VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

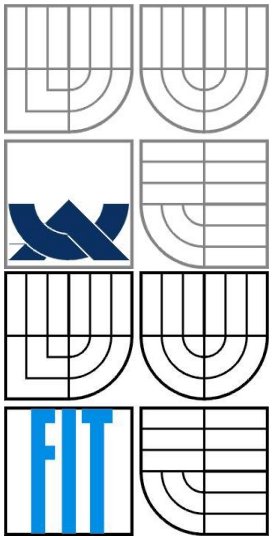
SÍŤOVÁ SYNCHRONIZACE DAT PRO OS WINDOWS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN HELLAR

BRNO 2014
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SÍŤOVÁ SYNCHRONIZACE DAT PRO OS WINDOWS

NETWORK SYNCHRONIZATION OF DATA UNDER OS WINDOWS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN HELLAR
AUTHOR

VEDOUCÍ PRÁCE — Ing. PAVEL OČENÁŠEK, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zabývá problémem návrhu a implementace nástroje pro synchronizaci souborů v síťovém prostředí pro OS Windows. Je poukázáno na problémy, které se při synchronizaci mohou vyskytnout, jako například správná detekce konfliktních změn, a jejich řešení. Dále je přiblížen problém ustanovení přímého spojení mezi dvěma zařízeními v internetu, pokud jsou tato zařízení k internetu připojena prostřednictvím zařízení NAT. V práci je ukázána konkrétní implementace řešení konfliktů, detekovaných během synchronizace, a aplikace některých metod pro překonávání zařízení NAT. Testováním byla ověřena funkčnost části výsledného nástroje pro řešení konfliktů a překonávání NAT. V závěru jsou diskutovány výsledky testování a navrhnuty možná rozšíření.

Abstract

This thesis deals with an implementation of a tool for file synchronization over a network for OS Windows. It also illustrates what issues are associated with synchronization of files, such as collision detection and resolution and the issue of setting up connection between two devices over the internet when connected to the internet via NAT devices. Thesis presents factual implementation of collision resolver and application of methods for NAT traversal. By testing the final solution correctness of collision resolver and NAT traversal technique was verified. Results of testing are discussed and possible extensions are proposed.

Klíčová slova

Synchronizace, řešení kolizí, NAT, peer-to-peer, UDP hole punching, Windows, .NET

Keywords

Synchronization, collision resolver, NAT, peer-to-peer, UDP hole punching, Windows, .NET

Citace

Jan Hellar: Síťová synchronizace dat pro OS Windows, bakalářská práce, Brno, FIT VUT v Brně, 2014

Síťová synchronizace dat pro OS Windows

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Pavla Očenáška, Ph. D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Hellar
14. května 2014

Poděkování

Zde bych chtěl poděkovat vedoucímu panu Ing. Pavlu Očenáškovi, Ph. D za odbornou pomoc při řešení této práce.

© Jan Hellar, 2014
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod

Osobní počítače a další zařízení, jako chytré telefony či tablety, jsou v současné době stále více finančně dostupné. Není tedy výjimkou, že uživatelé vlastní více než jedno takové zařízení a často tak vzniká potřeba řešit otázku dostupnosti dat. V ideálním případě by byla všechna uživatelova data jednoduše dostupná z každého jeho zařízení. Tento problém lze řešit pomocí synchronizace, při které jsou data mezi úložišti replikována. Vzhledem k stále vyšší dostupnosti internetového připojení je možnost umístit data do internetových, takzvaných cloudových[1], úložišť a ze zařízení připojenému k internetu k nim jednoduše přistupovat. Nespornou výhodou tohoto řešení je vysoká dostupnost dat. Aby uživatel mohl ke svým datům přistoupit, stačí mu pouze jeho přihlašovací údaje a jakékoli zařízení s přístupem k internetu a webovým prohlížečem. Nástroje jako Dropbox či Microsoft SkyDrive kombinují oba přístupy, tedy synchronizují data skrze cloudové úložiště. Ukládání dat do cloudového úložiště, pokud ho sám uživatel nevlastní, přináší problém spojený s otázkou soukromí. Pokud uživatel svoje data uloží na jiné než svoje vlastní zařízení, nemůže mu být zaručeno, že nebudou využita někým jiným.

Dalšími představiteli nástrojů pro synchronizaci dat jsou systémy pro správu verzí, které jsou často využívány při vývoji softwaru. Mezi tyto nástroje patří například Git nebo CVS. Tyto nástroje předpokládají, že synchronizovaná data jsou zároveň upravována více uživateli a kromě samotné synchronizace nabízejí zálohování synchronizovaných dat.

Další současné nástroje pro synchronizaci dat, využívající decentralizovaný přístup, většinou nenabízejí takový komfort a jednoduchost ovládání jako nástroje využívající cloudových úložišť. Cílem této práce je tedy návrh a implementace nástroje, který by uživateli umožňoval jednoduchou, bezpečnou, efektivní a decentralizovanou synchronizaci jeho dat mezi zařízeními s operačním systémem Windows. Naopak, cílem této práce není vytvořit nástroj podobný systémům pro správu verzí.

V této práci budou nejdříve detailněji prozkoumána současná řešení. Dále budou uvedeny některé práce zabývající se problematikou specifikace a implementace nástrojů pro synchronizaci dat. Na základě toho bude ukázán konkrétní návrh a implementace části synchronizačního algoritmu pro řešení konfliktů, které jsou detekovány během synchronizace. Zařízení bývají často připojena k internetu prostřednictvím NAT[2]. Bude tedy ukázán problém ustanovení komunikace mezi takovými zařízeními a jeho možná řešení.

[1] Cloud computing je model pro poskytování přístupu ke sdílené skupině výpočetních zdrojů (např. serverům, úložištím, aplikacím a službám) po síti.
[2] Network Address Translator (NAT) je zařízení pro překlad IP adres.

2 Existující řešení

V této části budou popsána některá ze současných řešení pro synchronizaci dat. Obecnými vlastnostmi současných řešení využívajících cloudových úložišť (Dropbox, Skydrive, Google Drive) je jednoduchost jejich použití, dostupnost dat (data v cloud úložišti jsou dostupná skrze webové rozhraní), podpora zálohování, zabezpečená komunikace po síti a šíře podporovaných operačních systémů.

2.1 Rsync a RDC

Rsync je nástroj pro platformu Unix pro jednosměrnou synchronizaci dat. Při jednosměrné synchronizaci jsou pouze na jedné z replik dovoleny provádět změny. Pomocí tohoto nástroje jsou potom ostatní repliky s touto hlavní synchronizovány. Tento nástroj využívá pro efektivní přenos dat rsync algoritmus.

