České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jan Minařík**

Studijní program: Otevřená informatika (magisterský)
Obor: Softwarové inženýrství

Název tématu: **Aplikace pro optimalizaci datového toku při synchronizaci souborů**

Pokyny pro vypracování:

Nastudujte algoritmy pro synchronizaci souborů mezi klientem a serverem, které minimalizují datový tok přenosové linky. Zvolte vhodné řešení pro projekt MsBox, který byl vyvíjen v rámci diplomových prací Ing. Martina Mudry a Ing. Daniela Kavana. Vybrané řešení implementujte a otestujte. Klientskou část implementujte ve formě programových samostatných knihoven pro platformy Windows Desktop, Windows Phone 8, Java a Android. Technické řešení řádně zdokumentujte. Při implementaci zohledněte konfigurovatelnost a více vláknové zpracování dat.

Seznam odborné literatury:

Komunikační protokol RSYNC: https://rsync.samba.org/
Diplomová práce Daniel Kavan FEL, ČVUT, 2012
U. Irmak, S. Mihaylov, T. Suel. Improved Single-Round Protocols for Remote File Synchronization. Polytechnic University, Brooklyn, NY 11201

Vedoucí: Ing. Martin Klíma, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

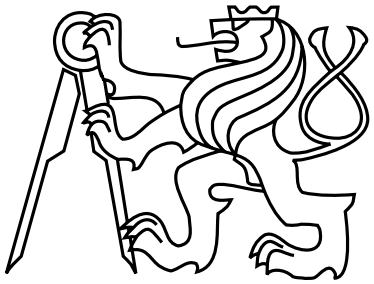
[redacted]
prof. Ing. Jiří Žára, CSc.
vedoucí katedry

L.S.

[redacted]
prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 21. 2. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Aplikace pro optimalizaci datového toku při synchronizaci souborů

Bc. Jan Minařík

Vedoucí práce: Ing. Martin Klíma, Ph.D.

Studijní program: Otevřená informatika, Navazující magisterský

Obor: Softwarové inženýrství

5. ledna 2015

Poděkování

Chtěl bych poděkovat především mému vedoucímu práce panu Ing. Martinu Klímovi, Ph.D. a také své rodině a přátelům za neutuchající podporu při vytváření této práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 5. 1. 2015 .....

Abstract

This thesis describes the design and implementation of server and clients for file synchronization with a reduction of the necessary data which must be exchanged between them. Emphasis is placed on the server scalability and the ability to run in a cloud environment, configurability and optimization of the data stream.

Keywords: file synchronization, optimization of data stream

Abstrakt

Tato práce se zabývá návrhem a implementací serverové strany a klientů pro synchronizaci souboru s redukcí potřebných dat, které si mezi sebou musí vyměnit. Důraz je kladen na škálovatelnost serverové strany a možnost běhu v cloudovém prostředí, konfigurovatelnost a optimalizaci datového toku.

Klíčová slova: synchronizace souborů, optimalizace datového toku

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

1.1 Motivace

V dnešní době je bežné, že si lidé zálohují soubory v cloudovém prostředí. V případě změny zálohovaného souboru je třeba tyto změny propagovat i do cloudového prostředí. To lze řešit posláním celého souboru. Pokud ale v cloudovém prostředí již existuje starší verze, je posílání celého souboru (tedy všech jeho dat) zbytečné. Protože v cloudovém prostředí je již nahrána starší verze souboru, mělo by být možné odeslat pouze změny nad daným souborem a dodatečné informace potřebné k jeho sestavení.

1.2 Popis problému

Problém se synchronizací souboru nastává v případě, kdy strana A vlastní soubor F_{new} a strana B vlastní soubor F_{old}. Strana B chce přijmout/poslat co nejméně dat, aby po jejich aplikaci na soubor F_{old} byla schopna sestavit soubor identický s F_{new}. Strana A ale nemá k dispozici soubor F_{old} (nebo je jeho získání velmi náročné) a tak nemůže snadno zjistit rozdíly mezi oběma soubory. Typicky se tento problém obecně řeší tak, že jedna strana soubor rozdělí na části, spočítají se otisky (hashe) každé takové části a ty se pošlou protistraně. Protistrana provede stejnou nebo podobnou proceduru, porovná obě množiny hashů a pro neshodující se hashe částí souboru zažádá o jejich data.

1.3 Specifikace cíle

Cílem této práce je navrhnout a implementovat řešení založené na vybraném algoritmu, které bude snižovat objem přenesených dat. Součástí řešení jsou nejen klienti, ale i serverová strana, která bude schopna běžet v cloudovém prostředí Windows Azure a bude snadno škálovatelná.

1.4 O systému MsBox

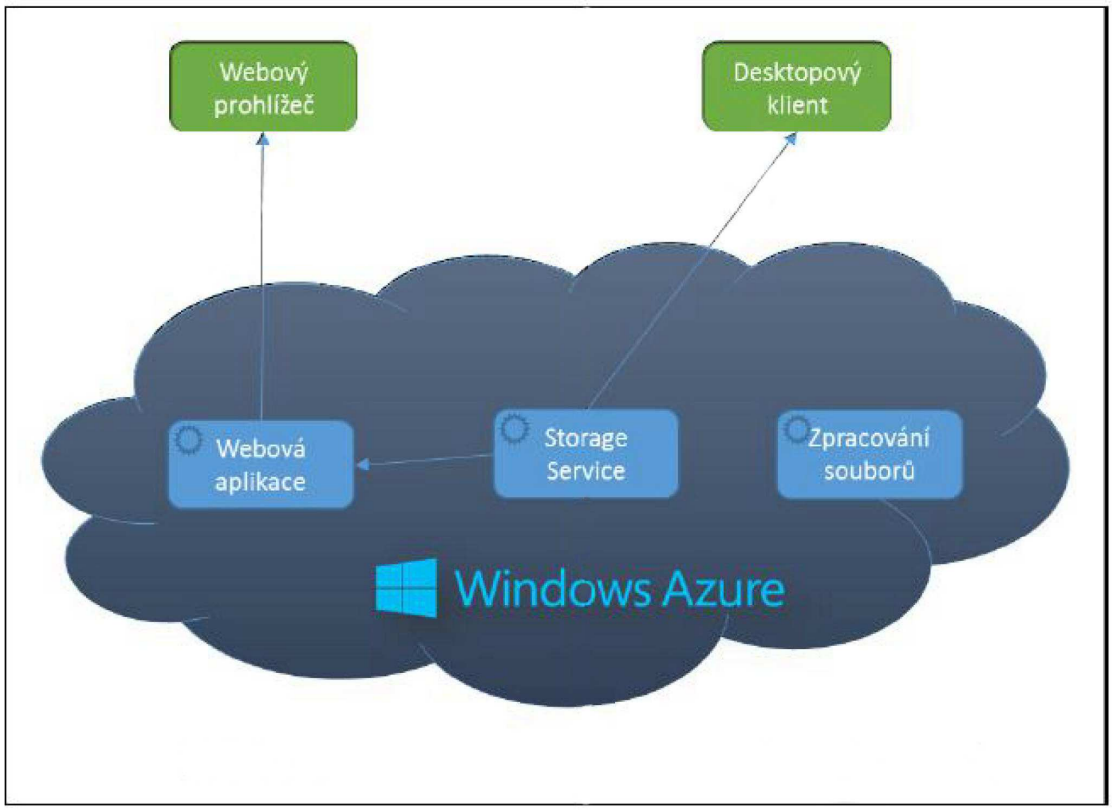
Systém MsBox je určen především pro zálohování souborů do cloudového prostředí Microsoft Azure. Umožňuje stahovat a nahrávat soubory a do jisté míry funguje i jako kolaborační prostředí díky podpoře sdílení s předáním práva zápisu a tvořením uživatelských skupin. Každá skupina pak může operovat nad společnými soubory. Tyto soubory jsou verzovány a uživatelům je umožněno se k dané verzi vrátit. Mimo jiné systém také řeší minimalizaci potřebného objemu diskového prostoru u dat uložených v cloudovém úložišti na úrovni jednotlivých verzí souborů.

Kapitola 2

Analýza

2.1 Analýza stávajícího systému MsBox

Původní systém MsBox, který je popsán v diplomové práci Daniela Kavana[11] byl kompletně předělán v rámci diplomové práce Martina Mudry a bakalářské práce Petra Messnera z důvodu špatné udržovatelnosti, rozšiřitelnosti, nedodržení best practices a aktualizací služeb Microsoft Azure, jak je popsáno v diplomové práci Martina Mudry[13]. Stávající verze systému využívá frameworku Windows Communication Foundation (WCF) a v současné době komunikuje pomocí SOAP (Simple Object Access Protocol) protokolu, který je založen na XML (Extensible Markup Language) zprávách. Pro ukládání informací o uživatelích, adresářích a metadat souborů používá databázový systém Azure SQL. Soubory, skupiny a uživatelé jsou pak identifikováni pomocí unikátních identifikátorů. Pro ukládání souborů je využito cloudové úložište Azure BLOB, kde se v současné době využívá konkrétně typ Block BLOB. Serverová část systému je implementována v jazyce C# 5.0 a využívá framework .NET verze 4.5. Rozdělení systému na jednotlivé služby a klienty je znázorněno na obrázku 2.1.



Obrázek 2.1: Rozdělení systému MsBox na jednotlivé služby a klienty. Inspirováno [13]

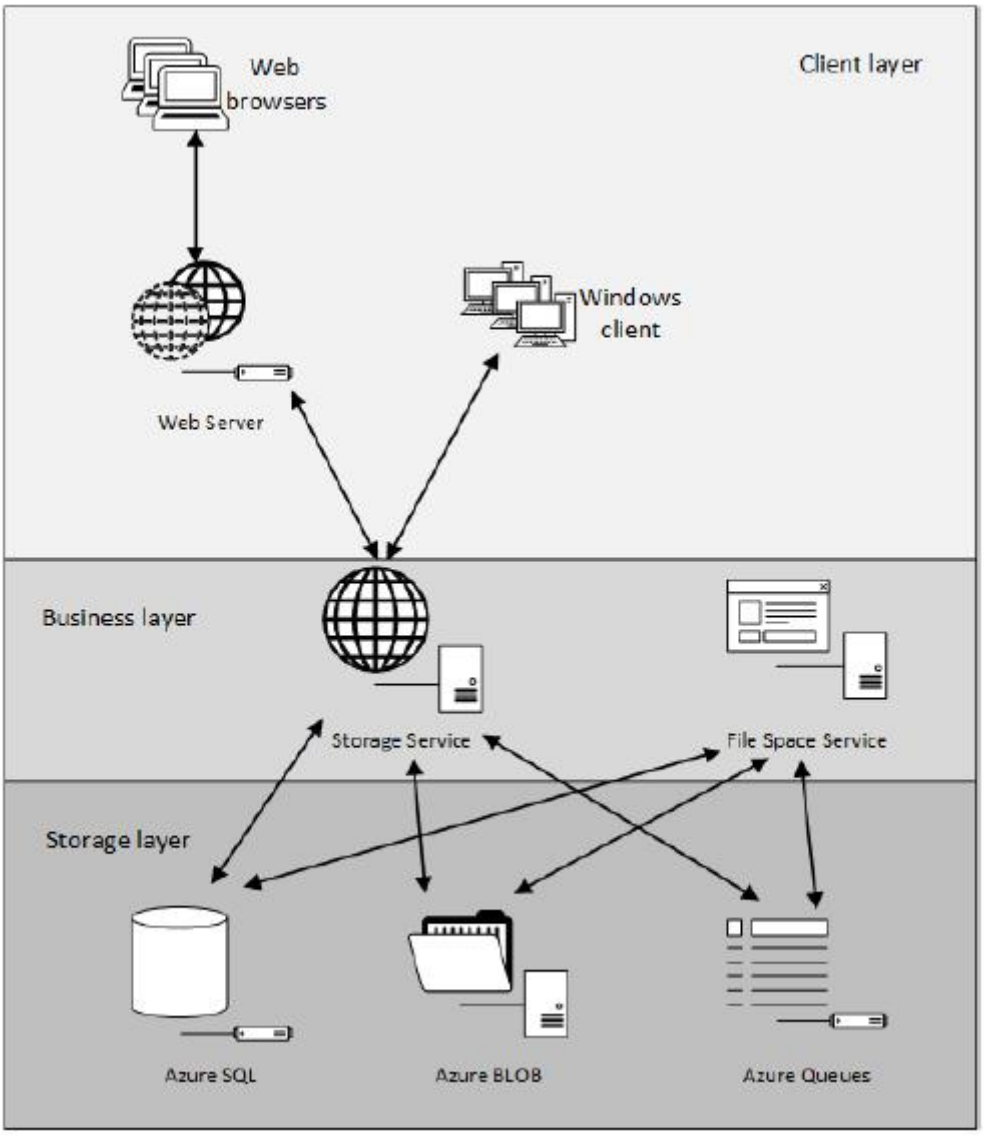
2.1.1 Architektura

Systém MsBox využívá client-server architektury a je rozdělen do 3 vrstev jak je znázorněno na obrázku 2.2.

Popis jednotlivých vrstev a jejich komponent[13]:

- *Client layer* - klientská vrstva poskytující potřebné komunikační rozhraní se serverem a nabízející uživateli dostupné služby s podporou vizualizace
  - *Web browser* - webový klient komunikující s web serverem
  - *Web server* - webový server obsluhující web klienty, který případně propaguje požadavky do *Business logic layer*
  - *Windows client* - tlustý klient pro platformu Windows Desktop
- *Business logic layer* - vrstva obsahující hlavní logiku serverové strany
  - *Storage Service* - obsluhuje všechny požadavky klientské vrstvy a poskytuje operace s uživateli, složkami a soubory.
  - *File Space Service* - řeší minimalizaci uložených dat na úrovni verzí souborů

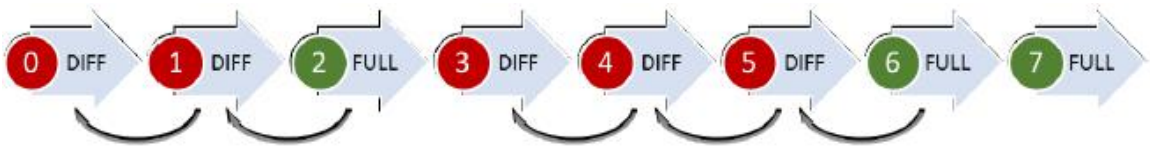
- *Storage layer* - vrstva obsahující perzistentní úložiště na data a metada a dočasné úložiště pro výměnu zpráv mezi službami
  - *Azure SQL* - perzistentní databázové úložiště uchovávající metadata o souborech a uživatelích
  - *Azure BLOB* - perzistentní úložiště uchovávající data souborů
  - *Azure Queues* - dočasné úložiště sloužící pro výměnu zpráv mezi službami



Obrázek 2.2: Architektura systému MsBox. Zdroj [13]

2.1.2 Práce se soubory

Klienti nestahují celé soubory ze služby, ale stahují pouze metadata jako je název souboru, identifikátor, velikost, typ, čas vytvoření a podobně. K reálnému stažení dojde až na vyžádání uživatelem. Tento systém také podporuje verzování souborů. Uživatel si může prohlížet historii verzí a v případě potřeby se může vrátit ke starší verzi souboru. V cloudovém úložišti ale nejsou uchovány všechny takovéto verze v plném datovém rozsahu. Uloženy plné verze souboru jsou pouze 2 poslední a následně každá 4. verze jak je znázorněno na obrázku 2.3. Z obrázku 2.3 je patrné, že pro získání 3. verze souboru je tedy nutno aplikovat posloupnost rozdílových souborů (patchů) od verze 6 až po verzi 3. Tento proces je výkonově, časově i paměťově náročný [13].



Obrázek 2.3: Uchovávání souborů v systému MsBox. Čísla označují verzi souboru. Zeleně jsou znázorněny plné verze souboru a červeně jsou znázorněny soubory, které lze získat po aplikaci patche na předchozí verzi. Šipky značí verzi souboru, ze které lze dopočítat plnou verzi. Zdroj [13]

2.2 Analýza požadavků

Analyzováním systému MsBox, zadání této práce a možných dalších požadavků jsme dospěli k následujícím funkčním a nefunkčním požadavkům.

- Funkční požadavky - určují jaké chování bude systém nabízet
- Nefunkční požadavky - specifikují vlastnosti nebo omezující podmínky systému

2.2.1 Funkční požadavky

- Systém bude umožňovat prostřednictvím klienta nahrát celý soubor na server.
- Systém bude umožňovat prostřednictvím klienta nahrát změny vůči jinému souboru na server.
- Systém bude umožňovat prostřednictvím klienta stáhnout celý soubor ze serveru.
- Systém bude umožňovat prostřednictvím klienta stáhnout změny vůči jinému souboru ze serveru.
- Systém bude umožňovat prostřednictvím klienta posílání souboru po menších částech.

