat historické verze (a to je u dlouhodobě fungujícího zálohovacího systému nezbytné), je potřeba implementovat řešení nějakým způsobem distribuující mazané změny do navazujících verzí, neboli přepočítávat navazující verze, aby zahrnovaly i mazané změny.

1.4 Automatizace

Výše popsané způsoby se dají snadno zautomatizovat. Pokud nevyžadují uživatelovo aktivní rozhodování během zálohování, lze naplánovat jejich pravidelné spouštění v naplánovaných intervalech.

Pro tuto automatizaci lze využít mnoho různých postupů. Buď může zálohovací systém stále běžet jako systémová služba (démon) a hlídat si čas plánovaného zálohování sám, nebo lze využít například v UNIXu dostupný *cron* na naplánování spouštění přednastaveného příkazu.

2. Vlastnosti zálohovacího systému FenixBackup

Zálohovací systém *FenixBackup* dostal své jméno podle mýtického ptáka fénixe, který dokázal znovu a znovu povstávat znovuzrozený ze svého popela. To je docela trefná paralela obnově zálohovaných dat, když nás potká nějaká nešťastná událost.

Nyní se pokusíme představit si hlavní myšlenky, na kterých FenixBackup staví, a základní principy jeho fungování.

2.1 Základní přehled

FenixBackup je souborově orientovaný zálohovací systém schopný si držet více historických verzí souborů. Každý soubor je v něm navíc ukládán nezávisle a tak je možné si od určitých souborů držet více historických záloh, než od jiných.

Ukládání jednotlivých verzí funguje na principu rozdílových záloh ve formátu VCDIFF s omezením na maximální hloubku na sebe navázaných delta rozdílů.

Další možností bylo ukládat vždy celé nové verze souborů, sdružovat je k sobě a komprimovat, jako to dělá třeba verzovací systém Git (stavějící na myšlence, že podobná data vedle sebe se při komprimaci vhodným kompresním algoritmem výrazně zmenší), ale zvolil jsem raději cestu přes VCDIFF. Při pokusech s několika většímu binárními soubory vycházely delta rozdíly menší, než když jsme jednotlivé verze vzal a dohromady zkomprimoval.

Velký důraz byl kladen také na snadnou, přehlednou, ale přitom dostatečně mocnou schopnost konfigurace, které by měla umožňovat popsat všechny myslitelné scénáře zálohování.

Dalším mým cílem bylo umožnit snadné rozšiřování systému a to jednak ve formě zajištění zpětné kompatibility dat, tak také ve snadném rozšiřování systému. Příkladem toho je systém *adaptérů* pro přístup k zálohovaným datům: Existuje společný interface pro komunikaci s adaptéry, ale to, jestli se data získají z lokálního filesystému, stáhnout přes SSHFS[1], nebo bude probíhat komunikace přes rsync protokol, by mělo být samotnému jádru zálohovacího systému jedno.

2.2 Pravidla konfigurace

Zásadní částí dobře fungujícího zálohovacího systému je dobře fungující a konfigurovatelný systém pravidel zálohování. Zápis pravidel do konfiguračního souboru je ukázán v kapitole vztahující se k uživatelské dokumentaci, zde se pokusíme rozebrat obecné fungování pravidel.

2.2.1 Zálohovací pravidla určená podle cest

Pravidla je možné specifikovat pro jakoukoliv cestu v zálohovaných datech (kořenová složka zálohy je označována pomocí lomítka: `/`) a stejnou cestu je možné

[1] SSH File System – připojení vzdáleného souborového systému přes protokol `SSH`

uvést i vícekrát – v takovém případě platí pro tuto cestu poslední specifikovaná pravidla.

Pokud pro nějakou cestu nejsou specifikována pravidla (v reálném případě to bude pravděpodobně většina cest), použijí se pro ní nejbližší vyšší pravidla po cestě ke kořeni. Je možné tak třeba zakázat zálohování určitých složek.

Přehled a význam parametrů, které lze nastavovat:

- `scan <true|false>`: Platí jen pro složky a říká, jestli vůbec tuto složku prozkoumávat. Defaultně `true`.
- `backup <true|false>`: Jestli zadaný soubor zálohovat, nebo ne. Defaultně `true`.
- `history <1-10>`: Parametr pro funkci na ředění historie, čím vyšší číslo, tím více verzí držet. Defaultně 1.
- `priority <1-10>`: Parametr pro prioritizační funkci. Určuje, které soubory zálohovat dříve, než ostatní. Čím vyšší číslo, tím důležitější. Defaultně 1.

2.2.2 Zálohovací pravidla podle specifičtějšího výběru

Mimo pravidel odvozených jen od cest existují také pravidla umožňující pokrýt třeba jen určitý typ souboru. Jsou také vázána na konkrétní cestu (a platí tak jen v určeném podstromě), ale je možné u nich zavést složitější způsoby filtrování.

Pro každou cestu je možné v konfiguraci uvést seznam *pravidel pro soubory*, kde každé pravidlo má parametry, které nastavuje, a pak filtry určující, na jaké soubory se má aplikovat. Je možné filtrovat podle následujících parametrů:

- Regex[2] názvu souboru (`regex`) – pokud je nevyplněný či prázdný, odpovídá jakýkoliv soubor, pokud je neprázdný, musí celý název souboru (jen souboru, bez cesty) odpovídat zadanému regexu.
- Regex cesty k souboru (`path_regex`) – podobně jako výše, nevyplněnému odpovídá vše, vyplněnému jen soubory, které se nacházejí (oproti aktuálnímu adresáři) na cestě odpovídající regexu.
- Minimální a maximální velikost souboru v bytech (parametry `size_at_least` a `size_at_most`).

Jednotlivá pravidla se vždy aplikují v pořadí, v jakém jsou uvedena v konfiguraci, a funguje zde princip přepisování (tedy pokud je splněna podmínka pravidla, jsou přepsány všechny parametry zálohování, které pravidlo nastavuje).