Rsync algoritmus slouží k aktualizování souboru na vzdáleném počítači a pracuje takto: Mějme počítač `A` se souborem `a` a počítač `B` se souborem `b`, který je `a` podobný. Soubor `b` je třeba aktualizovat tak, aby byl stejný jako soubor `a`. Počítač `B` rozdělí soubor `b` na stejně velké části. Pro každou takovou část spočítá dva kontrolní součty, slabší tzv. rolling 32bitový kontrolní součet a silnější 128bitový MD4 kontrolní součet. Tyto kontrolní součty odešle na počítač `A`, který pomocí nich identifikuje části v souboru `a`, které jsou stejné jako v souboru `b`. Nejdříve porovnává části souboru pomocí slabšího a rychlejšího kontrolního součtu a v případě shody pak ještě porovná pomocí silnějšího. Rozdílné části odešle počítači `B`. Počítač `B` tyto přijme a pomocí nich a souboru `b` sestaví nový soubor `b'`, který je s `a` identický. [1]

Remode Differential Compression (RDC) je mechanizmus vyvinutý firmou Microsoft, který implementuje algoritmus podobný algoritmu rsync. Přístup k funkcím RDC je v operačním systému Windows umožněn pomocí rozhraní COM. [2]

2.2 Dropbox

Dropbox je typickým představitelem synchronizačních nástrojů vyúžívající cloudové úložiště. Pro uložení uživatelských dat využívá úložiště Amazon S3[3]. Hlavním implementačním jazykem je python. Dropbox podporuje verzování souborů, což je způsob zálohování, při němž jsou změny v souborech zaznamenávány a lze tak zpětně obnovit původní verzi souboru. Pro efektivní přenos dat využívá knihovnu librsync[4], která implementuje rsync algoritmus. [3] Tím je docíleno toho, že při změně v souboru na jedné z replik je namísto přenášení celého souboru přenášena pouze tato změna. Další z funkcí, které přispívají k efektivnější synchronizaci, je detekce přejmenování nebo přesunutí souboru nebo složky. V případě, že je soubor/složka v jedné z replik přejmenován/přesunut, může toto synchronizační nástroj interpretovat jako odstranění souboru a vytvoření nového, což potom vede k zbytečnému přesunu celého souboru po síti. Dropbox také dokáže rozpoznat, když se dvě synchronizovaná zařízení nacházejí ve stejné síti LAN. V takovém případě probíhá přenos dat přímo mezi zařízeními.

Aplikace je dostupná pro platformy Mac OS X, Windows, Linux, Android, iOS, BlackBerry a Kindle Fire. Základní verze je limitována velikostí synchronizované složky do 2GB a je dostupná zdarma. [4]

[3] Amazon Simple Storage Service (S3)
[4] Knihovna je dostupná na http://librsync.sourceforge.net/

2.3 BitTorrent Sync

BitTorrent Sync je nástroj pro synchronizaci souborů založený na peer-to-peer protokolu zvaném BitTorrent. V době psaní této práce je dostupný v testovací verzi. Před zahájením synchronizace musí uživatel do zařízení, které chce synchronizovat, zadat jedinečný tajný klíč (secret), který je pro něj aplikací vygenerován. Veškerá komunikace mezi klienty je šifrována pomocí AES-128[5] a symetrického klíče. Klienti se autentizují pomocí protokolu SRP[6]. Aplikace je dostupná pro platformy Mac OS X, Windows, Linux, FreeBSD, Android, iOS a Windows Phone. [5]

2.3.1 BitTorrent protokol

BitTorrent protokol slouží k efektivnímu stahování velkých souborů. Řeší problém, kdy velké množství klientů stahuje jeden soubor, který je dostupný pouze z jednoho zdroje – serveru. Aplikováním tohoto protokolu je soubor rozdělen na stejně dlouhé části (většinou 256KB). Ty, které již mají klienti staženy, si pak navzájem posílají. Server na začátku slouží jako takzvaný seed, který má k dispozici celý soubor. Všichni klienti, kteří soubor stahují, jsou označováni jako roj (angl. swarm). Klient iniciuje stahování pomocí tzv. metainfo souboru (s příponou .torrent), který byl vytvořen na straně serveru. Metainfo soubor obsahuje mimo jiné URL adresu trackeru a seznam SHA-1[7] otisků částí stahovaného souboru. Tracker je serverová aplikace, která klientům poskytuje seznam ostatních klientů v roji.

Překonávání zařízení NAT při ustanovování komunikace mezi klienty je řešeno pomocí metody UDP Hole Punching (tato metoda je popsána v podkapitole 3.2.1), kde tracker vystupuje jako prostředník. V seznamu klientů, který klient obdržel od trackeru, jsou uvedeny i veřejné IP adresy a porty klientů, skrze které je možno s klienty ustanovit spojení. Informace v předchozích dvou odstavcích byly čerpány z [6].

DHT Protocol (distributed hash table) je rozšířením BitTorrent protokolu a slouží k ukládání a šíření adres jednotlivých klientů bez použití tracker serveru. Pomocí tohoto protokolu každý klient zastupuje funkci tracker serveru. [7]

2.3.2 Vzájemné nalezení klientů a překonávání NAT

Aby mezi sebou mohli klienti zahájit synchronizaci, musí vzájemně zjistit svoji adresu v síti. Toho je v BitTorrent Syncu dosaženo více mechanismy. Jedním z nich je využití dříve zmíněného tracker serveru, který pak klienty spojuje na základě kombinace jimi poskytnutého SHA-1 otisku tajného klíče a jejich IP adresy a portu. Další použitou metodou je vyhledávání klientů v lokální síti odesíláním broadcastových zpráv. Pokud v ní je jiný klient se stejným tajným klíčem, odpoví mu a ustanoví s ním spojení.

Kromě metody UDP Hole Punching, používá aplikace BitTorrent Sync, při navazování komunikace mezi klienty, také metodu pro mapování portů pomocí standardu UPnP[8]. Pokud zařízení NAT na straně klienta tento standard podporuje, aplikace může vytvořit statická pravidla v překladové tabulce NAT. Ke klientovi je tak potom možno se připojit z veřejné sítě.

[5] Advanced Encryption Standard (AES) je standardizovaný algoritmus používaný k šifrování elektronických dat.

[6] Secure Remote Password (SRP) je protokol pro vzdálené bezpečné ověření identity pomocí krátkých hesel.

[7] SHA-1 je jedním z bezpečných hešovacích algoritmů standardizovaných National Institute of Standards and Technology (NIST).

[8] Universal Plug and Play (UPnP) je standard definující protokoly sloužící jako univerzální prostředek pro komunikaci mezi různými zařízeními v sítích založených na IP protokolu.[30]

V případě, že se klienti nacházejí za tzv. symetrickými NATy nebo firewally s pravidly znemožňujícími ustanovit komunikaci mezi klienty pomocí dříve uvedených technik, umožňuje aplikace také zprostředkovat komunikaci skrze takzvaný relay server. Relay server slouží jako prostředník, ke kterému se klienti připojí a ten potom přeposílá jejich zprávy. [5].