2.2.2 Nefunkční požadavky

- Serverová část bude implementována v jazyce C# 5.0 s využitím frameworku .NET 4.5.
- Serverová část bude schopna využívat databázový systém Microsoft SQL Server či Azure SQL.
- Serverová část bude schopna využívat Azure Storage či lokální souborový systém pro ukládání souborů.
- Serverová část bude snadno konfigurovatelná.
- Serverová část bude snadno škálovatelná do šířky (navyšování počtu instancí).
- Serverová část bude schopna nasazení do cloudu Microsoft Azure.
- Klientská část nebude používat databázový systém.
- Klientská část bude implementována ve formě programových samostatných knihoven.
- Použitý software pro řešení nesmí být vystaven pod copyleftovou (či jinak nakažlivou) licencí.

2.3 Popis zkoumaných existujících algortimů

V této části jsou popsány jednotlivé algoritmy, které byly zvažovány při výběru vhodného algoritmu pro synchronizaci. Z podstaty ukládání starších verzí souborů v neúplném formátu a jejich náročného získání v plné verzi jak bylo popsáno v části 2.1.2, nemohlo být využito algoritmů, které by na serverové straně lokálně vyhodnotily rozdíl mezi současnou verzí souboru uživatele vůči aktuální verzí souboru na serveru a tyto změny vrátily klientovi.

2.3.1 Rsync

Rsync je dnes často používaný algoritmus pro synchronizaci souborů či celých souborových systémů[9]. Pro zlepšení efektivity výpočtu používá dva typy hashů a hashovacích funkcí, a to silné MD5 (128 bitů) a slabé Rolling Hash (32 bitů). Slabá hashovací funkce musí splňovat požadavek, že výpočet hashe následujícího bloku se dá snadno vypočítat z hashe předchozího bloku. Takovouto hashovací funkcí je například Adler-32, která je v upravené verzi použita v implementaci Rsyncu na systému UNIX a vypadá následovně[7]:

$$a(k,l) = (\sum_{i=k}^{l} X_i) \; mod \; M$$

$$b(k,l) = (\sum_{i=k}^{l} (l-i+1)X_i) \; mod \; M$$

$$s(k,l) = a(k,l) + M * b(k,l)$$

,kde X_i je hodnota bytu na pozici i a $s(k,l)$ je rolling hash bloku bytů $X_k, \ldots, X_l$.

Po posunutí hashovacího okna (posunutí bloku) o jednu pozici pak vypočítáme nový hash takto:

$$a(k+1, l+1) = (a(k,l) - X_k + X_{l+1}) \; mod \; M$$

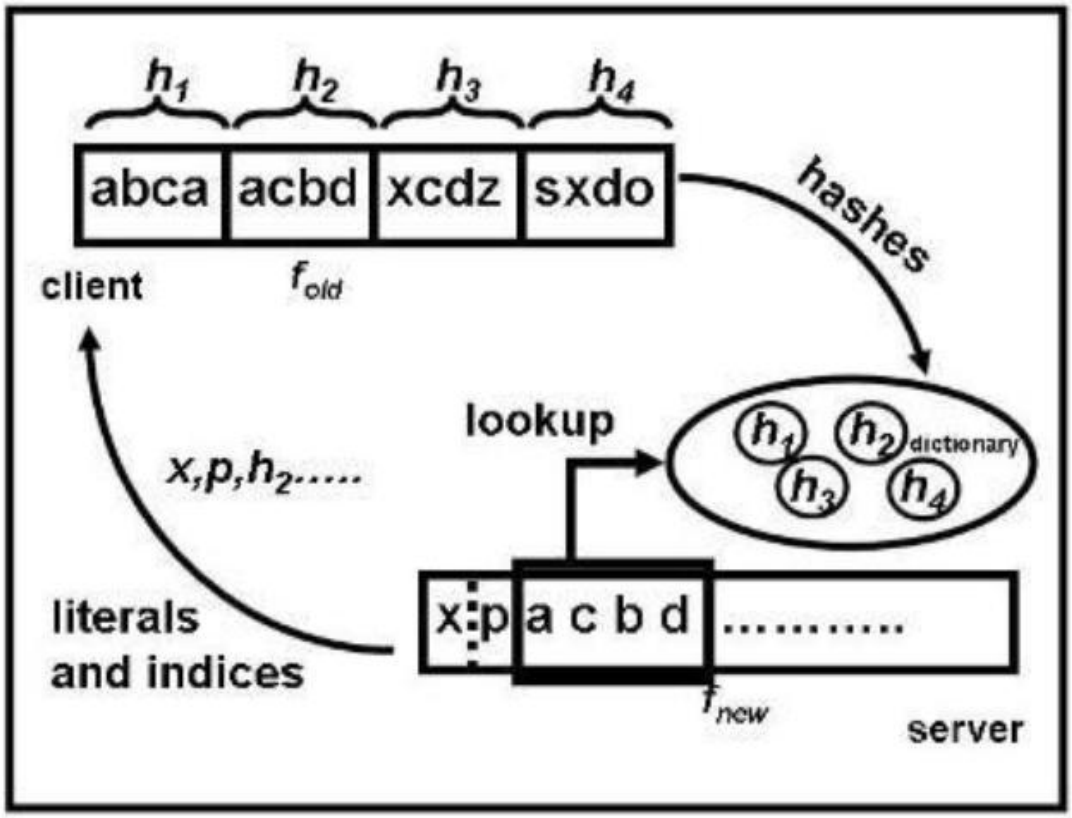
$$b(k+1, l+1) = (b(k,l) - (l-k+1)X_k + a(k+1, l+1)) \; mod \; M$$

$$s(k+1, l+1) = a(k+1, l+1) + M * b(k+1, l+1)$$

Strana, která má novější verzi souboru F_{new} vystupuje jako server (nazývá se Sender) a strana která má starší verzi souboru F_{old} vystupuje jako klient (nazývá se Reciever). Díky efektivitě výpočtu slabého hashe dochází na straně serveru k náročnému výpočtu silného hashe pouze v případě shody slabých hashů. Nástin, jak algoritmus funguje, je znázorněn na obrázku 2.4.

Rsync algoritmus se skládá z těchto kroků[8]:

1. Klient rozdělí soubor soubor F_{old} na nepřekrývající se bloky o pevné délce K bytů. Poslední blok může být kratší.
2. Pro každý takovýto blok klient vypočítá silný a slabý hash
3. Klient pošle všechny takto napočítané hashe serveru
4. Server poté počítá proti souboru F_{new} jednotlivě slabý hash pro blok o velikosti K s libovolným offsetem (tedy pro každý možný blok o velikosti K a ne pouze násobky), který vždy porovná se slabými hashi ze serveru a pokud najde shodu, vypočítá silný hash a ověří i shodu silných hashů. Pokud se i silné hashe shodují, pak si server uloží k odeslání klientovy nové i staré indexy bloku, aby mohl klient blok zkopírovat lokálně. Když se silné hashe neshodují, jednalo se pouze o "falešný poplach" a hledání shody pokračuje dále (pouze nastala kolize slabých hashů). Pokud pro zkoumaný blok nenastala shoda, pak se hashovací okno posune o jednu pozici, byte na staré pozici se označí k odeslání, vypočítá se slabý hash nového bloku a celá procedura se takto opakuje až do konce souboru.
5. Klient podle získaných dat od serveru sestaví nový soubor, který je identický s F_{new}



Obrázek 2.4: Nástin funkce Rsync algoritmu. Klient pošle množinu hashů a server odpoví streamem neshodujících se dat a indexů shodných bloků. Zdroj [22]

2.3.2 Remote Differential Compression

Remote Differential Compression (dále jen RDC) je protokol, který byl navržen jako efektivnější alternativa k Rsyncu a LFBS (Low-bandwidth Network FileSystem) a je určen k synchronizaci souborů přes síť s malou šířkou pásma[9]. Vylepšená byla především identifikace změněných souborů (zatímco Rsync používá time-stamp souboru či jeho 128-bitový hash, RDC používá pouze 96 bitů), určení podobnosti souborů (odhadovaná editační vzdálenost) pomocí uložených metadat a určování dynamické velikosti bloků na základě lokálního maxima. Přijaté hashe souboru F_{new} ze serveru klient porovnává nejen s hashi souboru F_{old}, ale i s hashi všechn souborů $FC_1, FC_2, \ldots, FC_n$, které byly souboru F_{new} dostatečně podobné (na základě odhadu editační vzdálenosti). Podobně jako u Rsyncu obě strany spojení mohou vystupovat jako klient či server.

RDC algoritmus pro jeden soubor se skládá z těchto kroků[9]:

1. Klient rozdělí všechny indetifikované soubory $FC_1, FC_2, \ldots, FC_n$ do bloků a vypočítá hash $SigC_{ik}$ pro každý blok k každého souboru FC_i.

2. Server rozdělí soubor FS do bloků a vypočítá jejich hashe $SigS_j$.

3. Server pošle seřazený list hashů a délky bloků $((SigS_1, LenS_1) \dots (SigS_n, LenS_n))$ klientovi.

4. Když klient obdrží list, tak porovná obdržené hashe s množinou vlastních hashů $(SigC_{11}, \dots, SigC_{1m}, \dots, SigC_{n1}, \dots, SigC_{nm})$. Klient si ukládá každý hash serveru, který se neshodoval

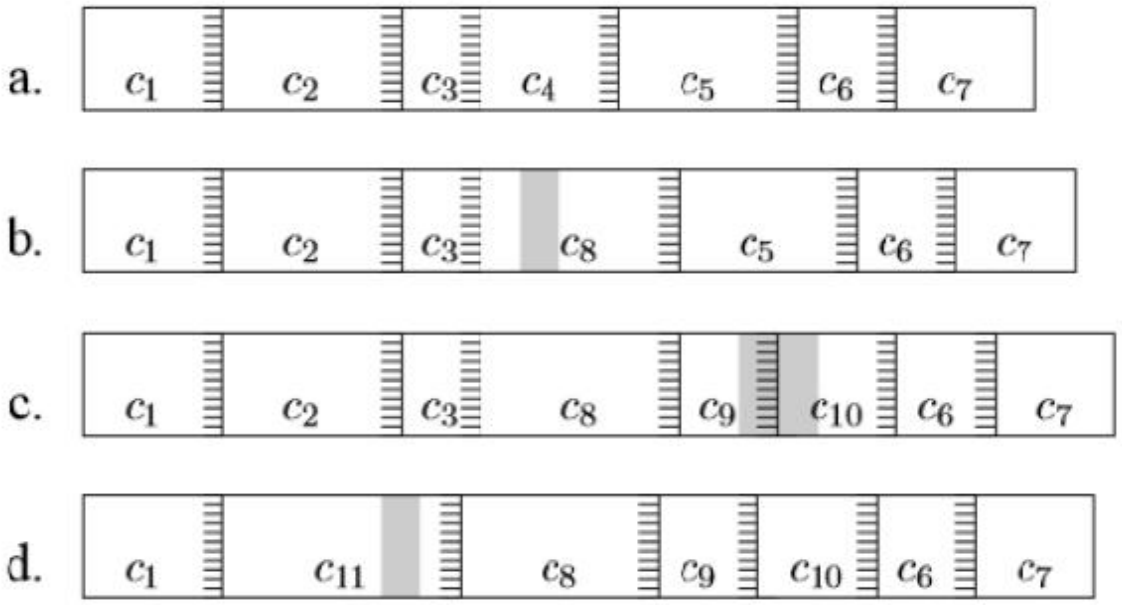
5. Klient požádá server o data všech hashů u kterých nenašel shodu

6. Server pošle data klientovi.

7. Klient z dat od serveru a shodných bloků v souborech $FC_1, FC_2, \dots, FC_n$ sestaví soubor FS

Jak již bylo zmíněno, velikost bloků není statická, ale dynamická a to na základě obsahu bloků. Podobně jako u Rsyncu se používá rolling hash hashovací funkce (založena na Rabin algoritmu [9]), pomocí které sumarizuje obsah hashovacího okna skládajícího se z posledních w bytů (typicky $w = 12 \dots 64$) do jedné 4-bytové hodnoty v. S přidáním každého nového bytu do hashovacího okna vypočítá RDC nový hash a porovná jeho hodnotu se starým hashem. V paměti si udržuje pouze maximum V_{max} ze všech doposud spočítaných hodnot hashů v a pozici bytu B_p při jeho výpočtu. V případě, že pozice V_{max} v hashovacím okně je $h + B_p$, kde h je počet bytů od začátku hashovacího okna, a současně spočítaná hodnota hashe $V_{act} < V_{max}$ a pozice V_{act} je $B_p + h$, pak bylo nalezeno lokální maximum V_{max} na pozici B_p. Pozice B_p je tedy koncem předchozího bloku (tzv. cut-point). Po nalezení cut-pointu se celá procedura nalezení lokálního maxima opakuje a takto se celý soubor rozdělí na rozdílně velké bloky. Při neoptimálním obsahu dat se může stát, že budou velikosti bloků příliš malé, nebo naopak velké. Z těchto důvodů je možné specifikovat hodnoty h_{min} a h_{max}, které zaručí, že velikost bloku je zdola i shora omezena.

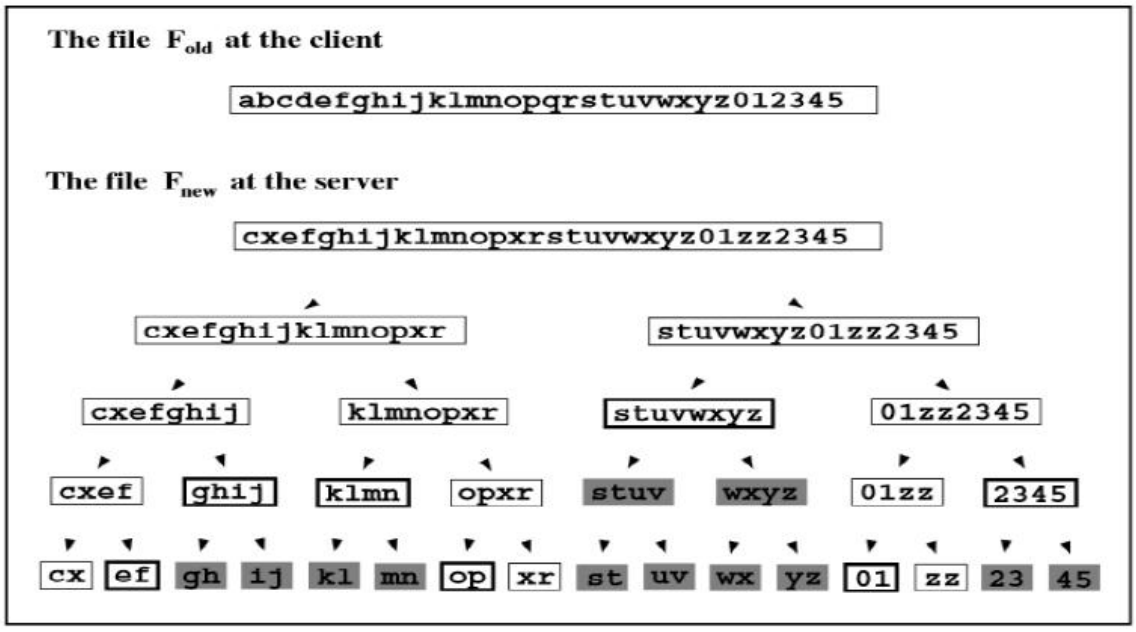
Na obrázku 2.5 jsou znázorněny případy změny obsahu souboru a jejich vliv na synchronizaci. V případě a je znázorněno rozdělení souboru na bloky s rozdílnou velikostí. V případě b byl přidán obsah do bloku c_4, který zvětšil jeho velikost. Tento změněný blok je označen jako c_8 a pouze on bude přenesen. V případě c byl do bloku c_5 přidán obsah, který zapříčinil rozpad bloku na bloky c_9 a c_{10}. Pouze tyto bloky budou přeneseny. Případ d ukazuje přidání obsahu, který sloučí bloky c_2 a c_3 (byl odstraněn cut-point) do bloku c_{11}. Celý blok c_{11} bude přenesen.



Obrázek 2.5: Vliv změny obsahu na okolní bloky v algoritmu RDC. Zdroj [6]

2.3.3 Erasure Codes

Tento algoritmus je postaven na ověření správnosti bloků či jejich podbloků a výpočtu hashů pro podbloky na úrovni $n+1$ z hashů na úrovni n a $2k$ tzv. erasure hashů, kde k je maximální počet hashů, které na dané úrovni nenajdou shodu (je tedy třeba znát maximální editační vzdálenost)[22]. Mějme jednoduchý algoritmus, kde se nejdříve spočítají hashe bloků o délce B_{max} a poté se počítají vždy hashe pro bloky o poloviční velikosti až po velikost bloku B_{min}. Na začátku tedy server rozdělí soubor do bloků o velikosti B_{max}, spočítá hash pro každý takový blok a tyto hashe pošle klientovi. Klient výpočítá hashe bloků o velikosti B_{max} s libovolným offsetem (posunutím) a pokusí se najít shodu s hashi obdrženými ze serveru. Poté klient pošle serveru vektor, ve kterém '1' znamená, že k přijatému hashi našel shodu a '0' znamená že pro daný hash bloku shoda nebyla nalezena. Takto server zjistí, které hashe byly rozpoznány (a klient má tedy k dispozici tyto data lokálně) a naopak které hashe se neshodovaly a klient tyto data nebo jejich podmnožinu lokálně zkopírovat nemůže. V dalším kroku server všechny bloky, jejichž hashe neměly shodu s klientskými, rozdělí na dvě poloviny a vypočítá hash každé z nich a tyto hashe menších bloků pošle opět klientovi. Klient provede porovnání s hashi všech polovin bloků u nichž předtím nenalezl shodu a opět pošle serveru vektor, který se skládá z '0' a '1'. Jakmile je délka bloků rovna B_{min}, tak server pošle klientovi data všech bloků či podbloků, u nichž nebyla nalezena shoda. Postup dělení bloků do podbloků a ověřování shody je znázorněno na obrázku 2.6.