[2] regulární výraz

Při dotazu na konkrétní soubor se pak parametry pravidel určí takto:

1. Vezmou se defaultní hodnoty pravidel a začne se v kořenové složce.
2. V aktuální složce se nejdříve aplikují pravidla složky (pokud existují) a poté se projdou všechny pravidla pro soubory:
3. Pokud vyhovuje filtr u pravidla pro soubory, aplikuje se.
4. Pokud se dá po zadané cestě sestoupit ještě o úroveň níže, sestoupí se a opakuje se znovu od bodu 2.

Nakonec se vrátí finální pravidla vzniklá aplikací všech pravidel, přes která se prošlo. Toto řešení nabízí velmi silné možnosti konfigurace, bohužel cacheování výsledků je zde vzhledem k pravidlům s filtry značně obtížné, pro běžně velké konfigurační soubory by ale rychlost i tak měla být dostatečná a úzkým hrdlem zde stále bude rychlost načítání dat z disku.

2.3 Model ukládání dat

FenixBackup představuje každou zálohu (ať už se v rámci ní zálohovaly všechny soubory, nebo se povedlo zálohovat pouze jediný) jedním souborovým stromem. Ten obsahuje všechny záznamy o všech souborech, které by se měly dostat do zálohy.

Při svém vzniku obsahuje tento strom jen informace zjistitelné o souborech bez nutnosti číst jejich obsah (to zajistí příslušný adaptér, viz následující podkapitola) a postupně se zpracováním obsahu jednotlivých souborů se v něm objevují i kontrolní součty (hashe) souborů, které odkazují na konkrétní datové bloky.

Každý uložený soubor je jednoznačně identifikován svým SHA256[3] hashem a představován stejnojmenným datovým blokem. Tento datový blok může být buď úplný, nebo je to jen VCDIFF oproti jinému datovému bloku.

Pro fungování zálohovacího systému předpokládáme, že nalezení kolize v kryptografické hashovací funkci jako je SHA256 je extrémně nepravděpodobné (zatím není známý ani cílený útok, který by uměl vyprodukovat kolizi, a velikost kolizní domény je 2^{256}). Na stejném předpokladu o nepravděpodobných kolizích v reálném nasazení staví i verzovací systém Git, který navíc používá jen kratší hashe generované SHA1.

Díky tomu, že jsou datové bloky pojmenovány hashem souboru, který mají představovat, řeší se velmi efektivním způsobem duplikáty souborů. Ačkoliv třeba dva stejné soubory mohou mít úplně rozdílnou historii, tak finálně je uložen v zálohovacím systému jen jeden datový blok, který představuje oba soubory.

Datové bloky a jejich případné rozdílové návaznosti tedy vůbec nemusí korespondovat s historií jednotlivých souborů uloženou v souborových stromech. Ale díky tomu, že v souborovém stromu je uložený hash každé jednotlivé verze souboru, lze všechny historické verze snadno zrekonstruovat.

Záznam o každém jednotlivém souboru se může navíc nacházet v jednom z následujících stavů:

[3]Kryptografická hashovací funkce, vytváří otisk o velikosti 256 bitů

- `UNKNOWN` – Nový soubor (bez známé minulé verze), u kterého zatím nemáme zálohovaný jeho obsah.
- `NEW` – Nový soubor (bez známé minulé verze), u kterého již máme uložený a zpracovaný jeho obsah.
- `UNCHANGED` – Soubor se známou minulou verzí, který se nezměnil (ani parametry, ani obsahem).
- `UPDATED_PARAMS` – Soubor se známou minulou verzí, u kterého se změnily pouze parametry, ale obsah ne.
- `NOT_UPDATED` – Soubor se známou minulou verzí, u kterého se změnil i obsah, a jenž ještě nemáme uložený.
- `UPDATED_FILE` – Soubor se známou minulou verzí, u kterého se změnil i obsah, a jenž již máme uložený a zpracovaný.
- `DELETED` – Soubor byl z nějakého důvodu vymazán (ať už na přímé přání uživatele, nebo automaticky při uvolňování místa).

2.4 Adaptéry pro přístup k zálohovaným datům

Zálohovací systém je psán tak, aby byl nezávislý na metodě přístupu k zálohovaným datům. Samotnému jádru je jedno, jak data získá, o to se starají takzvané *adaptéry*.

V úloze zálohování se dají jejich úkoly rozdělit do dvou fází. V první fázi získávají adaptéry informace o souborech a předávají je jádru zálohovacího systému, ve druhé fázi jim pak jádro zálohovacího systému dodá seznam souborů, které chce získat, a adaptér nějakým způsobem opatří jejich obsah a dodá je nazpět jádru.

První fáze začíná tím, že adaptér začne skenovat systém souborů od zadaného místa, načítá parametry souborů (vlastník, práva, velikost, ...) a ukládá tyto záznamy do souborového stromu. Jádro zálohovacího systému všechny ukládané záznamy zpracovává a může případně operativně rozhodnout, že ho informace v nějakém podstromě již nezajímají (třeba v případě nastavení `scan: false` pro nějakou cestu v konfiguraci). V takovém případě to dá domluveným způsobem vědět adaptéru a ten by již tento podstrom neměl dále procházet.

Během této fáze se jádro zálohovacího systému pokouší (podle cesty, času poslední modifikace a velikosti souborů) rozhodnout, které soubory se nezměnily, u kterých se změnily pouze parametry, a které soubory bude potřeba stáhnout pro uložení jejich nové verze.

Seznam požadovaných souborů (utříděný podle prioritizační funkce, viz dále) pak předá nazpět adaptéru a ten by měl zařídit získání obsahu těchto souborů a předání obsahu jádru zálohovacího systému.

To si spočte hash souborů, zjistí, zdali případně nemá již soubor se stejným hashem uložený, a pokud ne, uloží nový datový blok (buď jako kompletní datový blok, nebo jako VCDIFF oproti jinému).

2.5 Priorita zálohovaných souborů

Různé soubory mohou mít konfigurací jinak nastavené priority zálohování, čímž lze zálohovacímu systému sdělit, o které soubory máme větší zájem.