2.4 Práce kolegů na obdobné téma

V roce 2013 vytvořili kolegové Adam Marťák, Jaromír Karmazín a Jakub Slováček v rámci svých bakalářských prací své řešení pro synchronizaci a zálohování dat. V jejich pracích byl zvolen decentralizovaný přístup. Každá práce se zabývala implementací pro jiný operační systém: Windows, Linux a Android. Všechny aplikace sdílejí stejný protokol (ne všechny implementují veškerá rozšíření) a umožňují verzování synchronizovaných souborů. Jejich řešení je inspirované nástrojem Git, což je distribuovaný systém správy verzí. Data přenášená během komunikace jsou serializována pomocí Protocol Buffers[9]. Autentizace klientů a zabezpečení samotné komunikace je zajištěno pomocí protokolu TLS[10] a certifikátů podepsaných sebou samými (angl. self-signed). V práci A. Marťáka je implementován efektivní přenos dat pomocí rsync algoritmu a knihovny librsync. Jedním z jejich navrhovaných rozšíření je právě umožnění komunikace mezi klienty za zařízeními NAT. [8] [9] [10]

[9] Protocol Buffers bude popsán v podkapitole 3.6.1
[10] Protokol TLS bude popsán v podkapitole 3.5.2

3 Rozbor použitých technologií a možnosti přidělené platformy

V této části budou vysvětleny problémy spojené se synchronizací souborů a ustanovením spojení mezi klienty za zařízeními NAT a uvedeny metody a technologie, které umožňují tyto problémy řešit. Dále zde budou popsány metody pro zabezpečení komunikace mezi zařízeními v síťovém prostředí a možnosti pro podporu řešení výše uvedených problémů, které jsou dostupné v rámci operačního systému Windows.

3.1 Synchronizace

Synchronizace dat je proces ustanovení a udržení konzistence dat mezi zdrojovým a cílovým datovým úložištěm a naopak. [11] V této podkapitole budou uvedeny problémy spojené se synchronizací souborů a možné metody pro jejich řešení.

3.1.1 Problémy spojené se synchronizací souborů

Pokud uvažujeme synchronizaci dvou složek přítomných na jednom zařízení, proces synchronizace by mohl vypadat následovně. V první fázi jsou nalezeny cesty k souborům/složkám, které se vyskytují pouze v jedné ze synchronizovaných složek, a soubory/složky, které odkazují, jsou zkopírovány do druhé složky. V druhé fázi jsou nalezeny cesty k souborům, které se nacházejí v obou složkách, a soubory, které tyto cesty odkazují, porovnat. Pokud se tyto soubory liší, soubor se starším časem poslední modifikace je přepsán novějším. Tento přístup sám o sobě nebude reflektovat uživatelův záměr odstranit některý ze souborů. Proto by bylo nutné sledovat a pamatovat si všechny operace odstranění, ke kterým ve složkách od poslední synchronizace došlo. Možným přístupem je odvození operací odstranění ze současného stavu složek a ze stavu složek po poslední synchronizaci. Uvažováním těchto operací může docházet ke konfliktům. Příkladem může být vytvoření souboru ve složce s cestou `p` v jedné ze synchronizovaných složek a odstranění složky s cestou `p` v druhé ze synchronizovaných složek. Konfliktní změny mohou být provedeny například uživatelem (ať záměrně nebo ne) nebo některou aplikací. Tyto konfliktní změny musí být detekovány a určitým způsobem řešeny.

V případě, že jsou synchronizovány soubory mezi dvěma zařízeními a je použit výše uvedený přístup, nebude ve druhé fázi možné na základě časů poslední modifikace správně rozhodnout, který ze souborů je aktuálnější, jelikož systémový čas na obou zařízeních se může lišit. I kdyby byla zaručena synchronizace těchto časů, musí být předpokládáno, že uživatelé (nebo aplikace) provádějí změny v některé ze synchronizovaných složek pouze na základě znalosti jejího současného stavu. Tedy pokud mezi dvěma procesy synchronizace dojde k upravení souborů se stejnou cestou ve více zařízeních, měli by být i tyto změny uvažovány jako konfliktní.

3.1.2 Pesimistická a optimistická replikace

Obecně lze pro synchronizaci (nejen souborů) využít dvou přístupů, pesimistické nebo optimistické replikace.

Následující dva odstavce byly převzaty z [11]. Tradiční pesimistická replikace se snaží konfliktům předejít a zaručit, že všechny repliky jsou neustále totožné, tedy tváří se, jakoby to byla pouze jedna kopie dat. Toho bývá docíleno mnoha způsoby. Základní koncept spočívá v tom, že pokud replika není

zaručeně aktuální, přístup k ní je blokován. Například primary-copy algoritmus zvolí jednu repliku jako primární, a ta je zodpovědná za veškerý přístup k určitému objektu. Primární zařízení, po provedené změně, synchronně zapíše tuto změnu na ostatní (sekundární) repliky. Pokud primární zařízení selže, zbylé repliky mezi sebou zvolí novou primární repliku. Tento přístup je možné využít v LAN sítích, kde spojení mezi zařízeními je rychlé a spolehlivé.

Optimistická replikace (angl. optimistic/lazy replication) dovoluje, aby repliky divergovaly. Dovoluje přístup k datům, aniž by bylo zaručeno, že jsou repliky synchronizovány, a to na základě optimistického předpokladu, že ke konfliktům během následné synchronizace dojde málokdy. Změny jsou propagovány na pozadí a občasné konflikty jsou opraveny teprve, když k nim dojde. Optimistickou replikaci je výhodné použít při synchronizaci zařízení, které k síti nejsou připojeny stále, nebo pokud jsou spojeny prostřednictvím nespolehlivého, pomalého připojení.

Optimistická replikace je využívána ve většině nástrojů pro synchronizaci souborů a vzhledem k povaze řešeného problému bude využita i v této práci.

3.1.3 Stavově založený algoritmus

Je-li použita optimistická replikace, je nutné přesně specifikovat, které změny budou považovány za konfliktní. Práce [12] takovou specifikaci nabízí a je založena na posuzování stavů souborového systému před a po synchronizaci Základní myšlenkou práce je, že synchronizační nástroj by měl detekovat konfliktní změny a nekonfliktní propagovat.

Následující část této podkapitoly, popisující samotné řešení, byla převzata z [12]. Je uvažována synchronizace dvou replik. Nejdříve se provede detekce změn na obou replikách `A`, `B` od poslední synchronizace. Výsledkem jsou predikáty `dirtyA` a `dirtyB`, které pro každou cestu ke každému souboru a složce v replikách `A` resp. `B` definují, zda v ní byla provedena změna. Tyto predikáty a současné stavy obou replik jsou vstupem pro usmiřovatele (angl. reconciler), který vypočítá nové stavy `A′`, `B′` obou replik tak, jak bylo uvedeno dříve – všechny nekonfliktní změny v jedné z replik jsou propagovány do druhé. Pokud nebyly detekovány žádné konfliktní změny, `A′` a `B′` jsou stejné.

Po každé synchronizaci je uložen stav replik, a to tak, že pro každý soubor nebo složku je uložen čas poslední modifikace a i-node[11] číslo (v práci byl uvažován unixový systém). Při další synchronizaci potom detektor změn označí cestu jako „*dirty*“, pokud její i-node není stejný jako uložený nebo její čas poslední modifikace je pozdější než čas poslední synchronizace.