Obrázek 2.6: Znázornění dělení bloků na podbloky u algoritmu Erasure Codes. Tučným rámečkem jsou znázorněny podbloky, u nichž nastala shoda hashe s podblokem hashe klienta a v dalších výpočtem nefigurují (neprovedené operace poslání a ověření hashe jsou znázorněny šedou výplní). Zdroj [22]

Algoritmus Erasure Codes funguje podobně jako výše popsaný jednoduchý algoritmus, ale namísto aby server posílal hashe pro každou úroveň dělení bloků, tak server spočítá pro každou úroveň $2k$ erasure hasů, ze kterých lze s pomocí hashů které se shodovaly na dané úrovni opravit k chybných hashů (hashe které se neshodovaly se zde považují za chybné). Kroky erasure code algoritmu vypadají takto [22]:

1. Server rozdělí soubor F_{new} rekurzivně do bloků o velikost B_{max} až B_{min} a pro každou úroveň spočítá hashe všech bloků
2. Server aplikuje výpočet erasure code hashů na každou úroveň kromě nejvyšší a spočítá $2k$ erasure hashů pro každou úroveň.
3. Server pošle všechny hashe nejvyšší úrovně a $2k$ erasure hasů pro každou úroveň klientovi
4. Klient poté spočítá hashe nejvyšší úrovně ze souboru F_{old} a pokusí se najít shodu s hashi nejvyšší úrovně ze serveru. Na další úrovni jsou spočítány běžným způsobem hashe bloků, jejich předci měli shodu s hashi ze serveru a $2k$ erasure hashů je použito pro získání nejvýše $2k$ hashů bloků, jejichž předci neměli shodu s hashi ze serveru
5. Na nejnižší úrovni s bloky o délce B_{min} se přepokládá, že hash je obsahem bloku (tedy vlastní data) a tak je klient schopen sestavit soubor F_{new}

2.3.4 Výběr algoritmu

RDC algoritmus je velmi komplexní a také náročný na implementaci. Dle zdroje [9] jsou kritické části implementovány pro zlepšení výkonu v assembleru a i tak se velmi blíží výsledkům Rsyncu při dostatečně podobných souborech. Při souborech více odlišných má naopak Rsync výpočetní výsledky lepší a podobné výsledky síťového přenosu[9]. Při použití algoritmu Erasure Codes musíme znát maximální editační vzdálenost mezi soubory, což je proti algoritmu Rsync dosti limitující. Algoritmy stejného typu jako Rsync také nabízejí dobrou rovnováhu mezi výpočetní zátěží a ušetřeným datovým tokem a zároveň mají menší požadavky na paměť[10]. Vybraným algoritmem pro realizaci je tedy algoritmus na bázi Rsync, který byl už jednou použit v implementaci pro projekt MsBox jak je popsáno v diplomové práci Daniela Kavana[11].

2.4 Existující řešení

V současné době je dostupných mnoho implementací Rsyncu.[1] Většinou se ale jedná o wrappery nad samotnou referenční implementací Rsyncu, které jsou (a z podstaty licence referenční implementace musí být) vystaveny pod copyleftovou licencí GNU GPL (General Public License). Protože jedním z nefukčních požadavků je vyvárování se použití takovýchto licencí, jsou tyto knihovny pro tento projekt nevhodné.

2.5 Microsoft Azure

Microsoft Azure (dříve Windows Azure) je flexibilní cloudová platforma a infrastruktura, která umožňuje vytvářet, nasazovat, škálovat a spravovat aplikace a služby v rámci globální sítě datacenter spravovaných společností Microsoft. Nabízí PaaS (Platform as a Service) i IaaS (Infrastructure as a Service) služby a podporuje širokou škálu programovacích jazyků, frameworků a nástrojů[17]. Tato platforma je založena na virtualizaci na systémech Windows Server podporujících virtualizaci prostřednictvím Microsoft Azure Hypervisor. O rozložení zátěže, škálování a spolehlivost instancí se stará Microsoft Azure Fabric Controller[17].

2.6 Analýza platforem

2.6.1 Windows desktop

Mezi platformu Windows Desktop se řadí vyspělé operační systémy firmy Microsoft, jako jsou například Windows 7 nebo Windows 8. Tyto operační systémy nemají žádná specifická omezení. Lze na nich využít jak programy napsané s podporou frameworku .NET tak i programy napsané pro platformu Java.

[1] Například java-rsync dostupný na adrese <https://code.google.com/p/java-rsync/> či rsync.net dostupný na adrese <http://github.com/MatthewSteeples/rsync.net>

2.6.2 Windows Phone 8

Windows Phone 8 je třetí generací operačního systému pro mobilní zařízení od společnosti Microsoft[20]. Jako uživatelské rozhraní je zde použito Modern UI (dříve známé jako Metro), které je dostupné i na platformě Windows 8. Jádro Windows Phone 8 je postaveno na jádře Windows NT a sdílí mnoho komponent s Windows 8, což umožňuje lepší přenositelnost mezi těmito dvěma platformami[20]. V popředí běží pouze jedna aplikace, zatímco ostatním aplikacím které neběží v popředí je umožněno využít tzv. background agenty[14]. Pro běh vlastního programu na pozadí lze využít Scheduled Tasks, které se dělí na dva základní typy:

- Periodic Task - je spouštěn pravidelně každých 30 minut na krátký časový interval. Typickým scenářem pro využití je posílání geolokace zařízení nebo synchronizace malého množství dat.
- Resource Intensive Task - umožňuje běh programu po dobu 10 minut za splnění jistých kritérií, mezi které patří připojení telefonu k internetu pomocí Wi-Fi, baterie telofonu musí být nabitá alespoň na 90 % , telefon musí být nabíjen, maximální využití paměti programem nepřekročí stanovený limit určený dle maximální dostupné paměti telefonu a to, že telefon musí být zamknut.

Ve Windows Phone 8 lze pro posílání souborů na pozadí také využít již dostupnou službu BackgroundTrasnferService, která k realizaci přenosu využíva metod GET a POST protokolu HTTP (Hypertext Transfer Protocol). Ikdyž pro tuto službu platí také jistá omezení, jsou tato omezení určována v závislosti na velikosti souboru a služba umožňuje uživateli při posílání souboru dále používat mobilní telefon.[15]

2.6.3 Java

Java platforma je množina softwarových produktů a specifikací, které poskytují systém pro vývoj a nasazení programu na mnoha různých platformách[18]. Primárním implementačním jazykem je Java, ale lze použít i jiné jazyky, pokud mají dostupný překladač. Implementace v daném jazyku se pomocí překladače přeloží do tzv. bytecodu, což je instrukční sada pro Java Virtual Machine (dále také JVM). Tato instrukční sada není závislá na typu procesoru ani operačním systému, ale právě na JVM. Java Virtual Machine, neboli virtuální stroj, je pak pro každou plaformu včetně překladače implementován samostatně (implementace JVM je tedy platformově závislá a různé operační systémy mohou mít různé implementace[18]). JVM se poté stará o překlad programu v bytecodu do platformově závislé instrukční sady pomocí Just In Time (JIT) překladače a jeho následný běh. Mezi hlavní výhody Javy patří platformová nezávislost, dobrá podpora vícevláknovosti, bezpečnosti a pokročilý garbage collector, který se stará o efektivní uvolňování paměti. Nevýhodami je pomalější spouštění programů z důvodu překladu bytecodu a podpora pouze znaménkových (signed) datových typů[21].

2.6.4 Android

Android je celosvětově nejpopulárnější mobilní platforma[12]. Je založena na linuxovém jádře a jako běhové prostředí pro aplikace je použito Dalvik Virtual Machine (dále jen DVM).

DVM je virtuální stroj, který využívá just-in-time kompilaci pro spuštění Dalvik Executable formátu, který býva získán překladem Java bytecodu[16]. Díky tomu také platforma Android umožňuje využití většiny běžných Java knihoven, které jsou obsaženy v Android SDK a také kódu napsaného v programovacím jazyce Java. Protože však Android SDK obsahuje pouze podmnožinu knihovních funkcí JDK, není možné například používat knihovnu javax.ws.* a většinu funkcí z knihovny javax.xml.*. V současné době je nejpoužívanější verze 4.1.x Jelly Bean jak je znázorněno v tabulce 2.1.

Verze	Název	API	Podíl (v %)
2.2	Froyo	8	1.0
2.3.3-2.3.7	Gingerbread	10	16.2
3.2	Honeycomb	13	0.1
4.0.3-4.0.4	Ice Cream Sandwich	15	13.4
4.1.x	Jelly Bean	16	33.5
4.2.x	Jelly Bean	17	18.8
4.3	Jelly Bean	18	8.5
4.4	KitKat	19	8.5

Tabulka 2.1: Zastoupení verzí platformy Android. Zdroj [12]

2.7 Řešení vhodná pro implementaci

2.7.1 Castle Windsor

Castle Windsor je Inversion of Control (IoC) container pro platformu .NET a Silverlight. Je součástí open source projektu Castle, který je vystaven pod licencí Apache License 2.0. Díky IoC containeru je možné spravovat životní cyklus, konfiguraci a závislosti instancí jednotlivých tříd[5]. V konfiguračním souboru lze například specifikovat které třídy má IoC container spravovat a definovat hodnoty parametrů konstruktoru na základě jmenné konvence. Toto řešení umožňuje snadnou konfigurovatelnost a znovupoužitelnost komponent a je tedy vhodné k použití na serverové straně. Příklad konfigurace je znázorněn v ukázce 2.1.

Listing 2.1: Ukázka konfigurace komponenty pro Castle Windsor

```
<component id="FileSystemBlobDao"
        service="CP.Server.Core.IFileSystemBlobDao,CP.Server.Core">
        type="CP.Server.Core.FileSystemBlobDao,CP.Server.Core">
        <parameters>
                <path>D:\tmp\</path>
        </parameters>
</component>
```

2.7.2 Apache log4net

Framework Apache log4net je port frameworku Apache log4j pro platformu Microsoft .NET, který se zaměřuje na logování událostí. Tento framework je vysoce konfigurovatelný pomocí XML konfigurace, umožnuje nastavit různé výstupy pro jednotlivé typy zpráv, má hierarchickou strukturu logování (každá komponenta má vlastní logger) a je také velmi nenáročný na výpočetní zdroje[3]. Tento framework je vystaven pod licencí Apache License 2.0.

2.7.3 NHibernate

NHibernate je objektově relační mapovací framework pro platformu Microsoft .NET, který usnadňuje práci s mapováním objektů na databázové entity a opačně. Tento framework je portem jádra frameworku Hibernate, které je implementováno pro platformu Java[4]. S pomocí mapovacích souborů (XML formát popisující mapovaní entit na třídy, jejich vztahy a vlastnosti) automaticky generuje SQL příkazy pro práci s databází. NHibernate je vystaven pod licencí LGPL-2.1.

2.7.4 KSOAP2

KSOAP2 je minimalistická knihovna pro platformu Android zaměřující se na komunikaci pomocí SOAP protokolu s důrazem na efektivnost a kompatibilitu s různými SOAP enginy[2]. Pro práci se SOAP zprávami používá XML parser kXML či XmlPullParser, který by měl být dostupný pro všechna zařízení s platformou Android. Protože KSOAP2 nepoužívá pro serializaci/deserializaci objektů komplexního typu reflexi, musejí serializovatelné objekty komplexního typu implementovat rozhraní KVMSerializable a být zaregistrovány u komponenty, která se stará o mapování při serializaci/deserializaci. Tato knihovna je vystavena pod MIT licencí.

2.7.5 Apache CXF

Apache CXF je open source framework, který se zaměřuje na generování komunikačního frontendu. Umožňuje generovat komunikační rozhraní jak bottom-up, tak i top-down přístupem. Díky tomu je možné použít WSDL soubour popisující rozhraní služby k vygenerování tříd, které se pak s pomocí knihoven z frameworku Apache CXF serializují do SOAP zpráv či naopak se SOAP zprávy deserializují na objekty takto vygenerovaných tříd. Framework se zaměřuje především na efektivnost, jednoduchost použití a rozšiřitelnost[1]. Podporuje různé typy transportních protokolů (HTTP, TCP) a binding protokolů (SOAP, REST/HTTP). Apache CXF je vystaven pod licencí Apache License 2.0.

2.7.6 Komponenta systému MsBox pro ukládání dat do Azure Storage

Pro ukládání dat do cloudového úložiště Azure Storage může být s výhodou použita již existující komponenta systému MsBox. Tato komponenta používá pro specifikaci úložiště konfiguraci obsaženou v konfiguračním souboru IoC containeru Windsor Castle. Komponenta se skládá z knihoven *Jewellery.dll*, *NGMsBox.Azure.dll*, *NGMsBox.Configuration.dll*, *NGMsBox.DataEntities.dll* a *NGMsBox.Shared.dll*, které byly pro účely této práce poskytnuty panem Ing. Martinem Mudrou za následujících podmínek:

- Pro užití v rámci diplomové práce pana Bc. Jana Minaříka byl poskytnut k výše jmenovaným knihovnám zdrojový kód i binární podoba.
- Pro účely diplomové práce pana Bc. Jana Minaříka je možné knihovny použít bez omezení, za předpokladu, že do diplomové práce přiloží tento text do části „analýza“.
- Další redistribuce pro jiné účely je možná pouze po domluvě s autorem.
- Použití pro jiné účely je možné pouze po domluvě s autorem.

Kapitola 3

Návrh řešení

3.1 Použitý algoritmus

Algoritmus Rsync byl převzat v základní podobě a pro systém MsBox byl mírně pozměněn. V původním algoritmu vystupuje jako server vždy strana, která vlastní novější verzi souboru F_{new}. Z důvodu jednotnosti a podstaty systému bylo rozhodnuto, že služba bude vždy vystupovat jako server a připojení klienti jako klient v daném algoritmu bez ohledu na to, kdo vlastní novější verzi souboru. Toto rozhodnutí přináší lepší přehlednost (protože typicky inicializaci spojení provádí klient) a řadu výhod, které jsou popsány níže.

3.1.1 Delegace výpočetní zátěže

Protože jsme jednoznačně určili role serveru a klienta v algoritmu bez závislosti na souboru, můžeme snadno delegovat výpočetní zátěž na klienty. V případě algoritmu Rsync klient rodělí soubor do nepřekrývajících se bloků o délce k a spočíta jejich hashe, zatímco server rolluje a počítá hashe všech možných bloků o délce k. V případě připojení většího počtu klientů by byla služba vystavena velké výpočetní zátěži a i její paměťové nároky by byly vysoké (musela by si držet všechny hashe každého klienta v paměti). Jednoduchou záměnou odpovědnosti práce ale docílíme toho, že server bude počítat pouze hashe nepřekrývajících se bloků o pevné délce k, zatímco klient bude počítat hashe všech možných bloků délky k. Touto změnou nebude server pod tak velkou výpočetní zátěží a bude moci obsloužit více klientů.

3.1.2 Předpočítání hashů

V systému MsBox mohou různí uživatelé provádět změny nad stejným souborem. V případě změny souboru sdíleného mezi více uživateli, a následné synchronizaci, by se musely všechny hashe pro tento soubor počítat znovu pro každého uživatele se kterým je tento soubor sdílen. Počítání stejných hashů několikrát je ale zbytečné. Protože jsme určili, že server bude vždy počítat hashe pouze nepřekrývajících se bloků, lze tyto hashe pro daný soubor spočítat pouze jednou, uložit je a v případě dalšího požadavku na synchronizaci je opět použít. Takto se ušetří nejen výpočetní výkon potřebný k výpočtu hashů, ale také se sníží čas odpovědi serveru, protože klient nebude muset čekat na jejich výpočet.

3.1.3 Vyjednávání parametrů

Aby klient mohl využít výhody přepočítaných hashů, musí si se serverem vyměnit informace o tom, s jakými parametry jsou předpočítané hashe dostupné. Klient se tedy může dotázat serveru na množinu nastavení přenosu, které určují jaké algoritmy, velikost bloku a další parametry byly použity při výpočtu hashů. Poté klient pošle na server vybrané nastavení pro přenos. Server si toto nastavení uloží a při dalších akcích pro tento přenos bude postupovat podle něj. Dojednávání parametrů je detailněji popsáno v sekci 3.7.5.

3.2 Konfigurovatelnost

V rámci konfigurovatelnosti musíme rozlišovat stranu klienta a serveru, protože každá strana má jiné možnosti a potřeby na konfiguraci.

