Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Vícebodová synchronizace dat protokolem WebDAV

Plzeň 2012 Jiří Praus

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 13. května 2012

Jiří Praus

Multipoint synchronization using WebDAV protocol

Abstract

The goal of this thesis is to verify whether the WebDAV communication protocol and its implementation by Kerio Connect email server is a suitable tool for data synchronization between more Kerio Connect servers in a distributed environment and to design fault-tolerant synchronization algorithm, which would use the WebDAV communication interface to transfer data between Kerio Connect servers. The work describes in detail the communication protocol WebDAV and various algorithms and technologies for maintaining data consistency in a distributed environment. Based on learned facts the algorithm is designed to efficiently synchronize data between servers in a distributed environment, deal with possible conflict situations and resolve possible error conditions. Behavior of the algorithm is extensively tested both in terms of functionality and reliability as well as performance and efficiency.

Obsah

1 Úvod

V moderní době jsou pro nás data uložená v počítači to nejcennější, čím náš počítač disponuje, bez nich je počítač bezcenný. Svá data chceme často sdílet s ostatními uživateli a publikovat na veřejných prostorech. Emailový server Kerio Connect od společnosti Kerio Technologies s.r.o. tuto možnost nabízí a dovoluje uživatelům sdílet důležitá data nebo události mezi sebou. Uživatelé mohou takto sdílená data libovolně upravovat a měnit.

Problém ovšem nastává v distribuované doméně, kde jsou uživatelé stejné domény distribuováni mezi více Kerio Connect serverů. Uživatelé mohou veřejná data nadále sdílet mezi sebou, ale pouze v rámci jednoho serveru. Motivací pro tuto práci je právě potřeba sdílená data distribuovat mezi všechny servery domény tak, aby každý z uživatelů distribuované domény mohl přistupovat ke stejným veřejným datům na libovolném serveru. Ideálním rozhraním pro takovou výměnu dat mezi servery je přitom otestované a funkční komunikační rozhraní WebDAV používané pro připojení Entourage emailového klienta ke Kerio Connect serveru.

Distribuce dat s sebou ovšem přináší nemalé problémy. Data jsou replikována mezi servery a jsou uložena ve vícenásobných kopiích. Uživatelé mohou stejné soubory měnit na libovolném serveru a vyvolat tím nutnost přenášet změny do ostatních kopií tak, aby všichni uživatelé viděli stejná data. Situace se dále komplikuje, pokud je změna stejného souboru provedena na více serverech současně. Vzniká tak konflikt, který je potřeba uspokojivě vyřešit. Distribuované systémy dále trpí častými výpadky a nestálostí sítě, čímž dochází k dočasné nedostupnosti jednotlivých uzlů systému.

Cílem práce je výše zmíněné problémy vyřešit v prostředí distribuované domény Kerio Connect serverů s použitím komunikačního protokolu WebDAV. Druhá kapitola práce se podrobně zabývá ověřením možností WebDAV komunikačního rozhraní implementovaného v Kerio Connect serveru a jeho vhodností pro replikaci dat mezi servery. Ve třetí kapitole je představen pojem replikace a detailně popsány různé způsoby udržení dat na více replikách v konzistentním stavu. Získané poznatky jsou ve čtvrté kapitole využity pro návrh vhodného synchronizačního algoritmu, který bude spolehlivě udržovat repliky v konzistentním stavu a vhodně řešit vzniklé konflikty. Navržený algoritmus je v závěrečné páté kapitole prověřen jak z hlediska funkčnosti a spolehlivosti, tak i výkonnosti a efektivity.

2 Protokol HTTP a WebDav

HTTP (HyperText Transfer Protocol) je jednoduchý aplikační protokol, který se dá využít nejen k přenosu hypertextových dokumentů a obrázků, ale má i mnoho zajímavých rozšíření a vlastností. A právě jeho významné rozšíření WebDAV, které umožňuje mimo klasického stahování obsahu i jeho správu, je použito v této práci jako komunikační prostředek pro přenos dokumentů. V této kapitole budou uvedeny základy protokolu HTTP a důležité vlastnosti WebDAV, které byly v dalším textu práce použity.

2.1 HTTP

Hypertext Transfer Protokol pochází z roku 1996 (respektive HTTP 1.1 z roku 1999) a byl používán pro přenos dat přes internet. V dnešní době je používán zejména pro WWW. HTTP je aplikační protokol nad transportním protokolem TCP a je založen na principu požadavek/odpověď. Klient vždy inicializuje spojení odesláním požadavku a server mu patřičně odpoví. Dotaz se vždy skládá z požadované metody, cesty k souboru (URL), verze protokolu a hlaviček. V některých případech následuje po hlavičkách i tělo dotazu oddělené jedním prázdným řádkem. Vše je přenášeno v textové podobě, je tedy možné požadavky jednoduše vytvářet nebo číst:

```
1 GET / HTTP/1.1
2 Host: www.example.com
```

Server požadavek vyřídí a vrátí odpověď, která může vypadat například takto:

```
1 HTTP/1.1 200 OK
2 Date: Sat, 10 Dec 2011 20:22:25 GMT
3 Server: Apache
4 Content-Length: 27072
5 Connection: Keep-Alive
6 Content-Type: text/html;charset=UTF-8
7
8 ...
```

Odpověď obsahuje verzi protokolu, návratový kód, hlavičky a tělo odpovědi. Opět je veškerý přenos prováděn pouze v textové podobě. Jak je možné z předchozího pozorovat, protokol HTTP je velice jednoduchý a proto také velmi oblíbený a používaný. Dále se budeme zabývat pouze nejnovější verzí protokolu HTTP 1.1.

2.1.1 Vybrané vlastnosti

Perzistentní spojení Server podporující protokol 1.0 po vrácení odpovědi ihned navázané spojení uzavře. Často je toto protokolu vyčítáno, protože otevření TCP spojení pro každý pár požadavek/odpověď je poměrně drahé. Protokol 1.1 přichází s perzistentním spojením, kdy server spojení neuzavře ihned, ale čeká na další požadavky. Klient může pokračovat v dotazování a spojení ukončit sám.

Bezstavovost Požaduje-li opakovaně klient stejná data, zašle nový požadavek a server vygeneruje novou odpověď. Případné uchování některých informací o komunikaci mezi klientem a serverem je zcela v režii klienta. Protokol si informace o již provedených spojeních neudržuje, je to protokol bezstavový.

MIME typ (formát) Pomocí hlavičky *Accept* může klient požadovat určitý MIME typ. Takže při požadavku na stejné URL, ale s jinou hlavičkou může dostat v případě *Accept: text/plain* stránku převedenou na prostý text, zatímco v případě *Accept: application/xhtml+xml* dostane hypertext. Stejně tak pomocí této hlavičky můžeme vyjednat se serverem vhodný formát například obrázků [RFC2616].

Bezpečnost Požadavky i odpovědi jsou přenášeny čistě v textové podobě, což může znamenat značná bezpečnostní rizika, a proto je možné přidat TLS vrstvu a vytvořit tak zabezpečený protokol nazývaný HTTPS [RFC2818].

2.1.2 Důležité metody

Použitou metodu HTTP protokolu specifikuje klient v požadavku a oznamuje tak serveru, jakou operaci chce provést. S metodami jsou svázány hlavičky a každá metoda se může chovat trochu jinak při použití rozdílných hlaviček, podrobnosti je možné nalézt v oficiální specifikaci [RFC1945].

Metoda GET představuje požadavek na zaslání dokumentu určeného pomocí URL, čili například stažení HTML stránky nebo obrázků z webového serveru.

Metoda HEAD je shodná s metodou GET, ale v odpovědi serveru jsou uvedeny pouze hlavičky vyžádaného dokumentu bez těla obsahující samotný dokument. Tato metoda se používá pro zjištění existence dokumentu nebo jeho vlastností.

Metoda POST se používá v případě, kdy má cílový server přijmout data z požadavku. Skutečná funkce metody závisí na implementaci a určené URL. Může se jednat o zaslání emailu, předání dat do procesu a podobně. U klasických webových serverů se jedná především o odeslání formuláře.

Metoda PUT představuje požadavek na uložení posílaných dat pod specifikované URL na server. Takto uložená data budou dostupná např. následnými dotazy GET. Webové servery toto samozřejmě nepodporují, klient nemá možnost měnit obsah na vzdáleném serveru.

Metoda DELETE je požadavkem na zrušení dokumentu na serveru. Rušený dokument je specifikován v URL. Stejně jako u metody PUT ji klasické webové servery nepodporují.

2.1.3 Návratové kódy

Na každý požadavek server odpoví jednou odpovědí, tato odpověď vždy obsahuje tříciferný návratový kód včetně popisujícího textu, který oznamuje stav požadavku. Návratový kód se dělí podle první cifry na pět tříd [RFC1945]:

- **1 - informativní** Požadavek byl v pořádku obdržen, ale není doposud kompletně zpracován.
- **2 - úspěch** Dotaz byl serverem pochopen, akceptován a případná data jsou v odpovědi. Nejčastější odpovědí je známá 200 OK.
- **3 - přesměrování** Požadovaný cíl existuje, ale byl přesunut, klient musí provést další akce pro jeho získání (zaslat nový požadavek). Například známé přesměrování 301 Moved Permanently.
- **4 - chybný požadavek** Klient položil chybný dotaz nebo nemá dostatečná oprávnění získat požadovaný cíl. Opět můžeme uvést příklad typické návratové hodnoty 404 Not Found.
- **5 - chyba serveru** Server není z nějakého důvodu schopen obsloužit požadavek. Často nepříjemná odpověď 500 Internal Server Error.

2.2 Protokol WebDAV

Web Distributed Authoring and Versioning - WebDAV - je množina rozšíření HTTP protokolu, která poskytuje možnost vzdálené správy souborů, adresářů a vlastností na datovém serveru. Protokol tak činí ze vzdáleného serveru zapisovatelné médium. Vytváří prostředí, ve kterém mohou klienti vytvářet, měnit, přesouvat a mazat soubory a adresáře. Prostředí organizuje soubory do standardní stromové adresářové struktury [RFC2818]

WebDAV spojuje dvě již propracované technologie HTTP a XML (Extensible Markup Language) [W3C]. Protokol HTTP je používán pro zajištění způsobu komunikace. Kromě standardních metod využívaných v rámci protokolu HTTP (GET, POST, PUT ...) zavádí WebDAV pro své potřeby nové metody a hlavičky. XML naopak WebDAV využívá jako nositele strukturovaných informací, které upřesňují jak dotaz klienta, tak i výsledek zasílaný serverem. V XML části komunikace jsou odesílány výsledky zpracování příkazů pro jednotlivé soubory, informace o souborech, adresářích a jiné. Nejlepší bude ukázat WebDAV požadavek a odpověď na jednoduché ukázce vytvoření nové kolekce.

Požadavek

```
1  MKCOL /folder/ HTTP/1.1
2  Host: www.example.com
3  Content−Type: text/xml
4
5  <?xml version=”1.0”?>
6  <d:propertyupdate xmlns:d=”DAV:”>
7    <d:set>
8      <d:prop>
9        <d:displayname>Folder</d:displayname>
10     </d:prop>
11   </d:set>
12 </d:propertyupdate>
```

Odpověď

```
1  HTTP/1.1 201 Created
```

V podstatě se nejedná o nic nového, požadavek je shodný s požadavkem v HTTP protokolu, s tím rozdílem, že tělo obsahuje XML dokument s podrobnostmi o požadavku, kterým se v tomto případě nastaví vlastnost kolekce *displayname* na hodnotu „Folder“.

2.2.1 Slovník pojmů

WebDAV protokol definuje několik nových pojmů, které jsou v následujícím textu objasněny, a tyto pojmy budou i v dalším textu této práce použity.

Zdroj - libovolná entita, se kterou je možné pracovat. Je identifikován jedinečnou URL nebo URI adresou. Neformálně řečeno se jedná o soubory, kolekce a vlastnosti (z angl. Resource [RFC4918]).

Vlastnost - pár jméno/hodnota obsahující popisné informace o zdroji - typ zdroje, velikost, autor ... (z angl. Property [RFC4918]).

Kolekce - zdroj, který se současně chová jako kontejner referencí na další zdroje. V podstatě se jedná o analogii adresáře v klasickém souborovém systému. Kolekce smí obsahovat vlastnosti a další zdroje, at' už kolekce nebo dokumenty (z angl. Collection [RFC4918]).

Dokument - zdroj, který není kolekcí. Dokumenty jsou analogií souborů v klasickém souborovém systému (z angl. Document).

Vnitřní člen kolekce - zdroj, který je přímým potomkem dané kolekce, čili kolekce obsahuje referenci na tento zdroj (z angl. Internal Member of a Collection [RFC4918]).

Člen kolekce - zdroj, který je potomek dané kolekce neboli je vnitřní člen dané kolekce nebo vnitřní člen libovolné vnořené kolekce (z angl. Member of a Collection [RFC4918]).

Aktivní vlastnost - vlastnost, jejíž sémantika a syntaxe je vynucená serverem. Příkladem takové aktivní vlastnosti může být *DAV:getcontentlength*, jejíž hodnota, vrácená v požadavku *GET*, je automaticky spočítána na straně serveru z aktuálního stavu zdroje (z angl. Live Property [RFC4918]).

Pasivní vlastnost - vlastnost jejíž sémantika a syntaxe není vynucená serverem. Server pouze hodnotu ukládá a za údržbu konzistence a správnosti sémantiky a syntaxe hodnoty je odpovědný klient. Příkladem této vlastnosti může být jméno autora zdroje (z angl. Dead Property [RFC4918]).

2.2.2 Odpověď 207 Multi-Status

WebDAV mimo klasických dotazů a odpovědí nad jedním konkrétním zdrojem, zavádí i hromadné (z angl. batch) operace, například pro smazání nebo nastavení vlastností zdroje. V tomto případě je ovšem nutné obdržet od serveru odpověď se stavem pro každý dílčí cíl hromadné akce. WebDAV proto zavádí odpověď 207 Multi-Status, čímž oznamuje, že stav provedených operací je k nalezení v XML těle odpovědi pro každý jednotlivý zdroj samostatně.

2.3 Exchange Store WebDAV

Celá specifikace protokolu WebDAV je k nalezení v oficiálním [RFC2818]. V této práci je ovšem nutné se zaměřit na specifickou část celého dokumentu, kterou podporuje server Kerio Connect. Kerio Connect je emailový server vyvíjený společností Kerio, který mimo jiné poskytuje i komunikační rozhraní se službami WebDAV [Connect].

Motivací pro implementaci WebDAV komunikačního rozhraní byla nutnost podporovat emailového klienta Entourage vyvíjeného společností Microsoft pro operační systém OS X. Microsoft Entourage byl součástí balíku Microsoft Office pro MAC a jednalo se o sourozence známého emailového klienta Microsoft Outlook. Tento klient uměl komunikovat se serverem Microsoft Exchange a to právě pomocí protokolu WebDAV.

Jak již to často bývá a zvláště u společnosti Microsoft, komerční společnost převezme oficiální specifikaci a upraví ji podle svých konkrétních potřeb a požadavků. Tak tomu je i v případě implementace WebDAV produktem Microsoft Exchange. Společnost Microsoft má svoji vlastní oficiální specifikaci protokolu WebDAV, která původní [RFC2818] rozšiřuje o nové vlastnosti. Podstata protokolu zůstává stále stejná, ale díky drobným odlišnostem musíme vycházet právě z této Microsoft Exchange specifikace [MSDN].

Do jisté míry limitujícím faktorem je to, že celá implementace komunikačního rozhraní WebDAV produktem Kerio Connect se omezuje pouze na potřeby emailového klienta Microsoft Entourage, který využívá jen některé z možností WebDAV protokolu. Nebylo tudíž nutné implementovat celou specifikaci protokolu, ale pouze ty metody a vlastnosti, které emailový klient opravdu používá. Komunikační rozhraní se tedy omezuje jen na určitou podmnožinu a tato podmnožina je v následujících řádcích uvedena. Všechny tyto vlastnosti byly zjištěny přímým testováním komunikačního rozhraní produktu Kerio Connect.

2.3.1 Metoda PROPFIND

Používá se pro získání vlastností zdroje specifikovaného podle URI. Vlastnosti, které chceme získat, můžeme uvést v XML těle požadavku nebo uvést zástupnou vlastnost *DAV:allprop*, která donutí server v odpovědi vrátit veškeré dostupné vlastnosti. Server vrací odpověď 207, která obsahuje částečné odpovědi 200 pro vlastnosti existující a dostupné a 403 pro vlastnosti neexistující. Použitím hlavičky *Brief* je možné serveru říci, ať neexistující vlastnosti vynechá a vrací pouze existující.

Navíc existuje speciální vlastnost *descriptor*, která vrací tzv. *security descriptor* neboli bezpečnostní popis zdroje. Tato vlastnost je specifická pro kolekce a obsahuje definici práv, kterými skupina uživatelů nebo uživatel disponují vzhledem k cílové kolekci [MSDN].

Hlavička Depth

Uvedením hlavičky *Depth* můžeme vyžádat kromě vlastností zdroje uvedeného v URL i vlastnosti všech jeho členů (samozřejmě relevantní pouze pro kolekce). Implicitní hodnota hlavičky Depth je „0“, což je pouze dotazovaný zdroj. Explicitním uvedením hodnoty „1“ získáme vlastnosti veškerých vnitřních členů kolekce a hodnotou „infinity“ vlastnosti všech členů kolekce [MSDN].

2.3.2 Metoda PROPPATCH

Metoda PROPPATCH umožňuje nastavit nebo odstranit vlastnosti cílového zdroje specifikovaného podle URI. Všechny vlastnosti jsou uvedeny v XML těle požadavku a mají příznak *DAV:set* pro nastavení nebo *DAV:remove* pro odebrání vlastnosti z množiny vlastností. Server stejně jako v případě PROPFIND vrací odpověď 207 s částečnými odpovědi 200 pro vlastnosti úspěšně nastavené a 403 pro vlastnosti nenastavené (vlastnost neexistuje nebo se jedná o aktivní vlastnost).

2.3.3 Metoda MKCOL

Metoda MKCOL vytváří novou kolekci umístěnou do stromové struktury podle URI požadavku. Server vrací odpověď 207 s částečnou odpovědí 200 pro úspěšně vytvořenou kolekci a 403 pro kolekci, kterou se nepodařilo vytvořit. Cesta ke každé kolekci, kterou chceme vytvořit, musí již existovat v době vytváření. WebDAV cestu rekurzivně nevytváří.

Při vytváření kolekce je vždy nutné uvést třídu kolekce, která se má vytvořit. Třída kolekce určuje sémantiku, s jakou se má s kolekcí v prostředí e-mailového serveru a klienta pracovat. Určuje například její vzhled a možný obsah. Kolekce se dělí celkem do pěti tříd:

- **IPF.Note** složka e-mailových zpráv,
- **IPF.Contact** složka obsahující kontakty,
- **IPF.Appointment** kalendář s událostmi,
- **IPF.Task** složka úkolů a
- **IPF.StickyNote** složka poznámek.

2.3.4 Metoda SEARCH

SEARCH je významnou metodou zavádějící možnost vyhledávat ve zdrojích dotazovacím jazykem SQL. Je možné tedy například vyhledat všechny kolekce nebo všechny zdroje určitého typu nebo velikosti. Metoda umožňuje použít 2 typy vyhledávání:

- **Hierarchical** stromové vyhledávání hledá ve členských kolekcích a
- **Shallow** mělké vyhledávání hledá ve všech vnitřních členech kolekce.

Kromě velice efektivního způsobu vyhledávání, je možné získat v odpovědi *Manifest of a Collection*, což je seznam změn v hledané kolekci [MSDN]. Každý nalezený zdroj je označen jedním ze tří možných stavů:

- **new** zdroj byl nově přidán,
- **change** vlastnost nebo obsah zdroje byly změněny nebo
- **delete** zdroj byl smazán.

K označení verze kolekce je použita neprůhledná textová známka *collblob*. Tato známka je vždy obsažena v odpovědi na metodu SEARCH a označuje novou verzi kolekce, kterou jsme právě odpovědí obdrželi. Tuto známku můžeme použít při dalším hledání metodou SEARCH a dostaneme seznam veškerých změn, které nastaly od verze označené známkou. Pokud použijeme známku prázdnou, vrací server všechny zdroje se stavem *new*.

2.3.5 Metoda LOCK a UNLOCK

K e-mailovému serveru může najednou přistupovat konkurenčně více klientů a proto je nutné při vytváření nebo úpravě zdroje tento zdroj explicitně zamknout pomocí metody LOCK a vytvořit tak novou transakci. Metoda LOCK umožňuje zamknout i neexistující zdroj a to pro případ, kdy potřebujeme nový zdroj vytvořit. Transakce se vytváří pouze pro nahrávání dokumentů metodou PUT a PROPPATCH, kdy přenos trvá řádově déle než u jiných operací jako DELETE nebo MKCOL. Transakce je vždy vytvořena na dobu jedné hodiny a po této době se automaticky zruší. Server vrací odpověď 200, pokud je transakce úspěšně vytvořena, nebo 403, pokud transakci nelze vytvořit nebo jiný zámek je již aktivní nad stejným zdrojem. V odpovědi je také uvedena neprůhledná známka *Lock-Token* identifikující transakci, která se použije v hlavičce *If* pro všechny metody transakce.

Transakci je možné ukončit explicitně metodou UNLOCK. V XML těle požadavku se uvádí, zda byla transakce úspěšně provedena a změny se mají zapsat (element *DAV:commit*) nebo transakce nebyla úspěšná a změny se mají zrušit (element *DAV:abort*). Po ukončení transakce je zámek uvolněn a jiný klient může přistupovat ke stejnému zdroji.