Specifikaci usmíření lze charakterizovat jako soubor podmínek, které by měli platit pro dvě repliky mezi jejich počátečními stavy `A`, `B` a smířenými stavy `C`, `D` pro každou cestu `p`. Neformální specifikace:

1. Pokud `p` není *dirty* v replice `A`, víme, že celý podstrom začínající v `p` nebyl v `A` změněn a jakékoli změny v odpovídajícím stromu v `B` by měly být propagovány do obou replik. To znamená, že `C(p)` a `D(p)` by mělo být stejné jako `B(p)`.
2. Pokud `p` není *dirty* v `B`, potom by mělo platit `C(p)=D(p)=A(p)`.
3. Pokud `p` odkazuje složku v `A` i `B`, pak by `p` mělo označovat složku i v `C` a `D`.
4. Pokud je `p` *dirty* v `A` i `B` a odkazuje soubor alespoň v jedné z replik `A` nebo `B`, potom se jedná o potenciální konfliktní změnu. V tomto případě, by se neměla změna propagovat: `C(p)=A(p)` a `D(p)=B(p)`.

[11] Index node (i-node) je datová struktura používaná v operačních systémech Unix pro reprezentaci objektů v souborovém systému.

3.1.4 Algoritmus založený na operacích

Ve výše uvedeném řešení jsou konfliktní změny detekovány, ale rozhodnutí o tom, jak s nimi naložit, je ponecháno na uživateli a teprve po jeho zásahu mohou být data na replikách znovu konzistentní. Tento problém řeší práce [13]. V řešení zde prezentovaném jsou namísto stavů posuzovány operace provedené na každé replice.

Následující tři odstavce jsou převzaty z [13]. Jsou zde uvažovány tři druhy operací: vytvoření - operace `create`, úprava - operace `edit` a smazání - operace `remove`. Přejmenování/přesunutí je uvažováno jako operace smazání a vytvoření. Uvažováním pouze těchto tří operací bylo umožněno vytvořit formální systém pro popis operací v souborovém systému užitečný při specifikaci a implementaci synchronizačního nástroje. Jak již bylo uvedeno v podkapitole 2.2, tento přístup nebude v případě operací přejmenování nebo přesunutí efektivní. V práci je proto uveden i princip, pomocí kterého lze řešení rozšířit o detekování těchto operací. Na základě vytvořeného formálního systému byly definovány tři části procesu synchronizace: detekce změn, usmíření a řešení konfliktů.

Detekce změn je realizována jako vytvoření minimální posloupnosti operací S_i, potřebných k uvedení stavu repliky R_i po poslední synchronizaci do stavu po provedení změn. Operace budou uvedeny ve formátu `typ_operace(cesta, výsledný_objekt)`. `π/π'` označuje cestu k objektu, který je v podstromu složky s cestou `π`. Výsledným objektem může být buďto složka nebo soubor, které budou označeny jako `Dir` respektive `File(obsah)`. Operace v minimální posloupnosti jsou uspořádány takto:

- Operace `edit(π, Dir)`.
- Operace `create`, seřazeny tak, aby vytvoření rodičovské složky předcházelo vytvoření jejích potomků.
- Operace `remove`, seřazeny tak, aby smazání potomků předcházelo smazání rodičovské složky.
- Operace `edit(π, File(x))`.

Výsledkem usmíření je jedna posloupnost S_i* a jedna množina S_iK operací pro každou ze synchronizovaných replik R_i. S_i* je posloupnost smířených operací, vzniklá propagováním nekonfliktních operací z ostatních replik. S_iK je množina konfliktních operací. Algoritmus pro usmíření:

```
for i ∈ 1..n do
   make Si* empty
for i ∈ 1..n do
   for j ∈ 1..n do
        for every command C ∈ Si do
             if C should be propagated to replica j then
                  append C to Sj*
```

Operace `C` má být propagována z repliky `i` do repliky `j`, pokud zároveň platí tři kritéria:

- Stejná operace nebyla na `j` provedena.
- Žádná z operací provedených na replice jiné než `i` není v konfliktu s `C`.
- Žádná z operací provedených na replice jiné než `i` není v konfliktu s operacemi, které musí předcházet `C`.

Formální zápis, který je v práci použit pro definici dvou konfliktních operací, zde nebudu uvádět, jelikož by to vyžadovalo vysvětlení značné části formálního systému. Intuitivně se ale jedná o dvojice operací ovlivňující soubor se stejnou cestou.

Práce [13] dále popisuje možné přístupy k řešení konfliktních operací. Jedna z možností je úprava konfliktních operací na nekonfliktní a aplikování těchto upravených operací na každou repliku. Pokud máme dvě konfliktní operace C_i a C_j, kdy C_i i C_j vytvářejí soubor `/soubor.dat`, ale s jiným obsahem, C_i je, připojením suffixu k názvu souboru, upravena na vytvoření souboru `/soubor_konflikt_z_i.dat` a C_j na `/soubor_konflikt_z_j.dat`. Tento přístup je využit například v Dropboxu. V případě, že máme soubor, který je na jedné z replik upraven a na jiné smazán, bylo by nežádoucí, kdyby byl tento aktualizovaný soubor smazán. V takovém případě je tedy operace smazání zrušena. Použitím těchto technik je zaručeno, že všechny konflikty budou vyřešeny, aniž by uživatel musel do procesu synchronizace zasahovat, a že po skončení každé synchronizace jsou všechny repliky konzistentní.

3.2 Překonávání NAT

Klienti spolu během synchronizace musejí komunikovat. Většina dnešních osobních počítačů je k internetu připojena skrze zařízení typu NAT, což znemožňuje jednoduché ustanovení spojení mezi klienty. Dále bude vysvětleno proč tomu tak je.

Network address translator (NAT) je zařízení, které mapuje jeden IP adresový prostor do jiného. Termín NAT lze použít pro širší skupinu zařízení nebo metod pro překlad adres. Dále se v tomto textu bude termínem NAT rozumět zařízení implementující metodu Network address port translation (NAPT), která je definována v RFC 3022 [14]. NAPT umožňuje více zařízením v privátní síti sdílet jednu veřejnou IP adresu, například v síti internet. NAPT je nejrozšířenějším typem NAT. Klienti za zařízením NAT nemají permanentně viditelné veřejné porty, na které by mohly být směrovány příchozí spojení od ostatních klientů v internetu. [15] To je důvod, kvůli kterému je proces ustanovení spojení mezi klienty za NAT znesnadněn.