3.2.1 Server

Na serveru budeme moci pomocí konfigurace rozdělit soubory na tři typy podle velikosti a to na malé, střední a velké soubory. U každého typu specifikujeme maximální možnou velikost daného typu souboru. U každého typu souboru lze dále pomocí konfigurace určit, kdy se počítají hashe nepřekrývajících se bloků a jaké jsou parametry algoritmu pro výpočet. Tyto hashe je možné počítat při sestavování souboru, až po jeho sestavení nebo předpočítávání plně zamezit. V rámci konfigurace je také možné nastavit, zda se k ukládání dat má používat úložiště typu Azure BLOB, nebo souborový systém dostupný serveru. V případě souborového systému je pak umožněno specifikovat přímo adresář, kam se data mají ukládat. Toto řešení může být vhodné, pokud uživatel bude chtít nasadit server na svém hardwaru. Pokud bude ale uživatel vyžadovat škálovatelnost do šířky (tedy že může nasadit více instancí serveru), musí využívat souborový systém namapovaného síťově dostupného datového úložiště, aby měly k datům přístup všechny instance serveru.

3.2.2 Klienti

Protože konfiguraci parametrů algoritmu pro synchronizaci určuje serverová strana, mohou mít klienti v rámci konfigurace pouze vlastnosti, které přímo neovlivňují daný algoritmus. Mezi tyto vlastnosti patří možnost nastavit velikost přenášených částí souboru a počet vláken, které budou přenos realizovat.

3.3 Vícevláknové zpracování

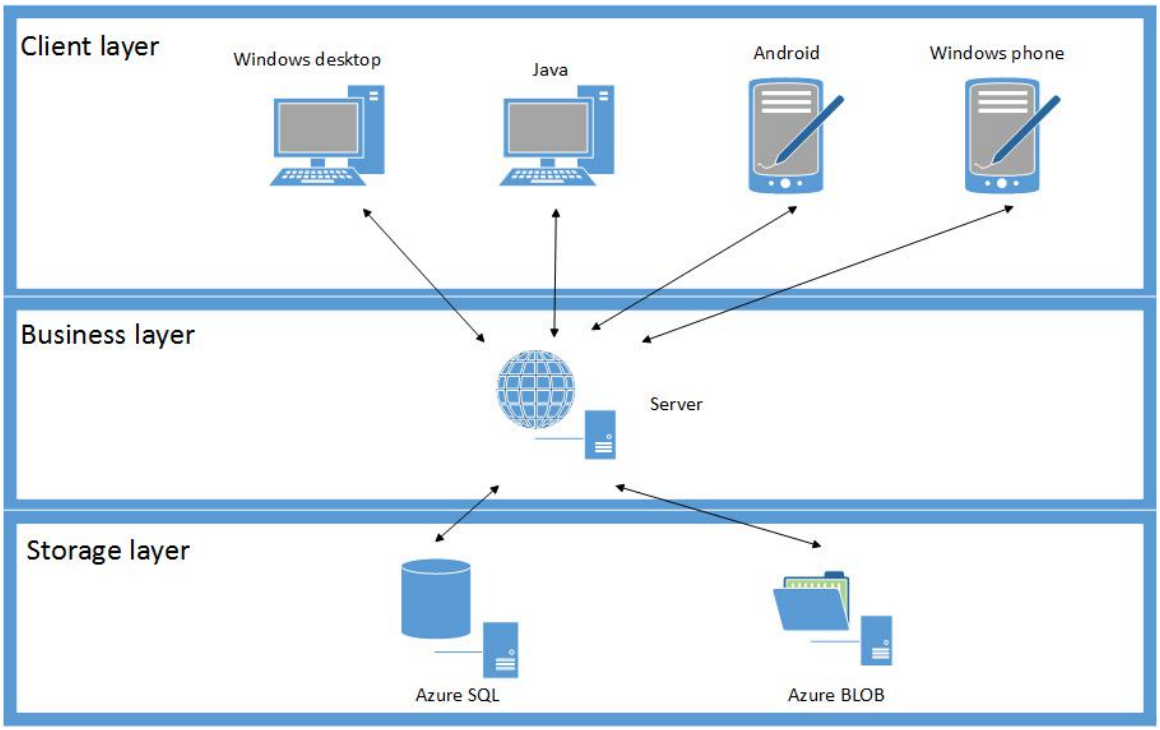
Data, která je třeba přenést, jsou na klientovi rozdělena a přenášena po částech. V případě dočasného výpadku připojení tedy selže pouze posílání právě přenášených částí, ale části které do té doby server obdržel se již znovu přenášet nemusí. V případě opětovného připojení pak klient dopošle chybějící části a není nutné znovu posílat celý objem dat. Protože v cloudovém prostředí může být některá z instancí serveru více vytížena, je dobré využít posílání dat pomocí více vláken. Každé posílající vlákno vytvoří nový požadavek, který může být

obsloužen méně vytíženou instancí a proto by se mohla zvýšit i rychlost samotného přenosu dat. Další možností vícevláknového přístupu je výpočet silného hashe bloku v samostatném vlákně na straně klienta při hledání shodných částí souborů. Vlákno počítající slabé hashe při shodě slabých hashů přidá požadavek na výpočet silného hashe do fronty. Tuto frontu pak může zpracovávat jiné vlákno (či více vláken), které pak vypočítá silný hash a rozhodne, zda jsou části souborů opravdu shodné, nebo se jednalo pouze o kolizi slabých hashů.

3.4 Architektura

Jak již bylo zmíněno v analytické části, systém MsBox využívá třívrstvé architektury, která efektivně odděluje zodpovědnost a funkcionalitu jednotlivých částí systému. Tato třívrstvá architekura je dobře škálovatelná a je vhodná pro řešení i tohoto projektu. Návrh architektury je znázorněn na obrázku 3.1 a dělí se na tyto vrstvy:

- Client layer - tato vrstva obsahuje funkcionalitu klientů
- Business layer - vrstva obsahující logiku serverové strany
- Storage layer - vrstva starající se o ukládání dat



Obrázek 3.1: Navržená architektura

3.4.1 Client layer

Klientská vrstva bude implementována jako sada samostatných knihoven pro platformy Windows Desktop, Windows Phone, Java a Android, které budou nabízet funkcionalitu potřebnou k realizaci synchronizace. Z důvodu přenositelnosti kódu mezi platformami a jeho údržbě by klientská část neměla být příliš složitá. V případě použití platformově závislé funkcionality je nutno zaručit existenci podobné funkcionality i na ostatních platformách nebo poskytnout její snadnou implementaci. Takovýmto případem může být například použití třídy *FileStream* dostupné v jazyce C# na platformách Windows Desktop a Windows Phone 8 pro náhodný přístup k datům souboru. Na platformě Java i platformě Android existuje podobný ekvivalent v rámci třídy *RandomAccessFile*. Klientská knihovna by měla umožňovat:

- Stažení celého souboru ze serveru po částech
- Stažení změn mezi lokálním souborem a souborem uloženým na serveru
- Nahrání celého souboru na server po částech
- Nahrání změn na server mezi lokálním souborem a souborem na serveru

3.4.2 Business layer

Business vrstvu představuje v našem návrhu server starající se o vyřizování klientových požadavků. Podmínkou pro realizaci běhu serveru v cloudu Microsoft Azure jako škálovatelné služby je jeho bezestavovost. Na straně serveru je třeba uchovávat stav o zpracovaných (přijatých) datech, samotná data, nastavení přenosu a předpočítané hashe. K tomuto účelu využívá služby nižší vrstvy *Storage layer*. K plnému obsloužení klientů by měl server klientům poskytovat tyto funkce:

- Nastavení parametrů přenosu
- Stažení části souboru
- Nahrání části souboru
- Počítání hashů nepřekrývajících se bloků souboru
- Přemapování částí souboru, o kterých klient určil, že je má server lokálně dostupné
- Ověření dostupnosti všech dat potřebných k sestavení souboru na straně serveru
- Sestavení souboru z částí souboru obdržených od klienta (včetně přemapovaných)
- Ověření, že byl soubor v pořádku přenesen a korektně sestaven

3.4.3 Storage layer

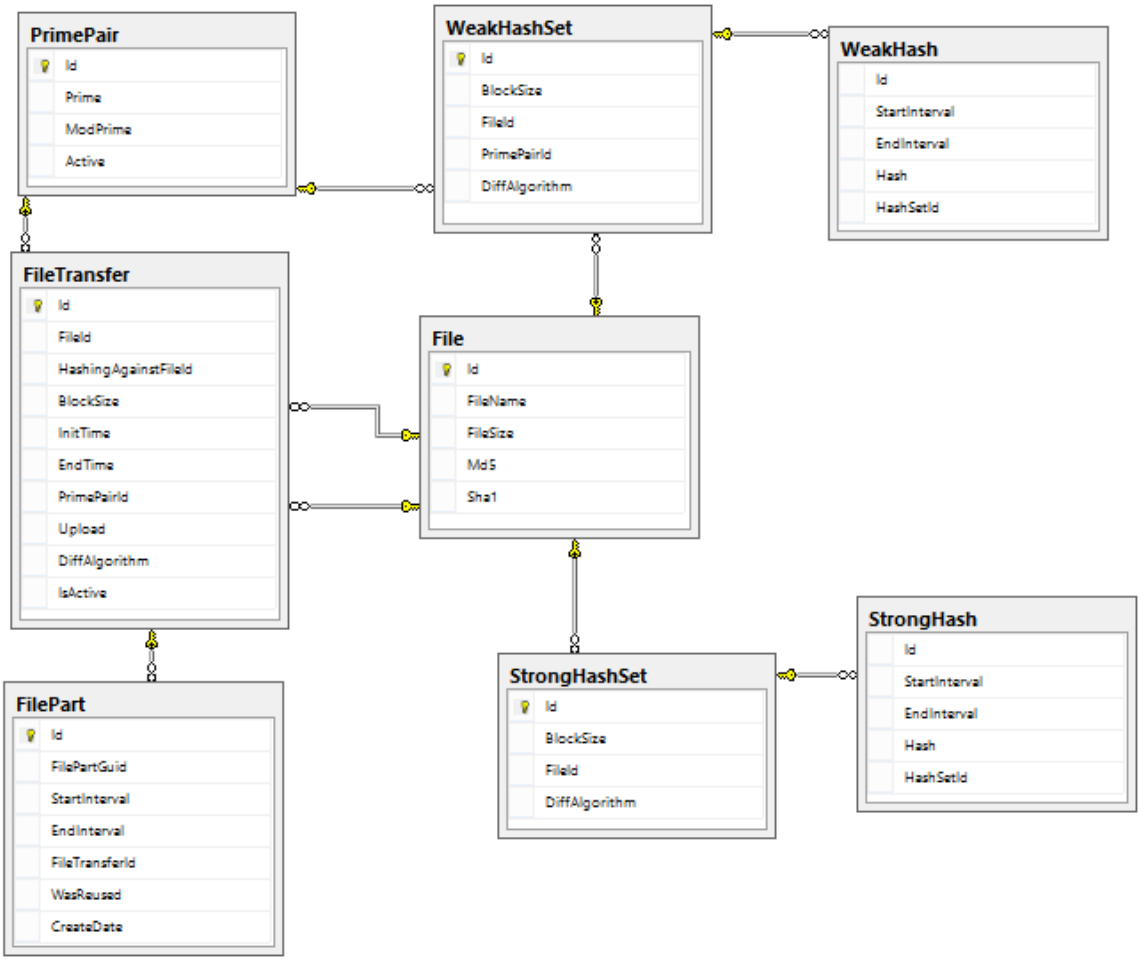
Tato vrstva se stará o ukládání dat a obsahuje dva typy úložišť. Relační databáze Azure SQL je využita pro ukládání všech informací o probíhajících přenosech, zpracovaných datech a napočítaných hashích. Pro ukládání dat většího objemu, jako jsou části souboru a sestavené soubory, je využita již existující komponenta systému MsBox (jež byla poskytnuta pro účely této práce), která tato data ukládá do úložiště Azure BLOB.

3.5 Komunikační protokol

Pro komunikaci byl zvolen protokol SOAP, který je založen na výměně zpráv ve formátu XML, a který se v současné době používá v projektu MsBox. Ve WCF frameworku má SOAP velmi dobrou podporu a také umožňuje díky WSDL (Web Services Description Language), které popisuje rozhraní služby, snadno generovat komunikační frontend. Díky velké rozšířenosti a dlouhodobé používanosti má SOAP dobrou podporu pro generování frontendu na různé platformy jako například .NET nebo Java. Mezi nevýhody patří větší nároky na přenášená data právě z důvodů použitého XML formátu (duplicita názvu u párového elementu, jmenné prostory), které by ale měly být zanedbatelné v porovnání s velikostí přenášených dat synchronizovaného souboru.

3.6 Struktura databáze

Relační databáze slouží k uchování informací o nastavení přenosu, zpracovaných souborech (nebo jejich částech) a předpočítaných hashích. Při návrhu bylo dbáno na to, aby navržená struktura databáze splňovala 3. normální formu[19]. Výsledná struktura je znázorněna na obrázku 3.2. Informace o aktivních přenosech a jejich nastavení (velikost bloků, typ přenosu, jaký soubor se přenáší, algoritmus použitý pro výpočet hashů a podobně) jsou uloženy v tabulce $FileTransfer$. V tabulce $PrimePair$ jsou uloženy kombinace prvočísel, které se používají při výpočtu slabého hashe, proto na ni mají také referenci tabulky $WeakHashSet$ a $FileTransfer$. Tabulka $File$ obsahuje informace o přenášených a také již přenesených souborech. Tabulky $WeakHashSet$ a $StrongHashSet$ obsahují souhrnné informace k množinám předpočítaných hashů, které jsou uloženy v tabulkách $WeakHash$ a $StrongHash$. Každá entita v tabulkách $WeakHash$ a $StrongHash$ mimo hodnoty hashe obsahuje i *StartInterval* a *EndInterval*, které určují pozici bloku, pro který je hash spočítán. V tabulce $FilePart$ jsou uloženy informace o částech souboru, které server přijal a jsou uloženy v Azure BLOB pod názvem *FilePartGUID*.



Obrázek 3.2: Schéma relační databáze

3.7 Popis průběhu synchronizace

V této části popíšeme v bodech jednotlivé kroky, které jsou prováděny při synchronizaci.

3.7.1 Stažení celého souboru

1. Klient pošle na server požadavek na stažení souboru, který je v požadavku reprezentován pomocí jeho identifikátoru.
2. Server vytvoří přenos, který si uloží do databáze a vrátí klientovi identifikátor přenosu (kterým se klient v dalších požadavcích bude identifikovat) a velikost souboru.
3. Klient inicializuje vlákna pro stahování, které probíhá následovně:
   (a) Každé vlákno posílá požadavek na server na stažení části souboru dokud nebylo požádáno o všechny části. V požadavku posílá identifikátor přenosu a interval

bloku dat, který určuje pozici požadovaných dat ve stahovaném souboru.

(b) Server vyhledá přenos podle obdrženého ID přenosu v požadavku. Podle obdrženého intervalu z požadavku vyhledá blok dat a ten vrátí klientovi.

(c) Klient obdrží blok dat, tyto data zapíše do souboru na danou pozici a uloží si informaci o úspěšném zpracování daného bloku.

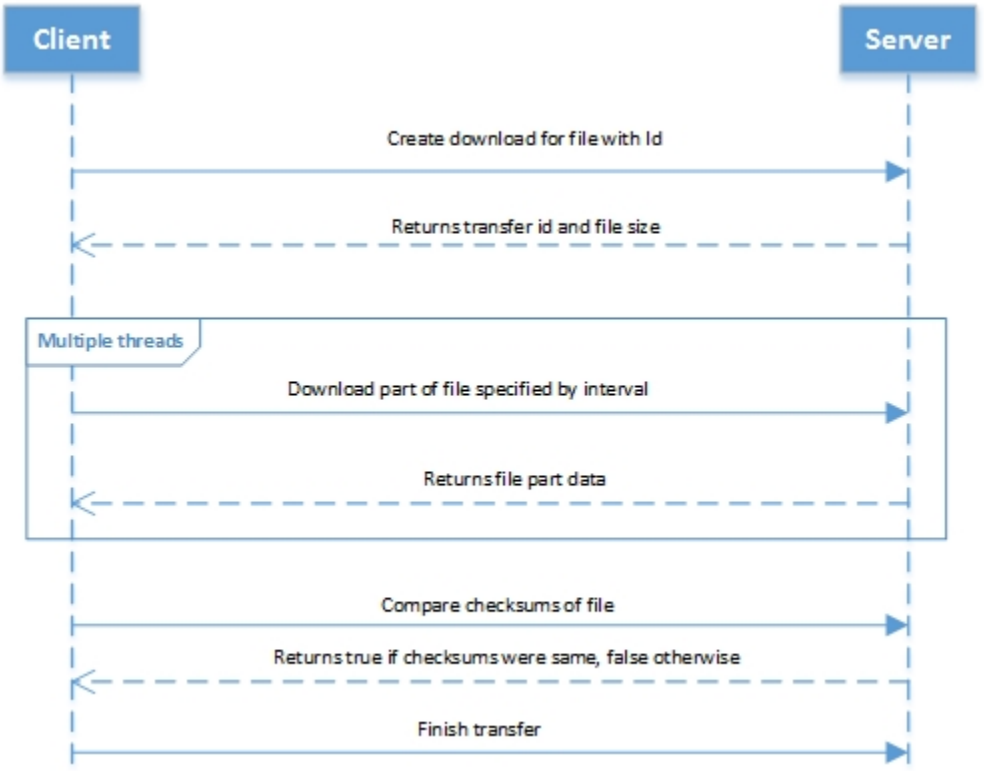
(d) Klient si ověří, že byly úspěšně zpracovány všechny bloky. O nezpracované bloky si klient požádá znovu.