2.3.6 Metoda SUBSCRIBE a POLL

Metodou SUBSCRIBE je možné přihlásit se ke sledování změn nad kolekcí specifikovanou v URI požadavku. Tímto způsobem je možné získat informace o změně v reálném čase hned, jakmile nastane. Přihlášení je identifikováno celočíselným *Subscription-ID*, které server vrací při úspěšné odpovědi 200. Přihlášení je možné provést pouze na kolekce, pokud se pokusíme vytvořit přihlášení na jiný zdroj, server odpoví 501. Přihlášení trvá implicitně jednu hodinu a před uplynutím této doby je možné jej obnovit opětovným voláním metody SUBSCRIBE, ale tentokrát s hlavičkou *Subscription-ID* obsahující identifikátor přihlášení. Klient může být o změnách informován dvěma způsoby:

- **Metodou NOTIFY** Klient oznámí serveru, na které URL bude čekat na oznámení změny, hlavičkou *Call-Back*, používající protokol HTTPU. Na tuto URL následně server při každé změně zavolá metodu NOTIFY, takže klient je informován o změně ihned, jakmile na straně serveru nastane. Nevýhodou je nutnost implementace HTTP serveru na straně klienta pro podporu naslouchání HTTP volání [MSDN].

- **Metodou POLL** Klient se periodicky ptá serveru metodou POLL s hlavičkou *Subscription-ID*, zda nastala změna nad daným přihlášením. Při volání metody POLL je možné uvést více identifikátorů přihlášení. Server v úspěšné odpovědi 207 rozdělí identifikátory do tří skupin: 200 pro existující přihlášení se změnou, 204 pro existující přihlášení beze změny a 403 pro neexistující přihlášení. Nevýhodou je, že oznámení o změně klient dostane, až ve chvíli kdy se zeptá. To samozřejme může být i výhodou, protože právě má čas změny obsloužit. Značnou výhodou je jednoduchost klienta, není nutné implementovat HTTP server [MSDN].

Přihlášení je zrušeno automaticky po uplynutí časového limitu, při zrušení kolekce nebo je možné jej zrušit explicitně voláním metody UNSUBSCRIBE s hlavičkou *Subscription-ID* obsahující identifikátor přihlášení.

Typy přihlášení

Metodou SUBSCRIBE je možné přihlásit se k odběru různých typů událostí do různé hloubky pomocí hlaviček *Depth* a *Notification-Type*:

- **Hlavička Depth** stejně jako v případě metody PROPFIND (2.3.1) umožňuje odebírat změny pouze pro vnitřní členy kolekce nebo pro všechny členy kolekce.
- **Hlavička Notification-Type** specifikuje sledovanou událost. E-mailový server Kerio Connect podporuje celkem dva typy událostí. Hodnotou *„update“* sledujeme změny obsahu nebo vlastností zdrojů. Hodnotou *„delete“* sledujeme událost smazání zdroje. Je nutné poznamenat, že událost *„update“* sleduje v implementaci i událost smazání, takže druhá hodnota není potřeba, pokud nechceme sledovat pouze odstranění zdroje.

2.4 Důvody pro použití WebDAV

- **Standard** WebDAV je velice dobře popsaný a jednoznačný standard prověřený mnoha použitími. Je to jedna z výhod pro použití protokolu WebDAV. Jakýkoliv jiný emailový server bude možné synchronizovat pomocí protokolu WebDAV, pokud bude server toto komunikační rozhraní implementovat. A i když se jedná o použití standardu od společnosti Microsoft, stále se jedná pouze o nadmnožinu oficiálního standardu RFC 4918.
- **Použití HTTP** Bez protokolu HTTP se žádný dnešní standardní počítač nemůže obejít, jelikož by žádný z uživatelů nemohl prohlížet webové stránky. Proto je tento protokol zaveden jako standardní ve veškerých firewallech, port 80 není blokován a není nutné pro něj nastavovat specifická pravidla.
- **Bezpečnost** Protokol HTTP umožňuje šifrovanou komunikaci přidáním SSL vrstvy s výměnou certifikátů, čímž vznikne jeho zabezpečená varianta HTTPS. Je proto možné místo standardního textového formátu posílat data zašifrovaně bez jakékoliv nutnosti změnit komunikační protokol. Navíc se opět jedná o standardní protokol na portu 443, který není blokován.
- **Komprimace** Protokol HTTP umožňuje přenášen komprimovaná data, čímž je možné značně ulehčit komunikačnímu médiu při přenosu dat.
- **Podpora replikace** Jak bude uvedeno dále, rozšíření WebDAV od společnosti Microsoft v sobě přímo zahrnuje podporu replikace dat mezi servery. Takže kromě *Manifest of a Collection* jsou k dispozici další data pomáhající při replikaci.
- **Kerio Connect** Největším důvodem pro použití WebDAV je ovšem existence otestovaného a funkčního komunikačního rozhraní v produktu Kerio Connect. Není tedy nutné pro synchronizaci emailových serverů měnit produkt Kerio Connect, ale pouze použít toto rozhraní.

3 Replikace

Replikace dat udržuje vícenásobné kopie dat, nazývané *repliky*, na oddělených počítačích. Tato velice důležitá technologie umožňuje existenci distribuovaných služeb. Replikace zvyšuje dostupnost a propustnost, jelikož data jsou uložena ve vícenásobných kopiích, ke kterým lze přistupovat najednou a odděleně [Saito].

3.1 Konzistence

Hlavním cílem všech replikačních technik a algoritmů je udržovat konzistenci dat [Saito]. Konzistence jsou pravidla, která musí replikační algoritmus dodržovat pro udržení všech replik v konzistentním stavu. Model konzistence definuje, za jak dlouho budou změny dostupné na ostatních replikách nebo v jakém pořadí se provádí operace. Modelů konzistence existuje celá řada, liší se především ve striktnosti pravidel. Následující text uvádí pro představu dva výrazně odlišné modely:

- **Striktní konzistence** znamená, že aby mohla být libovolná změna na jedné z replik úspěšná, je nutné, aby tato změna byla okamžitě zreplikována na všech ostatních replikách. Jakmile je obdrženo potvrzení o úspěšném provedení na všech replikách, je i volající operace úspěšná. Jediným možným způsobem jak této konzistence dosáhnout je použití zámků a všechny repliky v době aktualizace uzamknout, což má samozřejmě fatální dopad na propustnost systému.

- **Eventual consistency** neformálně znamená, že všechny repliky nakonec (z angl. eventually) dosáhnou konečného stejného stavu, pokud uživatel přestane vytvářet nové operace [Saito]. Tento model striktně nepředepisuje, kdy mají repliky dosáhnout stejného stavu, jen říká, že ho nakonec musí dosáhnout. V dnešní době se jedná o velmi populární koncept u systémů s optimistickou replikací (viz sekce 3.3), systémy jsou jednodušší a lépe škálovatelné. Tento koncept používá například Google File System [Ghemawat].

3.2 Pesimistická replikace

Tradiční technikou replikace je udržování jedné primární kopie na primárním serveru v distribuovaném systému. Primární replika zpracovává veškeré požadavky na změnu dat. Jakmile je změna provedena, je asynchronně zreplikována na ostatní, sekundární repliky. V době, kdy jsou data na sekundárních replikách neaktuální, je přístup k nim blokován. Pokud primární kopie vypadne, ostatní repliky si zvolí novou primární kopii a systém funguje dále. Tato technika se nazývá *„pesimistická replikace“* [Saito].

Pesimistická technika je jednoduchá a nemusí řešit žádné konflikty, jelikož veškeré aktualizace se provádí vždy nad aktuálními daty. Tato technika funguje dobře na lokálních sítích, kde latence mezi replikami je malá a výpadky nepravděpodobné. Nemůžeme od ní ovšem očekávat dobré výkony v prostředí internetu, s vysokou latencí, malou propustností a častými výpadky. Propagace změn na sekundární repliky je pomalá a propustnost významně klesá, díky blokování neaktuálních dat. Navíc je zde velká pravděpodobnost ztráty

dat při výpadku primární kopie, pokud se změny nestačí zreplikovat na sekundární repliky. Je velice těžké vytvořit systém s pesimistickou replikací v prostředí internetu s častou potřebou zápisů, jelikož jeho propustnost dramaticky klesá s narůstajícím počtem uživatelů. V tomto prostředí nalezne využití replikace optimistická [Saito].

3.3 Optimistická replikace

Klíčovou myšlenkou pro optimistickou replikaci je povolit přístup pro zápis na libovolné replice bez dřívější synchronizace a optimisticky předpokládat, že konflikt více zápisů do jednoho objektu nastane velice vzácně, pokud vůbec kdy nastane. Všechny provedené změny jsou propagovány na pozadí, a pokud vícenásobný zápis do jednoho objektu nastane, je konflikt vyřešen během synchronizace [Saito].

Optimistická replikace nabízí mnoho výhod. Dostupnost dat se zvýší a data jsou zpracovávána i při výpadku libovolné repliky. Umožňuje asynchronní spolupráci mezi více uživateli. Repliky zůstávají stále autonomní a nezávislé na primární replice [Saito].

Oproti pesimistické replikaci je však nutné řešit konflikty. Tam kde pesimistická replikace čeká na aktuální data, optimistická replikace musí hádat jak vyřešit konflikt mezi konkurentními operacemi. Takže je použitelná pouze pro systémy, které tolerují občasné konflikty a nekonzistentní data. Optimistickou replikaci používají například systémy DNS, CVS a SVN.

3.3.1 Single-master

V distribuovaném prostředí je často mnohem složitější problém distribuovaného algoritmu převeden na jednodušší algoritmus centralizovaný například provedením distribuovaného výběru 1 z N pro zvolení centrálního prvku. Stejně je tomu i v případě architektury **single-master**. Systém stále dovoluje zápis do všech replik, ale za plánování pořadí operací a řešení konfliktů je zodpovědný pouze jediný uzel, tzv. „master“. Master přijímá operace od ostatních replik a po vhodném naplánování odešle operace ostatním replikám. Systém je podstatně jednoduší, jelikož konflikty jsou řešeny centralizovaně a pořadí určuje pouze jeden uzel. Pokud tento uzel vypadne, je zvolen nový uzel opět použitím algoritmu výběru 1 z N. Nevýhodou je samozřejmě omezená propustnost při velkém počtu operací, ale systém není blokován jako u pesimistické replikace.

3.3.2 Multi-master

Naproti tomu existují systémy **multi-master**, kde několik nebo všechny repliky systému rozhodují o pořadí operací. Replika odešle operaci, která na ní byla vykonána, ostatním replikám. Pokud se na těchto replikách neprováděla ve stejnou dobu konfliktní operace, je operace úspěšně provedena. Pokud ovšem konfliktní operace proběhla, je nutné naplánovat na všech replikách správné a hlavně stejné pořadí operací, aby byla udržena konzistence. Systém musí tedy disponovat algoritmem shody (z angl. commitment protocol), kterým se repliky shodnou na společném výsledku provádění operací [Saito]. To s sebou ovšem přináší velkou režii při vysokém počtu konfliktních operací. Na druhou stranu, pokud systém nepředpokládá vysoký počet konfliktů, je tato architektura mnohem výkonnější a škálova-

telná, díky chybějícímu úzkému místu jako v architektuře single-master. Pro optimalizaci režie je také možné zvolit různý počet masterů.

3.4 Souběžnost operací

Optimistická replikace přináší již zmíněnou komplikaci v podobě nutnosti detekovat a řešit konflikty vzniklé zápisem do stejného objektu na různých replikách. Úkolem systému je naplánování, seřazení a detekce konfliktů těchto operací tak, aby výsledkem byl opět konzistentní stav.

Systém ovšem není schopný správně seřadit a naplánovat operace, pokud nezná jejich vzájemné pořadí, tzv. *happened-before* relaci [Lamport]. Bohužel v distribuovaném prostředí, ve kterém komunikační zpoždění je nepředvídatelné, nemůžeme jednoduše určit pořadí událostí podle pořadí příchodu, jako je tomu u běžných aplikací.

Představme si dvě operace α a β, které mohli vzniknout na jedné z replik i nebo j. Operace α nastala před operací β (α happened before β), právě když:

- $i = j$ a operace α nastala před operací β (operace se udály na stejné replice v jiný časový okamžik) nebo
- $i \neq j$ a β nastala až poté, co j obdržela a vykonala operaci α, nebo
- existuje jiná operace γ, kdy α předchází γ a γ předchází β.

Pokud ani jedna z těchto podmínek není splněna, není možné jednoduše určit happened-before relaci a obě operace jsou považovány za **souběžné** [Lamport]. Existují dvě základní skupiny technik k určení pořadí a plánování operací, a to syntaktické a sémantické. Syntaktické plánování určuje výsledné pořadí operací z času a místa vzniku dané operace. Naproti tomu sémantické plánování využívá sémantiku operací.

3.4.1 Syntaktické plánovače

Syntaktický plánovač určuje pořadí operací pouze na základě informace kdy, kde a kým byla operace vytvořena [Saito]. Syntaktické plánovače jsou jednoduché a obecné, ale mohou vyvolat zbytečné konflikty. Uvažujme například systém pro rezervaci dataprojektorů, kde následující tři operace vzniknou souběžně: uživatel A si vypůjčí projektor (1), uživatel B si vypůjčí projektor (2) a uživatel C vrátí projektor (3). Syntaktický plánovač tyto operace naplánuje v pořadí vzniku, tedy 1,2 a 3, což znamená, že operace 2 se nezdaří, jelikož projektor je již vypůjčen. Typickým příkladem syntaktického plánovače jsou časové značky.

Reálné hodiny

Nejjednodušší technikou pro určení happened-before relace je použití hodin reálného času. Každá z replik tímto časem disponuje a jednoduše ho uvede v operaci. Porovnání těchto časů mezi replikami je ovšem směrodatné pouze pokud jsou hodiny na všech replikách správně synchronizovány. Kupříkladu mějme dvě operace, α na replice i a β na replice j.

Čas operace β může stále předcházet čas operace α, i když j již dávno operaci α přijalo, jelikož hodiny j můžou být o hodně pozadu oproti i.

Tento problém ovšem není neřešitelný. V dnešní době je velkou snahou právě správná synchronizace časů v distribuovaném prostředí. Existuje například moderní algoritmus **NTP** (Network Time Protocol), který dokáže udržet synchronizaci hodin s přesností na desítky milisekund s téměř zanedbatelnými náklady [Saito]. Časy replik tak můžou být dostatečně přesné pro určení většiny happened-before relací.

Logické hodiny

Logické hodiny, nazývané také *Lamportovy hodiny*, jsou rostoucí skalární časová značka udržovaná na každé jednotlivé replice [Lamport]. Ve chvíli, kdy je provedena operace α, replika zvýší svoji časovou značku a přiřadí novou hodnotu k operaci α. Tato hodnota se označuje C_α. Replika přijímající operaci od jiné repliky změní svoji časovou značku tak, aby byla vyšší než časová značka příchozí operace a zároveň vyšší než aktuální hodnota logických hodin. Pokud tedy operace α nastala před operací β musí platit, že $C_\alpha < C_\beta$. Bohužel logické hodiny (ani jiné skalární hodiny) nemohou detekovat souběžnost, jelikož $C_\alpha < C_\beta$ nutně neznamená, že operace α nastala před operací β [Lamport].

Vektorové hodiny

Vektorové hodiny jsou rozšířením klasických logických hodin a je dokázáno, že jsou nejmenší datovou strukturou pro přesné měření happened-before relace [Saito]. Vektorové hodiny jsou datová struktura VC_i uložená na replice i obsahující M logických hodin, kde M je počet master replik, čili obsahuje informace o logických hodinách všech master replik. Při provedení nové operace α jsou navýšeny pouze logické hodiny dané repliky $VC_i[i]$ a k operaci je přiřazena nová hodnota VC_i. Replika přijímající operaci zamění logické hodiny pro repliku, na které operace nastala, za přijatou hodnotu a zvýší svoje logické hodiny.

Na základě výše uvedeného principu můžeme určit, že operace α nastala před operací β právě tehdy, jsou-li vektorové hodiny rozdílné a zároveň všechny logické hodiny VC_α jsou rovny nebo menší než logické hodiny VC_β pro stejné pozice, neboli VC_α je dominantní nad VC_β. Pokud ani jedna z vektorových hodin není dominantní, jsou operace souběžné. Pokud $VC_i[j] = t$, replika i přijala veškeré operace od j do logického času t [Lamport]. Obecným problémem vektorových hodin je jejich velikost pro velký počet master replik [Saito].

3.4.2 Sémantické plánovače

Sémantické plánování uvažuje sémantické relace mezi jednotlivými operacemi, čili zkoumá operace do hloubky a zjišťuje například jejich komutativitu nebo idempotenci. Pokud použijeme sémantické plánování na příklad vypůjčení dataprojektorů ze sekce 3.4.1, sémantický plánovač tyto operace naplánuje v pořadí 1, 3 a 2, čili všechny operace se zdaří. Sémantické plánování se používá buď to samostatně nebo v kombinaci s happened-before relací určenou syntaktickou metodou, kdy syntaktické plánování nemůže již dále rozhodnout, jelikož se jedná například o souběžné operace [Saito]. Sémantické plánovače jsou často aplikačně specifické a výpočetně náročné na rozdíl od obecných syntaktických plánovačů.

Komutativita

Pokud dvě po sobě jdoucí operace α a β jsou navzájem komutativní, mohou být provedeny v libovolném pořadí, i když nezachovají happened-before relaci, jelikož tyto operace jsou na sobě nezávislé [Saito]. Běžným příkladem je operace zápisu do dvou nezávislých objektů. Díky tomu nemusíme provádět rollback pokaždé, když obdržíme komutativní operaci mimo pořadí.

Kanonické pořadí

Ramsey a Csirmaz [Ramsey] definovali sémantická pravidla pro řazení operací v souborovém systému. Nesouběžné operace jsou řazeny podle syntaktické happened-before relace a pro každý možný pár souběžných operací jsou definována pravidla, která říkají, jak je možné dané operace seřadit. Mějme například operaci vytvoření dvou souborů a/b a a/c. Tyto operace mohou být provedeny v libovolném pořadí, i když modifikují společný objekt - složku a. Jako další příklad můžeme uvést operace smazání a modifikace stejného souboru. Tyto operace jsou označeny jako konfliktní a uživatel je vyzván k jeho vyřešení.

Ramsey a Csirmaz ve svém souborovém systému uvažují pouze tři operace *create*, *remove* a *edit*, kdy například neuvažují operaci *move*, která by značně systém zkomplikovala, jelikož pracuje celkem se třemi objekty - dva adresáře a jeden soubor. I přesto bylo však nutné specifikovat celkem 36 pravidel, s kterými dokázali, že repliky používající tento systém zůstávají konzistentní [Ramsey].

Transformace operací

Transformace operací je technika vyvinutá pro umožnění spolupráce více editorů [Saito]. Editor upraví text objektu na lokální replice a tato opearace (přidání nebo smazání textu) je propagována na ostatní repliky ve formě příkazu - $delete(x)$ nebo $insert(y)$. Ostatní repliky provádějí příchozí operace v pořadí jejich příjmu. Z čehož vyplývá, že dvě repliky mohou vykonat stejné příkazy v jiném pořadí. Pro každý takový možný pár operací jsou definována transformační pravidla, která příchozí příkaz přepíší tak, aby výsledek operace byl stejný, i když je proveden v jiném pořadí.

Mějme například text „abc“ sdílený mezi dvěma editory na replikách i a j. Editor na replice i provede přidání „X“ na začátek textu - $insert(X)$ - a operace je odeslána na repliku j. Editor na replice j provede smazání prvního znaku - $delete(1)$ - a stejně tak odešle operaci na repliku i. Pokud bychom nepoužili transformaci operací, dostali bychom na replice j správný řetězec „Xbc“, ale na replice i nesprávný řetězec „abc“, jelikož operace smazala první znak místo znaku „a“. S použitím transformací dojde k přepsání operace $delete(1)$ na $delete(2)$ a obě repliky zůstávají konzistentní. Systémem těchto přepisovacích pravidel se zabývá například studie *A Calculus for Concurrent Update* [Cormack]. Existují i varianty pro použití v tabulkových editorem nebo souborových systémech [Saito].

Tato sémantická technika i přesto, že je nutné definovat velice složitá přepisovací pravidla i pro jednoduché operace, přináší obrovskou výhodu - není nutné na již aplikované operace provádět rollback a modifikace objektů tak probíhá pouze směrem dopředu.

Optimalizační přístup

Jednou ze sofistikovaných metod pro sémantické řazení je převedení tohoto problému na problém optimalizační [Preguica]. Operace v této metodě jsou navzájem podmíněné. Metoda podporuje několik různých podmínek a mimo jiné například i závislost (operace α musí být provedena po operaci β), implikaci (pokud je provedena operace α, je provedena i operace β) a výběr (jedna z operací α nebo β bude provedena, ale nikdy obě).

Například uživatel může provést rezervaci místnosti A nebo B (výběr), pokud má rezervovanou místnost, musí si vypůjčit dataprojektor (implikace), ale pouze pokud má dostatek kreditů (závislost). Metoda se k těmto operacím chová jako k optimalizačnímu problému, kdy se snaží naplánovat nejlepší pořadí operací pro uspokojení podmínek. Implementaci této metody používá například systém IceCube, který díky efektivnímu hill-climbing algoritmu dokáže seřadit 10.000 operací během pouhých 3 sekund, i když je optimalizace NP-těžkým problémem [Preguica].

3.5 Zpracování konfliktů

Operace α je konfliktní, pokud její předpoklady nejsou splněny. Nejlepším přístupem je vzniku konfliktů zabránit úplně - tzv. **prevence**. Systémy s pesimistickou replikací konfliktům zabraňují zablokováním nebo zrušením operací. Bohužel toto řešení má velkou nevýhodu v tom, že snižuje propustnost systém, jak je popsáno v sekci 3.2.

Některé systémy konflikty dokonce **ignorují**. Jakákoliv operace, která by mohla být konfliktní, je jednoduše přepsána novější operací jako například v případě použití *Thomas write rule* algoritmu [Thomas]. Tyto ztracené operace mohou být nepodstatné, pokud je jejich počet zanedbatelný nebo pokud se uživatel těmto operacím dobrovolně vyhne [Saito].