Do rozhodovacího procesu se také promítá to, jakou nejmladší verzi daného souboru již máme uloženou. Systém je totiž připravený i na to, že ne vždy se povede provést kompletní zálohu, například u přenosných počítačů připojených k rychlé síti jen po omezenou dobu. V takovém případě zůstanou některé soubory ve stromu se stavem `UNKNOWN` nebo `NOT_UPDATED` a další zálohy na to berou ohled (soubory, jež se nepovedlo zálohovat delší dobu, mají při příštím zálohování přednost).

Konkrétně si jádro zálohovacího systému přiděluje každému souboru číselné *skóre* odpovídající následujícímu vztahu:

$$[\text{čas od poslední zálohované verze}] \times [\text{priorita souboru}]$$

Pokud soubor nemá známou minulou verzi, tak se jako čas poslední zálohované verze bere čas minulého spuštění zálohování – protože tento nový soubor se mohl objevit kdykoliv v mezičase mezi touto poslední zálohou a současností.

Tímto postupem by se mělo zajistit zálohování postupně všech souborů i v případě, že zálohovaný stroj bývá připojen k zálohovacímu systému pravidelně jen po omezenou dobu.

3. Uživatelská dokumentace

FenixBackup se v současnosti ovládá z příkazové řádky, ale do budoucna není problém nad samotným jádrem postavit více jinak fungujících rozhraní (třeba GUI). Pro jeho fungování je nezbytný konfigurační soubor, který popisuje co, jakým způsobem a kam se má zálohovat.

3.1 Konfigurační soubor

Konfigurační soubor se vztahuje vždy k jednomu zálohovanému stroji a obsahuje tři části. První specifikuje úložiště zálohovaných dat, druhá z nich popisuje způsob připojení k zálohovanému stroji – volba adaptéru a parametry pro něj (například cesta v rámci lokálního souborového systému, nebo adresu a port stroje, na který se připojit) a poslední a nejobsáhlejší nakonec popisuje zálohovací pravidla pro jednotlivé soubory.

Při spuštění zálohovacího systému je nejdříve načtena celá konfigurace a teprve, když se jí povede celou zpracovat bez chyby, přistupuje se k provádění příkazů.

3.1.1 Ukázková konfigurace

Konfigurace pro zápis používá v mnoha projektech rozšířený zápis poskytovaný knihovnou *libconfig*.[1]

Protože je velmi časté používat společná zálohovací pravidla pro více strojů, dají se tato společná pravidla sdružit do společných souborů a ty pomocí direktivy `@include` v konfiguraci používat opakovaně na více místech.

Ukázka je připojena níže.

```
baseDir: "./backup_dir"
dataSubdir: "data"
treeSubdir" "trees"

adapter: {
  type: "local_filesystem"
  path: "/home/jirka/fenix_backup"
}

paths: (
  { path: "/"
    file_rules: (
      @include "no_exe.config"
    )
  },
  {
    path: "/.git"
    scan: false
  },
```

[1] `http://www.hyperrealm.com/libconfig/libconfig_manual.html`

```
  { path: "/test_backup/"
    scan: true
    file_rules: (
      { backup: false },
      { regex: ".*\.fenixtree"
        size_at_least: 10
        size_at_most: 10485760
        backup: true
      }
    )
  }
)

# Obsah souboru no_exe.config:
{ regex: ".*\.exe"
  backup: false
}
```

První položka `baseDir` je povinná a popisuje, kam přesně má zálohovací systém ukládat svá data. Pak lze volitelně modifikovat názvy podsložek na ukládání dat souborů a souborových stromečků (`dataSubdir` a `treeSubdir`).

Druhá položka `adapter` specifikuje způsob získávání zálohovaných dat. V první řadě je specifikováno, který adaptér se má použít, a zbytek položek je pak předán tomuto adaptéru jako jeho nastavení.

Poslední části konfigurace popisuje pravidla pro cesty a specifičtější pravidla určovaná filtry, jejichž fungování jsme rozebrali v minulé kapitole.

3.2 Použití z příkazové řádky

FenixBackup je psaný primárně pro spouštění příkazů z příkazové řádky s cílem umožnit snadné plánování akcí třeba pomocí `cronu`.

Základem je spuštění příkazu `fenix`, kterému se vždy předá cesta ke konfiguračnímu souboru a poté příkaz, který chceme provést. Inspirací pro rozhraní bylo zčásti rozhraní verzovacího systému Git (viz [1]).

Pomocí tohoto rozhraní lze provádět všechny potřebné akce:

- Zobrazit seznam záloh
- Zobrazit seznam souborů v záloze
- Sledovat historii jednoho konkrétního souboru
- Provést zálohu nebo čištění
- Obnovit jeden soubor, podstrom nebo všechny soubory z určené zálohy buď do původního místa, nebo do specifikované cesty

3.2.1 Popis příkazů

Přesné použití rozhraní nejlépe popisuje usage-note příkazu `fenix`:

```
Usage: ./fenix <config_file>
And one of these commands:
  show backups (displays list of all backups)
  show files [<backup>] (displays list of all files in given backup)
  show history <backup> <path>(displays known history of given file)
  backup (run backup)
  restore full <backup>(run full restore to original path)
  restore full <backup> <path>(run full restore to given path)
  restore subtree <backup> <subtree_path>
(restore subtree to original path)
  restore subtree <backup> <subtree_path> <path>
(restore subtree to given path)
  restore file <backup> <file_path>
(restore one file to original path)
  restore file <backup> <file_path> <path>
(restore one file to given path)
  cleanup [<x>] (run <x> rounds of cleanup, default 1)
```

4. Implementace zálohovacího systému FenixBackup

Zálohovací systém je napsaný v jazyce C++, konkrétně podle specifikace C++11. Vyvíjen byl na platformě operačního systému Linux a je optimalizován pro jeho systém práv a parametrů souborového systému, ale měl by fungovat na všech hlavních platformách.