4. Klient vypočítá pomocí hashovacích funkcí MD5 a SHA1 hashe takto staženého souboru a ty pošle s identifikátorem souboru na server.

5. Server dohledá hashe souboru podle obdrženého identifikátoru a porovná je s hashi v požadavku klienta. V odpovědi vrátí klientovi informaci o tom, zda se hashe shodovaly.

6. Klient podle odpovědi serveru rozpozná jestli byl soubor v pořádku stažen a korektně sestaven. V případě úspěšného stažení pošle serveru zprávu o ukončení přenosu.

Nástin komunikace mezi klientem a serverem při stahování celého souboru je znázorněn na obrázku 3.3.

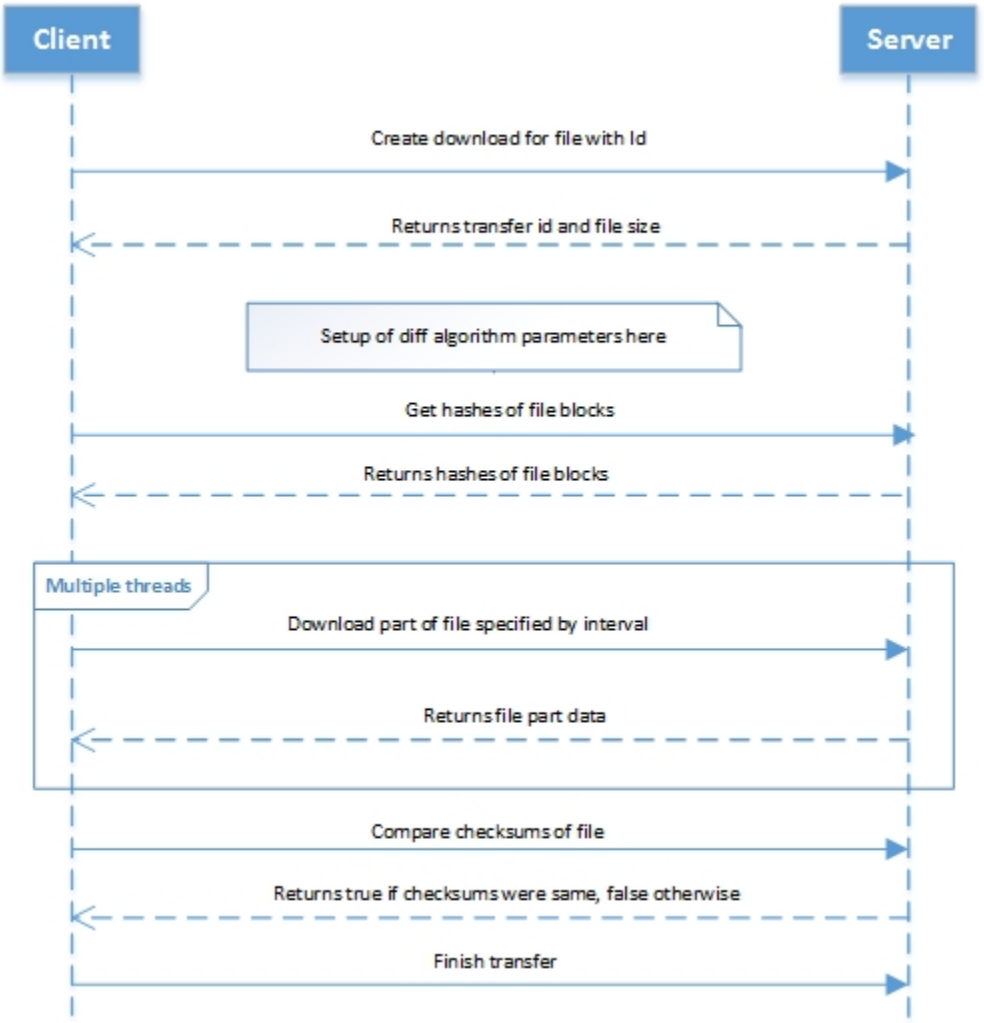


Obrázek 3.3: Nástin komunikace mezi klientem a serverem při stahování celého souboru

3.7.2 Stažení rozdílných částí souboru

1. Klient pošle na server požadavek na stažení souboru, který je v požadavku reprezentován pomocí jeho identifikátoru.
2. Server vytvoří přenos, který si uloží do databáze a vrátí klientovi identifikátor přenosu (kterým se klient v dalších požadavcích bude identifikovat) a velikost souboru.
3. Klient se serverem dojedná parametry algoritmu jak je znázorněno v části 3.7.5
4. Klient požádá server o hashe všech nepřekrývajících se bloků stahovaného souboru.
5. Server si tyto hashe načte z databáze a vrátí je klientovi s informacemi o pozicích ve stahovaném souboru.
6. Klient vyhledá pomocí zvoleného algoritmu shodné bloky s lokálně dostupným souborem a u každého shodného bloku si uloží informace o původní pozici a nové pozici (která je shodná s pozicí obdrženou od serveru).
7. Klient si vytvoří cílový soubor a nalezené shodné bloky dat zkopíruje ze zdrojového souboru na správné pozice. Informace o chybějících blocích dat jsou dopočítány a uloženy jako požadované části souboru.
8. Klient inicializuje vlákna pro stahování, které probíhá následovně:
   (a) Každé vlákno posílá požadavek na server na stažení části souboru dokud nebylo požádáno o všechny potřebné části. V požadavku posílá identifikátor přenosu a interval bloku dat, který určuje pozici požadovaných dat ve stahovaném souboru.
   (b) Server vyhledá přenos podle obdrženého ID přenosu v požadavku. Podle obdrženého intervalu z požadavku vyhledá blok dat a ten vrátí klientovi.
   (c) Klient obdrží blok dat, tyto data zapíše do cílového souboru na danou pozici a uloží si informaci o úspěšném zpracování daného bloku.
   (d) Klient si ověří, že byly úspěšně zpracovány všechny bloky. O nezpracované bloky si klient požádá znovu.
9. Klient vypočítá pomocí hashovacích funkcí MD5 a SHA1 hashe takto staženého souboru a ty pošle s identifikátorem souboru na server.
10. Server dohledá hashe souboru podle obdrženého identifikátoru a porovná je s hashi v požadavku klienta. V odpovědi vrátí klientovi informaci o tom, zda se hashe shodovaly.
11. Klient podle odpovědi serveru rozpozná jestli byl soubor v pořádku stažen a korektně sestaven. V případě úspěšného stažení pošle serveru zprávu o ukončení přenosu.

Nástin komunikace mezi klientem a serverem při stahování rozdílů souborů je znázorněn na obrázku 3.4.

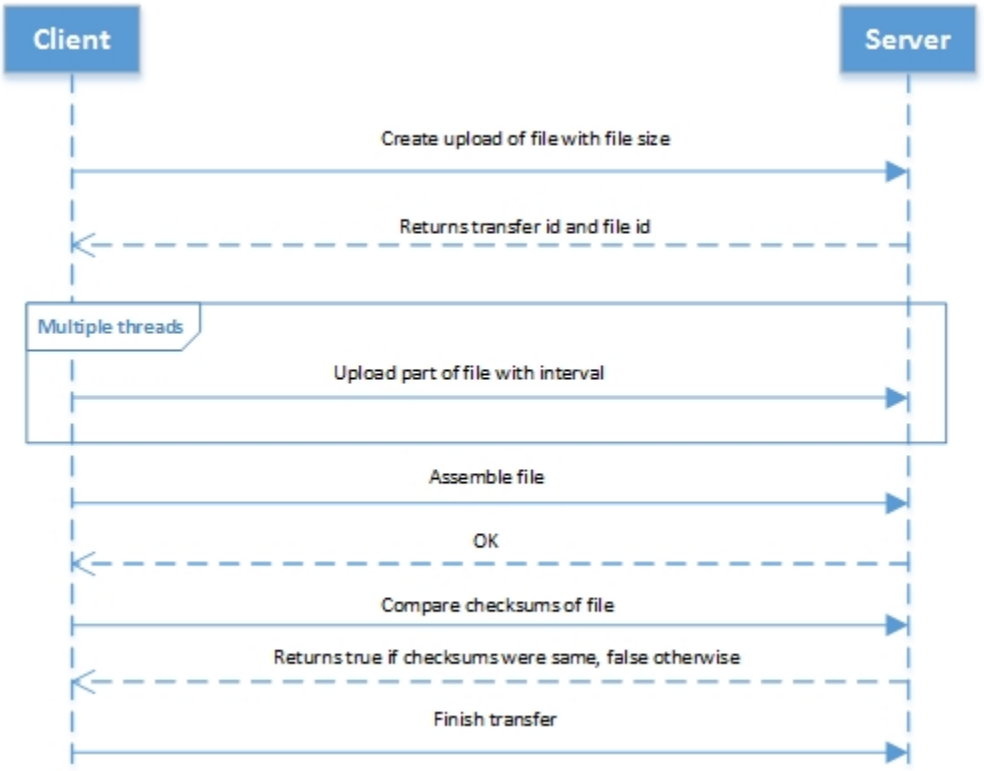


Obrázek 3.4: Nástin komunikace mezi klientem a serverem při stahování rozdílných částí souboru

3.7.3 Nahrání celého souboru

1. Klient pošle na server požadavek na nahrání souboru. V požadavku je uvedena velikost souboru.
2. Server vytvoří v databázi entity přenosu a nahrávaného souboru. Klientovi vrátí identifikátor přenosu (kterým se klient v dalších požadavcích bude identifikovat) a identifikátor souboru.
3. Klient inicializuje vlákna pro nahrání soubrou, které probíhá následovně:
   (a) Každé vlákno posílá požadavek na server na nahrání části souboru dokud nebyly poslány všechny části. V požadavku posílá identifikátor přenosu, data nahrávané části a interval bloku dat, který určuje pozici dat v nahrávaném souboru.
   (b) Server uloží obdržená data do úložiště Azure BLOB a uloží si informaci o jejich přijetí do databáze včetně obdrženého intervalu.
   (c) Klient pošle dotaz na server, zda obdržel všechny části souboru.
   (d) Server vypočítá chybějící bloky a případně je vrátí v odpovědi klientovi.
   (e) Klient chybějící bloky na serveru pošle znovu.
4. Klient pošle na server požadavek na sestavení souboru.
5. Server s pomocí informací uložených v databázi postupně načítá uložené části souboru a sestavuje výsledný soubor. Při sestavování také počítá hash celého souboru pomocí hashovacích funkcí MD5 a SHA1. O úspěšném sestavení informuje klienta v odpovědi.
6. Klient vypočítá pomocí hashovacích funkcí MD5 a SHA1 hashe nahráváného souboru a ty pošle s identifikátorem souboru na server.
7. Server dohledá hashe souboru podle obdrženého identifikátoru a porovná je s hashi v požadavku klienta. V odpovědi vrátí klientovi informaci o tom, zda se hashe shodovaly.
8. Klient podle odpovědi serveru rozpozná jestli byl soubor na server v pořádku nahrán a korektně sestaven. V případě úspěšného sestavení pošle serveru zprávu o ukončení přenosu.

Nástin komunikace mezi klientem a serverem při nahrávání celého souboru je znázorněn na obrázku 3.5.



Obrázek 3.5: Nástin komunikace mezi klientem a serverem při nahrávání celého souboru

3.7.4 Nahrání rozdílných částí souboru

1. Klient pošle na server požadavek na nahrání souboru. V požadavku je uvedena velikost souboru.

2. Server vytvoří v databázi entity přenosu a nahrávaného souboru. Klientovi vrátí identifikátor přenosu (kterým se klient v dalších požadavcích bude identifikovat) a identifikátor souboru.

3. Klient se serverem dojedná parametry algoritmu jak je znázorněno v části 3.7.5 včetně nastavení souboru na serveru, proti kterému se počítají rozdíly.

4. Klient požádá server o hashe všech nepřekrývajících se bloků souboru, proti kterému se počítají rozdíly.

5. Server si tyto hashe načte z databáze a vrátí je klientovi s informacemi o pozicích v daném souboru.

6. Klient vyhledá pomocí zvoleného algoritmu shodné bloky s lokálně dostupným souborem a u každého shodného bloku si uloží informace o původní pozici (která je shodná s pozicí obdrženou od serveru) a nové pozici.

7. Klient pošle na server požadavek na přemapování shodných bloků dat. V požadavku je seznam původních pozic bloků dat (které jsou shodné s pozicemi v souboru na serveru, proti kterému se počítají rozdíly) a nových pozic, kam se data mají uložit v nahrávaném souboru. Tento seznam je ještě před odesláním klientem seřazen vzestupně podle původních pozic. Díky tomuto kroku může server číst data ze souboru lineárně.

8. Server si shodná data načte a uloží podle částí do samostaných souborů. Do databáze si uloží informace o každé takovéto uložené části včetně nových pozic v nahrávaném souboru.

9. Klient pošle dotaz na server, jaké mu chybějí části.

10. Server vypočítá chybějící bloky (tedy bloky co nebyly přemapovány) a informace o nich vrátí v odpovědi klientovi.

11. Klient inicializuje vlákna pro nahrání chybějících částí soubrou, které probíhá následovně:

    (a) Každé vlákno posílá požadavek na server na nahrání části souboru dokud nebyly poslány všechny požadované části. V požadavku posílá identifikátor přenosu, data nahrávané části a interval bloku dat, který určuje pozici dat v nahrávaném souboru.

    (b) Server uloží obdržená data do úložiště Azure BLOB a uloží si informaci o jejich přijetí do databáze včetně obdrženého intervalu.

    (c) Klient pošle dotaz na server, zda obdržel všechny části souboru.

    (d) Server vypočítá chybějící bloky a případně je vrátí v odpovědi klientovi.

    (e) Klient chybějící bloky na serveru pošle znovu.

12. Klient pošle na server požadavek na sestavení souboru.

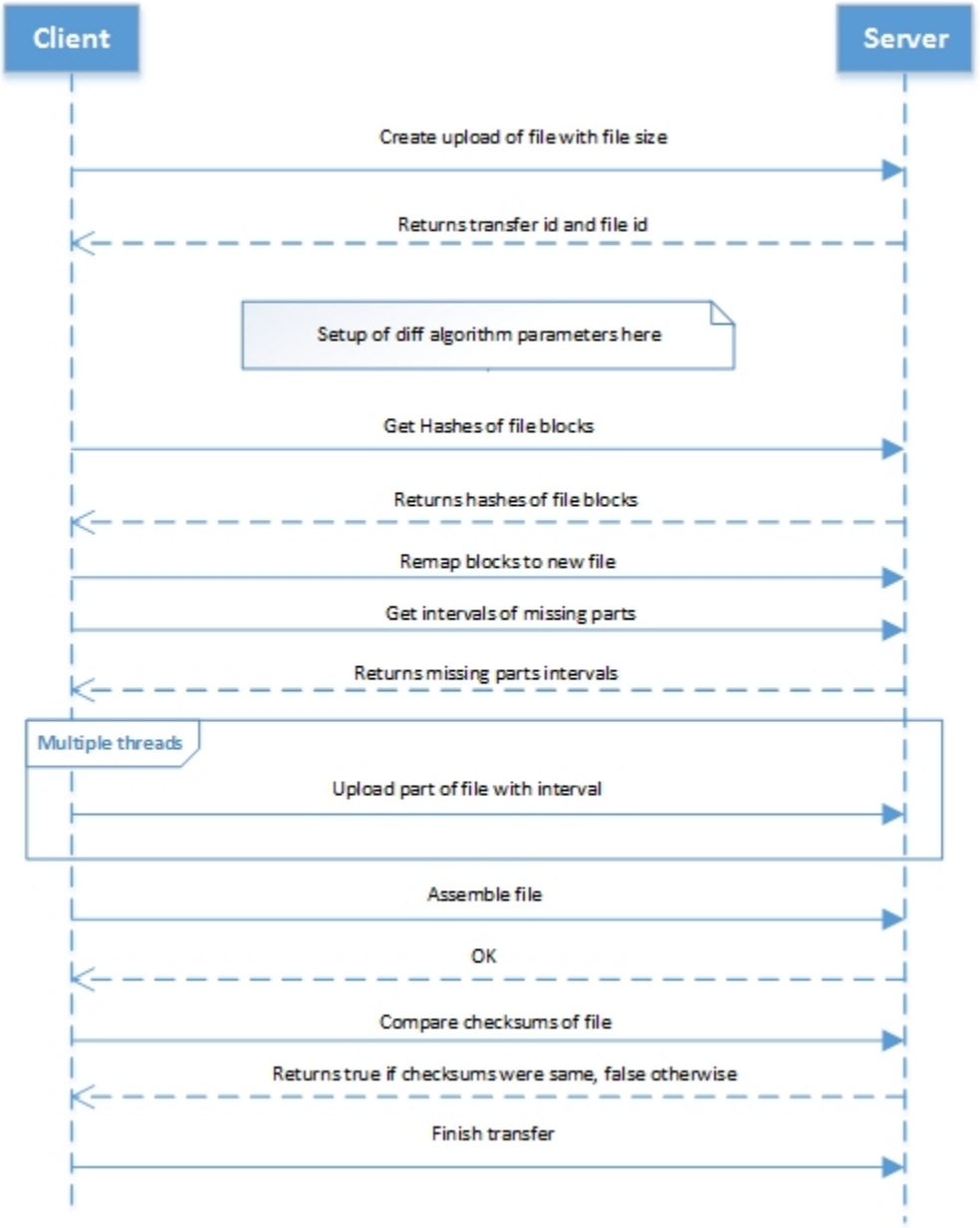
13. Server s pomocí informací uložených v databázi postupně načítá uložené části souboru a sestavuje výsledný soubor. Při sestavování také počítá hash celého souboru pomocí hashovacích funkcí MD5 a SHA1. O úspěšném sestavení informuje klienta v odpovědi.