Dalším možným způsobem, který je možné kombinovat s jinými, je počet konfliktů **redukovat** častou aktualizací replik mezi sebou nebo rozdělením operací na menší části, čímž se podstatně sníží pravděpodobnost výskytu konfliktních operací [Saito].

Nejčastějším způsobem zpracování konfliktů je však jejich vznik povolit a následně je **detekovat a odstranit**. Takové systémy jsou daleko uživatelsky hodnotnější, než systémy, které konflikty ignorují, jelikož nedochází ke ztrátám konfliktních operací. Metody pro detekci kolizí se stejně jako u metod plánování dělí do dvou skupin - syntaktické a sémantické. Syntaktické metody využívají happened-before relaci nebo některou její aproximaci k označení konfliktů. Nejčastěji to znamená, že pokud jsou operace souběžné, jedna z nich je konfliktní. Sémantický přístup využívá znalosti sémantiky operací k detekci kolizí. Například v případě souborového systému je vytvoření dvou nezávislých souborů souběžně nekonfliktní, ale aktualizace stejného adresáře již konfliktní je [Saito]. Opět ovšem platí, že sémantické metody jsou často aplikačně specifické, protože musí znát sémantiku všech operací.

Cílem metod pro odstranění konfliktu je přepsání nebo zrušení konfliktní operace tak, aby byl konflikt odstraněn. Odstranění konfliktů může být provedeno buď automaticky, nebo manuálně. Manuální odstranění konfliktů jednoduše neprovede konfliktní operaci, ale vytvoří dvě nové verze objektu. První verze objektu bez provedené operace a druhá verze objektu s provedenou operací. Uživatel musí následně z těchto verzí jednu vybrat, konflikt vyřešit a odeslat správnou verzi na ostatní repliky. Automatické vyřešení konfliktů je aplikačně specifické a jedná se v podstatě o shodný postup jako v případě manuálního

odstranění konfliktu. Metoda přijímá dva objekty, které sloučí, a vrací nový objekt, který je odeslán na ostatní repliky [Saito]. Často jsou v systémech použity oba přístupy. Například systém SVN se snaží automaticky vyřešit konflikty v souborech a pokud nemůže rozhodnout, je vyzván uživatel, aby konflikt vyřešil manuálně.

3.6 Synchronizace replik

V sekcích 3.5 a 3.4 byly uvedeny klíčové techniky v prostředí distribuovaného systému replik, nyní je čas uvést, jak se zmíněné techniky využívají pro synchronizaci. Synchronizace se dělí podle typu systému do dvou skupin - *state-transfer* a *operation-transfer*.

State-transfer systémy přenášejí mezi replikami celé pozměněné objekty, kdežto operation-transfer systémy přenášejí pouze operace, které změny způsobily. Je možné se na state-transfer systém dívat jako na operation-transfer pouze se třemi operacemi - čtení, zápis a smazání celého objektu. State-transfer systémy jsou jednoduché, protože udržování konzistence znamená odeslání celého nového objektu na ostatní repliky. Operation-transfer systém naproti tomu musí udržovat historii operací a dohodnout se na jejich společném pořadí s ostatními. Na druhou stranu tyto systémy jsou mnohem efektivnější, hlavně v případě, kdy objekty jsou velké a operace vysokoúrovňové. State-transfer systém musí při změně jednoho bitu přenést celý objekt, operation-transfer pouze danou operaci [Saito].

WebDAV protokol je state-transfer systém, protože se omezuje na tři operace s celými objekty - zápis metodou PUT, čtení metodou GET a smazání metodou DELETE. Nebudeme se tedy dále zabývat složitými operation-transfer, ale pouze jednoduchými state-transfer systémy.

3.6.1 Pomalá synchronizace

Při prvotním propojení více replik do jednoho systému je nejdříve nutné provést tzv. pomalou synchronizaci. Tedy přenést všechny objekty mezi všemi replikami a spárovat je pro pozdější změnovou synchronizaci. Pokud se jedná o vytvoření prázdného systému, není pomalá synchronizace nutná, jelikož nejsou žádné objekty, které je nutné přenést. Problém ovšem nastává, pokud se jedná o propojení více replik po výpadku nebo vynucení pomalé synchronizace uživatelem. Mějme objekt α_i na replice i a jeho kopii β_j na replice j. Oba tyto objekty jsou shodné a byly před vynucením pomalé synchronizace synchronizovány proti sobě. U naivní implementace by se vytvořili nové kopie α_j na replice j a β_i na repliace i, což vede k nevyžádané duplikaci objektů. Cílem je tedy tyto objekty i po výpadku spárovat, aby se opět chovaly jako jeden objekt. Standardní metodou je použití unikátního identifikátoru pro označení objektů a následné spárování stejných identifikátorů. Pokud není možné objekty označit shodně na různých replikách, přichází na řadu sémantické metody. Například pro souborové systémy je možné využít absolutní cesty k souboru. Při párování je samozřejmě nutné zpracovat případné konflikty.

3.6.2 Změnová synchronizace

Změnová synchronizace, nebo také rychlá synchronizace, se provádí opakovaně, jakmile jsou repliky spárovány. Je možné přenášet pouze změněné objekty, což je výrazně efektivnější, než provedení pomalé synchronizace všech objektů. U state-transfer systémů se vět-

šinou používají pro detekci změn syntaktické metody. Výhodou state-transfer systému je pro udržení konzistence nutnost přenést pouze nejnovější stav objektu a vynechat všechny předchozí.

Thomas write rule

Každá replika uchovává pro každý objekt časovou značku, která označuje „novost“ objektu. Při synchronizaci repliky i s replikou j je porovnána časová značka C_i a C_j a pokud je C_j novější, objekt je zkopírován z repliky j a časová značka je aktualizována. Tento algoritmus je velice jednoduchý a efektivní, ale nedetekuje konflikty - možným rozšířením je two timestamps algoritmus [Thomas].

S použitím Thomas write rule algoritmu vyžaduje smazání objektu speciální zacházení. Jednoduché smazání objektu z repliky i způsobí při další synchronizaci obnovu objektu z jiné repliky, jelikož její časová značka, v porovnání s neexistující časovou značkou, je novější [Saito]. K řešení tohoto problému je možné použít například Culling tombstones algoritmus popsaný níže.

Two timestamps

Two timestamps algoritmus je rozšířením Thomas write rule umožňující detekci konfliktů. Každá replika uchovává u každého objektu navíc časovou značku poslední aktualizace. Konflikt je jednoduše detekován, pokud při synchronizaci dvou replik je čas aktualizace rozdílný. Nevýhodou této techniky je ovšem detekce tzv. falešných konfliktů při synchronizaci více jak dvou replik, takže je vhodný zejména pro systémy s malou frekvencí konfliktů a malým množstvím replik [Saito].

Modified bits

Modified bits algoritmus je zjednodušením two timestamps algoritmu. Tento algoritmus funguje správně pouze pokud dvě repliky jsou synchronizovány opakovaně - je použit například pro synchronizaci Pocket PC se stolním počítačem [Saito]. Každý objekt obsahuje modified bit. Pokud je objekt modifikován v PC i Pocket PC jsou bity na obou stranách nastaveny a objekt je označen jako konfliktní. Tento algoritmus je možné použít pouze pro změnovou synchronizaci mezi dvěma replikami. V případě použití při pomalé synchronizaci jsou bity ignorovány a veškeré spárované objekty jsou označeny jako konfliktní.

Hash histories

Základní myšlenkou v Hash histories algoritmu je použití otisku objektu místo časové značky pro reprezentaci stavu objektu (například MD5 funkce). Je tak možné sledovat nejen jak se objekt změnil, ale i jeho větvení a slučování v čase. Tabulka historie otisků s počtem replik a změn značně roste, je proto nutné použít algoritmus pro odstranění starých záznamů [Saito].

Culling tombstones

V případě algoritmu Thomas write rule bylo uvedeno, že smazání objektu není triviální záležitostí, jelikož dochází k jeho obnově při synchronizaci. Jednou z metod je po smazaném

objektu zanechat „náhrobek“ s časovou značkou objektu. Kdykoliv poté dojde k synchronizaci s jinou replikou je nalezen náhrobek a objekt je smazán. Nevýhodou algoritmu je narůstající počet náhrobků po smazaných objektech, které ovšem není možné smazat pro případ, kdy se připojí replika, která byla dlouho nedostupná [Saito].

3.6.3 Aktivní synchronizace

Výše zmíněné metody synchronizace se řadí do skupiny pasivních synchronizací (poll metody), kdy replika libovolným způsobem vyhodnotí, že je nutné provést pomalou nebo změnovou synchronizaci a ta je provedena. Spouštěčem může být vynucení uživatelem nebo vnitřní časovač. Mnohem efektivnější metodou je aktivní synchronizace (push metody). Replika začne novou změnu okamžitě aktivně propagovat na ostatní repliky, což značně redukuje zpoždění přenosu změn a eliminuje zatížení při propagaci mnoha změn u pasivní replikace.

3.7 Replikace a WebDAV

Komunikační protokol WebDAV (viz kapitola 2.2) již ve svém základu [RFC4918] obsahuje několik vlastností pro podporu replikace. Společnost Microsoft ve své implementaci pro Microsoft Exchange [MSDN] tento koncept rozšířila o několik nových vlastností zdrojů a již výše zmíněný *Manifest of collection*:

- **UID** je unikátní identifikátor v kontextu celé repliky. Umožňuje tak identifikaci zdroje v procesu změnové synchronizace. UID je automaticky generováno v sémantice serveru.
- **Resourcetag** je neprůhledný textový řetězec obsahující aktuální stav zdroje. Pokud je zdroj jakkoliv modifikován, je společně s ním vygenerován nový *resourcetag*, čímž je možné detekovat modifikované zdroje, případně ho použít pro historii změn pro algoritmus *Hash histories* popsaný v sekci 3.6.2.
- **Last-Modified** obsahuje časovou značku ve formátu ISO 8601 udávající čas poslední změny, což je možné použít pro algoritmy *Thomas write rule* nebo *Two timestamps* popsané v sekci 3.6.2.
- **Metoda SUBSCRIBE** umožňuje provádět aktivní synchronizaci. Jednoduše je možné přihlásit se ke sledování vzniku událostí a jakmile daná událost nastane, je možné provést ihned synchronizaci modifikovaných zdrojů.
- **Manifest of collection** umožňuje provádět změnovou synchronizaci, jelikož vrací seznam modifikovaných zdrojů v kolekci od posledního manifestu. Seznam obsahuje nové, modifikované a smazané zdroje.

4 Návrh modelu synchronizace

V předchozích kapitolách byl představen komunikační protokol WebDAV, který umožňuje úplnou práci s datovým úložištěm Kerio Connect serveru, a existující modely replikace distribuovaných datových úložišť, které je možné pro synchronizaci mezi datovými úložišti použít. Nyní je nutné navrhnout s použitím těchto znalostí fungující model synchronizace, čímž se zabývá celá tato kapitola.

4.1 Požadavky a předpoklady

Před započetím návrhu modelu synchronizace je potřeba stanovit základní požadavky na model synchronizace a množinu předpokladů, které je nutné při jeho návrhu respektovat:

- **Prostředí je nespolehlivé** - Servery nebo komunikační linky mohou kdykoliv vypadnout a to nejen mezi synchronizacemi, ale i v průběhu synchronizace, kdy server nemusí na požadavek odpovědět nebo odpoví chybovou odpovědí.
- **Servery o sobě neví** - Jednotlivé servery jsou od sebe odděleny, neví o sobě a nekomunikují spolu. Není tedy možné synchronizaci navrhnout jako rozšíření Kerio Connect serveru, ale jako proces, který samostatně poběží mimo prostředí Kerio Connect serveru a bude zprostředkovávat komunikaci mezi servery.
- **Synchronizace může být přerušena** - Synchronizace může být zásahem administrátora nebo operačního systému přerušena a poté opět spuštěna.
- **Konflikty objektů** - Objekt může být vícenásobně změněn na více serverech najednou, což může vést ke konfliktu, který je nutné dle povahy objektu korektně vyřešit.
- **Objem dat** - Každý cílový Kerio Connect server obsahuje řádově desítky složek, které mohou obsahovat tisíce zdrojů.
- **Operace jsou state-transfer** - Komunikační protokol WebDAV umožňuje přenášet pouze celé objekty. Synchronizačnímu modelu tedy nepropaguje, v jaké části objektu změna nastala, ale pouze to, že změna v objektu nastala.
- **Uživatel nemůže zasáhnout** - Není možné se jakýmkoliv způsobem spojit s uživatelem a požádat ho o vyřešení konfliktu, všechny konflikty je nutné řešit automaticky.
- **Servery nejsou časově synchronizovány** - Jednotlivé Kerio Connect servery nejsou časově synchronizovány a jejich reálné hodiny se mohou rozbíhat (viz sekce 3.4.1).
- **Konzistence** - Změny musí být propagovány v nejbližším možném termínu synchronizace, ale nemusí být striktně propagovány ihned a systém toleruje občasná nekonzistentní data.

- **Dostupnost** - Synchronizace nesmí omezit dostupnost serverů a musí být dostatečně optimální, aby nevytěžovala dostupné prostředky serveru naplno.

4.2 Cíle synchronizace

Každé Kerio Connect datové úložiště je, dle terminologie WebDAV, rozděleno do stromové struktury kolekcí, které obsahují strukturované nebo nestrukturované zdroje. Kolekce jsou jednoznačně identifikovány prostřednictvím URI, na které se nachází, a mohou být hierarchicky vnořeny do jiné kolekce. Kolekce navíc obsahuje přístupová práva (tzv. *security descriptor*), která definují, jakými právy uživatelé vzhledem ke kolekci a jejím členům disponují. Kolekce je možné pouze vytvářet, mazat nebo upravovat jejich přístupová práva.

Každá kolekce může obsahovat jeden až řádově tisíce dokumentů. Dokumenty nemohou obsahovat další dokumenty ani kolekce. Dokumenty jsou jednoznačně identifikovány prostřednictvím vlastnosti UID, která se na různých replikách pro stejný dokument liší, a dělí se do dvou skupin - nestrukturované dokumenty, u kterých nelze zjistit význam obsahu, a strukturované dokumenty, které mají přesně specifikovanou vnitřní strukturu, a je tedy možné určit význam obsahu. V prostředí Kerio Connect se vyskytuje celkem šest typů dokumentů - emailová zpráva, kalendářová událost, úkol, kontakt, distribuční seznam a poznámka. Poznámka však není podporována pro replikaci a tudíž následující text se o ní nebude zmiňovat. Kolekce se podle typů vnitřních dokumentů dělí podobně jako dokumenty do pěti typů - poštovní schránka, kalendář, úkoly, kontakty a poznámky.

Emailová zpráva

Emailová zpráva je nestrukturovaný dokument, který je specifikován ve standardu [RFC2822]. Kromě obsahu jsou k dokumentu přidruženy i aktivní vlastnosti jako například příjemce nebo předmět zprávy.

Kalendářová událost a Úkol

Kalendářová událost a Úkol jsou strukturované dokumenty ve formátu iCalendar verze 2.0 [RFC2445]. Kalendářová událost obsahuje právě jednu kalendářovou událost VEVENT a úkol obsahuje právě jeden úkol VTODO. Následující dokument je ukázkou jednoduché kalendářové události „Marketingová porada“:

```
1 BEGIN:VCALENDAR
2 BEGIN:VEVENT
3 DTSTAMP:20120223T153700Z
4 UID:503b1180−29cf−4f63−8898−38e633f8b06c
5 ORGANIZER;CN=”Test␣User␣3”:mailto:u3@test.lab
6 SUMMARY:Marketingová porada
7 DTSTART:20120224T153000
8 DTEND:20120224T160000
9 BEGIN:VALARM
10 ACTION:DISPLAY
11 TRIGGER;RELATED=START:−PT15M
12 END:VALARM
13 END:VEVENT
14 END:VCALENDAR
```

Kontakt a Distribuční seznam

Kontakt a Distribuční seznam jsou strukturované dokumenty ve formátu vCard verze 3.0 [RFC2426]. Distribuční seznam je standardní kontakt rozšířený o vlastnosti pro reprezentaci distribučních seznamů. Formát dokumentu je stejný jako u [RFC2445], jen použité vlastnosti se mění. Následující dokument je ukázkou jednoduchého kontaktu „Bc. Jiří Praus“:

```
1 BEGIN:VCARD
2 N:Praus;Jiří;;Bc.;
3 FN:Bc. Jiří Praus
4 URL;TYPE=WORK:www.test.lab
5 TITLE:Programátor
6 ICQ:123456789
7 BDAY;VALUE=DATE:20120208
8 EMAIL;TYPE=PREF,HOME:praus@test.lab
9 UID:60876c2e−3e7b−4678−9153−e133f893f47d
10 END:VCARD
```

Vlastnosti

Ke každému zdroji jsou přiřazeny strukturované WebDAV vlastnosti, které přidávají dokumentu další význam, jako například přečtená/nepřečtená emailová zpráva. Každý zdroj obsahuje velké množství vlastností, ale většina z nich je určena pouze ke čtení nebo jejich obsah je duplicitní s obsahem dokumentu a není je tak nutné synchronizovat.

- Pro kolekce je možné synchronizovat pouze vlatnost *http://schemas.microsoft.com/exchange/security/descriptor*, ostatní jsou určeny pouze pro čtení.
- Pro všechny typy dokumentů je nutné replikovat podmnožinu vlastností jmenného prostoru *http://schemas.microsoft.com/mapi/proptag/*:
  - *x0e070003* speciální MS Outlook příznaky,
  - *x0E2B0003* speciální význam,

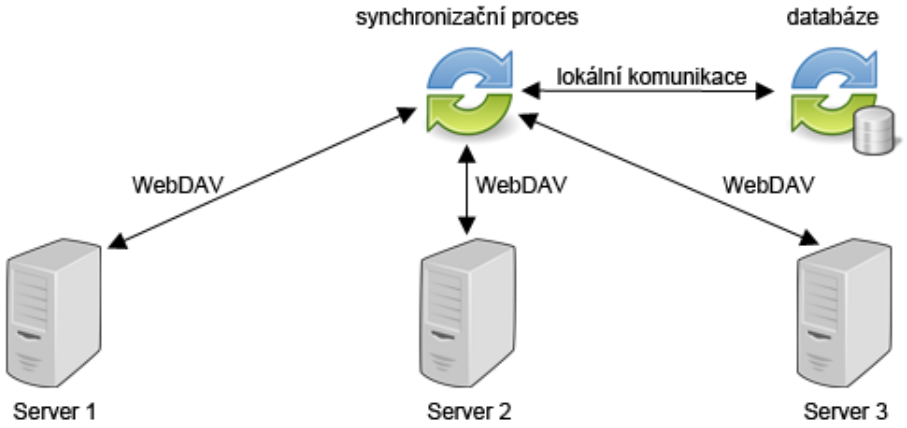
  - *x10900003* stav zprávy,
  - *x10910040* čas označení zprávy stavem,
  - *x10950003* identifikátor ikony stavu,
  - *x67aa000b* označení skryté nebo normální zprávy.

- Pro emailové zprávy je navíc nutné replikovat celé množiny vlastností ze jmenných prostorů:
  - *urn:schemas:mailheader:* pro hlavičku emailu,
  - *urn:schemas:httpmail:* pro další atributy emailu.

4.3 Architektura

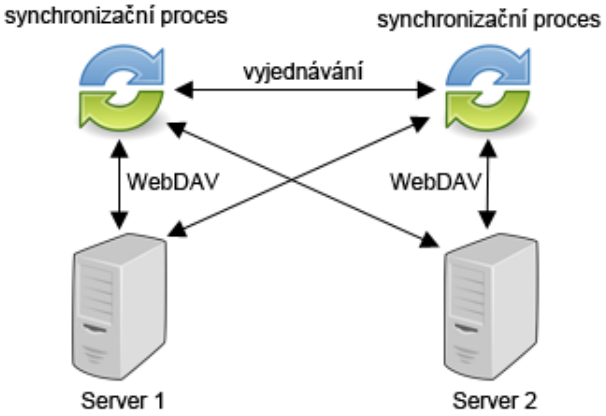
Kerio Connect server je autonomní datové úložiště umístěné v libovolném místě v síti. Jednotlivé servery o sobě navzájem neví a nekomunikují spolu. Uživatelé mohou na každém datovém úložišti vykonávat libovolné operace nezávisle na ostatních serverech, z čehož vyplývá, že model se bude zabývat **optimistickou replikací** blíže popsanou v sekci 3.3.

Jelikož do serverů není možné zasahovat a servery se neznají, synchronizace mezi úložišti musí probíhat pomocí externího procesu, který o serverech ví a bude využívat komunikační rozhraní WebDAV. Každý server může před započetím synchronizace obsahovat libovolná data, která musí být synchronizačním procesem korektně zreplikována na ostatní servery.



Obrázek 4.1: Architektura single-master

Obrázek 4.1 ukazuje návrh architektury synchronizace pro tři Kerio Connect servery. Všechny servery disponují komunikačním rozhraním WebDAV, kterým jsou dotazovány procesem synchronizace. Proces synchronizace běží na jednom ze serverů a replikuje data mezi servery po naznačených komunikačních kanálech. Informace o synchronizaci je nutné ukládat do lokální databáze, jelikož není možné zasahovat do prostředí Kerio Connect serverů, které neposkytují dostatečné informace pro efektivní synchronizaci. Celou architekturu můžeme tedy označit jako optimistickou replikaci single-master.