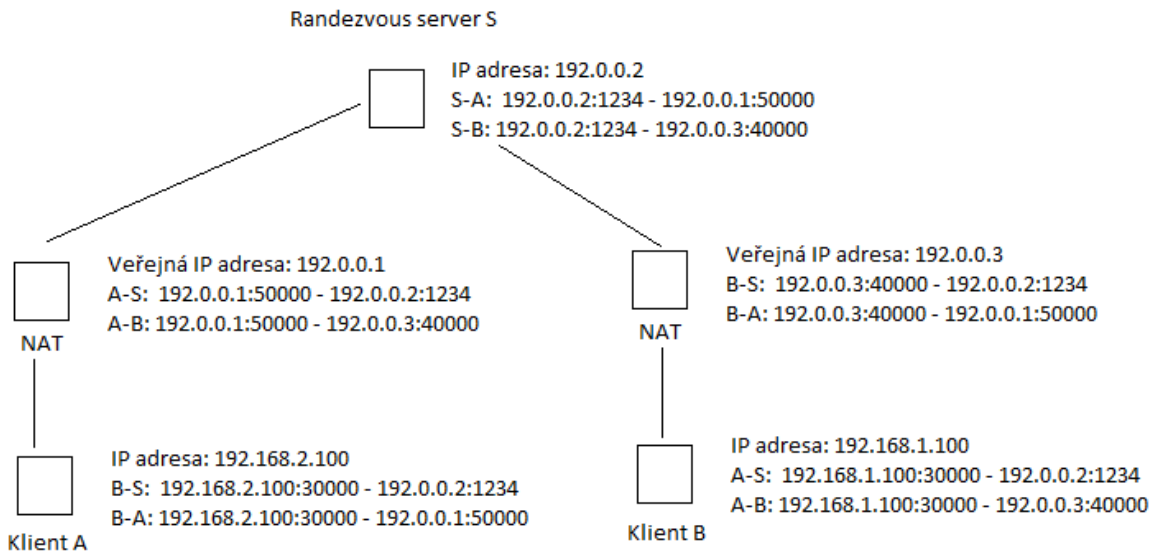
Endpoint je n-tice reprezentující jedno z komunikujících zařízení v rámci jedné pobíhající komunikace. V případě protokolů UDP nebo TCP je to dvojice (IP adresa, port). [15] Při iniciování spojení zařízením `A` za NAT se zařízením `B` ve veřejné síti, je endpoint zařízení `A` mapován na veřejnou adresu a některý z portů zařízení NAT. Toto mapování zajišťuje, že pakety posílané zařízením `B` jsou správně přeloženy a přeposlány zařízení `A`. [15] Chování zařízení NAT při překladu se může pro konkrétní typy NAT lišit. Typy zařízení NAT lze definovat na základě způsobu dříve zmíněného mapování endpointů a dále filtrování endpointů. Jsou rozlišovány dva způsoby mapování: endpoint-dependent a endpoint-independent mapping. Při endpoint-independent mapping zachovává NAT mapování pro pakety odesílané ze stejné vnitřní IP adresy a portu nezávisle na vnější IP adrese a portu. Endpoint-dependent mapping při mapování zohledňuje vnější IP adresu a port. Při filtrování jsou zahazovány pakety přicházející z veřejné sítě, pokud nejsou určeny pro vnitřní endpoint, pro který bylo dříve vytvořeno mapování. Možné způsoby filtrování endpointů jsou definovány obdobně jako v případě mapování endpointů: endpoint-independent filtering a endpoint-dependent filtering. Při endpoint-dependent filtering jsou navíc zahazovány pakety z veřejné adresy Y (a portu y) určené pro vnitřní endpoint X:x, pokud z X:x nebyl dříve odeslán paket na veřejnou adresu Y (může být zohledňován i port y). [15]

Existuje více způsobů jak umožnit komunikaci mezi zařízeními za NAT. Techniky, které toto umožňují, se nazývají NAT Traversal. Jednou z možných metod je použití tzv. relay serveru ve veřejné síti, který by sloužil jako prostředník, který by zprávy klientů přeposílal. Takové řešení by nutně

snižovalo propustnost komunikace. Toto řešení je zároveň jediné, které bude vždy zaručeně fungovat, pokud budou mít obě zařízení za NAT přístup k relay serveru. Žádná z ostatních metod nemůže zaručit funkčnost pro všechny typy NAT.

3.2.1 UDP Hole Punching

Technika UDP Hole Punching využívá vlastností Endpoint-Independent Mapping NAT (EIM-NAT) zařízení. Dále bude uveden příklad ustanovení komunikace pomocí této metody mezi dvěma klienty za různými zařízeními NAT. Pro veškerou komunikaci musí být na transportní vrstvě použit UDP protokol. Klienti `A` a `B` se nejprve spojí s tzv. rendezvous serverem S. Rendezvous server slouží k vzájemnému nalezení klientů a poskytnutí jejich veřejných IP adres a portů. Chce-li klient A zahájit komunikaci s klientem B, zašle žádost serveru S. Server tuto žádost přepošle klientovi B, s kterým má otevřeno spojení. Klient A začne odesílat pakety na veřejnou adresu klienta B, kterou obdržel od serveru, a naopak klient B začne odesílat pakety na veřejnou adresu a port klienta A. Klienti musí použít stejný zdrojový port, který použili při komunikaci se serverem S, aby bylo zachováno mapování, které bylo použito pro komunikaci se serverem. Předpokládejme, že zařízení NAT jsou typu endpoint-dependent filtering, a že paket odeslaný klientem A se dostane k NAT zařízení klienta B dříve než paket odeslaný klientem B. V tomto případě ještě není v NAT klienta B vytvořeno pravidlo pro překlad a paket je zahozen. Paket odeslaný klientem B už ale NAT klienta A přijme, jelikož je pro něj vytvořené překladové pravidlo, a správně ho doručí klientovi A. V toto chvíli je ustanovení spojení dokončeno a klienti spolu mohou začít komunikovat. [15] V obrázku je ilustrován konkrétní případ mapování adres zařízeními NAT a endpointy probíhající komunikace.



Obrázek 3.1

Dále bude uvažován případ, kdy jsou dva klienti připojeni v stejné síti LAN a ke stejnému zařízení NAT. Aby byl výše uvedený proces ustanovení spojení mezi těmito klienty úspěšný, musí NAT podporovat takzvaný hairpinning. Hairpinning zajišťuje, že je-li pro klienta A přidělen endpoint (A',a') na vnější straně NAT, NAT musí zaručit, že pakety posílané klientem B na tento endpoint budou správně doručeny klientovi A. Ne všechny zařízení NAT hairpinning implementují. I v případě, že by

NAT implementoval hairpinning, bylo by výhodnější ustanovit přímé spojení mezi klienty. Toto je v metodě UDP hole punching řešeno tak, že klient při své registraci u rendezvous serveru odešle i údaje o svých privátních adresách a portech. Ostatní klienti tak mohou obdržet i tyto adresy a ve fázi odesílání paketů na své veřejné adresy a porty zároveň odesílat pakety i na tyto privátní adresy a porty. Pokud se podaří ustanovit mezi klienty více než jedno spojení, klienti se mohou dohodnout na nejvhodnějším z nich. [15]

Další metodou pro překonávání zařízení NAT je metoda TCP hole punching, která je metodě UDP hole punching podobná. Při otevírání spojení je třeba, aby oba klienti odesílali pakety SYN. Úspěch této metody závisí na konkrétní implementaci zařízení NAT. [15] V [16] jsou uvedeny výsledky testování úspěšnosti metod UDP/TCP hole punching s různými zařízeními NAT. Metoda UDP hole punching byla úspěšná u více než 80% testovaných zařízení. Metoda TCP hole punching byla úspěšná na více než 60% zařízení.

3.2.2 Teredo

Teredo je technologie, která zprostředkovává IPv6 spojení mezi klienty za jedním nebo více zařízeními NAT tunelováním paketů skrze IPv4 síť. Součástí infrastruktury jsou Teredo klienti, Teredo server a Teredo relay. Chce-li klient využít služby Teredo, musí podporovat jak IPv4, tak IPv6. [17] Překonávání zařízení NAT je založeno na technice UDP Hole Punching, kde Teredo server slouží jako rendezvous server. Teredo relay slouží v roli IPv6 routeru mezi Teredo službou a nativní IPv6 sítí.