4.1 Obecné návrhové vzory

Při implementaci jsem se držel snahy mít v hlavičkových souborech tříd co možná nejméně položek nepotřebných jako rozhraní pro komunikaci s ostatními třídami. Většina privátních proměnných a funkcí je tedy (podle PIMPL[1] idiomu) zapouzdřena uvnitř vnořených tříd

Důvodem pro tuto snahu je za prvé přehlednost hlavičkových souborů a za druhé odstranění potřeby překompilování všech souborů, které daný hlavičkový soubor includují při změně v privátní metodě.

Dále se v celém systému hojně používají chytré pointery zavedené v C++11, které na mnoha místech zjednodušují zacházení s instancemi tříd, pokud si na ně potřebujeme držet odkaz z více míst zároveň.

Vše bydlí ve jmenném prostoru *FenixBackup*.

4.2 Přehled tříd a jejich funkce

V minulé části jsme si nastínili schéma stromečků reprezentujících jednotlivé zálohy a držící informace o souborech, a schéma datových bloků.

Souborové stromečky jsou představovány třídou *FileTree* a obsahují většinu logiky související se zálohováním souborů. V jednotlivých jeho uzlech pak bydlí instance třídy *FileInfo*, jež si drží informace vždy o jedné složce, normálním souboru nebo symlinku. V případě složky si pamatují pointery na v ní obsažené soubory, jinak si pamatují hash obsahu.

Hashe obsahu odkazují na instance třídy *FileChunk*, které již drží finální data. To je vždy VCDIFF oproti prázdnému souboru, nebo jinému souboru reprezentovanému instancí třídy *FileChunk*.

4.2.1 FileTree

Hlavní logika jádra zálohovacího systému je soustředěna do třídy *FileTree*, jež obsahuje tři metody pro registraci položek do souborového stromu:

- `AddDirectory(rodic, jmeno_souboru, parametry)`
- `AddFile(rodic, jmeno_souboru, parametry)`

[1] Návrhový idiom „private implementation“ pokoušející se skrýt co nejvíce implementace před vnějším světem

- `AddSymlink(rodic, jmeno_souboru, parametry)`

Každá z těchto metod vrací odkaz na nově přidanou položku (který se pak třeba v případě složky dá využít pro přidání položek do ní), nebo `nullptr`, pokud daná věc nemá být uložena ve stromě souborů (a tedy nemá například smysl procházet podstrom souborů pod touto složkou).

Dále obsahuje třída *FileTree* metodu pro vrácení seznamu souborů, které si zálohovací systém přeje dodat (setříděného podle důležitosti) a několik pomocných metod pro uložení stromu, vrácení kořenu celého stromu kvůli předání první vrstvě položek jako rodiče aj.):

- `FinishTree()` → vektor odkazů na položky stromu
- `GetRoot()` → odkaz na *FileInfo* reprezentující kořen

Nakonec třída obsahuje statické metody pro načtení stromu konkrétního jména z úložiště (ta vrací odkaz na strom nebo `nullptr`) a pro vrácení seznamu všech stromů v úložišti.

4.2.2 FileInfo

Hlavním úkolem této třídy je držet si informace o jednom souboru, složce či symlinku, o kterých si ve všech případech drží parametry jako jsou práva, čas poslední modifikace a podobně. V případě složky navíc obsahuje vektor obsažených souborů, v ostatních případech pak hash odkazující na příslušný datový blok.

Mezi základní metody patří metody pro zpracování obsahu souboru nebo jeho vrácení. Jako nepovinný parametr mohou dostat odkaz na aktuálně konstruovaný strom souborů a pomocí něj vyhledávat soubory se stejným hashem v minulé verzi stromu:

- `ProcessFileContent(inputstream, strom = nullptr)`
- `GetFileContent(outputstream)` → `outputstream` (vrácení umožňuje použití v syntaxi `<<` streamů)

Další důležité metody jsou následující tři sloužící pro správu obsahu složek. Zbytek veřejných metod jsou pak buď settery nebo gettery na vlastnosti souboru:

- `AddChild(jmeno, odkaz_na_instanci_FileInfo)`
- `GetChild(jmeno)` → odkaz na instanci *FileInfo*
- `GetChilds()` → odkaz na asociativní pole odkazů na *FileInfo*

4.2.3 BackupClean

Tato třída se stará o mazání starých verzí záloh. Obsahuje dvě metody, z nichž první si načte do paměti všechny souborové stromy, analyzuje je a spočítá badness (viz kapitola tomu věnovaná) a druhá metoda vždy obstará smazání jednoho datového bloku a vrátí, kolik dat bylo celkově uvolněno:

- `LoadData()`
- `Clean()` → uvolněné místo (v bytech)

4.2.4 Ostatní třídy

Třída *FileChunk* představuje jednotlivé datové bloky. Každá instance třídy si záznam, jestli je odvozená jako rozdíl od jiné instance, nebo jestli je takzvaně na „nulté úrovni“ a na ničme nezávisí.

Dále existuje třída představující konfiguraci *Config*, která vrací jednotlivé položky načtené konfigurace a pak řeší dotazy na pravidla pro jednotlivé zálohované soubory (opět podle načtené konfigurace).

Pro komunikaci s adaptéry je pak definována abstraktní třída *Adapter* obsahující metody pro proskenování a vytvoření souborového stromu, pro zpracování obsahu souboru a pro obnovení souboru na původní, nebo na zadané místo.

A konečně, jako uživatelské rozhraní existuje třída *CLI* (zkratka od *command-line-interface*).

4.3 Externí knihovny

Zálohovací systém využívá několik externích knihoven a to pro zpracování konfiguračních souborů, pro serializaci dat, pro počítání SHA256 hashů VCDIFF rozdílů souborů a pro práci se souborovým systémem.

Důvody pro použití externích knihoven jsou hlavně dva: Pokud existuje fungující a efektivní řešení daného problému, není potřeba „znovu vynalézat kolo“, a za druhé, na mnohá z těchto řešení mohou být uživatelé zvyklí (například konfigurace) a použití známých komponent usnadní uživatelům používání.