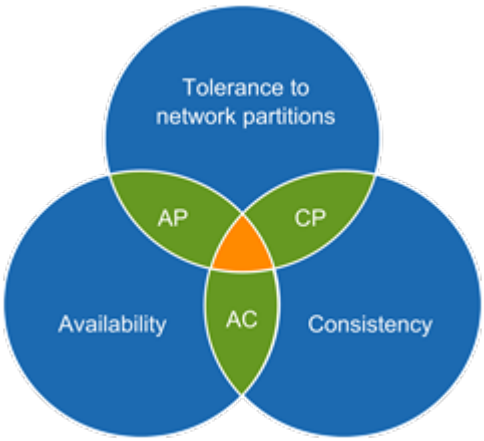


Obrázek 4.2: Architektura multi-master

Obrázek 4.2 naznačuje, jak by komunikace mohla vypadat při použití architektury multi-master pro dva servery, kdy je každý server obsluhován vlastním synchronizačním procesem. Tato architektura je ovšem značně nevýhodná. Stále musíme k serveru přistupovat komunikačním rozhraním WebDAV, kdy je při lokální komunikaci sice sníženo zpoždění, ale za cenu nutnosti implementace distribuovaného algoritmu shody a udržování všech procesů aktivních. Navíc bychom ztratili výhodu protokolu WebDAV - není blokován na firewallech. Toto řešení by mělo smysl, pokud by byla synchronizace přímo uvnitř Kerio Connect serveru, kdy by odpadla lokální WebDAV komunikace. Závěrem tedy je, že pro návrh modelu synchronizace bude použita architektura optimistické replikace single-master.

4.4 Konzistence

Důležitou roli při návrhu modelu synchronizace hraje zvolený model konzistence. Volba modelu konzistence je vždy kompromis mezi třemi ukazateli - dostupnost, striktní konzistence a tolerance výpadků sítě - tzv. *The CAP theorem* [Gilbert].



Obrázek 4.3: The CAP theorem

Obrázek 4.3 zobrazuje tři vyjmenované ukazatele vytvářející možné skupiny modelů

konzistence. Zelené varianty jsou modely, které lze dosáhnout na úkor třetího ukazatele. Systém splňující všechny tři ukazatele (oranžové pole) není možné sestavit, jelikož pokud toleruje výpadky sítě a je striktně konzistentní, nemůže být při výpadku dostupný.

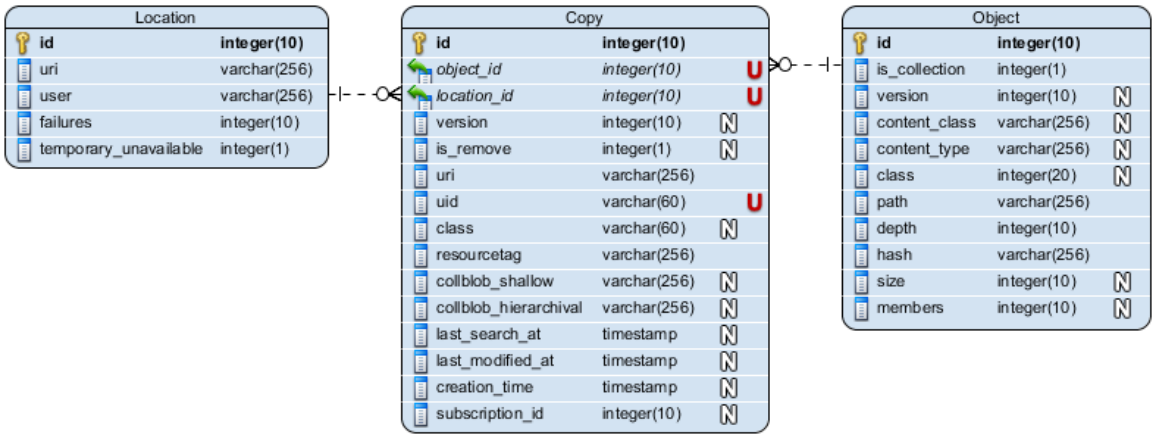
Ukazatel, ze kterého musíme vycházet, je **dostupnost**, protože důraz je kladen hlavně na dostupnost Kerio Connect serverů. Navíc ani není možné do chodu serveru zasahovat, protože komunikační rozhraní WebDAV to neumožňuje. Je tedy možné volit mezi striktní konzistencí a tolerancí výpadků. Striktní konzistence není požadována a navíc by jí nebylo možné dosáhnout. Je proto mnohem optimálnější tolerovat výpadky a dopustit, že na více serverech nebudou aktuální data, než že na žádném serveru nebudou aktuální data. Jako druhý nejvhodnější ukazatel se proto jeví právě **tolerance výpadků**.

Z obrázku 4.3 je možné vyčíst, že jsme zvolili model AP a za tento model lze považovat **eventual consistency**. Synchronizace bude tedy dodržovat eventual consistency, bude tolerovat občasnou nekonzistenci, ale s výhodami lepší škálovatelnosti a propustnosti celého systému.

4.5 Pravdivostní databáze

Jelikož není možné zasahovat do prostředí Kerio Connect serverů, které neposkytují dostatečné informace pro synchronizaci, je nutné informace ukládat do lokální databáze - pravdivostní databáze. Ukládáním průběžných informací do perzistentní databáze se zvýší nejen efektivita, ale i spolehlivost synchronizace, protože při přerušení synchronizačního procesu budou dostupné původní informace o synchronizaci.

Databáze slouží zejména pro účely párování. Každý zdroj může být na každém serveru uložen pod jiným jménem a UID, je proto nutné párovat explicitně s využitím databáze (viz sekce 4.9).



Obrázek 4.4: Model databáze pro podporu synchronizace

Model na obrázku 4.4 je relační databáze o třech tabulkách, které uchovávají dostatečné a přesto minimální množství informací pro efektivní algoritmus synchronizace Kerio Connect serverů protokolem WebDAV. Celý koncept databáze bude postupně vysvětlen v následujících sekcích.

4.5.1 Lokace

V tabulce *Location* jsou uchovávány konfigurace všech serverů, které budou mezi sebou replikovány. Server je identifikován základní URI, na které se nachází kořenová kolekce, a uživatelským jménem pro přístup na zabezpečený server.

4.5.2 Objekt

Tabulka *Object* obsahuje sloučenou stromovou reprezentaci všech zdrojů, které jsou na všech replikách, tak jak by repliky měly vypadat v konzistentním stavu. Kolekce jsou od ostatních zdrojů odlišeny atributem *is_collection*, který je nastaven na hodnotu 1.

Stromová reprezentace je zajištěna algoritmem pro reprezentaci stromu v databázi - materializovanou cestou s atributy *path* udávající název rodičovské kolekce a *depth* reprezentující hloubku vnoření zdroje. Tento algoritmus má složitost výběru celého stromu $O(1)$ a složitost úpravy stromu $O(N)$. Jelikož operace *move* není detekována protokolem WebDAV, úpravu stromu stejně nevyužijeme, což znamená, že je velice efektivní pro reprezentaci stromu oproti například *adjacency list* algoritmu.

4.5.3 Kopie

Díky eventual consistency a možnosti výpadku serverů nemůžeme vycházet z předpokladu, že obsah všech replik je totožný. Nemůžeme proto použít pouze tabulku Object pro reprezentaci stavu synchronizace. Potřebujeme i informace o tom, na jakém serveru je jaký zdroj v jakém stavu přítomný. K tomuto účelu slouží tabulka *Copy*, která reprezentuje aktuální stav všech serverů uložených v tabulce *Location* vůči celkové struktuře v tabulce *Object*. Záznamy jsou pevně svázány se zdroji na serverech atributem *UID*, který je jednoznačně identifikuje na daném serveru. Pokud server neobsahuje záznam *Copy*, není daný zdroj na serveru zreplikován.

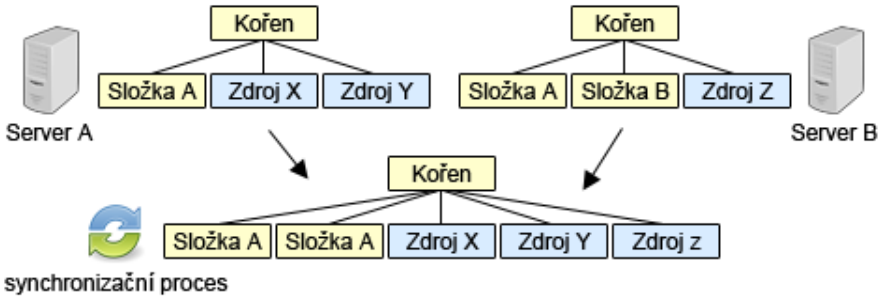
4.6 Synchronizace

4.6.1 Synchronizace kolekce

Základním procesem synchronizace je synchronizace jedné kolekce mezi všemi replikami. Na začátku známe pouze URI kolekce, která existuje na všech serverech. Úkolem synchronizace je kolekci prozkoumat na všech replikách a zreplikovat její obsah. Následující pseudokód naznačuje, jak synchronizace jedné kolekce probíhá.

```
1  Pro všechny servery {
2      Detekuj změněné kolekce pro collblob_hierarchical
3      Detekuj změněné dokumenty pro collblob_shallow
4  }
5  Pokud je počet detekovaných změn > 0 {
6      Pro všechny kopie s příznakem copy.is_remove = 1 {
7          Smaž kopii na serveru
8      }
9      Pro všechny neaktuální objekty seřazené podle object.depth vzestupně {
10         Aktualizuj objekt na serverech
11     }
12 }
```

Algoritmus je rozdělen do dvou fází. První fází je detekce změn a struktury kolekce na všech serverech, abychom spárovali veškerý obsah a vyhnuli se tak nechtěné duplikaci (viz sekce 3.6.1). Pokud bychom totiž prohledali nejprve jeden server, obsah zreplikovali a poté až další server, při opakované pomalé synchronizaci bychom nechtěně duplikovali shodný obsah mezi servery.



Obrázek 4.5: Detekce struktury kolekce kořen

Obrázek 4.5 naznačuje, jak může probíhat sloučení obsahu dvou stejných kolekcí na dvou replikách s rozdílným obsahem. Podrobněji se detekcí změn v kolekcích zabývá sekce 4.8. V druhé fázi algoritmu probíhá aktualizace detekovaných změn na jednotlivé repliky. Prioritu mají operace smazání a po jejich dokončení následují operace přidání nebo modifikace. Aktualizací replik se zabývá podrobněji sekce 4.11.

Data jsou v první fázi perzistentně uložena do lokální pravdivostní databáze tak, aby při chybě nebo selhání v době aktualizace obsahu nedošlo ke ztrátě detekovaných změn a tak ztrátě provedených operací.

4.6.2 Pomalá synchronizace

Při prvním spuštění synchronizace je databáze synchronizačního procesu prázdná a známe pouze kořenovou kolekci celé synchronizace, je tedy nutné prozkoumat strukturu replik, které budeme synchronizovat. Následující pseudokód popisuje algoritmus pomalé synchronizace.

```
1 Vytvoř objekt a kopie pro kořenové složky všech serverů
2 Dokud existuje kopie kolekce s copy.last_searched_at = −1 {
3     Vyber první kolekci s nejnižším object.depth
4     Synchronizuj kolekci
5 }
```

Algoritmus pracuje iterativně a postupuje do hloubky stromu kolekcí. Nejprve jsou vytvořeny prázdné kopie pro kořenové složky serverů, jelikož předpokládáme, že kořenové složky existují, jinak synchronizace nemá smysl. Následně je vybrána kolekce na nejvyšší úrovni, která ještě nebyla synchronizována, jelikož má neinicializovaný čas posledního hledání - $copy.last\_searched\_at = -1$. Kolekce je synchronizována podle algoritmu v sekci 4.6.1 a pokud byly v kolekci nalezeny vnořené kolekce, algoritmus je najde jako ještě neprohledané a provede synchronizaci. Tímto způsobem algoritmus pokračuje, dokud všechny kolekce na všech replikách nejsou prohledány a sesynchronizovány, čímž vytvoří kompletní strom zdrojů a uvede repliky do konzistentního stavu.

4.6.3 Změnová synchronizace

Po prvním provedení pomalé synchronizace je inicializována pravdivostní databáze a je známa struktura všech replik, stačí tedy provádět o poznání rychlejší změnovou synchronizaci.

```
1 Nastav copy.last_searched_at := −1 pro všechny kolekce
2 Dokud existuje kopie kolekce s copy.last_searched_at = −1 {
3     Vyber první kolekci s nejnižším object.depth
4     Synchronizuj kolekci
5 }
```

Algoritmus je v podstatě shodný s pomalou synchronizací s tím rozdílem, že je na začátku místo vytváření kořenové kolekce resetován čas posledního hledání všech kolekcí, tak aby došlo k prohledání všech kolekcí stromu. Rozdíl oproti pomalé synchronizaci je v tom, že již známe stav replik a je možné detekovat pouze nové změny oproti předchozí synchronizaci. Více o detekci změn v sekci 4.8.

4.6.4 Aktivní synchronizace

Protokol WebDAV umožňuje provádět aktivní synchronizaci metodou SUBSCRIBE, což umožňuje efektivně detekovat nové změny na replikách. Nejdříve je nutné se k odběru změn přihlásit metodou SUBSCRIBE a protože je aktivní synchronizace poměrně drahá, je nutné omezit kolekce, ke kterým se budeme přihlašovat, na ty, u kterých to má opravdu smysl:

- Kolekce obsahující kalendářové události a úkoly, jelikož tyto dokumenty jsou často modifikovány a je vhodné pro ně zavést aktivní synchronizaci.
- Kolekce na první úrovni, protože je k nim často přistupováno a jejich obsah je nejviditelnější.

Přihlášení je jednoznačně identifikováno celočíselným *Subscription-ID*, které je uloženo ke každé kopii kolekce v databázi. Pro detekci změněných kolekcí je použita metoda POLL, jelikož je podstatně jednodušší na implementaci a nevyžaduje komunikaci směrem k synchronizačnímu procesu. Umožňuje také aktivní synchronizaci provést v době, kdy má synchronizační proces dostatek zdrojů k její obsluze. Synchronizační proces jednoduše pošle požadavek POLL se všemi *Subscription-ID*, které má uloženy v databázi, a server v odpovědi vrátí, které z nich byly změněny. Změněné kolekce jsou synchronizovány tak, jak je popsáno v sekci 4.6.1.

4.7 Nedostupnost serverů

Jelikož v průběhu synchronizace může dojít k výpadku nebo chybě na libovolném serveru, obsahuje tabulka *Location* atribut *location.temporary_unavailable*, který nabývá hodnoty 1, pokud je server momentálně nedostupný. Tento server je vynechán z jakékoliv aktivity synchronizačního procesu a je otestován na dostupnost až v další iteraci synchronizace. Tímto způsobem se sníží zátěž v podobě zbytečných dotazů na nedostupný server, které by stejně skončily neúspěšně. Tabulka navíc obsahuje čítač $location.failures$ obsahující celkový počet výpadků, díky čemuž je možné zjistit, jaký server je nespolehlivý.

K výpadku serveru dojde kdykoliv, když se nepodaří se serverem spojit protokolem WebDAV, autorizace se nezdaří nebo server odpoví nečekanou odpovědí (například kořenová kolekce neexistuje). Zabrání se tím propagaci dočasně chybné nekonzistence z nedostupného serveru na ostatní repliky.

4.8 Detekce změn

K detekci změn disponuje WebDAV metodou SEARCH, která díky funkci *Manifest of collection* umožňuje detekovat změny v členech kolekce provedené od poslední synchronizace. Implementace této metody produktem Kerio Connect má však jistá omezení. Metoda umožňuje vyhledávat pouze odděleně změny v kolekcích nebo dokumentech a je možné vyhledávat pouze v přímých členech kolekce. Pro detekci změněných kolekcí použijeme metodu SEARCH s SQL dotazem

```
1 SELECT *
2 FROM SCOPE ('SHALLOW␣TRAVERSAL␣OF␣"<cesta>"');
3 WHERE "DAV:isfolder" = false
```

a pro detekci změněných dokumentů použijeme metodu s SQL dotazem

```
1 SELECT *
2 FROM SCOPE ('HIERARCHICAL␣TRAVERSAL␣OF␣"<cesta>"');
3 WHERE "DAV:isfolder" = true AND "DAV:ishidden" = false
```

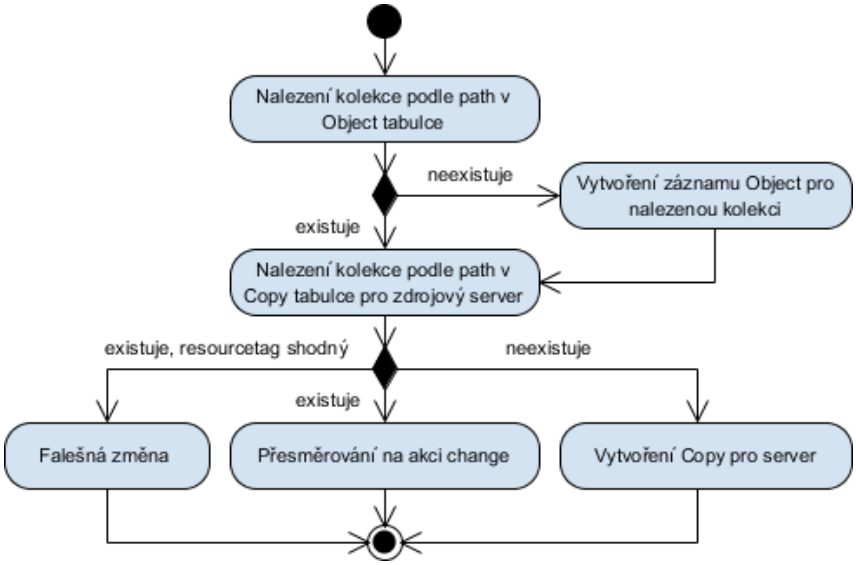
Server v odpovědi vrací seznam všech změněných zdrojů označených jedním ze tří příznaků - *new*, *change* a *delete*. Nalezené změny jsou vráceny od posledního stavu *collblob*, který je předán v požadavku na hledání metodou SEARCH. Při pomalé synchronizaci je zřejmé, že *collblob* je prázdný, jelikož neznáme stav datového úložiště, takže všechny položky seznamu mají příznak *new* - změna od vytvoření úložiště. Všechny změny jsou

zpracovány a uloženy do pravdivostní databáze, ze které proběhne v další fázi synchronizace aktualizace serverů podle nalezených změn. Obecně předpokládáme, že nahlášené příznaky jsou nespolehlivé a jsou vždy ověřeny vůči lokální databázi.

Po úspěšném prohledání kolekce je uložen vrácený nový stav kolekce *collblob* do atributu *copy.collblob_hierarchical* pro hledání změněných kolekcí a *copy.collblob_shallow* pro hledání změněných dokumentů tak, aby při příštím hledání byly nalezeny pouze nové neaplikované změny. Stejně tak je aktualizována časová značka posledního úspěšného hledání *copy.last_searched_at*.

4.8.1 Nová kolekce

Pokud v odpovědi nalezneme kolekci s příznakem *new*, vyhledáme, zda již neexistuje objekt se stejnou URI - kolekce jsou jednoznačně identifikovány relativní URI. Pokud tento objekt existuje, znamená to, že kolekce již byla detekována na jiném serveru a jedná se o spárování s jinou shodnou kolekcí. V opačném případě, kdy objekt neexistuje, je vytvořen nový objekt.



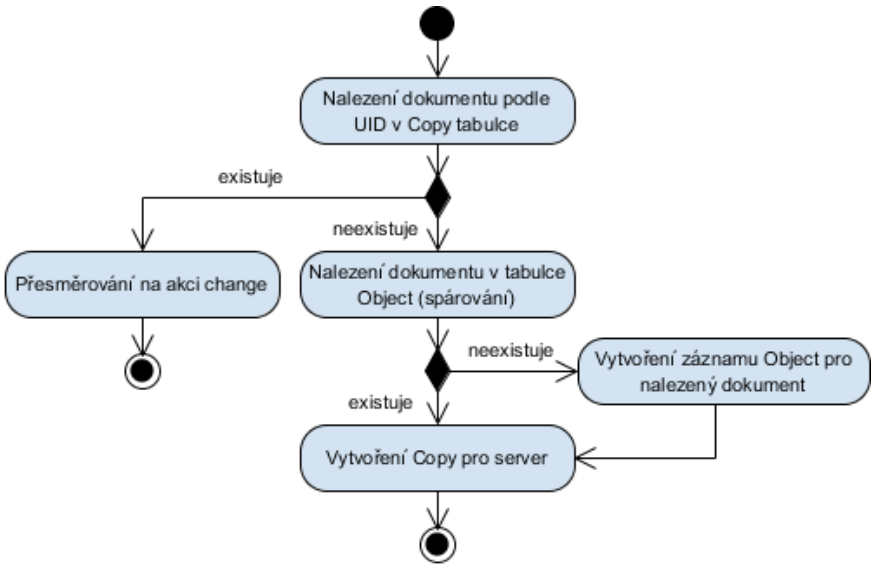
Obrázek 4.6: Stavový diagram algoritmu zpracování nové kolekce

K objektu se poté vyhledá kopie pro server, na kterém proběhlo hledání. Pokud je kopie kolekce nalezena, je nejprve porovnán její *resourcetag*, značící verzi zdroje na serveru. Pokud se *resourcetag* změnil, jedná se o špatně označenou změnu (*new* místo *change*), například při hledání bez použití *collblob*, a je vyvolána obsluha změněné kolekce. Pokud se ovšem *resourcetag* nezměnil, jde o falešnou změnu popsanou v sekci 4.8.6. Poslední variantou je neexistující kopie, kdy je vytvořena nová kopie pro danou repliku. Obrázek 4.6 zobrazuje diagram pro výše popsaný algoritmus.

4.8.2 Nový dokument

Pro nový dokument s příznakem *new* je na rozdíl od kolekcí nejprve hledána kopie objektu pro zdrojový server podle UID - dokumenty jsou jednoznačně identifikovány prostřednic-

tvím UID. Jestliže tuto kopie nalezneme, jedná se stejně jako u nové kolekce o špatně označenou změnu (*new* místo *change*).