Teredo bylo vyvinutu firmou Microsoft a standardizováno IETF v RFC 4380. Podpora pro Teredo je implementována v operačním systému Windows od verze XP s SP2. [17]

Miredo[12] je open-source implementace Teredo technologie pro operační systémy Linux a BSD. Miredo implementuje všechny komponenty Teredo specifikace, tedy klienta, relay a server.

3.2.3 UDT

UDT (UDP-based Data Transfer Protocol) je protokol aplikační vrstvy pro spolehlivý přenos dat využívající UDP jako transportní protokol. Tento protokol byl vyvinut především pro vysokorychlostní přenos dat skrze sítě WAN. Jelikož je postaven na UDP protokolu a zajišťuje spolehlivou komunikaci, lze ho s výhodou použít při aplikaci metody UDP hole punching. Protokol UDT kromě ustanovení tradičního spojení typu klient-server umožňuje ustanovit spojení pomocí tzv. rendezvous módu, kdy se spolu oba klienti snaží aktivně spojit. [18]

UDT protokol je implementován ve stejnojmenné open-source knihovně napsané v jazyce C++. UDT.Net[13] je knihovna pro .NET framework, která obaluje nativní API[14] UDT knihovny. Obě knihovny umožňují implementovat vlastní mechanizmus pro předcházení zahlcení.

3.3 .NET

.NET je rozhraní vyvinuté firmou Microsoft. Skládá se z běhového prostředí Common Language Runtime (CLR) a knihovny tříd. Při vývoji aplikací pro .NET může být využito více jazyků, například C#, Visual Basic nebo C++. Všechny jazyky sdílí stejný typový systém, který definuje základní datové typy, vyznačují se objektovým přístupem, mají přístup k stejným knihovnám tříd .NET frameworku a jsou překládány do mezikódu, tzv. managed code. [19]

[12] Nástroj je dostupný na adrese http://www.remlab.net/miredo/.
[13] Nástroj je dostupný na adrese https://code.google.com/p/udt-net/.
[14] Application programming interface (API) specifikuje rozhraní softwarové komponenty.

Běhové prostředí CLR obstarává mimo jiné správu paměti, překlad managed kódu do nativního kódu a přístup ke knihovnám tříd .NET Frameworku. Garbage collector je součást systému správy paměti, která automaticky uvolňuje prostředky alokované objektům v paměti, na které již neexistují reference. Překlad managed kódu je zajištěn pomocí takzvané just-in-time[15] (JIT) kompilace, obdobně jako v běhovém prostředí Javy. [20] Součástí standartní instalace OS Windows 8 je .NET Framework ve verzi 4.5, ve Windows 7 je to verze 3.5.

3.4 Zabezpečení komunikace

Pro zabezpečení komunikace je nutné zajistit vzájemnou autentizaci klientů. Data posílaná po síti by měla být zašifrována, aby bylo zabráněno případnému man-in-the-middle útoku.

3.4.1 AES

Advanced Encryption Standard (AES) je algoritmus používaný k šifrování elektronických dat standardizovaný National Institute of Standards and Technology (NIST). Jedná se o symetrickou blokovou šifru šifrující/dešifrující bloky dat o velikosti 128 bitů použitím klíče délky 128, 196 nebo 256 bitů. AES je založen na algoritmu zvaném Rijndael. [21]

Operační systém Windows nabízí implementaci AES algoritmu skrze své API zvané CryptoAPI. [22] Knihovna tříd .NET obsahuje třídu AesCryptoServiceProvider, která využívá implementace CryptoAPI, a třídu AesManaged, která je implementací AES algoritmu v managed kódu.

3.4.2 TLS/SSL

Cílem Transport Layer Security (TLS) protokolu je především zabezpečit komunikaci mezi dvěma aplikacemi proti odposlechu nebo podvrhnutí zpráv a je nástupcem protokolu Secure Sockets Layer (SSL). Protokol se skládá ze dvou vrstev, TLS Record protokolu a TLS Handshake protokolu. Celý TLS protokol leží mezi transportní a aplikační vrstvou. Požadavkem na protokol v transportní vrstvě je, aby umožňoval spolehlivý přenos dat. TLS Record protokol se nachází přímo nad transportní vrstvou a zajišťuje bezpečnou komunikaci pomocí symetrické kryptografie (například AES). Pro každé nové spojení je generován nový symetrický klíč pro šifrování dat. Tento klíč je generován na základě tajného klíče, na kterém se obě strany dohodnou pomocí TLS Handshake protokolu (může být použit i jiný protokol). TLS Handshake protokol slouží k dohodnutí algoritmu a klíče pro šifrování dat. Protokol zprostředkovává vzájemnou autentizaci obou stran pomocí asymetrické kryptografie. [23]

Schannel je implementace SSL a TLS protokolů dostupná v operačním systému Windows skrze Security Support Provider Interface (SSPI) API. [24] SslStream je třída dostupná v knihovně tříd .NET, která využívá Schannel.

Součástí Microsoft Software Development Kit (SDK) jsou nástroje pro vytváření a úpravu certifikátů [22]:

- MakeCert.exe – nástroj pro vytváření X.509 certifikátů.
- Cert2SPC.exe – nástroj pro vytvoření Software Publisher Certificate (SPC) souboru z existujících X.509 certifikátů.
- Pvk2pfx.exe – nástroj pro vytvoření Personal Information Exchange (PFX) souboru ze SPC souboru a Private Key (.pvk) souborů.

[15] Just-in-time je kompilace probíhající v době běhu programu.

3.5 Přenos dat po síti

Data posílaná mezi klienty musí mít předem stanovený formát. Aplikace může být později rozšířena na další platformy, pokud je takový formát přesně definován.

3.5.1 Protocol Buffers

Protocol Buffers je mechanizmus určený pro binární serializaci datových struktur. Je nezávislý na programovacím jazyku i platformě. Uživatel musí nejdříve definovat datové struktury, které chce později serializovat. Ty jsou uloženy do souboru s příponou .proto. Tento soubor je potom vstupem pro kompilátor, který vytvoří definice tříd pro serializaci, deserializaci a přístup k datům struktur. Kompilátory .proto souborů jsou dostupné pro jazyky C++, Java a Python. [25] Existuje množství dalších kompilátorů pro ostatní jazyky, které byly vyvinuty třetími stranami. Protobuf-net[16] je jedním z nich a implementuje Protocol Buffers pro platformu .NET.

3.6 Sledování změn ve složce

Aby mohla aplikace ihned zareagovat na změnu v synchronizované složce, ať už způsobenou uživatelem nebo jinou spuštěnou aplikací, je vhodné využít některou funkci systému pro sledování změn ve složce.