Konfigurace – libconfig

Byla použita mezi mnoha projekty rozšířená a přizpůsobivá knihovna `libconfig`[2]. Důvodem pro její volbu je velká rozšířenost mezi C a C++projekty a tedy její známost mezi uživateli. Program potřebuje být slinkován s knihovnou `libconfig++`.

Serializace dat – Cereal

Pro serializaci instancí tříd *FileTree* (s navázanými instancemi třídy *FileInfo*) a *FileChunk* se používá serializační knihovna *Cereal*[3], což je projekt vzniklý cíleně pro C++11 a novější využívající vlastností chytrých pointerů pro serializaci složitých datových struktur.

Sídlí čistě v hlavičkových souborech přiložených společně s projektem, takže není nutné cokoliv dynamicky linkovat.

Hashe a rozdíly – sha256.h a open-vcdiff

Pro počítání SHA256 je použita část hashovací knihovny, kterou napsal Stephan Brumme[4], sídlí jen v hlavičkových souborech, tedy se opět nic dynamicky nelinkuje.

[2] `http://www.hyperrealm.com/libconfig/`
[3] `http://uscilab.github.io/cereal/index.html`
[4] `http://create.stephan-brumme.com/hash-library/`

Pro vytváření a zpracování binárních rozdílů souborů je použitaVCDIFF implementace `open-vcdiff`[5], která vyžaduje slinkování s externími knihovnami `libvcdcom`, `libvcdenc` a `libvcddec`.

Souborový systém – boost::filesystem

Na místech interagujících nějakým složitějším způsobem se souborovým systémem byla použita implementace souborového systému z C++projektu `boost`[6]. Poskytuje na platformě nezávislé rozhraní a při případném použití zálohovacího systému na jiné platformě, než na které byl vyvinut, by tak neměla nastat žádné významné problémy. Pro běh musí být program slinkován s knihovnami `libboost_system` a `libboost_filesystem`.

4.4 Ukládání dat

Souborové stromy a datové bloky jsou ukládány na oddělená místa. Každý souborový strom, stejně jako popis datového bloku, sídlí v samostatném souboru na disku. Soubory jsou ukládány v binárním formátu. Ke každé serializované instanci třídy *FileChunk* je ještě připojen blok dat ve formátu VCDIFF popisující rozdíl oproti prázdnému souboru, nebo oproti nějakému jinému uloženému datovému bloku.

Formát uložení dat určuje serializační knihovna *Cereal* (která podporuje více druhů serializačních archivů, zálohovací systém využívá binárního formátu zápisu dat). Formát dat je explicitně verzovaný, což dovoluje do budoucna přidávat či ubírat položky se zachováním zpětné kompatibility.

Cereal dovoluje pojmenovávat ukládané položky, což se využívá při výstupu v lidsky čitelných formátech typu JSON nebo XML (toho bylo využíváno při vývoji), ale v binárním formátu jsou za sebe serializovaná jen samotná binární data. Pokud je serializován chytrý pointer na objektu, je nejdříve uveden čtyřbytový identifikátor typu objektu a pak buď serializovaná data objektu, nebo čtyřbytové pořadové číslo již použitého chytrého pointeru na tento typ objektu.

Každému archivu předchází hlavička a čtyřbytové číslo verze, níže následují seřazené tabulky položek pro obě serializované třídy. S jejich znalostí je možné snadno pomoci jakékoliv verze knihovny Cereal získat původní data i bez zálohovacího systému

4.4.1 Formát dat třídy FileTree

Položka	**datový typ**
Název stromu	`std::string`
Čas vytvoření	`time_t`
Název předchozího stromu	`std::string`
SHA256 hash předchozího stromu	`std::string`
Odkaz na kořen stromu	`std::shared_ptr<FileInfo>`

[5] `https://code.google.com/p/open-vcdiff/`

[6] `http://www.boost.org/doc/libs/1_46_0/libs/filesystem/v3/doc/index.htm`

4.4.2 Formát dat třídy FileInfo

Položka	datový typ
Jméno souboru	`std::string`
Typ souboru	`enum{DIR, FILE, SYMLINK}`
Stav verzování	`enum{UNKNOWN, NEW, UNCHANGED, UPDATED_PARAMS, UPDATED_FILE, NOT_UPDATED, DELETED}`
Parametry	`struct file_params`
Index souboru	`uint32_t`
Index minulé verze souboru	`uint32_t`
Odkaz na rodiče	`std::shared_ptr<FileInfo>`
Hash souboru	`std::string`
Asociativní pole synů	`std::unordered_map<std::string, std::shared_ptr<FileInfo>>`

4.4.3 Formát dat struktury file_params

Položka	datový typ
Číslo zařízení	`dev_t`
Číslo inode	`ino_t`
Souborová práva	`mode_t`
UID vlastníka	`uid_t`
GID skupiny	`gid_t`
Velikost souboru	`size_t`
Čas modifikace (sekundy)	`timespec.tv_sec`
Čas modifikace (nanosekundy)	`timespec.tv_nsec`

4.4.4 Formát dat třídy FileChunk

Položka	datový typ
Název (hash)	`std::string`
Název (hash) předchůdce	`std::string`
Hloubka zanoření	`int`
Velikost dat	`size_t`
Závislé instance	`std::vector<std::string>`

4.5 Postup uložení nové verze souboru

Prvním krokem je vždy obstarání si stromu souborů, neboli instance třídy *FileTree*. Většinou si vyrobíme aktuální ze současného stavu zálohovaných dat pomocí vhodného adaptéru, ale je možné vzít i starší instanci třídy *FileTree* a modifikovat tu (je pak ale nutné počítat s tím, že nebudou souhlasit kontrolní součty v navazujících instancích *FileTree*).