Obrázek 4.7: Stavový diagram algoritmu zpracování nového dokumentu

Pokud se ovšem kopii nepodaří najít, je nutné zjistit, zda již stejný dokument nebyl nalezen na jiném serveru. Pokusíme se tedy spárovat dokument s objektem a pokud se nám to podaří, vytvoříme pro něj novou kopii. Pokud se dokument spárovat nepodaří, je vytvořen nový objekt i nová kopie, viz diagram na obrázku 4.7.

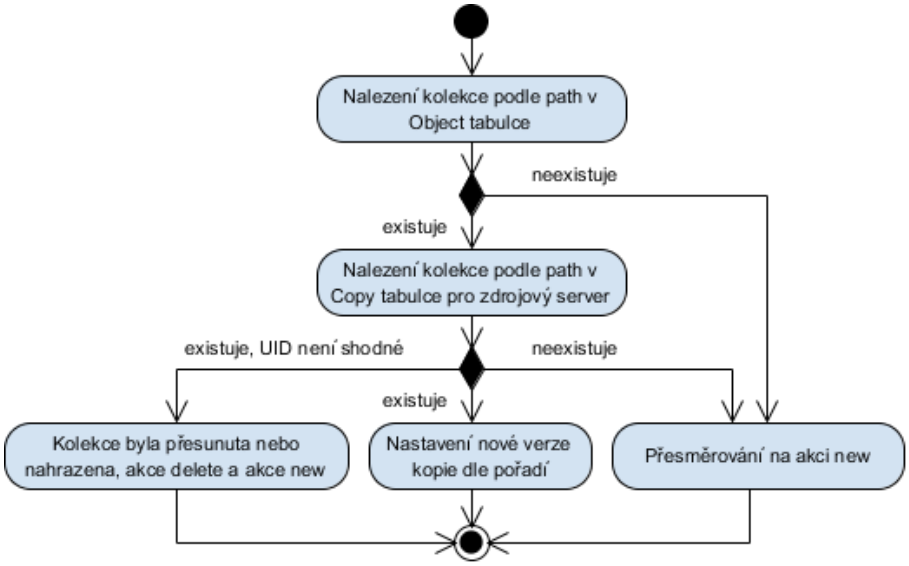
4.8.3 Změněná kolekce

Při akci nad kolekcí s příznakem *change* by se mělo jednat o změnu již existující kolekce. Pokusíme se tedy najít objekt i kopii kolekce a pokud alespoň jeden záznam není nalezen, je akce přesměrována na akci nová kolekce, jelikož nejspíše došlo k dřívějšímu vynechání akce *new* nebo chybnému příznaku ze strany Kerio Connect serveru.

Jestliže jsou ovšem objekt i kopie nalezeny, jedná se zcela jistě o modifikovanou kolekci. Může se ovšem stát, že kolekce byla nahrazena jinou kolekcí se stejným URI. Zkontrolujeme tedy ještě UID a pokud nejsou shodné, kolekci nejdříve smažeme pro ostatní servery a poté ji opět přidáme ze zdrojového serveru s novými vlastnostmi, čímž je přesunutá kolekce propagována na ostatní repliky. Pokud jsou UID shodné, nastavíme pouze novou verzi modifikace podle algoritmu popsaného v sekci 4.10. Výše zmíněný algoritmus je schématicky zobrazen na diagramu 4.8.

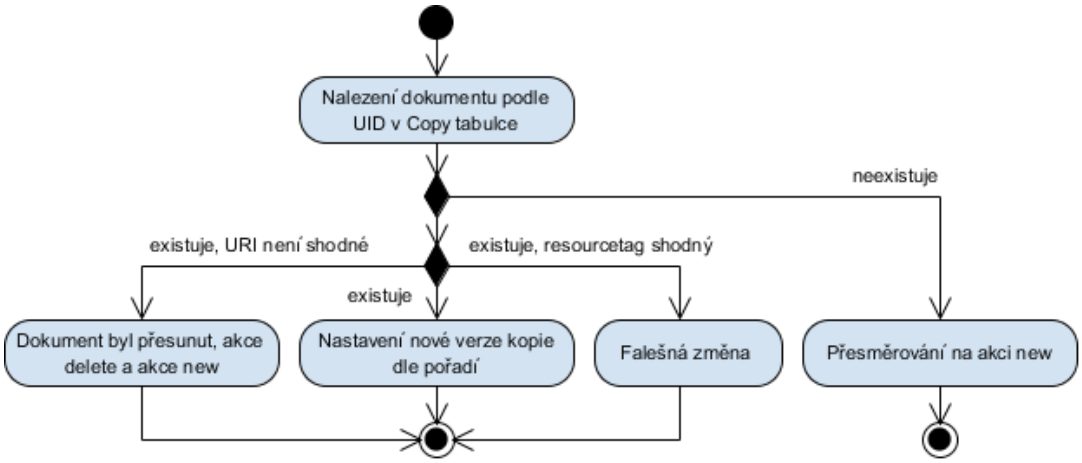
4.8.4 Změněný dokument

Pokud je detekována změna dokumentu, je nalezena kopie dokument podle UID. Jestliže není kopie nalezena, je situace podobná jako u nenalezené změněné kolekce a akce je přesměrována na nový dokument. Pokud je ovšem dokument nalezen, je nutné porovnat URI v kopii a na serveru. Může se totiž stát, že dokument byl přesunut do jiné kolekce a je tedy nutné ho přesunout i na ostatních replikách. Stejně jako u kolekce je v takovém



Obrázek 4.8: Stavový diagram algoritmu zpracování změněné kolekce

případě dokument smazán pro ostatní servery a vytvořen nový ze zdrojového serveru, čímž se zajistí jeho přesun na ostatních replikách. Na druhou stranu může mít dokument stejný *resourcetag*, přičemž se jedná o falešnou změnu popsanou v sekci 4.8.6. Posledním případem je standardní změna, kdy URI je shodná a *resourcetag* jiný. Pro tento případ nastavíme pouze novou verzi modifikace podle algoritmu popsaného v sekci 4.10. Popsaný algoritmus je schématicky zobrazen na diagramu 4.9.



Obrázek 4.9: Stavový diagram algoritmu zpracování změněného dokumentu

4.8.5 Smazaný zdroj

Algoritmus smazání je pro kolekce i dokumenty shodný. Všem kopiím daného zdroje pro všechny servery je v pravdivostní databázi nastaven příznak *copy.is_remove* na hodnotu 1. Akce smazání je vyvolána i v případě, kdy metoda *SEARCH* vrací odpovědí *404 Not found* - kolekce nebyla nalezena, tudíž byla smazána. To ale platí pouze v případě, kdy

kolekce není kořenovou kolekcí pro daný server, v takovém případě je server označen jako dočasně nedostupný, protože kořenová kolekce nemůže být smazána.

4.8.6 Detekce falešných změn

Významnou komplikací při detekci změn je tzv. falešná změna. Falešná změna nastane v případě, kdy je zdroj změněn na replikách synchronizačním procesem. V dalším zavolání metody SEARCH je totiž tato změna také nahlášena. V naivní implementaci by to způsobilo nekonečné aktualizace, protože při jakékoliv aktualizaci serveru by byla změna nahlášená zpět a provedena na jiné replice a opět nahlášena.

Je proto nutné falešné změny detekovat. Při každé aktualizaci je získán *resourcetag* změněného zdroje a uložen do databáze. Při dalším použití metody *SEARCH* je resourcetag porovnán a pokud je shodný, je daná změna ignorována. U kolekcí je nutné detekovat falešné změny pouze u nových kolekcí, změněné kolekce nejsou nikdy falešně nahlášeny. Tímto jednoduchým algoritmem jsou spolehlivě odlišeny falešné změny od skutečných změn.

4.9 Párování zdrojů

Jak již bylo naznačeno v sekci 4.6.1, je nutné při detekci změn v kolekcích párovat stejné zdroje z různých replik, tak aby nedocházelo k nechtěným duplikacím obsahu.

Párování stejných kolekcí je intuitivně řešeno použitím URI kolekce, jelikož stejné kolekce na různých replikách mají stejné URI - jsou umístěné na stejné pozici ve stromě a mají shodný název.

Párování dokumentů je složitější problém, jelikož je nelze párovat intuitivně jako kolekce. Stejné dokumenty mohou na různých serverech být uloženy pod jinými názvy a jejich UID se také liší, jelikož je přidělováno replikami a je určeno pouze pro čtení. Pro dokumenty je tedy nutné navrhnout sofistikovanější algoritmus. Ideální je použití obsahu dokumentu, jelikož pokud jsou dokumenty stejné, je stejný i jejich obsah. K tomuto účelu je pro každý dokument počítán *hash* řetězec MD5 funkcí, který je následně uložen pod atribut *object.hash*, jelikož je pro všechny kopie zdroje stejný. Hash řetězec je kompaktnější reprezentace obsahu než celý obsah, je mnohem rychleji porovnáván a pravděpodobnost shodného hashe je zanedbatelná. Hash řetězec je možné jednoduše generovat u nestrukturovaných dokumentů z celého obsahu, jelikož je obsah dokumentu stálý. Situace se ovšem komplikuje u strukturovaných dokumentů, které obsahují pro každou repliku specifické identifikátory a časové značky. Následující seznam předepisuje, jaké používané atributy je nutné ze strukturovaných dokumentů vynechat před generováním hashe:

- **UID** - identifikátor dokumentu stejný jako vlastnost UID,
- **REV** - časová značka modifikace dokumentu,
- **X-NEXT-ALARM** a blok **VALARM** - čas upozornění na úkol nebo kalendářovou událost, který se často mění a je proto vhodné ho vynechat, aby nezpůsoboval falešné duplikace,

- **PERCENT-COMPLETE** a **STATUS** - speciální vlastnosti úkolu, které se často mění a je proto vhodné je vynechat.

Pro snížení pravděpodobnosti špatného spárování při vygenerování stejného hash řetězce pro různé dokumenty, je navíc párování prováděno s použitím URI rodičovské kolekce a typu dokumentu. Spárování dokumentů tímto algoritmem je spolehlivé, má ovšem značnou nevýhodu. Pro výpočet hash řetězce je nutné stáhnout celý obsah dokumentu na lokální disk, což způsobuje zátěž jak na straně serveru, tak na straně synchronizačního procesu. Ideálním řešením by byla možnost uložit na repliku vlastní synchronizační *ID*, kterým by se dokumenty párovaly, nebo alespoň poskytovat pasivní vlastnost *hash*, kde by byl hash řetězec již spočítán a přenesen pouze jako krátký textový řetězec. Bohužel není možné do Kerio Connect serveru zasahovat a tak výše zmíněný algoritmus je nejlepším spolehlivým řešením.

4.9.1 Konflikt tříd kolekcí

Při párování kolekcí může dojít k tomu, že kolekce se stejnou URI, a tudíž i stejným jménem, může být detekována na více replikách s rozdílnou třídou. Rozdílné třídy kolekcí vyvolají konflikt, jelikož zcela jistě každá z kolekcí obsahuje různé vnitřní členy, a není tedy možné tyto kolekce synchronizovat. Konflikt je nutné vyřešit automaticky, protože nelze požádat uživatele o výběr správné třídy kolekce. První nalezená kolekce je uznána jako správná a nekonfliktní, což znamená, že třída objektu i kopie jsou shodné ($object.container\_class = copy.class$). Jakmile je s objektem spárována jiná kolekce, jejíž třída je rozdílná ($object.container\_class \neq copy.class$), je označena pro synchronizační proces jako konfliktní:

- V konfliktních kolekcích se nehledají změny, protože stejně není možné je replikovat do nekonfliktních kolekcí,
- Do konfliktních kolekcí se nereplikují členové nekonfliktních kolekcí, protože není jisté, zda tam patří,
- Operace odstranění konfliktní i nekonfliktní kolekce se propaguje pouze na kopie stejné třídy, protože se jedná o odstranění konfliktu.

Konflikt tříd kolekcí je vyřešen, jakmile je odstraněna kolekce s konfliktní třídou. Odstranění se nepropaguje na nekonfliktní kolekce a tak je konfliktní kolekce ihned po odstranění nahrazena kolekcí nekonfliktní. Pokud dojde k odstranění nekonfliktní kolekce a zároveň existují konfliktní kolekce, je třída objektu $object.container\_class$ přepsána třídou první konfliktní kolekce, čímž je konflikt odstraněn a kolekce zreplikována. Jiným možným řešením by bylo navrhnutou konfliktní kolekci odstranit a nahradit nekonfliktní, případně konfliktní kolekci přejmenovat a tím uvolnit místo pro kolekci nekonfliktní. Obě řešení však zasahují do struktury serverů bez přičinění uživatele, což je nevhodné chování, které může působit jako chybné.

Navržený algoritmus počítá i se vznikem vícenásobného konfliktu, kdy více jak dvě třídy jsou přítomny nad stejnou kolekcí, a umí jej vyřešit. Podobný konflikt není u dokumentů možný, jelikož párování probíhá i podle typu dokumentu.

4.10 Pořadí operací

Servery u každého zdroje ukládají reálný čas poslední modifikace - vlastnost *Last-modified* je tedy možné určit happend-before relaci s použitím syntaktického plánování reálnými hodinami, jak je popsáno v sekci 3.4.1. Servery ovšem nemusí být časově synchronizovány a jejich reálné hodiny se mohou rozcházet. Neexistuje ovšem jiná metoda pro určení happend-before relace než použití reálných hodin, proto musíme metodu použít, i když nemusí být vždy přesná. I přesto je stále lepší než určovat happend-before relaci náhodně, což víme, že přesné nebude nikdy.

Pro plánování můžeme použití i sémantické plánování popsané v sekci 3.4.2. Operace provedené nad různými zdroji jsou na sobě nezávislé, protože zdroje se navzájem neovlivňují. Tyto operace proto můžeme označit za komutativní a provádět je v libovolném pořadí. Operace je dále možné řadit podle jejich sémantiky - vytvoření, úprava a smazání.

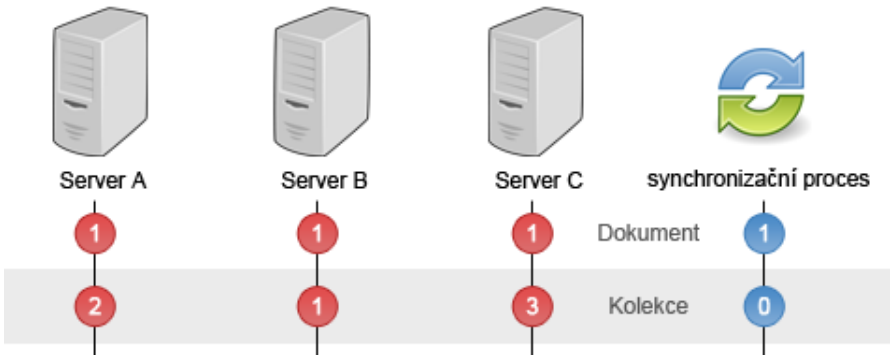
Pro určení pořadí operací definujeme z výše zmíněného jednoduchá pravidla, kterými se bude synchronizační algoritmus řídit:

- Operace nad různými zdroji jsou komutativní a je možné je provést v libovolném pořadí.
- Operace přidání bude vždy předcházet operaci úpravy nad stejným zdrojem, protože není možné upravovat neexistující dokument.
- Operace smazání bude vždy předcházet operaci úpravy nad stejným zdrojem. Pokud totiž bude zdroj smazán, nemá smysl ho předtím modifikovat a operace úpravy mohou být vynechány.
- Stejné operace úpravy nad stejným zdrojem budou řazeny podle časových značek provedené úpravy.
- Operace se stejnou časovou značkou úpravy budou seřazeny náhodně, protože není možné dále určit jejich pořadí.
- Operace přidání nad stejným zdrojem není nutné řadit, protože systém je state-transfer a dovoluje provést pouze jednu operaci přidání a ostatní operace přidání jsou operací úpravy.
- Operace smazání nad stejným zdrojem není nutné řadit, protože se opět jedná o state-transfer systém, ve kterém je operace smazání provedena pouze jednou.

Z výše zmíněných pravidel je možné sestavit deterministické pořadí všech operací, které se v systému vyskytnou. Aby toto pořadí bylo persistentní i při výpadku replik a nezdařené aktualizaci, je nutné jej uložit do databáze. A jelikož stačí pouze ukládání pořadí operací úpravy, neboť ostatní operace mají pouze jedno pevné pořadí, je možné k reprezentaci pořadí použít inkrementální číselné řady - verze, které uložíme u každé *kopie* objektu v databázi, čímž řekneme, v jaké verzi je daná kopie. Pořadí operací úpravy je určeno pořadím verzí. Aby bylo možné poznat, co se změnilo od poslední aktualizace, obsahuje *objekt* další verzi, která označuje poslední zreplikovanou verzi na všech replikách. Pro verze tedy platí následující pravidla:

- Pokud $copy_\alpha.version = object.version$, kopie zdroje na replice α je aktuální.
- Pokud $copy_\alpha.version < object.version$, kopie zdroje na replice α je zastaralá, jelikož nebyla z důvodů výpadku nahrána poslední aktualizace.
- Pokud $copy_\alpha.version > object.version$, nad kopií zdroje na repliace α byla provedena operace úpravy.
- Pokud $copy_\alpha.version > copy_\beta.version$, operace úpravy zdroje na replice α byla provedena po úpravě kopie na replice β.
- Pokud $\forall copy.version > object.version$, žádná z replik nemá aktuální verzi, jelikož na všech replikách byla provedena úprava.

Tento jednoduchý a průhledný systém dvou verzí plně pokryje požadavky na pořadí operací a vyhne se tak nutnosti složitého uchovávání všech provedených operací pro každou repliku. Zacházení s verzemi je názorně ukázáno na následujících příkladech.

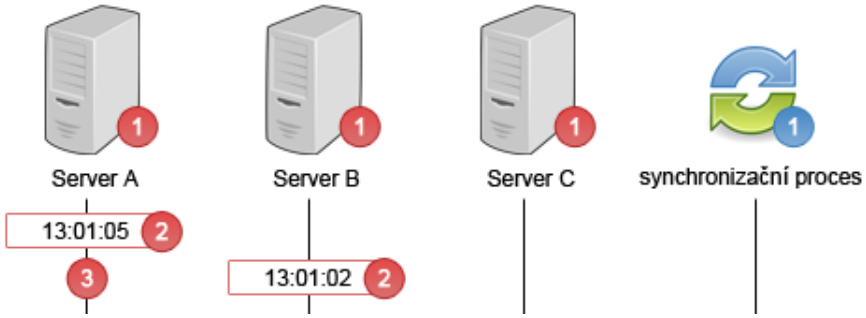


Obrázek 4.10: Nově spárovaný zdroj při pomalé synchronizaci

Obrázek 4.10 ilustruje detekci a spárování tří stejných zdrojů na třech replikách při pomalé synchronizaci pro dokument a kolekci. V horní části obrázku, kde došlo ke správnému spárování všech tří dokumentů, není nutné operace nijak řadit, protože dokumenty musí být stejné (viz sekce 4.9), a verze je nastavena na 1. Toto ovšem neplatí pro kolekce, kde není párování prováděno podle obsahu a kolekce nemusí být stejné. Operace přidání kolekce při pomalé synchronizaci jsou proto řazeny stejně jako operace úpravy. Druhá část obrázku 4.10 naznačuje stejnou situaci pro kolekce, kdy verze kolekcí jsou seřazeny B, A a C podle časů modifikace stejně jako u operací úpravy a na žádné replice není aktuální verze, aby došlo k aktualizaci všech replik.

Na dalším příkladu na obrázku 4.11 je zdroj na všech serverech ve shodné verzi 1. Verze poslední aktualizace uložená v synchronizačním procesu je také 1, tedy objekt je konzistentní. Detekce změn je prováděna na serverech abecedně. První byla detekována změna provedená v 13:01:05 na serveru A, je tedy nastavena verze kopie na hodnotu 2. Jako druhá je detekována změna ze serveru B provedená v 13:01:02, čímž se řadí před změnu provedenou na serveru A. Verze jsou tedy přeplánovány tak, aby změna na serveru A byla zařazena za změnu na serveru B.

Nakonec je nutné reprezentovat persistentně i operace přidání a smazání, které vždy nastanou pouze jednou pro jeden zdroj. Operace přidání je reprezentována neexistující



Obrázek 4.11: Vícenásobná operace úpravy

vazbou *kopie* mezi *objektem* a *lokací*. Jakmile je tedy detekováno přidání, je vytvořen objekt a kopie pouze pro lokaci, na které bylo přidání detekováno. Operace smazání je indikována nastavením příznaku *copy.is_remove* na hodnotu 1 - jedná se o použití algoritmu Culling tombstones ze sekce 3.6.2. Jednoduše je tak zajištěna ochrana při výpadku během provádění operací.

4.11 Aktualizace

Po dokončení detekce provedených operací a určení jejich pořadí je nutné tyto změny zreplikovat. Jako první se provádí operace smazání a po dokončení všech operací smazání jsou provedeny operace přidání a úpravy zdrojů.