API operačního systému Windows umožňuje aplikacím sledovat změny ve složce a případně i v jejích podsložkách. Pomocí API funkce FindFirstChangeNotification lze specifikovat, které operace je třeba sledovat, v které složce a zda mají být sledovány i operace v podsložkách. Funkce vrací manipulátor, který může být použit v takzvaných wait funkcích. Je-li potřeba čekat na oznámení o změně ve složce, je zavolána některá z wait funkcí, které je manipulátor předán. Vlákno aplikace je tak blokováno, dokud nedojde k sledované operaci. [26]

Knihovna tříd .NET Frameworku nabízí třídu FileSystemWatcher [27], kterou lze použít ke stejnému účelu. Instanci této třídy lze nastavit cestu k monitorované složce, operace, které mají být sledovány a funkce pro obsluhu událostí vzniklých provedením změny ve složce. Jsou rozlišovány operace vytvoření, změny, odstranění a přejmenování/přesunutí. Zachytí-li systém některou z těchto operací, vytvoří pro ni událost. Funkce, které obsluhují tyto události, obdrží kromě typu provedené operace i cestu, v které byla provedena.

3.7 Jednoznačná identifikace zařízení

Synchronizovaná zařízení by měla mít přiděleno určité jedinečné číslo pro jejich identifikaci. Universal Unique Identifier (UUID) je identifikátor jednoznačný, jak v čase, tak v prostoru. Pro jeho vygenerování není potřeba registrační autority. Popis generování tohoto identifikátoru je specifikován v [28]. GUID je implementací UUID používanou v operačním systému Windows. [29]

[16] Knihovna je dostupná na adrese https://code.google.com/p/protobuf-net/.

4 Návrh

V této kapitole bude popsán návrh nástroje pro synchronizaci dat v OS Windows. Bude popsána architektura celého systému, způsob zabezpečení komunikace a samotný proces synchronizace.

4.1 Specifikace požadavků

Seznam požadavků na výslednou aplikaci pro synchronizaci souborů:

- Podpora běhu na operačním systému Windows.
- Synchronizace mezi klienty připojenými k síti internet.
- Uživatelova data nesmí být ukládána na centrální server.
- Zajištění spojení klientů za zařízeními NAT.
- Zajištění bezpečné komunikace po síti.
- Jednoduchost ovládání.

4.2 Uživatelské rozhraní

Uživatelské rozhraní by mělo být co nejjednodušší a přitom nabízet veškeré potřebné funkce. Uživatel by měl mít možnost specifikovat složku, která bude synchronizována, pozastavit synchronizaci a být informován o stavu synchronizace.

Synchronizace by měla probíhat na pozadí. Uživatel může být informován o stavu synchronizace prostřednictvím notifikací v notifikační oblasti systému Windows. Z kontextového menu notifikační ikony může být umožněn přístup k nastavení aplikace a pozastavení synchronizace.

4.3 Synchronizace

Vzhledem k tomu, že by synchronizace měla probíhat automaticky bez uživatelova explicitního zásahu, je potřeba určit, kdy a s kým má být proces synchronizace zahájen. Jednoduše řečeno, proces synchronizace by měl být zahájen po každé změně v synchronizované složce na libovolném zařízení se všemi dostupnými zařízeními. K tomu bude zapotřebí mechanizmus pro sledování změn v synchronizované složce a efektivní způsob, jakým by klienti rozpoznali, že jsou data na všech replikách identická. Proces synchronizace bude založen na práci [13].

4.3.1 Verze repliky

Efektivním způsobem, jakým by klient rozpoznal, že jsou data na všech replikách identická, by mohlo být udržování čísla verze repliky. Verze by měla určitým způsobem zachycovat stav repliky. Pokud by bylo každé zařízení identifikováno jednoznačným klíčem a jeho replika by měla přiděleno číslo, které by vyjadřovalo počet změn provedených na této replice, tak množina dvojic (klíč, počet_změn) by vyjadřovala stav replik po provedení synchronizace všech uživatelových replik. Verze tedy bude reprezentována jako tato množina. Každý klient si bude udržovat záznam o své verzi repliky. Při každé provedené změně na jeho replice zvýší svoje číslo ve verzi. Po synchronizaci s jiným klientem budou mít oba klienti stejnou verzi zachycující změny provedené na obou synchronizovaných replikách. Změnou v replice může být myšleno například provedení jedné operace, ale může také

vyjadřovat posloupnost několika operací. Důležité je, že pokud v replice dojde ke změně, číslo této repliky ve verzi musí být zvětšeno před spuštěním další synchronizace.

Replika	Verze	=> změna v A a B	Replika	Verze
A	A:0; B:0; C:0		A	A:1; B:0; C:0
B	A:0; B:0; C:0		B	A:0; B:1; C:0
C	A:0; B:0; C:0		C	A:0; B:0; C:0

Replika	Verze	=> synchronizace A s B	Replika	Verze
A	A:1; B:0; C:0	poslední synchronizovaný stav:	A	A:1; B:1; C:0
B	A:0; B:1; C:0	A:0; B:0; C:0	B	A:1; B:1; C:0
C	A:0; B:0; C:0		C	A:0; B:0; C:0

Tabulka 4.1 Ilustrace způsobu přidělení verzí replikám

Pokud by každý klient znal verzi ostatních klientů dostupných v síti, mohl by zahájit proces synchronizace pokaždé, pokud by zjistil, že existuje klient s jinou verzí.

4.3.2 Definování pravidel pro řešení kolizí

Označme množinu všech možných dvojic operací, které může algoritmus pro usmíření detekovat ve chvíli, kdy bude rozhodovat o tom, zda jsou v konfliktu, jako množinu `O`. Potom vybereme-li všechny dvojice operací z množiny `O`, které jsou konfliktní, dostaneme těchto sedm dvojic operací:

```
1.Create(π, X); Create(π, Y)
2.Create(π/π', X); Remove(π)
3.Create(π/π', X); Edit(π, File(Y))
4.Edit(π, X); Remove(π)
5.Edit(π, X); Edit(π, Y)
6.Edit(π/π', X); Remove(π)
7.Edit(π/π', X); Edit(π, File(Y))
```

Pro konfliktní dvojice lze přesně definovat, jak budou operace upraveny. Konflikty budou řešeny tak, jak bylo popsáno v části věnované synchronizačním algoritmům, tedy úpravou konfliktních operací na nekonfliktní:

- Řešení pro 1. a 5. dvojici: k názvům souborů budou připojeny suffixy s upozorněním na konflikt.
- Řešení pro 2., 4. a 6. dvojici: operace `remove` bude zrušena.
- Řešení pro 3. a 7. dvojici: Operace `edit` je změněna na operaci `create` a k názvu souboru je připojen suffix s upozorněním na konflikt.

Jelikož jsou takto upravovány operace, které již byly na jedné z replik provedeny, musí být pro každou takovou repliku vytvořena další sekvence operací, která ji, před samotným aplikováním sekvence smířených operací, uvede do stavu reflektujícího úpravu operací. Tyto sekvence mohou obsahovat i operace pro přesunutí/přejmenování.

4.3.3 Definice operace

Usmiřovatel během své činnosti zjišťuje, zda již operace nebyla provedena v druhé replice a v takovém případě ji nepropaguje. Toto vyžaduje, aby v případě operace `create` nebo `edit` souboru

byl součástí definice operace i obsah tohoto souboru. Namísto celého obsahu souboru může být v definici operace použit pouze hash souboru.