4.5.1 Záznam do stromu souborů a hledání minulé verze

Pokud si strom souborů vyrábíme, tak v nějakou chvíli dojde k přidání námi sledovaného souboru (voláním jedné z metod třídy *FileTree*). Přitom proběhne následující posloupnost operací:

1. Obstarají se pravidla pro tento soubor (od třídy *Config*) a případně se ukončí zpracování, pokud se soubor nemá zálohovat.
2. Vytvoří se nová instance *FileInfo* a přidá se odkaz do rodiče.
3. Uloží se známé parametry souboru (vlastník, velikost, čas modifikace)
4. Pokud je znám minulý strom: Systém se pokusí nalézt soubor se stejnou cestou (dotazem na svého rodiče, jestli zná svůj starší ekvivalent, a z něj jedním dotazem na jméno souboru). Pokud není nalezen, je nastaven stav `UNKNOWN` a zpracování se ukončí.
5. Pokud nebyla poslední verze zpracována, i když se změnila (status `UNKNOWN`, `NOT_UPDATED` nebo `DELETED`), nastav stejný stav (mimo `DELETED`, v tom případě nastav `NOT_UPDATED`) a skonči.
6. Pokud je nalezený soubor stejného typu, velikosti, času modifikace a parametrů → status `UNCHANGED` a přiřadí se stejný hash (odkazující na stejný datový blok).
7. Pokud je nalezený soubor stejného typu, velikosti, času modifikace, ale liší se v parametrech → status `UPDATED_STATUS` a přiřadí se stejný hash (odkazující na stejný datový blok).
8. Jinak přiřaď status `NOT_UPDATED`.

Pokud soubor skončí ve stavu `NOT_UPDATED` nebo `UNKNOWN`, přidá se do seznamu souborů k obstarání a spočítá se pro něj skóre důležitosti, jak již bylo ukázáno.

4.5.2 Zpracování obsahu souboru

Pokud se zálohovací systém rozhodne, že má zájem o obsah souboru, tak ho adaptér nějakým způsobem obstará a poté voláním metody na třídě *FileInfo* předá tomuto záznamu obsah souboru.

Nejdříve se spočítá SHA256 hash a pokud není známá minulá verze souboru, porovná se, jestli neexistuje v minulém stromě soubor se stejným hashem. Pokud ano, přiřadí se jako minulá verze, k souboru se uloží hash a skončí se, protože není nutné obsah dále zpracovávat (takto systém pozná přesuny souborů).

Pokud toto neuspěje, je potřeba obsah souboru uložit. Nejdříve systém prozkoumá, jestli již nemá uložen soubor se stejným hashem (tedy pokud zachováme důvěru v kryptografickou hashovací funkci, tak i se stejným obsahem) a pokud ano, není nutné nic nového ukládat a použijeme tento uložený datový blok.

Pokud stejný datový blok neexistuje, tak systém vytvoří novou instanci třídy *FileChunk* reprezentující soubor s tímto hashem a nechá ji zpracovat obsah. Pokud je znám předek, tak se zkusí vytvořit VCDIFF oproti poslední verzi (získají se

data této poslední verze a udělá se rozdíl mezi nimi), jinak se dělá rozdíl oproti prázdnému souboru.

Pokud by byl při vytváření rozdílu vůči starší verzi překročen limit na maximální hloubku na sebe navazujících rozdílů, tak se vytvoří rozdíl oproti první známé verzi souboru (tedy oproti kořeni, který sám vznikl rozdílem oproti prázdnému souboru), čímž se opět dosáhne nižší hloubky.

Alternativou by bylo vytvářet novou kompletní kopii takového souboru, ale rozdíl oproti nějaké základní verzi zabere nejvýše tolik místa, kolik by zabral rozdíl oproti prázdnému souboru. Takže toto řešení je nejméně stejně dobré, jako řešení s vyráběním nových záznamů s o jedna nižší maximální hloubkou.

4.6 Obnova dat

Při obnově dat se dá zvolit několik strategií. Vždy se zvolí konkrétní záloha (konkrétní strom), ze kterého nás zajímají daná data, a jádro zálohovacího systému pak poskytuje podporu pro:

- Obnovu jednoho konkrétního souboru nebo celého podstromu
- Obnovu do původního místa, do jiného místa na původním (vzdáleném) stroji a do jiného místa v lokálním souborovém systému
- Obnovu pouze dat souboru, pouze parametrů souboru, nebo všeho

Navíc se dá specifikovat taktika, co se má dělat, pokud v aktuálním stromu není soubor dostupný, určuje se podle dalšího parametru předávaného obnovovacím funkcím. Výchozí chování je to, že v takovém případě se pokusí systém nalézt nejnovější zálohovanou verzi souboru a obnoví tu (zde si můžeme všimnout, že starší verzi má smysl hledat jen u souborů označených jako `NOT_UPDATED`, u souborů označených `UNKNOWN` ne). Alternativním chováním je obnova pouze těch dat, která jsou odkazovaná aktuálním stromem (neuložené soubory jsou přeskakovány).

Při obnovování je navíc potřeba dát pozor na hardlinky vedoucí mimo obnovovanou oblast. V takovém případě by se mohlo stát, že se obnovením nějakého souboru přes hardlink projeví změna i na úplně jiném místě, což pravděpodobně není požadovaný cíl. Zálohovací systém tedy přijal taktiku nejdříve obnovovaný soubor celý smazat (tím se zruší případné provázání hardlinkem) a pak na stejném místě vytvořit znovu.

4.6.1 Postup získání obsahu souboru

Obnova jednoho souboru začíná dotazem na jeho typ – pokud to je složka, tak jsou všechny potřebné informace uložené již ve stromě souborů a není nutné získávat jakákoliv data z datových bloků. Pokud to není složka, tak se přes třídu *FileInfo* dostane žádost o obnovení dat až na datový blok reprezentující danou verzi souboru.

Pokud se datový blok (uložený ve formátu VCDIFF) neodkazuje na žádný jiný, je jeho obsah přímo obnoven jako diff od prázdného souboru a vrácen. Pokud se odkazuje na jiný datový blok, je nejdříve dotazem na tento blok (rekurzivně) získán jeho obsah, proti kterému je pak aplikováním rozdílu vyroben obsah tohoto datového bloku a ten je vrácen.