4.11.1 Operace smazání

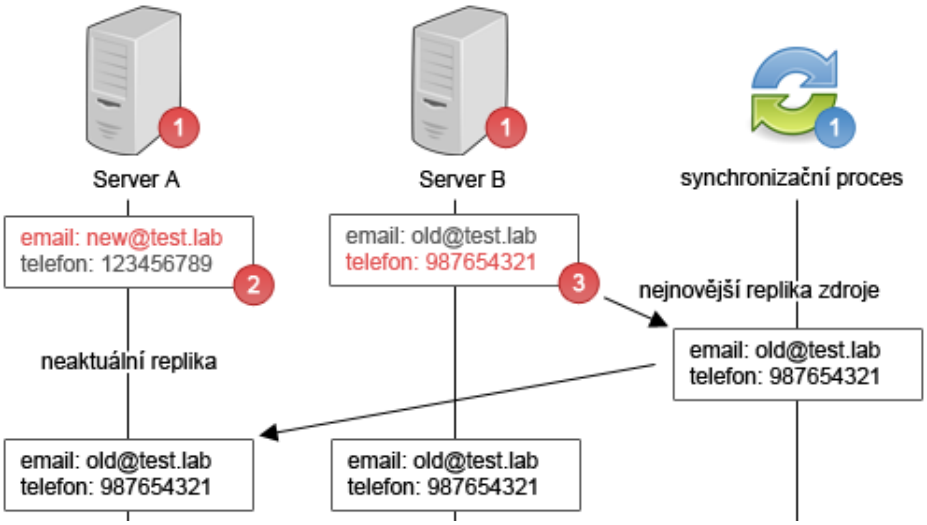
Zdroj na replice má být smazán, pokud $copy.is\_remove = 1$. Při smazání zdroje na jednom serveru musí dojít ke smazání stejného zdroje na všech serverech. Pokud navíc je zdroj kolekcí, je nutné smazat i všechny její členy. Naštěstí protokol WebDAV umožňuje smazat kolekci a všechny její členy jedním požadavkem *DELETE*, nemusíme tedy smazat vždy všechny členy, ale pouze zdroj, který byl smazán. Jakmile je kopie s příznakem *copy.is_remove* smazána fyzicky z repliky, je smazána i její kopie z databáze a pokud neexistuje již žádná kopie zdroje, je smazán i objekt a zdroj přestane existovat. Tímto způsobem se předejde situaci, kdy v době smazání bude některý ze serverů nedostupný a zdroj by se z tohoto serveru obnovil zpět na ostatní servery.

4.11.2 Aktualizace zdroje

Zdroj $copy_\alpha$ na replice α je považován za neaktuální, pokud je splněna alespoň jedna z následujících podmínek:

- Pokud neexistuje záznam o $copy_\alpha$, zdroj byl vytvořen na jiné replice a musí být nahrán na repliku α v nejnovější verzi.
- Pokud $\exists copy_i.version > object.version$ a $copy_i.version > copy_\alpha.version$, existuje jiná novější verze zdroje než $copy_\alpha$ a je nutné zdroj na replice α aktualizovat.

- Pokud $copy_{\alpha}.version < object.version$, je zdroj na replice α neaktuální kvůli předchozímu výpadku a nezdařilé aktualizaci na verzi $object.version$.



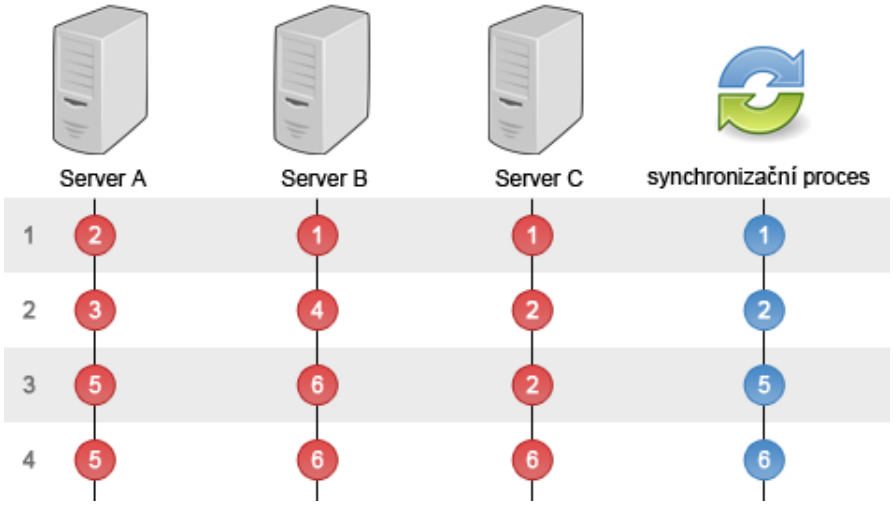
Obrázek 4.12: Použití Thomas Write Rule pro aktualizaci

Aktualizace zdrojů je prováděna vždy od kořene stromu po listy, aby se předešlo případům, kdy chceme nahrát na repliku zdroj, ale rodičovská kolekce zatím neexistuje, což by vyvolalo chybu a neúspěšnou replikaci. Aktualizaci je jednoduše možné provést s použitím Thomas Write Rule tak, že nalezneme nejnovější verzi zdroje, kterou zreplikujeme na ostatní repliky. Tento jednoduchý algoritmus je ovšem čistě state-transfer a neřeší konflikty v operacích, což je v pořádku například pro nestrukturované dokumenty, do kterých není možné zasahovat. Pokud bychom ovšem stejný algoritmus použili pro strukturované dokumenty, například kontakt, tak by při změně jména na replice α a telefonu na replice β byla na repliky propagována jen jedna změna a druhá změna by byla zahozena a navrácena do původního stavu jak zobrazuje obrázek 4.12.

4.11.3 Detekce konfliktů

Abychom vylepšili algoritmus uvedený v sekci 4.11.2, je nutné implementovat zpracování konfliktů. Detekce konfliktů je již vyřešena ukládanými verzemi. Konflikt nastává, jakmile existuje více jak jedna verze kopie $copy.version$ vyšší než $object.version$, což znamená, že nastala více než jedna operace úpravy zdroje.

Obrázek 4.13 představuje čtyři možné příklady stavu verzí kopií zdroje. V prvním případě byla detekována pouze jedna změna a operace není konfliktní, je možné provést klasický state-transfer přenos nejnovější verze. Případ dvě zobrazuje dvě konfliktní operace změny na serverech A a B, kdy je nutné konflikt vyřešit a zreplikovat sloučený zdroj na všechny repliky. Příklad tři obsahuje tři odlišné verze kopií, ale pouze jedna je novější než aktuální verze zdroje, jelikož server C je neaktuální. V tomto případě je detekována jen jedna operace úpravy, tudíž nenastal konflikt. V posledním případě nebyla provedena žádná změna, jen server A je neaktuální. Opět je tedy možné použít přímý state-transfer přenos.



Obrázek 4.13: Možné případy aktualizace verzí

4.11.4 Zpracování konfliktů

Jakmile je detekován konflikt, je nutné jej adekvátně vyřešit, aby nedošlo k žádné nebo k co nejmenší ztrátě provedených operací. Bohužel WebDAV protokol je čistě state-transfer a oznamuje pouze, že změna ve zdroji nastala, ale již neříká, co přesně se ve zdroji nebo vlastnostech zdroje změnilo. Musíme proto navrhnout vlastní algoritmus, který bude umožňovat detekci změn ve zdrojích a bude plně automatický, jelikož není možné dotazovat se uživatele.

Aby bylo možné zjistit, co se ve zdroji změnilo, je nutné mít k dispozici referenční verzi zdroje. Referenční verze je zdroj v původní nezměněné verzi, než byla na replice změna provedena. Jako referenční zdroj je ideálně možné použít libovolnou kopii, pro kterou platí $copy.version = object.version$, čili libovolnou aktuální a nezměněnou verzi zdroje. Jakmile je k dispozici referenční zdroj, je možné zjistit, jaké vlastnosti a atributy jsou ve zdroji nové, změněné nebo smazané.

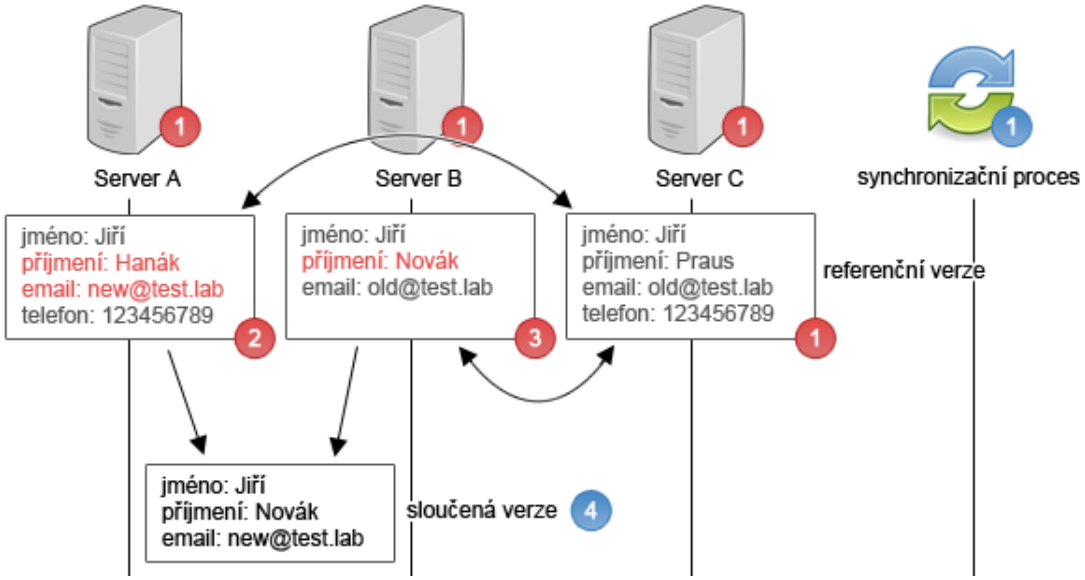
Vstupem do řešitele konfliktů je referenční zdroj a konfliktní změny a výstupem je sloučená verze zdroje. Sloučení probíhá iterativně tak, že konfliktní změny jsou seřazeny vzestupně dle verzí, a jsou porovnávány s referenční verzí zdroje. Na začátku je nová verze zdroje stejná jako referenční verze zdroje a postupně jsou detekované změny propagovány do sloučené verze. Mějme atribut α, referenční verzi zdroje R, změněnou verzi zdroje A a nově vytvářený zdroj M. Nový zdroj M je aktualizován během řešení konfliktů podle následujících pravidel:

- Pokud $R.\alpha = A.\alpha$, je atribut nezměněn a tudíž vynechán, aby nebyla přepsána již provedená změna v M. Jeho hodnota se již v novém zdroji M nachází, jelikož byla zkopírována z referenční verze přiřazením na začátku.
- Pokud $R.\alpha \neq A.\alpha$, je atribut změněn a přepsán v nově vytvářeném zdroji $M.\alpha := A.\alpha$. Pokud bude atribut změněn i v další verzi, bude hodnota opět přepsána.
- Pokud $\exists R.\alpha$ a $\nexists A.\alpha$, byl atribut odstraněn a je odstraněn i z vytvářeného zdroje M. Pokud bude v další verzi zdroje atribut změněn, bude znovu přidán, jelikož

smazaný atribut znamená pro uživatele nastavení atributu na prázdnou hodnotu, což je v podstatě typ změny.

- Pokud $\nexists R.\alpha$ a $\exists A.\alpha$, byl atribut nově přidán a je přidán i do vytvářeného zdroje $M.\alpha = A.\alpha$. Pokud bude přidán i v další verzi, bude hodnota stejně jako v případě změny přepsána.

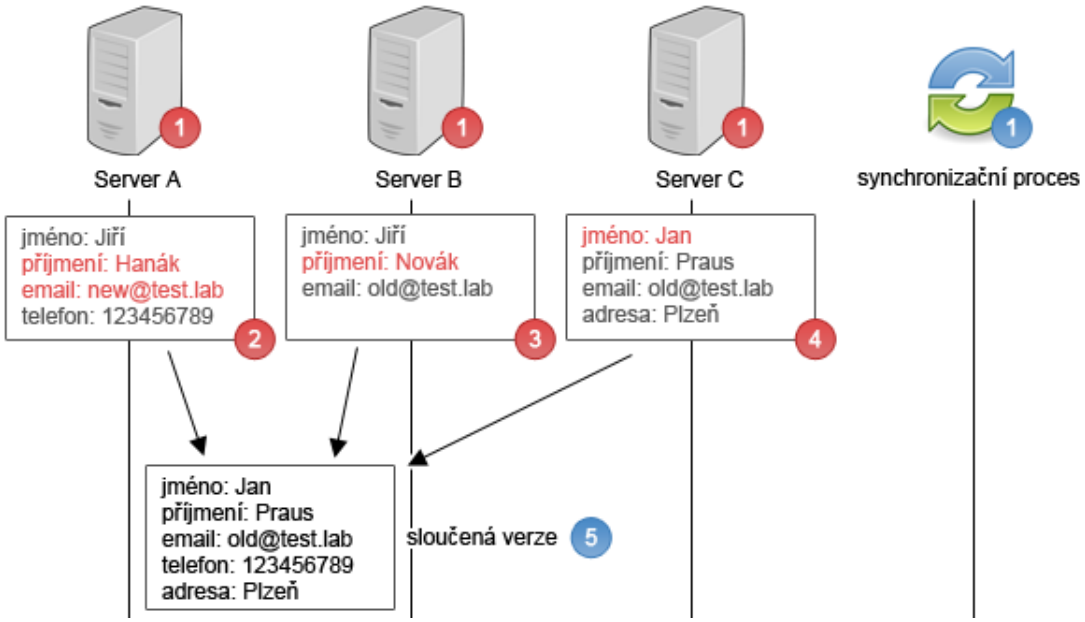
Z pravidel vyplývá, že operace přidání a operace smazání jsou variacemi operace změny. Pokud je atribut smazán, byla provedena operace změny na prázdnou hodnotu, a pokud je atribut přidán, byla provedena operace změny na neprázdnou hodnotu. Obrázek 4.14 ilustruje výše uvedená pravidla na příkladě sloučení dvou konfliktních změn na serverech A a B vůči referenčnímu zdroji na serveru C do nového zdroje. Nový zdroj, který vznikne sloučením konfliktních změn, je nutné označit zcela novou verzí, aby byla splněna pravidla ze sekce 4.11.2.



Obrázek 4.14: Zpracování konfliktu dvou změn vůči referenčnímu zdroji

Problém ovšem nastává, pokud nemáme k dispozici referenční verzi zdroje. Takový případ nastane, pokud jsou změny detekovány na všech replikách a tudíž žádný ze zdrojů není v aktuální verzi. Pro tyto případy by bylo nutné uchovávat vždy aktuální verzi zdroje na lokálním disku, jelikož protokol WebDAV neumožňuje získat dokument v požadované verzi, a je možné si jednoduše představit objem takto uchovávaných dat. Vznik takového případu je ovšem velice nepravděpodobný a kvůli zanedbatelné pravděpodobnosti nemá smysl komplikovat a zvyšovat paměťovou náročnost algoritmu. Tento problém je tedy zanedbán a i když nastane, je uspokojivě vyřešen, jak naznačuje obrázek 4.15. Změna byla detekována na všech serverech A, B i C a není tudíž možné získat referenční verzi zdroje ve verzi 1. Je možné si povšimnou, že hodnoty se pokaždé přepisují novější verzí, jelikož je vždy detekováno přidání nového atributu vůči prázdnému referenčnímu zdroji. Operace změny ani odstranění atributů nejsou detekovány.

Algoritmus řešení konfliktů bez referenčního objektu by bylo možné vylepšit tak, že by se nové atributy přidávali i do prázdného referenčního objektu. Bylo by tak možné



Obrázek 4.15: Zpracování konfliktu tří změn bez referenčního zdroje

detekovat v dalších slučovaných verzích změny v hodnotách atributů. Problém by ovšem nastal, pokud by v dalších verzích zdroje atribut chyběl, jelikož by bylo detekováno odstranění atributu. Není totiž možné automaticky určit, zda je správně operace odstranění nebo přidání. Nejlepší je zvolit takový kompromis, který bude co nejméně destruktuvní, a takovým je právě vypnutí detekce odstranění. Uživatel se snáz vyrovná s tím, že data byla nechtěně vyplněna, než že data zmizela.

Navrhnutý algoritmus je možné použít jak pro vlastnosti zdroje, tak pro strukturované dokumenty, u kterých jsou vlastnosti navíc hierarchicky vnořeny. Pro nestrukturované dokumenty nemá smysl, jelikož nedokážeme rozpoznat vnitřní strukturu. Algoritmus by se dal dále rozšířit o detekci operací na úrovni textových hodnot jednotlivých atributů, například odstranění slova a zároveň přidání slova na jiné replice. Toto rozšíření by však bylo příliš složité a nespolehlivé v komplikovaných situacích, což by mohlo způsobit vznik nesmyslných dat.

4.12 Plánovač

Sekce 4.6 uvedla hned tři možné způsoby synchronizace Kerio Connect serverů - pomalou synchronizaci, změnovou synchronizaci a aktivní synchronizaci. Jednotlivé metody synchronizace je nutné plánovat tak, aby byly všechny změny dostatečně rychle propagovány, ale servery nesmí být přetíženy. Jelikož není možné protokolem WebDAV zjistit momentální zatížení serverů, je nutné synchronizaci plánovat intuitivně a inteligentně:

- Pomalá synchronizace značně zatěžuje jak server, tak synchronizační proces, protože musí prohledat veškeré kolekce a dokumenty v nich a spárovat je mezi sebou. Naštěstí je nutné ji spouštět pouze jednou, server totiž změny spolehlivě oznamuje při každé změnové synchronizaci, protože se vždy ptáme oproti poslednímu úspěšnému stavu.

- Změnová synchronizace je oproti pomalé synchronizaci mnohem méně náročná, ale stále je nutné prohledat veškeré kolekce na všech serverech. Pokud se nahromadí velké množství změn, je srovnatelná s pomalou synchronizací. Je proto nutné ji spouštět dostatečně často, aby nedošlo k náhlému zahlcení synchronizace, ale ne tak často, aby došlo k přetížení serveru hledáním změn.
- Aktivní synchronizace je nejméně náročná ze všech metod. Ale není možné ji aplikovat na všechny zdroje, jelikož by dotazování na změny vyvolalo přílišnou zátěž serveru. Je proto nutné ji kombinovat se změnovou synchronizací.

Spouštění těchto metod je potřeba načasovat tak, aby nedošlo k přílišnému zatížení serveru častým spouštěním synchronizací nebo naopak skokovému přetížení serveru při nahromadění operací. Správné časování je nutné zvolit dle konkrétního řešení a struktury synchronizačního stromu.

4.12.1 Heuristická synchronizace

Jelikož změnová synchronizace prohledává všechny složky a je natolik náročná, že nejde spouštět často, a aktivní synchronizace je možné aplikovat pouze na některé složky, je přidána nová synchronizační metoda s inteligencí. Heuristická metoda vybere nejpravděpodobnější kolekci, ve které mohlo dojít ke změně, a provede synchronizaci této kolekce. Jelikož dochází k hledání změn jen v jedné kolekci, je možné tuto synchronizaci spouštět častěji a tudíž docílit rychlé propagace změn a rozložení zátěže. Vzorcem

$$p = object.members * \frac{time - copy.last\_searched\_at}{delay}$$

získáme pořadí p ve výsledku vyhledávání pro každou kolekci. Proměnná *time* je aktuální časová značka a *delay* je počet sekund mezi heuristickými synchronizacemi. Počet členů je přímo úměrný počtu provedených heuristických synchronizací od času poslední synchronizace v dané kolekci. Vzorec by bylo možné doplnit o hodnotu *object.depth* hloubky ve stromě, ale tato hodnota není směrodatná, protože hluboko vnořená kolekce s velkým množstvím dokumentů může být stejně významná, jako kolekce se stejným množstvím na první úrovni. Výpočet klade důraz na to, aby při použití pouze heuristické synchronizace mohlo dojít k synchronizaci všech kolekcí a právě proto je úměrnou hodnotou čas poslední synchronizace. Synchronizovány jsou kolekce s nejvyšším pořadím.

4.12.2 Rychlost propagace změn

Rychlost propagace změn je závislá na umístění zdroje, který byl změněn, a na plánovači operací. Nejmenším možným časem propagace je zpoždění mezí heuristickou nebo aktivní synchronizací. Nejvyšším možným časem propagace je zpoždění mezi změnovou synchronizací, která detekuje veškeré změny. Průměrnou rychlost propagace je však možné určit až se znalostí konkrétní struktury zdrojů na serveru a jejich typů.

5 Implementace

Závěrečnou částí této práce je navrženou architekturu a algoritmus synchronizace z kapitoly 4 implementovat a otestovat v podobě spustitelného programu a ověřit tak jeho funkčnost a efektivitu. Implementace probíhá přímo proti existujícím Kerio Connect serverům, na kterých je možné celou implementaci dostatečně otestovat.

Implementace musí být přenositelná mezi různými platformami jako Linux/UNIX, Windows a MAC OS. Běh programu probíhá vždy na pozadí bez grafického rozhraní a přičinění uživatele, do budoucna by totiž program měl být součástí Kerio Connect serveru.

5.1 Programové prostředky

Možné programové prostředky splňující výše uvedené požadavky jsou Java SE nebo C++. Oba programovací jazyky jsou objektově orientované. Java SE však vyniká svojí jednoduchostí a vysokoúrovňovými prostředky, je také velice snadno přenositelná na různé platformy. Java SE je orientována na rapidní vývoj za cenu neefektivního použití prostředků. Oproti tomu C++ programovací jazyk je určen pro implementaci kritických aplikací s požadavkem na výkon. Nedostatek vysokoúrovňových prostředků kompenzuje efektivita a rychlost použití nízkoúrovňových prostředků. Nevýhodou C++ je ovšem zhoršená přenositelnost díky přímému použití prostředků operačního systému.

Pro účely implementace se přesto více hodí právě C++, jelikož je možné velice efektivně využívat programové prostředky a vytvořit tak výkonný synchronizační program, což je důležitým měřítkem. Navíc Kerio Connect server je napsán také v C++.

5.2 Knihovny

Při vývoji moderních aplikací je kladen důraz na použití již existujících řešení místo vlastní implementace. Existující knihovny jsou otestovány a odladěny mnohým používáním a často jsou multiplatformní, je proto velice jednoduché je použít v nové aplikaci. Požadavky, které jsou kladeny na knihovny pro použití v této práci, jsou:

- Knihovna musí být vydána pod licencí *LGPL*, *MIT*, *Apache* nebo *BSD*, aby bylo možné využít knihovnu pro komerční účely a nebylo nutné distribuovat zdrojové kódy programu,
- Knihovna musí být dobře zdokumentována.
- Vývoj knihovny musí být stále živý, aby bylo zaručeno, že knihovna je stále používaná, odladěná a není zastaralá.
- Knihovna by měla být Thread-Safe, aby se předešlo problémům při případném použití vícevláknového programu.
- Knihovna musí být multiplatformní, aby bylo možné program přeložit pro operační systémy UNIX/Linux, Windows a MAC OS.