14. Klient vypočítá pomocí hashovacích funkcí MD5 a SHA1 hashe nahráváného souboru a ty pošle s identifikátorem souboru na server.

15. Server dohledá hashe souboru podle obdrženého identifikátoru a porovná je s hashi v požadavku klienta. V odpovědi vrátí klientovi informaci o tom, zda se hashe shodovaly.

16. Klient podle odpovědi serveru rozpozná jestli byl soubor na server v pořádku nahrán a korektně sestaven. V případě úspěšného sestavení pošle serveru zprávu o ukončení přenosu.

Nástin komunikace mezi klientem a serverem při nahrávání rozdílů souborů je znázorněn na obrázku 3.6.

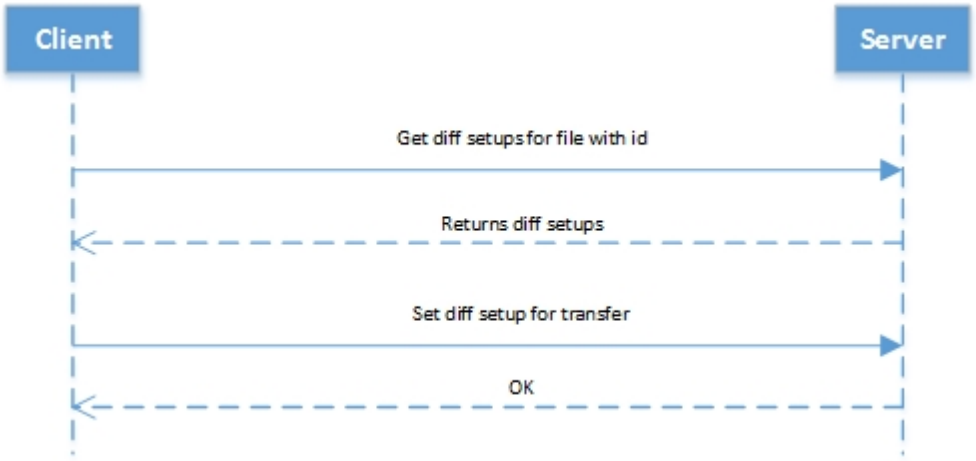


Obrázek 3.6: Nástin komunikace mezi klientem a serverem při nahrávání rozdílných částí souboru

3.7.5 Dojednávání parametrů algoritmu

1. Klient pošle dotaz serveru na možná nastavení algoritmu (velikost hashovacího okna, použitá prvočísla atd.).
2. Server vrátí klientovi množinu možných nastavení algoritmu.
3. Klient vybere jedno z možných nastavení, to si uloží do paměti pro pozdější použití a dále vybrané nastavení pošle serveru.
4. Server si uloží obdržené nastavení pro daný přenos klienta do databáze a vrátí klientovi zprávu o úspěšném nastavení.

Nástin komunikace mezi klientem a serverem při dojednávání parametrů algoritmu je znázorněn na obrázku 3.7.



Obrázek 3.7: Nástin komunikace mezi klientem a serverem při dojednávání parametrů algoritmu

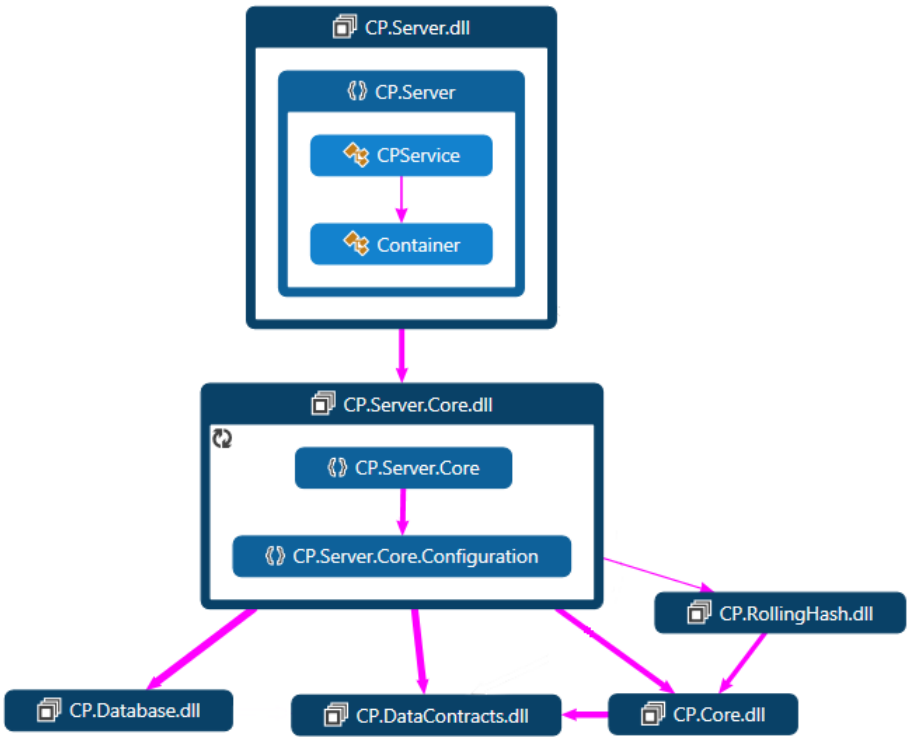
Kapitola 4

Realizace

4.1 Server

Z důvodu realizace pro cloudovou platformu Microsoft Azure, a vzhledem k nefunkčním požadavkům, byla serverová strana implementována v jazyce C#, který je vyspělým, silně typovaným a objektově orientovaným jazykem. Pro práci s databází byl použit mapovací framework NHibernate. Pro správu instancí, podporu dependency injection a konfiguraci byl použit IoC container Windsor Castle. Pro logování událostí byl pak použit framework Apache log4net. Serverová strana byla implementována jako webová služba nasaditelná do IIS (Internet Information Services), z důvodu snadného nasazení na Microsoft Azure platformu. Logika serverové strany byla rozdělena do několika komponent podle zodpovědnosti práce. Rozdělení na komponenty je znázorněno na obrázku 4.1. Funkcionalita jednotlivých komponent serveru je následující:

- CP.Server - Komponenta představující rozhraní služby
- CP.Server.Core - Obsahuje hlavní logiku serverové strany
- CP.Core - Obsahuje logiku, která je společná pro serverovou i klientskou stranu
- CP.Database - Obsahuje logiku pro práci s relační databází
- CP.RollingHash - Obsahuje logiku pro počítání hashů
- CP.DataContracts - Obsahuje kontrakty komunikačního rozhraní serveru

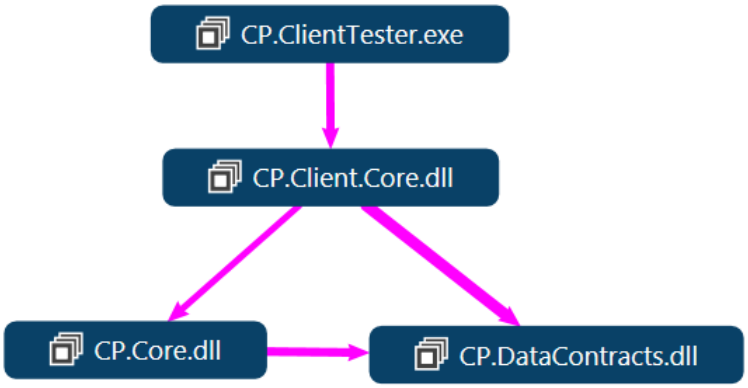


Obrázek 4.1: Diagram komponent serverové strany

4.2 Windows Desktop

Pro Windows Desktop klienta byl zvolen stejně jako v případě serveru implementační jazyk C# a bylo využito frameworku .NET 4.5. Pro logování událostí byl použit framework Apache log4net. Řešení bylo opět rozděleno na komponenty s výhodou využití některých komponent ze serverové strany, které jsou pro klientskou knihovnu shodné (jako jsou například kontrakty komunikačního rozhraní). Jednotlivé komponenty klienta jsou znázorněny na obrázku 4.2 a jejich funkcionalita je následující:

- CP.ClientTester - Konzolová aplikace využívající klientské knihovny
- CP.Client.Core - Obsahuje hlavní logiku klientské knihovny
- CP.Core - Obsahuje logiku, která je společná pro serverovou i klientskou stranu
- CP.DataContracts - Obsahuje kontrakty komunikačního rozhraní se serverem



Obrázek 4.2: Diagram komponent klientské strany pro Windows Desktop

4.3 Java

Pro platformu Java bylo využito standardní edice ve verzi 7 a implementačního jazyka Java. Komunikační rozhraní pro Java klienta bylo vygenerováno z WSDL serverové služby pomocí frameworku Apache CXF za použití následujícího příkazu:

```
wsdl2java -client -p nazev_balicku cesta_k_WSDL
```

Před vygenerováním komunikačního rozhraní bylo nejdříve nutné upravit WSDL, které je automaticky generované pomocí WCF, protože Java a WCF WSDL nejsou plně kompatibilní. Rozdílem je popis hlaviček, které jsou ve WCF generovaném WSDL popsány samostatně v další zprávě a jejich popis vypadá následovně:

```
<wsdl:message name="ChunkedFileStreamMessage">
<wsdl:part name="parameters" element="tns:ChunkedFileStreamMessage"/>
</wsdl:message>

<wsdl:message name="ChunkedFileStreamMessage_Headers">
<wsdl:part name="FileTransferId" element="tns:FileTransferId"/>
<wsdl:part name="Indexes" element="tns:Indexes"/>
</wsdl:message>
```

Služby postavené na platformě Java ale mají zápis hlaviček specifikován jako část zprávy. Upravená kritická část vypadá takto:

```
<wsdl:message name="ChunkedFileStreamMessage">
    <wsdl:part name="parameters" element="tns:ChunkedFileStreamMessage" />
   <wsdl:part name="FileTransferId" element="tns:FileTransferId" />
    <wsdl:part name="Indexes" element="tns:Indexes" />
</wsdl:message>
```

Kód knihovny obsahující hlavní logiku klienta byl v základní podobě přenesen z kódu knihovny pro Windows Dekstop a specifické funkce jazyka C# byly nahrazeny dostupnými ekvivalentními funkcemi v jazyce Java.

4.4 Windows Phone 8

Na platformě Windows Phone 8 bylo využito pro realizaci přenosu agenta BackgroundTransferService, který nezamezuje používání telefonu při synchronizaci, jak bylo popsáno v části 2.6.2. Protože tato služba umožňuje pouze stažení či poslání celého souboru pomocí protokolu HTTP, nemohla být úspora datového toku na této platformě realizována. Na serverové straně byly pro tento typ přenosu vystaveny metody s REST (Representational State Transfer) rozhraním.

4.5 Android

Jako vhodná verze platformy Android pro implementaci byla vybrána v současné době nejpoužívanější verze 4.1.x jak bylo zmíněno v části 2.6.4. Ikdyž na Android platformě je možnost využití většiny funkcí Javy, nebylo zde možné použít k vygenerování komunikačního rozhraní framework CXF, protože využívá jmenné prostory knihovních funkcí, které nejsou povoleny (též bylo zmíněno v části 2.6.4). Pro účely generování komunikačního rozhraní bylo využito služby easywsdl[1], která umožňuje volné šíření a modifikace vygenerovaného kódu bez jakýchkoliv dalších omezení. Takto vygenerované rozhraní využívá pouze knihovnu KSOAP2, která je použita pro podporu SOAP protokolu na zařízeních Android. Kód knihovny obsahující hlavní logiku klienta byl převzat z klienta pro Java platformu. Pozměněny byly specifické části, které využívaly funkcionalitu, jež nebyla dostupná na platformě Android. V rámci platformy Android byla implementována synchronizace s optimalizací datového toku pouze ve směru stahování. Před úplnou implementací by bylo vhodné otestovat vliv synchronizace na vlastnosti zařízení (jako je například výdrž baterie).

[1] dostupná online na adrese <easywsdl.com>

Kapitola 5

Testování a výsledky

Jelikož implementované řešení optimalizace datového toku je určeno pro přenášení uživatelských souborů, musela být otestována nejen efektivita optimalizace, ale i spolehlivost přenosu. Protože v rámci implementace byla výstupem pouze serverová strana a klientské knihovny, bylo pro otestování nutné napsat klientské aplikace, pomocí kterých se klientské knihovny daly otestovat. Tyto aplikace testují klientské knihovny jako black-box a mohou být v budoucnu s výhodou též použity jako příklad použití klientských knihoven. K testování byly použity počítače s těmito specifikacemi:

PC-1

- Procesor: Intel Core i5-3210M 4 jádra (8 vláken) s taktem 2.50 GHz (v režimu turbo až 3.1 GHz)
- Paměť RAM: 8 GB DDR3 s frekvencí 1333 MHz
- Operační systém: Windows 8.1 Pro, 64-bit
- Pevný disk: SSD Crucial MX-100 256 GB

PC-2

- Procesor: Intel Core i5 750 4 jádra (4 vlákna) s taktem 2.67 GHz (v režimu turbo až 3.2 GHz)
- Paměť RAM: 16 GB DDR3 s frekvencí 1333 MHz
- Operační systém: Windows 8.1 Pro, 64-bit
- Pevný disk: HDD Seagate Barracuda 7200.14 ST3000DM001 3TB

5.1 Testování spolehlivosti

Pro otestování spolehlivosti přenosu byla serverová strana nasazena na počítači PC-1 do lokálního emulátoru *Azure Cloud Services*, který emuluje nasazení instance v cloudovém

prostředí. Klientské knihovny pak byly na témže počítači spouštěny pomocí výše zmíněných testovacích aplikací s různými parametry, které byly zvoleny tak, aby se otestovala veškerá implementovaná funkcionalita. Korektní průběh přenosu byl monitorován pomocí analýzy informací zapsaných do logu (na serverové straně bylo logování nastavené do souboru, zatímco na klientské straně bylo logování nastavené do konzole). Výsledný soubor, který byl výsledkem synchronizace, byl poté porovnán se souborem cílovým jak implementovanou knihovnou (porovnání cílového a výsledného souboru je v rámci implementace knihoven prováděno vždy po přenosu a to pomocí porovnání hashů obou souborů získaných prostřednictvím hashovacích funkcí MD5 a SHA1), tak programem třetí strany HxD[1].

5.2 Testování optimalizace datového toku

Při testování optimalizace datového toku bylo třeba prozkoumat vliv synchronizace na čas potřebný pro přenos, objem přenesených dat a také závislost těchto veličin na nastavení přenosu a synchronizovaných datech. Pro testování synchronizace byly jako testovací data zvoleny textové soubory, které umožnily dostatečnou kontrolu nad provedenými změnami bez vlivu na naměřené výsledky (v tomto případě neuvažujeme typy souborů, které při změně uživatelem změní celou svoji datovou strukturu, protože tento typ souborů typicky není vhodný pro synchronizaci). Protože implementovaný algoritmus pracuje přímo s daty a bez dalších znalostí, výsledky synchronizace textových souborů jsou ekvivalentní s výsledky synchronizace jiných typů souborů (například binárních) za předpokladu, že data jsou dostatečně náhodná. Pro testovací účely byl tedy vytvořen cílový soubor obsahující náhodný anglický text, který byl vygenerován pomocí skriptu napsaném v jazyce Perl[2]. Tento skript náhodně vybírá slova z databáze slov, která je dostupná na většině linuxových distribucí (soubory byly generovány na distribuci Linux Mint 17) a generuje konkrétní počet slov na řádek a specifikovaný počet řádků. Poté vygenerováním dalších textových souborů a jejich kombinací s cílovým souborem byly získány testovací soubory, které odrážejí různě velké změny v různých částech cílového souboru.