4.7 Mazání uložených dat – uvolňování místa

Zálohovací systém se může čas od času potkat s nutností uvolnit nějaké místo. K uvolnění místa je nutné zahodit některé informace, které si zálohovací systém pamatuje, a když už se to provádí, měl by systém nějakým vhodným způsobem zvolit podle něj nejméně důležitá data a ta zahodit.

Toliko abstraktní představa, teď konkrétněji. Uvolnit místo se dá smazáním některých datových bloků a úkolem mazání je vybrat ten blok, který bude „nejméně scházet“. Na co se nesmí zapomenout je to, že na jeden datový blok se může odkazovat více záznamů o souborech.

Vybrání bloku ke smazání probíhá tak, že se pro každý záznam o zálohovaném souboru spočítá *badness* („míra špatnosti“), která se odvíjí mimo jiné i od stáří zálohy a jejich nahuštěnosti – je větší snaha držet novější zálohy a více do historie může pokrytí řídnout (přesná funkce viz níže).

Z takto spočítaných badness pro každý záznam o souboru se pro každý datový blok vybere ta nejmenší badness ze souborů, které daný blok využívají, čímž se efektivně ohodnotí bloky podle toho, jak moc jsou významné pro historii. Pak se mohou bloky postupně od největší badness vybírat a mazat. Přitom je nutné dát pozor na to, že při smazání nějakého bloku (a tím několik verzí souborů) se může změnit badness okolním souborům v historii a ty se musí přepočítat. Dá se však vypozorovat, že se badness vždy jenom sníží, čehož se dá využít při praktickém nasazení ke zjednodušení implementace.

4.7.1 Postup počítání badness

Čistící třída *BackupClean* obsahuje jednu metodu, jejímž voláním se načte kompletní seznam všech záloh a obsažených souborů, a druhou metodu sloužící ke smazání vždy jednoho datového bloku s největší badness. Postup je tedy takový, že se jednou načte kompletní seznam a pak se opakovaně volá mazání bloků.

Při načítání se nejprve vytvoří pro každou známou navazující posloupnost souborů jeden jejich vektor (aby bylo možné snadno nalézt o jedna novější a o jedna starší verzi souboru). Pak se pro každý záznam o souboru spočítá badness podle vzorce:

$$B = \frac{[\text{stáří zálohy}]}{[\text{vzdálenost nejbližší starší zálohy}]} + \frac{[\text{stáří zálohy}]}{[\text{vzdálenost nejbližší mladší zálohy}]}$$

$$\text{badness} = \frac{B}{[\text{hodnota } \texttt{history} \text{ z konfigurace}]}$$

Tento vzorec vlastně znamená, že čím starší záloha, tím větší volné místo okolo sebe vyžaduje pro zachování stejně badness, což je přesně to, co jsme chtěli.

Ze spočítaných badness se vybere pro každý datový blok ta nejmenší a pak se tato hodnota ještě vydělí pro každý datový blok počtem na něj navazujících bloků (to je heuristika beroucí v úvahu to, že při vícero navazujících blocích by se namísto zmenšení mohl celkový datový objem zvětšit, viz dále).

Pak se datové bloky utřídí od největší hodnoty. Po smazání datového bloku se provede označení všech spojených záznamů o souborech jako `DELETED` a všem okolním záznamům se přepočítá badness. Pak je systém připraven pro mazání dalšího bloku.

4.7.2 Smazání datového bloku

Datový blok nelze odstranit jen tak jednoduše, protože pokud by od něj byly odvozeny jiné datové bloky, tak by se tímto znehodnotily. Proto je nutné nejdříve všechny navazující datové bloky nechat přepočítat, aby se odkazovaly na předka mazaného bloku, a teprve poté je možné blok odstranit.

Přepočítání proběhne tak, že se provede obnovení dat předka mazaného souboru a následníků mazaného souboru a spočítá se nový VCDIFF mezi nimi. Tím se změny pokrývané tímto blokem rozředí do změn v dalších blocích.

V některých případech se může stát, že se tímto krokem celkový objem dat dokonce zvětší, protože se velký rozdíl pokrývaný tímto souborem odsune do více navazujících souborů. Proto je součástí výpočtu badness datových bloků také dělení počtem navazujících bloků

Závěr

Zálohovací systém FenixBackup se pokusil nabídnout další řešení problému zálohování dat. V ideálním světe by taková věc vůbec potřeba nebyla, bohužel náš svět ideální není, ke ztrátám či poškození dat v něm dochází a systémy jako FenixBackup jsou potřeba.

Častým problémem zálohování je neochota uživatelů strávit čas s jeho nastavováním, nebo jeho složité použití ve chvíli ztráty dat. Proto jsem se pokusil při psaní FenixBackup vycházet z principů držet pro uživatele vše co možná nejpřímočařejší.

Současně jsem se pokusil vyřešit i problém omezeného místa pro zálohy, což vnímám jako další důvod, proč se někdy se zálohováním vyskytují problémy. Často uživatelé jednou nakonfigurují zálohování a pak věří tomu, že už bude fungovat napořád – bohužel se stává, že zálohovacímu systému dojde místo a uživatelé tomu nevěnují pozornost. FenixBackup se k tomu staví tak, že má nastavený limit pro velikost a když je překročen, spustí čištění záloha a maže zálohované verze souborů s největším spočteným záporným skóre, dokud nevymaže dostatek dat.

Vytvořený zálohovací systém je doufám na počátku dalšího slibného vývoje a zkusím zde nastínit, jakými dalšími směry by se jeho vývoj mohl ubírat.

Jedním směrem vývoje je určitě přidání většího množství adaptérů pro získávání dat. V prvotní fázi vývoje byl vytvořen jen adaptér pro lokální souborový systém, ale další adaptéry se nabízejí:

- Adaptér připojující vzdálený souborový systém pomocí SSHFS a dále fungující jako adaptér pro lokální souborový systém.
- Adaptér využívající jako protistranu na zálohovaném stroji a jako přenosový protokol `rsync`.
- Adaptér připojující se přes protokol FTP nebo SMB[7] na zálohovaný stroj.
- Adaptér využívající na zálohovaném stroji běžícího vlastního klienta.