Sqlite 3.7.11

SQLite je C knihovna, která implementuje transakční SQL databázi bez nutnosti serveru a konfigurace. *SQLite* je nejrozšířenější SQL databázový systém na světě. Tato knihovna se výborně hodí pro reprezentaci perzistentní relační databáze představené v kapitole 4.5, pro podporu synchronizačního procesu. *SQLite* verze 3 je vydávána bez licence.

Neon 0.29.6

Neon je HTTP a WebDAV klientská knihovna používající rozhraní v jazyce C. Knihovna obsahuje jak nízkoúrovňové funkce pro komunikaci protokolem HTTP tak vysokoúrovňové funkce pro komunikaci protokolem WebDAV. Knihovna je vydávána pod licencí LGPL.

Libxml 2.7.8

Jelikož WebDAV používá XML technologii, je nutné najít knihovnu, která by uměla jednoduše parsovat XML dokumenty. Takovou knihovnou je *Libxml2*, která obsahuje SAX parser pro čtení obsahu XML dokumentů. Výhodou knihovny je také její přímá podpora v knihovně *Neon*. *Libxml2* je vydávána pod licencí MIT.

OpenSSL 1.0.1

Knihovna *OpenSSL* implementuje Secure Sockets Layer (SSL v2/v3) a Transport Layer Security (TLS v1) protokoly a mimo jiné i všeobecné kryptografické algoritmy. Tuto knihovnu přímo implementuje *Neon* pro komunikaci prostřednictvím HTTPS protokolu. *OpenSSL* je vydávána pod licencí *Apache*.

Log4cpp 0.3.4

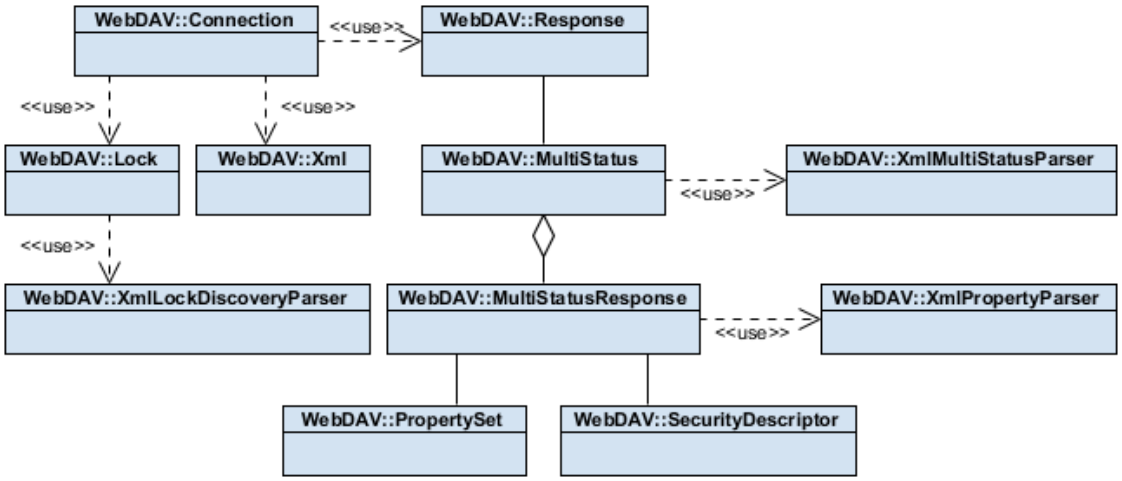
Log4cpp je C++ knihovna pro flexibilní logování do souborů, standardního výstupu a jiných cílů. Jedná se o přepsání známé *Log4j* Java knihovny do C++ prostředí s důrazem na zachování rozhraní. *Log4cpp* umožňuje efektivní logování na různých úrovních díky konfiguračním souborům. Knihovna je používána pro veškerý výstup do souborů nebo konzole. *Log4cpp* je vydávána pod licencí BSD.

5.3 Architektura

Architektura aplikace je dekomponována do čtyř komponent, které jsou popsány v následujících sekcích. Komponenty jsou popsány pouze z vnějšku a přesnou implementaci je možné dohledat v dobře zdokumentovaných zdrojových souborech. Celá aplikace je koncipována jako proces s jedním vláknem s možností běhu na pozadí bez uživatelského rozhraní.

5.3.1 WebDAV klient

První komponentou aplikace je nadstavba knihovny *Neon* implementující potřebné metody a funkce WebDAV protokolu a zapouzdřuje je do jednoduchých metod a objektů. Diagram na obrázku 5.1 představuje objektový model WebDAV komponenty.



Obrázek 5.1: Objektový model WebDAV klienta

Při používání WebDAV klienta je nutné znát pouze třídu *Connection*, která obsahuje implementaci volání všech potřebných WebDAV metod. Třída se stará o vytváření a udržování HTTP spojení na server, jeho zrušení při ukončení volání a také o autentifikaci a ověření certifikátu serveru. Zpracování probíhá čistě proudově, aby nemohlo dojít k vyčerpání operační paměti u rozsáhlých požadavků a odpovědí. Třída *Connection* definuje sadu výjimek, které jsou vyvolány, jakmile nastane chyba uvnitř klienta:

- *UnauthorizedException* - autentifikace se nezdařila,
- *NotFoundException* - zdroj nebyl nalezen,
- *ConnectionException* - spojení se nepodařilo navázat nebo bylo zrušeno.

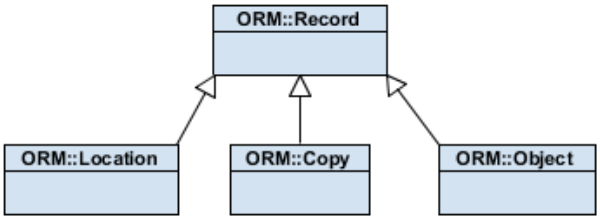
Při volání většiny metod je jako návratová hodnota vrácena instance třídy *Response* zapouzdřující odpověď serveru - hlavičky, návratový kód a tělo odpovědi. Pokud je návratový kód odpovědi 207 Multistatus, je odpověď dále zpracovávána tak, že obsahuje instanci třídy *MultiStatus* reprezentující hromadnou odpověď serveru. Třída *MultiStatus* zajišťuje zároveň přečtení XML těla odpovědi a vytvoření částečné odpovědi pro každý jednotlivý vnitřní stav hromadné odpovědi jako instanci třídy *MultiStatusResponse*.

Pokud mají být v odpovědi vráceny vlastnosti zdroje, jsou vždy uloženy v částečných odpovědích 207 Multistatus návratového kódu. Instance třídy *MultiStatusResponse* se proto stará o další zpracování částečné odpovědi a přečtení všech vlastností, které jsou nalezeny. Vlastnosti jsou uloženy jako pár jméno a hodnota v datovém kontejneru *PropertySet*, který je možné použít i pro následný zápis vlastností na server. Kolekce mohou obsahovat speciální strukturovanou vlastnost *security descriptor*, která je ve WebDAV klientu reprezentována jako instance třídy *SecurityDescriptor*.

Transakční mechanismus je zapouzdřen do třídy *Lock*, která se stará o zpracování metod LOCK a UNLOCK se speciálním tvarem XML odpovědi. Třída zároveň uchovává veškeré transakce platné pro aktuální spojení a automaticky přidává hlavičku *If* do požadavků všech metod, které jsou volány v rámci transakce, dokud není transakce ukončena metodou UNLOCK.

5.3.2 Objektový model

Aby bylo možné jednoduše pracovat s databází, která je nutná pro ukládání perzistentních dat, jak bylo naznačeno v sekci 4.5, je připojení do databáze a práce s databází zapouzdřeno do ORM tříd. Instance těchto tříd reprezentují jeden záznam v databázi a zapouzdřují SQL dotazy ve svých metodách. Například pro uložení záznamu do databáze tak není nutné vytvářet SQL dotaz, ale přímo zavolat metodu *save*, která se o vše postará a vrátí návratový kód provedení operace nad databází. Naproti tomu statické metody pracují s celou databází a slouží pro vyhledávání a hromadné úpravy.



Obrázek 5.2: Objektový model mapování databáze na objekty ORM

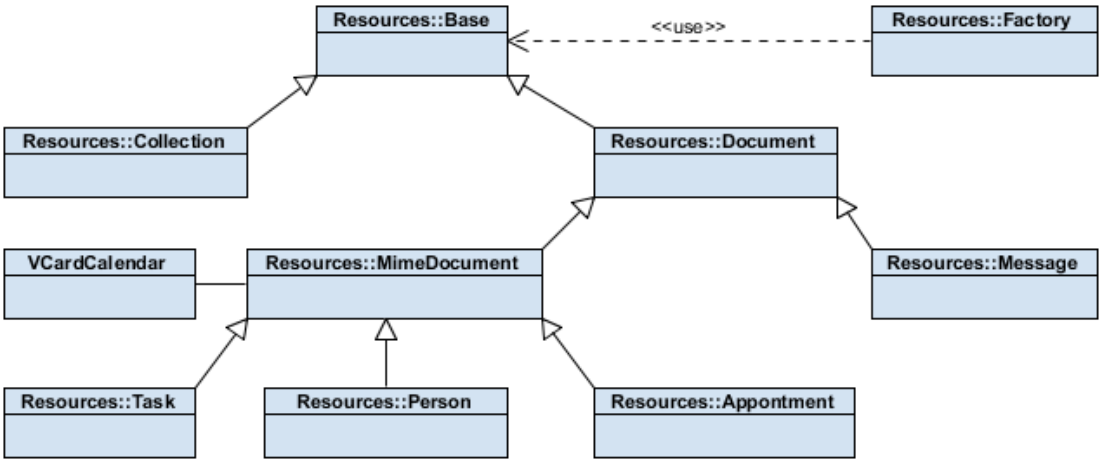
Komponenta se skládá ze tří tříd, které reprezentují každá jednu z tabulek databáze, a jednoho společného abstraktního předka, který definuje obecné metody a základní rozhraní ORM tříd, jak zobrazuje objektový model na obrázku 5.2. Všechny potřebné SQL dotazy a volání jsou takto uloženy na jednom místě a je možné je velice jednoduše a efektivně spravovat nebo rozšiřovat. Pokud se změní schéma databáze, je nutné upravit pouze tyto třídy.

Konzistence databáze je automaticky udržována při každém vytvoření ovladače databáze. Pokud je zjištěno, že databáze je poškozena, je celý proces synchronizace restartován a databáze vyprázdněna, čímž dochází k vyvolání pomalé synchronizace a obnovení procesu synchronizace. Poškození databáze je nahlášeno vyvoláním výjimky *DatabaseCorruptedException*.

5.3.3 Reprezentace zdrojů

Další komponentou aplikace je množina tříd reprezentující zdroje WebDAV serveru - kolekce a strukturované i nestrukturované dokumenty. Zdroje jsou reprezentovány hierar-

chicky, jak ukazuje objektový model na obrázku 5.3. Základem je obecné rozhraní *Base*, které definuje základní metody všech tříd pro reprezentaci zdrojů.



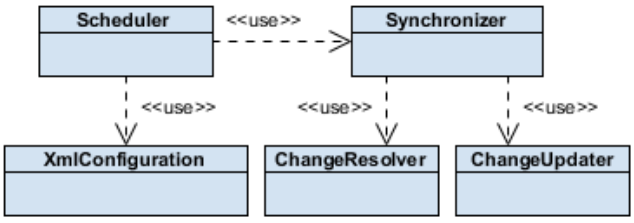
Obrázek 5.3: Objektový model reprezentace zdrojů

Díky této architektuře je možné jednoduše implementovat model synchronizace, který je pro všechny zdroje naprosto shodný, a použít pouze metody rozhraní a specifické vlastnosti různých zdrojů pokrýt až na úrovni implementace metod rozhraní v jednotlivých třídách. Jedná se o využití polymorfismu z objektově orientovaného programování. Tovární třída *Factory* rozhoduje o vytvoření konkrétní instance třídy na základě obdržených vlastností zdroje.

Příkladem může být implementace nahrání zdroje na serveru, kdy při nahrání kolekce je nutné použít metodu MKCOL, kdežto u dokumentu musíme vytvořit transakci a použít metodu PUT. U klasické implementace by se algoritmus nepřehledně větvil při každé odlišnosti. Aplikace je navíc snadno rozšiřitelná o nové typy zdrojů přidáním nové třídy a její implementace do stromu bez nutnosti zásahu do synchronizačního algoritmu.

5.3.4 Synchronizace

Poslední a také hlavní komponentou je implementace synchronizačního algoritmu z kapitoly 4. Komponenta využívá všechny předchozí komponenty a logicky z nich skládá finální synchronizační proces.



Obrázek 5.4: Objektový model synchronizace

Třída *ChangeResolver* implementuje algoritmy ze sekce 4.8, které se starají o detekci a zápis nalezených operací do databáze. Nalezené změny jsou zpracovány třídou

ChangeUpdater, která se postará o detekci a odstranění konfliktů s použitím komponenty reprezentace zdrojů a aktualizuje neaktuální repliky. Zmíněné dvě třídy logicky spojuje hlavní třída *Synchronizer* implementující všechny čtyři synchronizační metody, jež jsou plánovány časovačem ve třídě *Scheduler* popsaným blíže v sekci 4.12. Objektový model komponenty je zobrazen na obrázku 5.4.

5.4 Testování

Základní testování implementace probíhalo mezi dvěma Kerio Connect servery α a β na různých místech sítě propojené internetem. Následující seznam obsahuje výčet všech testovacích případů, které byly navrhnuty a použity pro ověření funkčnosti implementace:

- Kolekce
  - Přidání nové kolekce
  - Přejmenování kolekce
  - Úprava práv kolekce
  - Přesunutí kolekce o úroveň výše
  - Přesunutí kolekce o úroveň níže
  - Přejmenování dvou kolekcí proti sobě na stejná jména
  - Smazání kolekce
  - Smazání a ihned přidání stejné kolekce před detekcí
  - Jméno kolekce s použitím všech povolených znaků
- Konfliktní kolekce
  - Přidání konfliktní kolekce
  - Vytvoření členské kolekce v konfliktní kolekci
  - Vytvoření členské kolekce v nekonfliktní kolekci
  - Úprava práv konfliktní kolekce
  - Úprava práv nekonfliktní kolekce
  - Přejmenování konfliktní kolekce
  - Přejmenování nekonfliktní kolekce
  - Přesunutí konfliktní kolekce
  - Přesunutí nekonfliktní kolekce
  - Přejmenování 2 konfliktních kolekcí proti sobě na stejná jména
  - Smazání konfliktní a ihned přidání nekonfliktní kolekce před detekcí
  - Smazání konfliktní kolekce
  - Smazání nekonfliktní kolekce
- Dokumenty
  - Přidání nového dokumentu
  - Přesunutí dokumentu
  - Úprava dokumentu
  - Konfliktní úprava na více replikách
  - Konfliktní úprava na všech replikách
  - Zkopírování dokumentu
  - Smazání dokumentu

5.5 Zátěžový test

Po ověření a doladění funkčnosti mezi dvěma servery byla aplikace nakonfigurována pro synchronizaci deseti Kerio Connect serverů, aby mohla být důkladně otestována škálovatelnost. V této konfiguraci bylo možné ověřit i odolnost vůči výpadkům v síti a dočasně nedostupným serverům.

Pro důkladné zátěžové otestování byl jeden ze serverů naplněn 3.000 náhodnými testovacími zdroji a byla provedena replikace mezi všemi deseti servery. Synchronizační proces byl spuštěn na samostatné stanici odděleně od serverů s hardwarovou konfigurací uvedenou v tabulce 5.1.

Komponenta	**Synchronizační proces**	**Kerio Connect server**
Procesor	AMD Phenom X4 2.4 GHz	Intel Xeon 3.73 GHz
Počet využitelných jader	1	1
Operační paměť	512 MB	1024 MB
Připojení k internetu	1 GB Ethernet	1 GB Ethernet
Klidové zatížení CPU	0 %	2 %

Tabulka 5.1: Hardwarová konfigurace testovacích stanic a serverů

Profilování první pomalé synchronizace, která replikovala zdroje na devět dalších replik, je možné vidět v tabulce 5.2. Je možné si povšimnout alarmující hodnoty reálného času, po který synchronizace běžela. Ostatní hodnoty jsou však velice uspokojivé. Synchronizace využívala procesor na maximálně 42 % s použitou operační pamětí do 40 MB. Čas strávený na procesoru je také velice nízký, je 83 krát nižší než reálný čas běhu synchronizace. Z naměřených hodnot jednoduše vyplývá, že synchronizační proces 98 % času čekal a nepracoval s procesorem.

Sledovaná veličina	**nejmenší**	**největší**	**střední**
Reálný čas			9h 20m
Procesorový čas			6m 48s
CPU zatížení	0 %	28 %	1 %
Paměť	3,5 MB	39 MB	39 MB
CPU zatížení serveru	2 %	18 %	6 %

Tabulka 5.2: Profilování první pomalé synchronizace

První podezření samozřejmě padlo na komunikaci s Kerio Connect servery. Aby bylo možné přesně identifikovat úzké místo synchronizace, byly všechny často používané WebDAV metody profilovány pro získání přehledu o jejich chování, jak zachycuje tabulka 5.3. Stejné hodnoty byly zjištěny i z logů Kerio Connect serveru, což vyvrací možnost komunikačního zpoždění.

Na první pohled je možné odhalit úzké místo synchronizace - metoda PUT. Metoda PUT se využívá při vytváření nebo úpravě dokumentů na replice, je to tedy nejvíce používaná metoda hned po metodě GET. Při první pomalé synchronizaci byla metoda použita přibližně 27 tisíc krát (3.000 dokumentů * 9 replik), což po vynásobení průměrným časem

Metoda	Odezva		
	nejmenší	**největší**	**střední**
SEARCH	100ms	1,4s	500ms
GET	10ms	40ms	20ms
PUT	0,7s	1,4s	1s
MKCOL	10ms	16ms	13ms
LOCK	8ms	10ms	9ms
UNLOCK	9ms	10ms	9ms

Tabulka 5.3: Profilování často používaných WebDAV metod

1 sekunda na požadavek činí 7,5 hodiny. Jednoduše tedy synchronizace 7,5 hodiny čekala na provedení metody PUT a pokud přičteme použití dalších metod, získáme zjištěných 98 % nevyužitého času.

Sledovaná veličina	**nejmenší**	**největší**	**střední**
Reálný čas			18m 13s
Procesorový čas			2m 38s
CPU zatížení	1 %	28 %	22 %
Paměť	3,5 MB	13 MB	11 MB
CPU zatížení serveru	2 %	18 %	3 %

Tabulka 5.4: Profilování druhé pomalé synchronizace

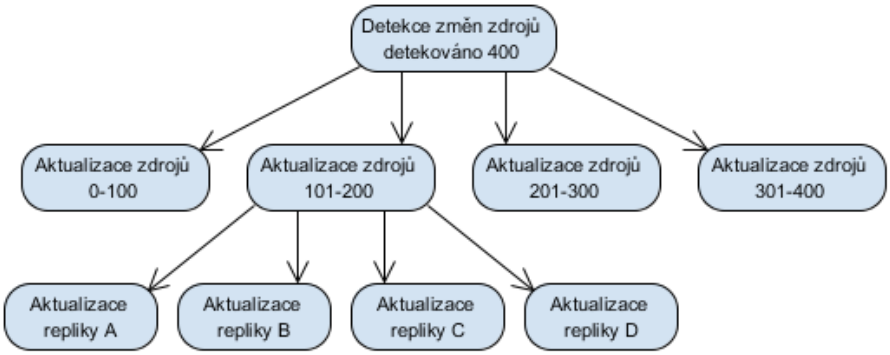
Po restartování perzistentní databáze a opětovném spuštění pomalé synchronizace již byly dosaženy mnohem příznivější hodnoty času běhu aplikace, jak ukazuje tabulka 5.4. Nižší hodnoty jsou samozřejmě způsobeny chybějícími replikacemi dokumentů, jelikož všechny dokumenty již na všech replikách existují a dokumenty jsou pouze spárovány a shledány jako aktuální na všech replikách. Server je více optimalizován pro stahování dokumentů, což je patrné ze zatížení serveru.

Ostatní synchronizační metody, změnová synchronizace, aktivní synchronizace a heuristická synchronizace, jsou nenáročné díky malému počtu replikujících dokumentů, pokud samozřejmě nedojde k nahrání velkého objemu dat, což u emailového serveru není typickým případem užití. Úzkým místem celé synchronizace je tedy replikace nově vytvořeného nebo změněného dokumentu na ostatní repliky při velkém počtu replik nebo velkém počtu operací.

5.6 Urychlení

Synchronizační proces je navrhnut jako jednovláknový a to z důvodu, že párování při detekci musí probíhat sekvenčně, jinak by mohlo dojít k nesprávnému spárování a zavedení nechtěné duplikace. Pokud by párování probíhalo paralelně, muselo by být každé hledání a následný zápis do databáze uzavřen do transakce, což by způsobovalo exkluzivní zamykání tabulek a v podstatě sekvenční běh. Urychlení by zde samozřejmě bylo možné, ale nebude mít takový efekt jako u druhé fáze synchronizace - aktualizace.

Urychlení je možné ideálně dosáhnout zavedením paralelizace při replikaci stejného dokumentu mezi více replik, jelikož repliky jsou nezávislé a na každé může být spuštěna operace PUT bez jakéhokoliv zpomalení nebo zvýšení zátěže na straně serveru. Další prostor pro paralelizaci se nachází při sekvenčním provádění všech operací nad jednotlivými zdroji. Jelikož již známe koncový stav provedení operací, které byly detekovány v první fázi synchronizace, můžeme všechny nové nebo změněné zdroje rozdělit mezi více paralelních procesů.