Množina testovacích souborů je tedy následující:

Soubor *S*

Popis: Cílový soubor, nebo-li soubor, který chce uživatel získat po dokončení synchronizace

Velikost: 5 548 kB

Soubor *A*

Popis: Synchronizovaný soubor, obsahující přibližně 0,5 MB změn v jednom navazujícím bloku (tedy v jedné části) oproti cílovému souboru *S*

Velikost: 5 548 kB

Soubor *A2*

[1] Tento program je volně dostupný na adrese <http://mh-nexus.de/en/hxd/>

[2] Tento skript byl převzat z webové stránky <http://www.skorks.com/2010/03/how-to-quickly-generate-a-large-file-on-the-command-line-with-linux/>

Popis: Synchronizovaný soubor, obsahující přibližně 0,5 MB ve dvou nenavazujících blocích (tedy ve dvou částech) oproti cílovému souboru *S*

Velikost: 5 548 kB

Soubor *B*

Popis: Synchronizovaný soubor, obsahující přibližně 1 MB změn v jednom navazujícím bloku (tedy v jedné části) oproti cílovému souboru *S*

Velikost: 5 548 kB

Soubor *B2*

Popis: Synchronizovaný soubor, obsahující přibližně 1 MB ve čtyřech nenavazujících blocích (tedy ve čtyřech částech) oproti cílovému souboru *S*

Velikost: 5 549 kB

Soubor *C*

Popis: Synchronizovaný soubor, obsahující přibližně 2 MB změn v jednom navazujícím bloku (tedy v jedné části) oproti cílovému souboru *S*

Velikost: 5 548 kB

Soubor *C2*

Popis: Synchronizovaný soubor, obsahující přibližně 2 MB v pěti nenavazujících blocích (tedy v pěti částech) oproti cílovému souboru *S*

Velikost: 5 548 kB

Soubor *D*

Popis: Synchronizovaný soubor, obsahující přibližně 4 MB změn v jednom navazujícím bloku (tedy v jedné části) oproti cílovému souboru *S*

Velikost: 5 549 kB

Soubor *D2*

Popis: Synchronizovaný soubor, obsahující přibližně 4 MB v sedmi nenavazujících blocích (tedy v sedmi částech) oproti cílovému souboru *S*

Velikost: 5 548 kB

Soubory *A*, *B*, *C* a *D* představují případ, kdy byl soubor změněn pouze v jedné části, zatímco soubory *A2*, *B2*, *C2* a *D2* představují případ, kde byl soubor změnen v různých částech, nebo se změna uživatelem propagovala do různých části souboru.

Testována byla synchronizace pouze v jednom směru a to stahování novějšího souboru ze strany serveru. Jelikož výpočet změn provádí vždy klient, tak by v opačném směru synchronizace měly být výsledky obdobné alespoň v rámci přenesených dat. Časy synchronizace se při měření nahrávání nové verze souboru na server mohou lišit, což je dáno potřebou na straně serveru sestavovat soubor z částí a případně počítáním hashů nově nahraného souboru již při

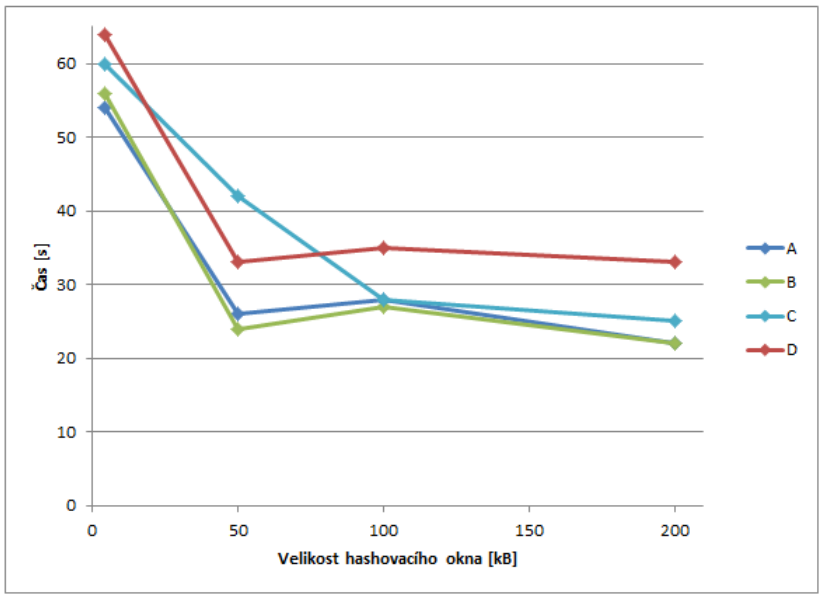
jeho sestavování. Tento fakt ale není problémem samotné synchronizace s omezením datového toku, ale právě ukládání nahrávaného souboru po částech, které je nutné díky omezení serverové strany s nemožností seekovat. Počítání hashů při sestavování souborů lze změnit v rámci konfigurace serverové strany a proto se tedy měřením přenosu v rámci nahrávání nové verze nebudeme zabývat.

Při měření efektivity synchronizace byla serverová část nasazena na počítači PC-1 do lokálního IIS serveru s využitím lokálního SQL Serveru 2014 pro databázi a Azure Storage emulátoru verze 3.3 pro ukládání dat. Klientská aplikace pro platformu Windows Desktop byla spouštěna na počítači PC-2. Klientské knihovny pro statní platformy nebyly na efektivitu synchronizace testovány, protože výsledky by měly být obdobné. Komunikace mezi PC-1 a PC-2 probíhala na lokální síti, ke které oba počítače byly připojeny pomocí Wi-Fi. Efektiva synchronizace je přímo závislá na velikosti hashovacího okna, které určuje granularitu s jakou lze najít shodné části. Jako vhodné velikosti hashovacích oken byly zvoleny hodnoty 4, 50, 100 a 200 kB. Menší hodnoty pro velikost hashovacího okna by zapříčinily nárůst výpočetní náročnosti a větší hodnoty by pro testování nebyly průkazné, protože by se ve většině případů přenesl celý soubor. Nejdříve byl na server nahrán textový soubor S s nastavením serveru na výpočet hashů s velikostí hashovacího okna 4 kB. Poté byla na serveru změněna konfigurace s nastavením velikosti hashovacího okna na 50 kB, server restartován a soubor S znovu poslán klientem. Takto bylo postupováno i pro zbývající velikosti hashovacích oken. Poté byl vždy klientem synchronizován soubor S s předpočítanými hashi s danou velikostí hashovacího okna oproti jednomu z ostatních testovacích souborů. Zaznenáván byl celkový čas potřebný pro danou synchronizaci (včetně dojednávání parametrů a kontroly shody souborů) a přenesená data, která byla požadována klientem pro úspěšné sestavení nové verze souboru. Pro ověření úspory datového toku byl též používan nástroj pro zachytávání síťové komunikace Fiddler[3]. V přenesených datech nejsou započítány poslané hashe částí souboru, protože ty lze v porovnání s velikostí přenesených dat zanedbat a také snadno dopočítat podle vzorce $velikost\ souboru / velikost\ okna * velikost\ hashe$, kde velikost jednoho hashe je 128 bitů nebo 32 bitů, podle toho jestli se jedná o silný či slabý hash. Například při stahování souboru o velikosti 5 MB a hashovacím oknem s velikostí 4 kB by velikost poslaných hashů byla rovna součtu $5000/4 * 16$ (silné hashe) a $5000/4 * 4$ (slabé hashe), tedy cca 25 kB což je přibližně 0.5 % z celkové velikosti souboru.

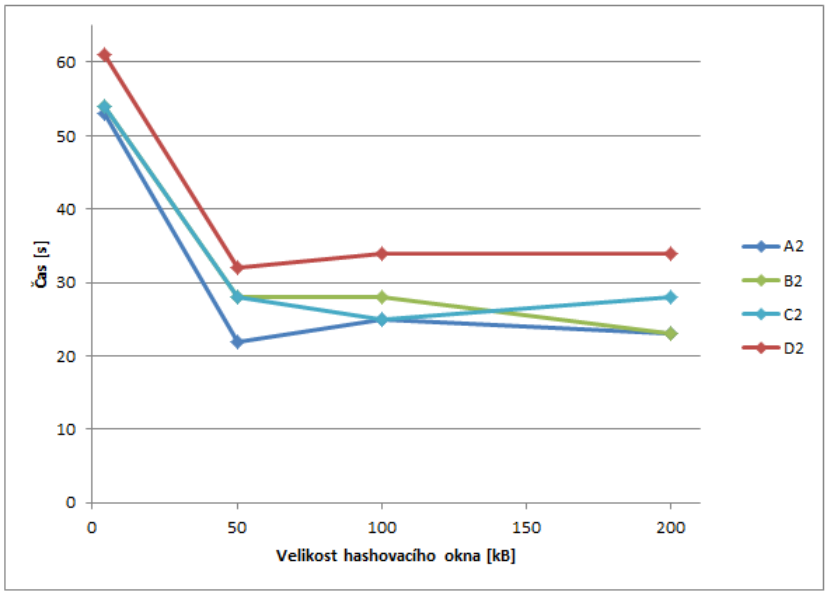
Naměřené hodnoty jsou zaznamenány v tabulkách, které jsou součástí přílohy C a byly použity pro sestrojení grafů zachycujících vliv velikosti hashovacího okna na čas potřebný pro synchronizaci a objem přenesených dat. Pro úplnost uvedeme, že stažení nové verze souboru bez použití synchronizace s optimalizací datového toku trvalo 16 sekund a přeneseno bylo 5 681 kB. Z grafů na obrázích 5.1 a 5.2 můžeme vypozorovat, že s malou velikostí hashovacího okna se zvyšuje náročnost na čas výpočtu rozdílů mezi soubory. To je způsobeno větším množstvím hashů cílového souboru S, které se musejí porovnávat při počítání shodných částí (dochází také častěji ke kolizím slabých hashů u bloků, které nejsou shodné). Hashovací okna s velikostí 50 kB a vyšší ale čas potřebný pro synchronizaci navýšily jen minimálně v řádu jednotek sekund a to v některých případech se značnou úsporou přenesených dat jak můžeme vypozorovat z grafů na obrázcích 5.3 a 5.4. Z grafů je také patrné, že soubory které mají málo shodných dat (soubory D či C) se synchronizují déle, protože se v jejich případě musejí rozdílná data také poslat, což zvyšuje časovou režii. Z porovnání grafů na

[3] Tento program je volně dostupný na adrese <http://www.telerik.com/fiddler>

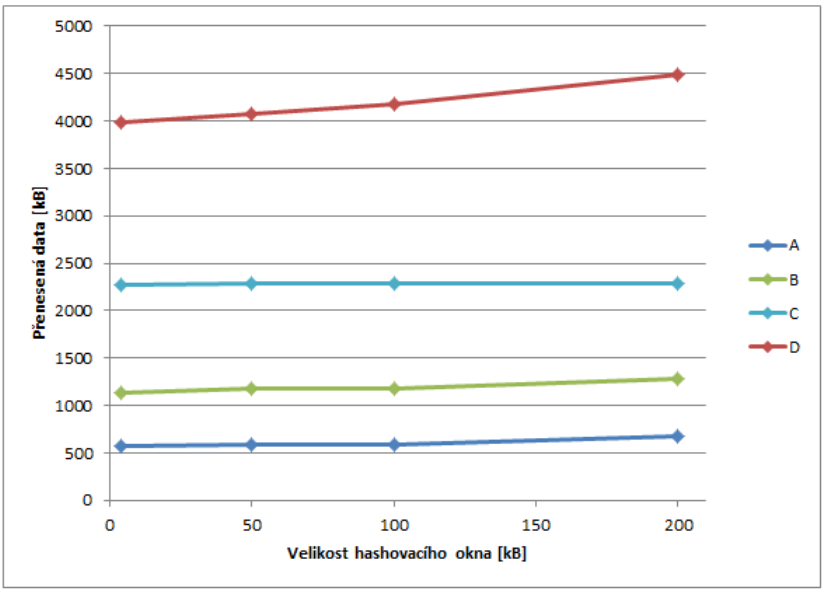
obrázcích 5.3 a 5.4 také vidíme, jak velikost hashovacího okna určuje granularitu se kterou je implementovaný algoritmus schopný nalézt shodné části. Zatímco na grafu 5.3 se pro soubor *C* přenáší prakticky stejný objem dat bez závislosti na velikosti hashovacího okna, na grafu 5.4 je pro soubor *C2* rozdíl přenesených dat téměř 1 MB pro hashovací okna s velikostí 4 a 200 kB.



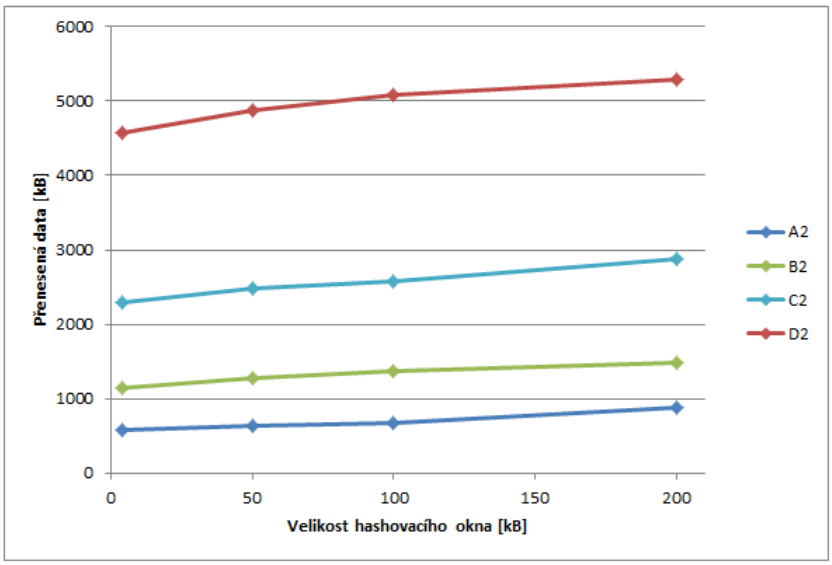
Obrázek 5.1: Graf času synchronizace pro soubory změnené v jedné části v závisloti na velikosti hashovacího okna



Obrázek 5.2: Graf času synchronizace pro soubory změnené v různých částech v závisloti na velikosti hashovacího okna



Obrázek 5.3: Graf přenesených dat pro soubory změnené v jedné části v závisloti na velikosti hashovacího okna



Obrázek 5.4: Graf přenesených dat pro soubory změnené v různých částech v závisloti na velikosti hashovacího okna

Kapitola 6

Závěr

6.1 Zhodnocení splnění cílů

Cílem této práce bylo navrhnout, implementovat a otestovat řešení pro synchronizaci souborů mezi klientem a serverem, které minimalizuje datový tok. V analytické části 2 jsme zvolili vhodný algoritmus minimalizující datový tok, který jsme pak v návrhové části 3 upravili tak, aby byl vhodný i pro systém MsBox. Takto upravený algoritmus byl v základní podobě implementován 4 a následně otestován jak na spolehlivost synchronizace, tak na efektivitu minimalizace datového toku v závislosti na času potřebném pro synchronizaci 5. Testování prokázalo úspěšnou minimalizaci datového toku a přijatelný časový nárůst pro potřebu synchronizace. Cíl této práce tedy můžeme považovat za splněný. Popis konfigurace serveru a rozhraní klientských knihoven, který je nedílnou součástí výstupu této práce, je obsažen v přílohách B.1 a B.2.

6.2 Návrhy na vylepšení

6.2.1 Ukládání předpočítaných hashů

V současné době se napočítané hashe pro soubor nahraný na server ukládájí do relační databáze. Protože objem takto uložených hashů bude se zvyšujícím se počtem souborů narůstat, bylo by vhodné tyto informace serializovat do samostatného souboru. V případě tohoto řešení bude ale nutné změřit vliv na čas potřebný k synchronizaci, protože parsování dat by mohlo synchronizaci zpomalit.

6.2.2 Dynamická velikost hashovacího okna

V kapitole zabývající se testováním efektivy synchronizace s úsporou datového toku jsme prokázali, že velikost hashovacího okna ovlivňuje čas potřebný pro synchronizaci i objem přenesených dat v určitých případech. Ikdyž serverová strana pomocí konfigurace umožňuje nastavení velikosti hashovacího okna v závisloti na kategorizaci souboru dle jeho objemu dat, nemusí toto řešení být dostačující. Řešením by bylo určovat velikost hashovacího okna dynamicky v závislosti na velikosti souboru (bez kategorizace) a to v rozumném měřítku (například by velikost hashovacího okna mohla být rovna 1 % z celkové velikosti souboru).

6.2.3 Předpočítávání hashů s různými parametry algoritmu

Ve stávajícím řečení se počítají hashe částí souboru pouze pro jedno nastavení algoritmu dané konfigurací. V budoucnu by bylo vhodné předpočítávat hashe částí souboru pro více nastavení algoritmu, aby si poté klient mohl zvolit, které nastavení mu vyhovuje nejvíce. Díky tomuto vylepšení by mohl klient zvolit například větší granularitu pro nalezení shodných částí na úkor vyšší časové a výpočetní náročnosti (či naopak) podle vlastních preferencí.