Dalším možným směrem vývoje je modifikovat datový formát tak, aby mohl obsahovat nepovinné rozšiřující položky – inspirace například hlavičkami IPv6 paketů, které obsahují základní společnou hlavičku a pak podle potřeby rozšiřitelné hlavičky. V těchto nepovinných položkách by mohl sídlit například systém práv ACL[8], který se u nějakých souborů vyskytuje, ale je zbytečné mít pro něj vyhrazené pevné datové položky u každého souboru.

Nápad na to použít takový systém ukládání dat přišel bohužel až v pozdější fázi vývoje systému, kdy již na předělávání současné implementace nebyl dostatek času, ale je to jeden z cílů, kterým se chci dále věnovat.

Jinou oblastí, ve které se může udělat ještě velký pokrok, je přidat více inteligence do hledání minulých verzí souborů, nebo do hledání vhodných datových bloků, na které navázat pomocí rozdílové zálohy. Zde je podle mého ještě velký prostor, kterým se může systém posunout. Implementace jádra systému je na to

[7] Server Message Block, síťový protokol (nejen) pro přenos souborů, implementace například projektem Samba

[8] Access control list, rozšířený systém práv

připravená, protože už nyní umožňuje specifikovat, proti kterému datovému bloku má vznikat rozdílová záloha (i když zatím se volí buď předchozí známá verze souboru, nebo prázdný soubor).

Poslední oblast, nad kterou aktuálně přemýšlím a která by neměla být příliš složitá k doimplementaci, je nasazení v prostředí zálohování více podobných strojů (typicky nějaká firma nebo škola). Dá se vypozorovat, že při zálohování většího množství podobných strojů je mnoho souborů napříč stroji stejných (systémové soubory, společná konfigurace aj.). V takovém prostředí by bylo velmi efektivní umět mezi zálohami různých strojů sdílet společná data a zmenšit tak celkovou velikost všech záloh.

Pokud bychom deaktivovali systém mazání starých záloh, umí to FenixBackup již v současném stavu – jednotlivé zálohy budou sdílet společnou datovou složku, ale budou mít jiné složky, kam ukládají souborové stromy. Protože jsou všechny informace, jako jsou práva, uloženy v souborových stromech, a protože jsou datové bloky identifikovány hashem souboru, který zastupují, tak je možné bez jakýkoliv problémů tyto datové bloky sdílet. Problém nastane jen, kdyby jedna ze záloh chtěla datové bloky smazat jako nepotřebné – v tu chvíli by bylo potřeba sledovat využití datových bloků napříč všemi zálohami ukládajícími do stejného místa, pro což je nutné přidat podporu (aby zálohy o sobě navzájem věděly).

FenixBackup je připraven k použití již v současném stavu, ale pevně věřím, že jeho vývoj bude dále pokračovat a dočká se rozšíření ve výše zmíněných oblastech a také časem většího nasazení, než jen v řádu jednotek zálohovaných strojů.

Seznam použité literatury

[1] Chacon, Scott. *Pro Git.* Praha: CZ.NIC, 2009. ISBN 978-80-904248-1-4.

[2] Tridgel, Andrew et al. *Rsync* [online]. The Australian National University 1999. [Cit. 28.7.2015] Dostupné z `http://rsync.samba.org/`.

[3] Lindner, Mark. *libconfig – C/C++ Configuration File Library* [online]. [Cit. 28.7.2015] Dostupné z `http://www.hyperrealm.com/libconfig/`

[4] Grant, Shane a Voorhies, Randolph. *Cereal – A C++11 library for serialization* [online]. University of Southern California. [Cit. 28.7.2015] Dostupné z `http://uscilab.github.io/cereal/`

[5] Brumme, Stephan. *Portable C++ Hashing Library* [online]. [Cit. 28.7.2015] Dostupné z `http://create.stephan-brumme.com/hash-library/`

[6] Google. *open-vcdiff* [online]. [Cit. 28.7.2015] Dostupné z `https://code.google.com/p/open-vcdiff/`

[7] Boost community. *Boost Filesystem Library Version 3* [online]. [Cit. 28.7.2015] Dostupné z `http://www.boost.org/doc/libs/1_46_0/libs/filesystem/v3/doc/index.htm`

Seznam použitých zkratek

ACL Access control list, rozšířený systém práv. 24

FTP File Transfer Protocol – jednoduchý síťový protokol pro přenos souborů. 3, 24

MD5 Kryptografická hashovací funkce vytvářející z libovolného vstupu výstup o velikosti 128 bitů. 4

PIMPL Návrhový idiom „private implementation“ pokoušející se skrýt co nejvíce implementace před vnějším světem. 15

RAID Redundant Array of Inexpensive/Independent Disks – systém ukládání dat na více nezávislých disků. 1

SCP Secure copy – zabezpečený protokol pro přenos souborů na bázi SSH. 3

SHA1 Secure Hash Algorithm – pokročilejší kryptografická hashovací funkce, vytváří otisk o velikosti 160 bitů. 4, 9

SHA256 Kryptografická hashovací funkce, vytváří otisk o velikosti 256 bitů. 9, 17, 20

SMB Server Message Block, síťový protokol (nejen) pro přenos souborů, implementace například projektem Samba. 24

SSH Secure shell – zabezpečený komunikační síťový protokol. 3, 27

SSHFS SSH File System – připojení vzdáleného souborového systému přes protokol `SSH`. 7, 24

UNIX Operační systém vzniklý v roce 1969, který dal vzniknout celé rodině odvozených systémů – Linux, BSD, Mac OS X, aj.. 4, 6

VCDIFF Formát a algoritmus definovaný v RFC 3284 založený na článku *Data Compression Using Long Common String*. 5, 7, 9, 10, 15, 17, 18, 20, 21, 23

Přílohy

Příloha 1: Zdrojové kódy zálohovacího systému FenixBackup (včetně ukázkové konfigurace a několika pomocných souborů).