Obrázek 5.5: Příklad urychlení synchronizace 400 operací mezi 5 replik

Diagram na obrázku 5.5 ukazuje příklad urychlení synchronizace pro 400 nových nebo změněných zdrojů. Detekce zdrojů probíhá opět sekvenčně, aby se předešlo chybnému spárování. Ovšem seznam nalezených operací je rozdělen mezi čtyři pracovní vlákna, která je paralelně vykonávají. Rozdělit seznam prací je možné až ve chvíli, kdy je dokončena detekce změn ze všech replik, jinak by mohlo dojít k chybné replikaci operace, protože by se například přepsala nedetekovaná operace. Pro každou jednotlivou operaci je po vyřešení konfliktů a přípravě zdroje k aktualizaci vytvořena sada aktualizačních vláken, která paralelně provedou aktualizaci zdroje na všechny servery současně, čímž dojde k eliminaci sekvenčního čekání na dokončení metody PUT. Nelze ovšem očekávat, že čím více vláken vytvoříme, tím většího urychlení dosáhneme, pro vlákna proto platí následující pravidla:

- Počet pracovních vláken se odvíjí od velikosti práce, kterou je potřeba provést. Nemá například smysl vytvářet deset vláken pro deset operací, kdy každé provede pouze jednu operaci, jelikož vytvoření vláken má jistou režii. Vlákno má smysl vytvářet pro deset a více operací.

- Počet pracovních vláken je limitován propustností databáze a Kerio Connect serveru, jelikož pracovní vlákna provádí více paralelních operací PUT na jedné replice, což může způsobit nárůst zátěže na serveru. Pracovní vlákna také provádí zápis do sdílené databáze, která se při zápisech zamyká. Při velkém počtu vláken by došlo k zahlcení systému a možnému zpomalení.

- Počet aktualizačních vláken je rovný počtu replik, mezi které má být provedena replikace zdroje. Smysl má ovšem vytvářet vlákno pouze pro dokumenty, u kterých byl změněn obsah a bude použita metoda PUT, která způsobuje největší zpomalení. Jinak by režie vytvoření vlákna převýšila užitečnou hodnotu vlákna.

- Celkový počet vláken není omezen počtem procesorů, jelikož vlákna nejsou výpočetně vázána, ale budou využívat čas, ve kterém ostatní vlákna čekají na dokončení komunikace s WebDAV serverem.

Je možné očekávat značné urychlení zejména při synchronizaci velkého množství replik, jelikož všechny repliky budou aktualizovány paralelně. V ideálním případě je možné očekávat urychlení rovné počtu replik, což samozřejmě z důvodů režie a transakcí v databázi není reálné. Další urychlení způsobí použití pracovních vláken, jelikož servery nejsou plně vytíženy při použití pouze jedné metody sekvenčně. U pracovních vláken je však nutné dávat pozor na přetížení serverů.

Po implementaci urychlení byla spuštěna opět pomalá synchronizace se stejnými daty jako v případě zátěžových testů. Výsledky profilování urychlené pomalé synchronizace s různým počtem pracovních vláknem zobrazuje tabulka 5.5. Synchronizace byla díky použití paralelního nahrávání dokumentů urychlena 7 krát pro 10 Kerio Connect serverů, což je poměrně značný rozdíl oproti neurychlené variantě. Při použití více pracovních vláken je urychlení dále zvyšováno, ale je možné si povšimnout značného nárůstu zatížení procesoru na straně serveru. Počet pracovních vláken by bylo možné dále zvyšovat, ale urychlení již nebude tak výrazné a musíme si uvědomit, že není možné Kerio Connect server plně zatížit jen prováděním synchronizace. Server musí být stále schopný vyřizovat požadavky klientů a přijímat poštu.

Počet pracovních vláken	1	2	3	4	5
Reálný čas	1h 20m	1h 5m	50m 59s	39m 25s	34m 32s
Procesorový čas	6m 38s	6m 43s	6m 48s	6m 51s	6m 59s
CPU zatížení	24 %	26 %	27 %	31 %	32 %
CPU zatížení serveru	12 %	23 %	31 %	42 %	49 %
Urychlení	7	8,6	11	14,3	16,4

Tabulka 5.5: Profilování urychlené pomalé synchronizace s různým počtem pracovních vláken

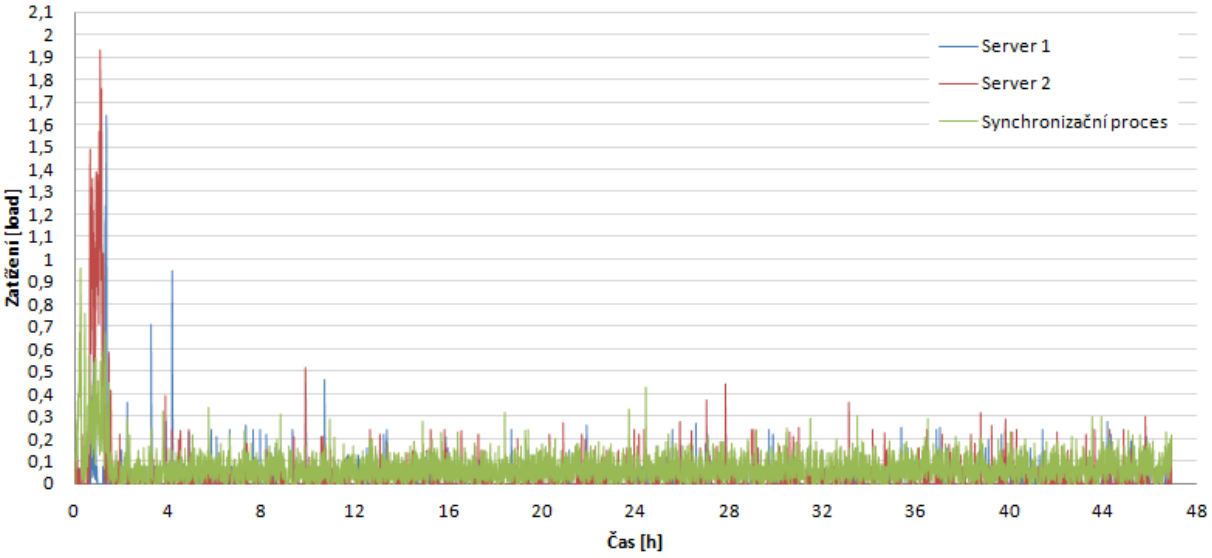
Urychlení se projevilo i u všech ostatních synchronizačních metod, nejen u pomalé synchronizace. Pomalá synchronizace je však nejhorším případem synchronizace a bylo nutné se na ni zaměřit.

5.7 Dlouhodobý test

Posledním provedeným testem byl dlouhodobý test pro ověření, zda synchronizační proces dokáže bezproblémově běžet několik dní bez přestání. Test byl spuštěn oproti dvoum testovacím serverům ve stejné konfuguraci jako při zátěžovém testu (viz sekce 5.5). Pro odzkoušení částečně reálné záteže, byl vytvořen generátor náhodných dokumentů, který generoval každých 10 sekund jeden dokument na server, čímž simuloval používání serveru. Synchronizační proces tak po celou dobu běhu mohl detekovat a synchronizovat nové dokumenty mezi servery. Nastavení časů pro plánovač synchronizačních metod je uvedeno v tabulce 5.6. Jeden ze serverů byl při spuštění synchronizace prázdný a druhý obsahoval 3000 náhodných dokumentů.

Změnová synchronizace	**Aktivní synchronizace**	**Heuristická synchronizace**
1 hodina	30 sekund	60 sekund

Tabulka 5.6: Nastavení plánovače pro dlouhodobý test



Obrázek 5.6: Graf zátěže při dlouhodobém testu

Test byl spuštěn po dobu 48 hodin a hlavní sledovanou veličinou bylo zatížení serverů jak uvádí graf na obrázku 5.6. Graf v první hodinu naznačuje průběh pomalé synchronizace, při které se přeneslo 3000 dokumentů na prázdný server. Průběh dalších 47 hodin je stálý a zatížení serverů se až na výjimky drží pod hodnotou $0,1$, což je velice dobrý výsledek. Viditelné výkyvy hodnot jsou způsobeny prováděním změnové synchronizace. Hodnota zatížení značí na kolik je celý server vytížen a kolik požadavků je možné ještě vyřídit (jedná se o ukazatel load z Unix systémů). Obsah obou synchronizovaných serverů byl po skončení synchronizačního procesu shodný, což dokazuje správnost replikace. Generátor náhodných dokumentů vygeneroval po celou dobu testu dalších 2720 dokumentů.

6 Závěr

Navržený a implementovaný optimistický replikační algoritmus spolehlivě synchronizuje sdílený obsah mezi servery včetně všech jejich vlastností. Pořadí operací při replikaci se řídí kombinací sémantické metody na základě povahy operací a syntaktické metody reálného času, kdy není možné správně řadit operace, pokud nejsou servery časově synchronizovány. Algoritmus dále detekuje konflikty při editaci stejného zdroje na více serverech současně a automaticky je řeší tak, aby nedošlo ke ztrátě provedených operací, i když se jedná o state-transfer protokol. Ošetřeny jsou i různé chybové stavy serverů, jako například konfliktní kolekce nebo dočasně nedostupný server.

Jak ukázal zátěžový test, má synchronizace s použitím WebDAV rozhraní jisté výkonnostní problémy. Metoda PUT má průměrnou dobu odpovědi 1 sekunda, což při potřebě replikovat velké množství dokumentů značně zpomaluje synchronizaci. Problém byl do jisté míry vyřešen urychlením aplikace použitím paralelních vláken, které spustí více metod PUT na serveru souběžně, jelikož server není jednou metodou plně vytížen. Synchronizace je tak libovolně škálovatelná co se týče množství serverů, které může synchronizovat. Zhoršený výkon se projeví až při nutnosti replikovat velký počet zdrojů a proto je nutné správně nastavit plánovač tak, aby nedocházelo ke skokovému přetížení.

Návrh algoritmu byl značně omezen jak možnostmi WebDAV protokolu a jeho implementace Kerio Connect serverem, tak i nemožností zasahovat do implementace Kerio Connect serveru, což by v některých případech značně zefektivnilo procesy. Algoritmus nedokáže při pomalé synchronizaci spárovat dokument, který byl před spárováním na jiné replice změněn. Párování probíhá podle obsahu dokumentu a změněný dokument se od původního liší, čímž dochází k duplikaci obsahu. Pomalá synchronizace však má být spuštěna pouze jednou pro inicializaci synchronizace, čímž se dá problému jednoduše vyhnout. Při velkých objemech dat může dojít k výraznému zpomalení, jelikož protokol WebDAV je state-transfer a je nutný vždy přenos celého změněného dokumentu. Vlastnost state-transfer také způsobuje, že server neví a neoznamuje, kde přesně v dokumentu změna nastala. Navržený algoritmus však umožňuje lokalizovat změny v dokumentu použitím referenčního dokumentu a dokáže tak řešit případné konflikty sloučením nalezených změn.

Protokol WebDAV je vhodný pro synchronizaci replik a to zejména díky své jednoduchosti, možnostem zabezpečení a spolehlivosti v Kerio Connect serveru. Výkon synchronizačního procesu je přitom přímo úměrný rychlosti obsluhy WebDAV metod, je tedy možné vylepšit Kerio Connect WebDAV rozhraní pro dosažení příznivějších výsledků.

Implementace algoritmu je provedená v programovacím jazyce C++ a efektivně využívá přidělené programové prostředky. Aplikaci je možné rozsáhle nastavit pro dosažení maximálního potenciálu dle konkrétní instance distribuované domény. Synchronizace je tak i bez zásahu do Kerio Connect serveru významným vylepšením distribuované domény.

Seznam zkratek

CVS	Concurrent Versions System
DNS	Domain Name System
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
HTTPU	HTTP protokolem UDP
MIME	Multipurpose Internet Mail Extensions
NTP	Network Time Protokol
ORM	Object Relation Mapping
RFC	Request for Comments
SAX	Simple API for XML
SQL	Structured Query Language
SVN	Subversion
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WebDAV	Web Distributed Authoring and Versioning
XML	Extensible Markup Language

Literatura

[Connect] *Kerio:* Kerio Connect
Dostupné na adrese: http://www.kerio.cz/cz/connect

[Cormack] Gordon V. Cormack: *A Calculus for Concurrent Update*. Department of Computer Science, University of Waterloo, 1995

[Ghemawat] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: *The Google File System*, 2003 Google

[Gilbert] Seth Gilbert, Nancy Lynch: *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 2000

[Lamport] Leslie Lamport: *Time, Clocks, and the Ordering of Events in a Distributed System*. Massachusetts Computer Associates, Inc., 1978

[Log4cpp] *Short introduction to Apache log4cx*
Dostupné na adrese: http://logging.apache.org/log4cxx/

[MSDN] *Microsoft Developer Network:* WebDAV Reference
Dostupné na adrese:
http://msdn.microsoft.com/en-us/library/aa486282%28v=exchg.65%29.aspx/

[Preguica] Nuno Preguiça, Marc Shapiro, Caroline Matheson: *Semantics-based reconciliation for collaborative and mobile environments*. Dep. Informática, FCT, Universidade Nova de Lisboa, Portugal, Microsoft Research Ltd., Cambridge, UK, 2010

[Ramsey] Norman Ramsey, Elöd Csirmaz: *An Algebraic Approach to File Synchronization*. Foundations of Software Engineering, 2001

[RFC1945] *RFC 1945:* Hypertext Transfer Protocol – HTTP/1.0
Dostupné na adrese: https://tools.ietf.org/html/rfc1945

[RFC2426] *RFC 2426:* vCard MIME Directory Profile
Dostupné na adrese: http://www.ietf.org/rfc/rfc2426.txt

[RFC2445] *RFC 2445:* Internet Calendaring and Scheduling Core Object Specification (iCalendar)
Dostupné na adrese: http://www.ietf.org/rfc/rfc2445.txt

[RFC2616] *RFC 2616:* Hypertext Transfer Protocol – HTTP/1.1
Dostupné na adrese: https://tools.ietf.org/html/rfc2616

[RFC2818] *RFC 2818:* HTTP Over TLS
Dostupné na adrese: http://www.ietf.org/rfc/rfc2818.txt

[RFC2822] *RFC 2822:* Internet Message Format
Dostupné na adrese: http://www.ietf.org/rfc/rfc2822.txt

[RFC4918] *RFC 4918:* HTTP Extensions for Distributed Authoring - WEBDAV
Dostupné na adrese: http://www.ietf.org/rfc/rfc4918.txt

[Saito] Yasushi Saito, Mark Shapiro: *Optimistic Replication*. ACM Computing Survey, Vol. V, No. N, 3 2005

[Thomas] Robert H. Thomas: *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases*. Bolt Beranek and Newman, Inc., 1979

[W3C] *W3C:* HTTP - Hypertext Transfer Protocol
Dostupné na adrese: http://www.w3.org/Protocols/ Massachusetts Computer Associates, Inc.

A Uživatelská příručka

Program je možné přeložit a spustit na platformách UNIX/Linux, Mac OS a Windows pro 32 i 64 bitové procesory. Minimální systémové požadavky programu jsou:

- procesor Intel Celeron 1,6 GHz nebo ekvivalentní,
- 512 MB RAM,
- 100 MB volných na disku.

A.1 Instalace

Před spuštěním instalace je nutné veškeré soubory a složky na přiloženém CD zkopírovat na zapisovatelné médium, ideálně pevný disk, jelikož jsou generovány soubory uvnitř složek.

A.1.1 UNIX/Linux

K instalaci a překladu synchronizačního procesu pro systém Linux je nutné nainstalovat následující nástroje a balíčky:

- g++
- make
- libslite3-dev
- liblog4cpp5-dev
- libneon27-dev

Předchozí balíčky je možné jednoduše nainstalovat u Debian systémů příkazem *apt-get install název_balíčku*. Pokud nechcete nebo není z nějakého důvodu možné nainstalovat balíčky knihoven *lib**, je možné použít dodané zdrojové soubory a knihovny přeložit. K překladu knihoven slouží skript *install-libraries.sh* umístěný v adresáři *unix/*. Skript spustíte jednoduše příkazem *bash install-libraries.sh* (použijte *sudo*, pokud nejste přihlášeni jako superuser) a automaticky jsou přeloženy všechny potřebné knihovny pro překlad hlavního programu.

Po úspěšné instalaci všech potřebných knihoven program přeložíte příkazem *make* provedeným v adresáři *unix/*. Vznikne tak přeložený spustitelný soubor *unix/sync*.

A.1.2 Windows

K instalaci a překladu na operačním systému Windows je připraven projekt pro volně dostupné Microsoft Visual Studio 2008 Express (dále jen VC8) umístěný v adresáři *windows/*. Soubor projektu *windows/KerioConnectSyncWin/KerioConnectSyncWin.vcproj* jednoduše otevřete ve VC8 a dostanete projekt se všemi potřebnými knihovnami. Projekt přeložíte příkazem *Build* a vznikne nový spustitelný soubor v adresáři *Release/* nebo *Debug/*, podle zvoleného profilu, s názvem *KerioConnectSyncWin.exe* včetně všech potřebných knihoven pro spuštění.

A.1.3 MAC OS

Překlad pro operační systém MAC OS probíhá obdobně jako pro platformu Linux. Nejdříve je nutné nainstalovat všechny potřebné závislosti skriptem *bash unix/install-libraries.sh* (použijte *sudo*, pokud nejste přihlášeni jako superuser). Spustitelný program *unix/sync* opět dostaneme spuštěním příkazu *make* v adresáři *unix/*.

A.2 Nastavení

Před prvním spuštěním synchronizačního procesu je nutné nastavit chování procesu a hlavně cílové servery, které mají být procesem synchronizovány. K nastavení programu sloučí XML konfigurační soubor *config.xml*, který může vypadat například následovně:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <config>
3      <database>database.sqlite</database>
4      <temporary_storage>tmp/</temporary_storage>
5      <acceleration>
6          <enabled>1</enabled>
7          <max_nb_workers>5</max_nb_workers>
8          <min_job_worker>10</min_job_worker>
9      </acceleration>
10     <delays>
11         <complete>60</complete>
12         <heuristic>20</heuristic>
13         <poll>20</poll>
14     </delays>
15     <synchronize>
16         <folder></folder>
17     </synchronize>
18     <locations>
19         <location>
20             <uri>https://195.113.184.42:443/public</uri>
21             <user>u1</user>
22             <password>***</password>
23             <certificate>cert/kerio-connect-42</certificate>
24         </location>
25     </locations>
26 </config>
```

kde

- *database* - libovolná cesta k souboru, na které bude uložena databáze synchronizačního procesu. Databáze bude vytvořena automaticky.
- *temporary_storage* - synchronizační proces potřebuje pro svou činnost průběžně ukládat dočasné soubory na disk, tyto soubory vytváří do nastavené složky. Složka musí být přístupná pro zápis.

- *acceleration* - synchronizační proces je možné urychlit, jak je popsáno v sekci 5.6, použitím vláken. Vlákna jsou ve výchozím nastavení vypnuta, ale je možné jejich použití zapnout nastavením *enabled* na hodnotu 1. Zapnutím urychlení se budou automaticky vytvářet aktualizační vlákna. Pro vytváření pracovních vláken slouží nastavení *max_nb_workers* pro určení nejvyššího počtu pracovních vláken a *min_job_worker* pro nastavení nejmenšího počtu operací, které budou jednotlivá vlákna zpracovávat.

- *delays* - pro optimalizaci plánovače synchronizace je možné nastavit jednotlivé časy v celých sekundách mezi spouštěním synchronizačních metod. Použijte *complete* pro nastavení času mezi spuštěním kompletní synchronizace, *heuristic* pro heuristickou metodu a *poll* pro aktivní synchronizaci.

- *locations* - proces může synchronizovat libovolné množství Kerio Connect serverů. Jednotlivé servery je možné nastavit vložením elementu *location*. Nastavení každého serveru obsahuje:

  - *uri* - URI cílového serveru do kořenové kolekce synchronizace včetně protokolu a portu,
  - *user* - uživatelské jméno pokud je vyžadována autentifikace,
  - *password* - heslo pro uživatelské jméno,
  - *certificate* - soubor certifikátu, pokud je použit protokol HTTPS a certifikát serveru není ověřen certifikační autoritou a je nutné jej ověřit ručně.

- *synchronize* - synchronizační proces je možné omezit na vybrané kolekce v kořenu synchronizace všech serverů. Je možné vložit libovolný počet *folder* elementů s relativní cestou ke kolekci, která má být v rámci procesu synchronizována.

A.3 Spuštění

Jakmile je program nastaven XML konfigurací, je možné jej spustit použitím spustitelného souboru vytvořeného při instalaci synchronizačního procesu. Program je pouze konzolový a je možné jej spustit na pozadí. Průběžné výstupy programu jsou zobrazovány na standardní výstup a zároveň ukládány do log souborů. Pro vynucení pomalé synchronizace při existující databázi je možné použít přepínač „-slow“ za příkazem pro spuštění programu, například:

```
1 ./sync -slow
```

Log soubor zaznamenávající běh aplikace je uložen standardně v *log/run.log* a statistiky synchronizace jsou uloženy v souboru *log/stats.log*. Aplikace umí logovat různé úrovně hlášení pro různé kategorie, vše je možné jednoduše nastavit úpravou souboru *logging.properties*, který lze libovolně změnit podle návodu pro Log4cpp [Log4cpp]. Pro běžný provoz aplikace je určena úroveň hlášení *NOTICE*. Aplikace dělí hlášení do šesti kategorií:

- *sql* - SQL dotazy a operace,
- *webdav* - WebDAV klient,
- *sync* - synchronizační procesy, detekce a aktualizace změn,

- *merge* - řešitel konfliktů,
- *parse* - čtení strukturovaných dokumentů,
- *stat* - statistiky operací.