Literatura

[1] *Apache CXF Features* [online]. [cit. 6. 5. 2014]. Dostupné z: <http://cxf.apache.org/index.html>.

[2] *KSoap2 About* [online]. [cit. 6. 5. 2014]. Dostupné z: <http://kobjects.org/ksoap2/index.html>.

[3] *Apache log4net Features* [online]. [cit. 6. 5. 2014]. Dostupné z: <http://logging.apache.org/log4net/release/features.html>.

[4] *NHibernate Project Summary* [online]. [cit. 6. 5. 2014]. Dostupné z: <https://www.openhub.net/p/nhibernate>.

[5] *Inversion of Control Container* [online]. [cit. 6. 5. 2014]. Dostupné z: <http://docs.castleproject.org/Windsor.Inversion-of-Control.ashx>.

[6] A. Muthitacharoen, B. Chen, D. Mazieres. A Low-bandwidth Network File System. *MIT Laboratory for Computer Science and NYU Department of Computer Science*. 2001.

[7] A. Tridgell. *Rsync Rolling Checksum* [online]. 1998. [cit. 6. 5. 2014]. Dostupné z: <http://rsync.samba.org/tech_report/node3.html>.

[8] A. Tridgell, P. Mackerras. The rsync algorithm. Technical report, 1996.

[9] D. Teodosiu, N. Bjørner, Y. Gurevich, M. Manasse, J. Porkka. Optimizing File Replication over Limited Bandwidth Networks using Remote Differential Compression. *Microsoft Corporation*. 2006.

[10] Dan Xu, Chuanyi Liu, Jianping Wu. Remote File Synchronization: A Performance Comparision and Analysis of Different Approaches. *Beijing University of Posts and Telecommunications*.

[11] Daniel Kavan. *Diplomová práce*. FEL, ČVUT, 2012.

[12] Google. *Android Developers* [online]. 2014. Dostupné z: <http://developer.android.com/>.

[13] Martin Mudra. *Diplomová práce*. FEL, ČVUT, 2013.

[14] Microsoft. *Background agents for Windows Phone 8* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202942%28v=vs.105%29.aspx>.

[15] Microsoft. *Background file transfers for Windows Phone 8* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202955%28v=vs.105%29.aspx>.

[16] Přispěvatelé Wikipedie. *Android (operating system)* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Android_(operating_system)>.

[17] Přispěvatelé Wikipedie. *Microsoft Azure* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Microsoft_Azure>.

[18] Přispěvatelé Wikipedie. *Java (software platform)* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Java_(software_platform)>.

[19] Přispěvatelé Wikipedie. *Třetí normální forma* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://cs.wikipedia.org/wiki/Třetí_normální_forma>.

[20] Přispěvatelé Wikipedie. *Windows Phone 8* [online]. 2014. [cit. 6. 5. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Windows_Phone_8>.

[21] T. Lindholm, F. Yellin, G. Bracha, A. Buckley. *The Java Virtual Machine Specification Java SE 8 Edition*. Oracle America, Inc., 2014.

[22] U. Irmak, S. Mihaylov, T. Suel. Improved Single-Round Protocols for Remote File Synchronization. *Polytechnic University, Brooklyn, NY 11201*.

Příloha A

Seznam použitých zkratek

WCF Windows Communication Foundation

SOAP Simple Object Access Protocol

WSDL Web Services Description Language

XML Extensible Markup Language

RDC Remote Differential Compression

SQL Structured Query Language

BLOB Binary Large Object

SDK Software Development Kit

JDK Java Development Kit

PAAS Platform As A Service

IAAS Infrastructure As A Service

IIS Internet Information Services

TCP Transmission Control Protocol

HTTP Hypertext Transfer Protocol

REST Representational State Transfer

JVM Java Virtual Machine

IOC Inversion Of Control

GPL General Public License

LGPL Lesser GPL

Příloha B

Instalační a uživatelská příručka

B.1 Nasazení a konfigurace serverové strany

B.1.1 Lokální nasazení

Pro nasazení serverové strany je zapotřebí mít nainstalováno Visual Studio 2013, SQL Server 2014 a Windows Azure Storage Emulator alespoň ve verzi 3.3. V první řadě je třeba zprovoznit relační databázi v následujících krocích:

1. Založit databázi s názvem CPDB
2. Přidělit databázi uživateli se *sysadmin* právy
3. Spustit skript *000_ createScricpt.sql*
4. Spustit skript *001_ createScricpt.sql*

Nyní je třeba nakonfigurovat serverovou stranu. Konfigurace serverové strany se nachází v souboru *CP.Server.Container.config*, který je součástí projektu *CP.Server*. Zde je pro úspěšné spojení služby s databází potřeba vyplnit v elementu *defaultConnectionString* korektní hodnoty, přičemž *User id* je název uživatele, kterému byla přidělena databáze CPDB a *Password* je jeho heslo. Dále je možné nastavit hodnoty dalších elementů, které ovlivňují samotný proces optimalizace datového toku při synchronizaci souboru. Tyto elementy jsou popsány v části konfigurace B.1.2. Poté již stačí projekt *CP.Server* spustit.

B.1.2 Konfigurace

Pro konfiguraci procesu optimalizace datového toku bylo využito frameworku Castle Windsor a načítání konfiguračních parametrů ze souboru *CP.Server.Container.config*, který je umístěn v projektu *CP.Server* a obsahuje veškerou konfiguraci pro serverovou stranu. Konfigurovatelnost některých vlastností optimalizace vztahujících se k souborům je rozdělena na 3 různá nastavení s prefixy *small*\|*medium* \|*large* a to podle velikosti souborů (velikost souborů, které spadají do dané kategorie lze nastavit). Dále místo zmíněných prefixů bude využito zástupného symbolu x, protože možné hodnoty v konfiguraci jsou shodné pro všechny 3 typy nastavení bez ohledu na prefix.

Možnosti konfigurace jsou následující:

Element	storagePath
Význam	Určuje cestu k souborovému systému pro ukládání souborů (pouze v případě že bude použita komponenta FileSystemBlobDao)
Hodnoty	Význam
string	Cesta k root adresáři, kam se budou ukládat soubory

Element	blobDao
Význam	Určuje komponentu, která bude zodpovědná za ukládání souborů
Hodnoty	Význam
${AzureStorageBlobClient}	Komponenta ukládající soubory do azure storage
${FileSystemBlobDao}	Komponenta ukládající soubory na filesystem specifikovaný pomocí *storagePath*

Element	maxAssembleFileAttemps
Význam	Určuje maximální počet pokusů o sestavení souboru z obdržených částí
Hodnoty	Význam
integer	Maximální počet pokusů na sestavení souboru

Element	maxBufferSize
Význam	Určuje maximální velikost bufferu alokovaného pro práci se souborem soubory
Hodnoty	Význam
integer	Maximální velikost bufferu

Element	max$x$FileSize
Význam	Určuje maximální velikost souboru spadající do konfigurace s prefixy x. Pro prefix *large* se nespecifikuje (automaticky do této konfigurace spadají soubory větší než pro kategorii *medium*)
Hodnoty	Význam
integer	Maximální velikost souboru pro konfiguraci x udávaná v bytech

Element	xFileHashingTime
Význam	Určuje, kdy se budou počítat hashe pro části právě nahraného souboru spadajícího do kategorie x.
Hodnoty	Význam
WithinAssembleFile	Hashe souboru se budou počítat již při jeho sestavování
AfterAssembleFile	Hashe souboru se budou počítat až po jeho sestavení
Never	Hashe souboru se nebudou počítat

Element	*x*FileBlockSize
Význam	Určuje velikost bloků souboru, nad kterými se počítají hashe (hashovací okno).
Hodnoty	Význam
integer	Velikost bloků po kterých se počítají hashe udávaná v bytech

Element	*x*FileAlgorithm
Význam	Určuje druh algoritmu, který se použije pro výpočet v hashů
Hodnoty	Význam
RollingHashAdler32	Pro výpočet slabých hashů (weak hashů) se použije algoritmus Adler32
RollingHashRabinKarp	Pro výpočet slabých hashů (weak hashů) se použije Rabin-Karpův algoritmus

Element	saveStatistics
Význam	Určuje, zda se budou uchovávat záznamy o přenosech
Hodnoty	Význam
true	Po dokončení přenosu se záznam o přenosu zachová a pouze se vyplní čas ukončení
false	Po dokončení přenosu se vymaže záznam o přenosu z databáze

B.2 Použití klientských knihoven a dokumentace rozhraní

B.2.1 Použití

Pro využití přenosu s optimalizovaným datovým tokem je třeba nareferencovat všechny potřebné klientské knihovny pro danou platformu. Ikdyž knihovny nemají úplně shodnou implementaci, dodržují unifikované rozhraní. Pro využití knihovny je nutné založit instanci třídy *CommunicationProtocol*, která je vstupním bodem klientské knihovny a poskytuje veškerou dostupnou funkcionalitu. Všechny metody obsažené v třídě *CommunicationProtocol* vracejí instanci třídy *TransferInfo*, která obsahuje základní informace o přeneseném souboru a přenosu samotném. V případě výskytu chyby mohou metody třídy *CommunicationProtocol* vyhodit výjimku typu *ClientException* (nebo případně její konkrétní potomky), která signalizuje chybu v klientské knihovně, nebo může také vyhodit vyjímku typu *ServerException*, která signalizuje chybu komunikace se serverovou stranou nebo chybu na straně serveru. Dokumentace jednotlivých tříd, které bude klientská aplikace využívat je uvedena v dokumentaci rozhraní knihoven B.2.2. Před použitím klientské knihovny je třeba správně nastavit cílovou adresu serverové strany. V případě klientských knihoven pro platformu Windows Phone a Android je adresa serveru předávána jako parametr konstruktoru třídy *CommunicationProtocol*. Při využití knihovny na platformě Java je jako parametr konstruktoru trídy *CommunicationProtocol* cesta k WSDL, ve kterém musí být správně nastavena adresa serveru, jak je znázorněno v ukázce B.1.

Listing B.1: Ukázka nastavení adresy serveru ve WSDL

```
<wsdl:service name="CPService">
  <wsdl:port name="BasicHttpBinding_ICommunicationProtocolContract"
  binding="tns:BasicHttpBinding_ICommunicationProtocolContract">
    <soap:address location="http://192.168.15.103/CP.Server/CPService.svc" />
  </wsdl:port>
</wsdl:service>
```

V případě klientské knihovny na platformě Windows Desktop musí klientská aplikace správně nastavit endpoint v souboru App.config jak je znázorněno v ukázce B.2.

Listing B.2: Ukázka nastavení adresy serveru v souboru App.config

```
<client>
  <endpoint address="http://192.168.15.103/CP.Server/CPService.svc"
            binding="basicHttpBinding" bindingConfiguration="NGMsBoxServiceBinding"
            contract="CP.DataContracts.ICommunicationProtocolContract"
            name="NGMsBoxServiceBinding" />
</client>
```

B.2.2 Dokumentace rozhraní knihoven

V této části je popsána programová dokumentace tříd klientských knihoven, které bude klientská aplikace používat.

CommunicationProtocol Class Reference

Entry point for client library providing all functionality.

Public Member Functions

TransferInfo	**DownloadAll** (long fileId, string downloadTargetFileName, string pathToTargetDir, int numOfDownloaderTasks) Downloads all file parts without using diff algorithm
TransferInfo	**UploadAll** (string fileName, string pathToDirectory, int numOfUploaderTasks) Uploads all file parts without using diff algorithm
TransferInfo	**DownloadWithDiff** (long fileId, string downloadTargetFileName, string pathToTargetDir, int numOfDownloaderTasks, string fileName, string pathToDirectory) Downloads only changed parts of file using diff algorithm
TransferInfo	**UploadWithDiff** (string fileName, string pathToDirectory, int numOfUploaderTasks, long hashAgainstFileId) Uploads only changed parts of file using diff algorithm

Member Function Documentation

TransferInfo **DownloadAll** (long **fileId**,
string **downloadTargetFileName**,
string **pathToTargetDir**,
int **numOfDownloaderTasks**
)

Downloads all file parts without using diff algorithm

Parameters

fileId	Id of file to be downloaded
downloadTargetFileName	Name of file created by download
pathToTargetDir	Path to directory where will be downloaded file saved
numOfDownloaderTasks	Number of paralel downloading tasks

Returns

Instance of TransferInfo class

```
TransferInfo DownloadWithDiff ( long   fileId,
                                string downloadTargetFileName,
                                string pathToTargetDir,
                                int    numOfDownloaderTasks,
                                string fileName,
                                string pathToDirectory
                              )
```

Downloads only changed parts of file using diff algorithm

Parameters

fileId	Id of file to be downloaded
downloadTargetFileName	Name of file created by download
pathToTargetDir	Path to directory where will be downloaded file saved
numOfDownloaderTasks	Number of paralel downloading tasks
fileName	Name of similiar file which will be used for computing diff
pathToDirectory	Path to directory where is similiar file placed

Returns

Instance of TransferInfo class

```
TransferInfo UploadAll ( string fileName,
                         string pathToDirectory,
                         int    numOfUploaderTasks
                       )
```

Uploads all file parts without using diff algorithm

Parameters

fileName	Name of source file to be uploaded
pathToDirectory	Path to directory where is source file
numOfUploaderTasks	Number of paralel uploading tasks

Returns

Instance of TransferInfo class

```
TransferInfo UploadWithDiff ( string fileName,
                              string pathToDirectory,
                              int    numOfUploaderTasks,
                              long   hashAgainstFileId
                            )
```

Uploads only changed parts of file using diff algorithm

Parameters

fileName	Name of source file to be uploaded
pathToDirectory	Path to directory where is source file
numOfUploaderTasks	Number of paralel uploading tasks
hashAgainstFileId	Id of similiar file on server side which will be used for computing diff

Returns

Instance of **TransferInfo** class

TransferInfo Class Reference

Properties

long **FileId** `[get, set]`
Server identificator of transferred file

long **FileSize** `[get, set]`
Size of transferred file in bytes

long **TransferId** `[get, set]`
Server identificator of file transfer

TimeSpan **ElapsedTime** `[get, set]`
Time duration of transfer

long **TransferredBytes** `[get, set]`
Sum of file's data which were really sent to server in bytes

ClientException Class Reference

The exception that is thrown when error occurs in client.

Inherits Exception.

Inherited by **DirectoryForUploadNotPresentedException**, **FileForUploadDoesNotExistException**, **FileIdForDownloadNotPresentedException**, **FileNameForUploadNotPresentedException**, **FiletransferAlreadyInStatistics**, **FileTransferIdNotPresentedException**, **HashAgainstFileCannotBeDiffed**, and **HashAgainstFileDoesNotExistException**.

Public Member Functions

ClientException (string message)

ClientException (string message, System.Exception innnerException)

ServerException Class Reference

The exception that is thrown when error occurs in communication with server.

Inherits Exception.

Public Member Functions

ServerException (string message)

ServerException (string message, System.Exception innnerException)

Příloha C

Tabulky výsledků měření

V této kapitole jsou uvedené tabulky s naměřenými hodnotami synchronizace pro různé soubory. Pro úplnost uvedeme, že testována byla synchronizace v rámci stahování novější verze souboru a to souboru S. V případě stahování souboru S bez synchronizace (tedy stahování celého souboru) přenos trval 16 sekund a přenesená data byla rovna velikosti souboru S (5 681 kB).

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	54	569
50	26	581
100	28	581
200	22	681

Tabulka C.1: Výsledky měření synchronizace pro soubor A

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	53	573
50	22	631
100	25	681
200	23	881

Tabulka C.2: Výsledky měření synchronizace pro soubor *A2*

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	56	1 138
50	24	1 181
100	27	1 181
200	22	1 281

Tabulka C.3: Výsledky měření synchronizace pro soubor B

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	54	1 151
50	28	1 281
100	28	1 381
200	23	1 481

Tabulka C.4: Výsledky měření synchronizace pro soubor *B2*

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	60	2 277
50	42	2 281
100	28	2 281
200	25	2 281

Tabulka C.5: Výsledky měření synchronizace pro soubor *C*

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	54	2 290
50	28	2 481
100	25	2 581
200	28	2 881

Tabulka C.6: Výsledky měření synchronizace pro soubor *C2*

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	64	3 989
50	33	4 081
100	35	4 182
200	33	4 481

Tabulka C.7: Výsledky měření synchronizace pro soubor *D*

Velikost hashovacího okna [kB]	Čas synchronizace [s]	Přenesená data [kB]
4	61	4 575
50	32	4 881
100	34	5 081
200	34	5 281

Tabulka C.8: Výsledky měření synchronizace pro soubor *D2*

Příloha D

Obsah přiloženého CD

src - složka obsahující zdrojové kódy serveru a knihoven

database - složka obsahující skripty pro vytvoření databáze

libs - složka obsahující sestavené knihovny a serverovou část

diploma - složka obsahující zdrojové kódy tohoto dokumentu

minarja4_2015dipl.pdf - tento dokument ve formátu pdf