Předpokládejme, že by uživatel ve dvou různých replikách vytvořil v synchronizované složce stejný soubor. Pokud by součástí definice operace nebyl hash obsahu souboru, usmiřovatel by detekoval kolizní pár operací a zbytečně by tuto kolizi řešil přenášením identických souborů z jedné repliky do druhé, a chybnému vytvoření druhého identického souboru se sufixem informujícím o konfliktu. Tento problém lze odstranit, pokud by si klient v případě detekování takového konfliktu dodatečně hash obsahu souboru vyžádal.

Vypočtení hash obsahu souboru je časově náročná operace. Lze předpokládat, že k výše uvedené situaci bude docházet zřídka, proto by byl přístup s dodatečným vyžádáním hashe efektivnějším řešením. Z důvodu jednodušší implementace bude použit přístup první, tedy součástí definice operace vytvoření nebo úpravy souboru bude i hash obsahu tohoto souboru. Pro vypočtení tohoto hashe bude použit algoritmus MD5.

4.3.4 Ukládání otisků verzí

Pamatování si otisků všech předchozích verzí repliky klienta může být paměťově náročné. Proto bude dovoleno, aby klient určitým způsobem zahazoval starší otisky. V případě, že by dva klienti během synchronizace nenalezli potřebný otisk, lze za poslední synchronizovaný stav považovat prázdnou repliku. Prázdná replika je stále validní stav v minulosti, kdy byly obě repliky synchronizovány, takže lze vygenerovat minimální seznamy operací a předat je algoritmu usmiřovatele. Důležité je, že vygenerované minimální seznamy operací budou obsahovat pouze operace vytvoření, čímž je zaručeno, že propagováním těchto operací nebudou uživatelovi chybně odstraněna některá data.

Příkladem můžou být klienti `A` a `B`, jejichž repliky po poslední synchronizaci obsahují jeden soubor. Předpokládejme, že je otisk těchto replik „ztracen“, soubor u klienta A je odstraněn a je spuštěn další proces synchronizace. Jako poslední synchronizovaný stav bude zvolena prázdná replika. Výsledkem synchronizace bude znovuvytvoření souboru u klienta A, namísto správného odstranění souboru u klienta B. Takovým situacím by se však mělo předcházet.

4.3.5 Celý proces synchronizace

Detekuje-li klient změnu v synchronizované složce, nebo zjistí-li, že je dostupný klient s jinou verzí repliky, zahájí proces synchronizace. V prvním případě může být synchronizace zahájena s libovolným dostupným klientem. V druhém případě s klientem s rozdílnou verzí repliky.

Synchronizace nebude úspěšná, pokud aplikace klienta nebude mít možnost provést některou z operací pro usmíření replik. Taková situace může nastat, například pokud má uživatel některý ze souborů otevřený. Dalším důvodem může být, že během procesu synchronizace provede uživatel zásah do repliky. V případě, že pouze vytvoří nové soubory nebo složky v replice, není to problém. Jakákoli jiná operace problémem je a měla by být klientem detekována. Je-li synchronizace úspěšná, klienti aktualizují číslo verze své repliky a uloží si její otisk. V případě, že úspěšná nebyla a klient již provedl některé z operací pro usmíření replik, klient bude tyto změny interpretovat jako změny, které by normálně provedl uživatel, tedy zvýší své číslo ve verzi. Klient musí zvýšit své číslo ve verzi i v případě, že budou detekovány kolizní operace, budou upravovány. To proto, že tímto došlo ke změně na jeho replice. Ostatní klienti tak potom mají možnost tuto změnu detekovat.

4.4 Síťová architektura

Pro vzájemné nalezení klientů bude zvoleno řešení využívající centrální server podobný trackeru v BitTorrent protokolu. Kromě samotného nalezení ostatních klientů je potřeba i mechanizmus pro překonávání zařízení NAT. Bude-li k dispozici dříve zmíněný server, lze ho relativně jednoduše použít jako rendezvous server pro navázání spojení mezi klienty metodou UDP Hole Punching. Pro tuto metodu musí být použit protokol UDP. Pro zajištění spolehlivé komunikace může být použito protokolu UDT. Tento protokol bude použit pro veškerou komunikace mezi klienty nebo mezi klientem a serverem.

Skupina klientů, mezi kterými bude probíhat synchronizace, musí být určitým způsobem identifikována. K tomu může sloužit klíč vygenerovaný z uživatelského jména a hesla, které zadá uživatel. Heslo uživatele bude po jeho zadání uloženo ve formě SHA-256 otisku. Klíč pro identifikaci uživatele bude generován takto: k uživatelskému jménu bude připojen řetězec číslic v šestnáctkové soustavě reprezentující otisk hesla. Klíč pro identifikaci uživatele bude SHA-256 otisk tohoto výsledného řetězce. Každý klient (zařízení) bude identifikován jednoznačným 128bitovým číslem (bylo naznačeno v podkapitole 4.3.1), který si při prvním spuštění vygeneruje. Tímto číslem bude GUID zmíněný v podkapitole 3.7.

Server si o každém klientovi povede záznam, obsahující identifikační klíč jeho skupiny, identifikační číslo samotného klienta, verzi jeho repliky, jeho veřejnou IP adresu a port a seznam jeho privátních IP adres a portů, které použije pro navázání spojení s ostatními klienty.

4.4.1 Zabezpečení komunikace

Komunikace mezi klientem a serverem bude zabezpečena pomocí protokolu TLS. Pro serverovou aplikaci bude vygenerován certifikát podepsaný sebou samým a privátní klíč. Privátní klíč bude uložen ve formě souboru PFX společně se souborem aplikace serveru. Aby mohla být identita serveru klientem ověřena, bude certifikát uložen mezi důvěryhodné certifikační autority v operačním systému, na kterém bude spouštěna aplikace klienta.

Komunikace mezi klienty bude zabezpečena šifrováním posílaných zpráv pomocí AES. Každá komunikace mezi dvěma klienty je iniciována skrze server. Server pro každou takovou komunikaci vygeneruje nový klíč, který předá klientům. Klienti tento klíč použijí pro šifrování a dešifrování zpráv. Pro jakoukoli posílanou zprávu nebo část zprávy bude vygenerován nový inicializační vektor IV, pomocí tohoto vektoru a klíče bude zpráva zašifrována. Před samotnou zašifrovanou zprávu bude připojena délka zašifrované zprávy a inicializační vektor. Délka zprávy bude zakódována jako 4B unsigned integer v kódování big-endian.

Délka zašifrované zprávy (4B)	IV (16B)	Zašifrovaná zpráva

Tabulka 4.2 Formát šifrované zprávy

4.4.2 Protokol

Zprávy budou serializovány pomocí Protocol Buffers. Jelikož data, která jsou výsledkem serializace zpráv pomocí Protocol Buffers, neobsahují údaj o délce dat, musí být tento údaj před každou takto serializovanou zprávu přidán. Tento údaj bude zakódován stejně jako v případě šifrování zpráv pomocí AES. Každá odeslána zpráva se skládá z údaje o velikosti zprávy, serializované zprávy typu `Message` a popřípadě zprávy s daty.

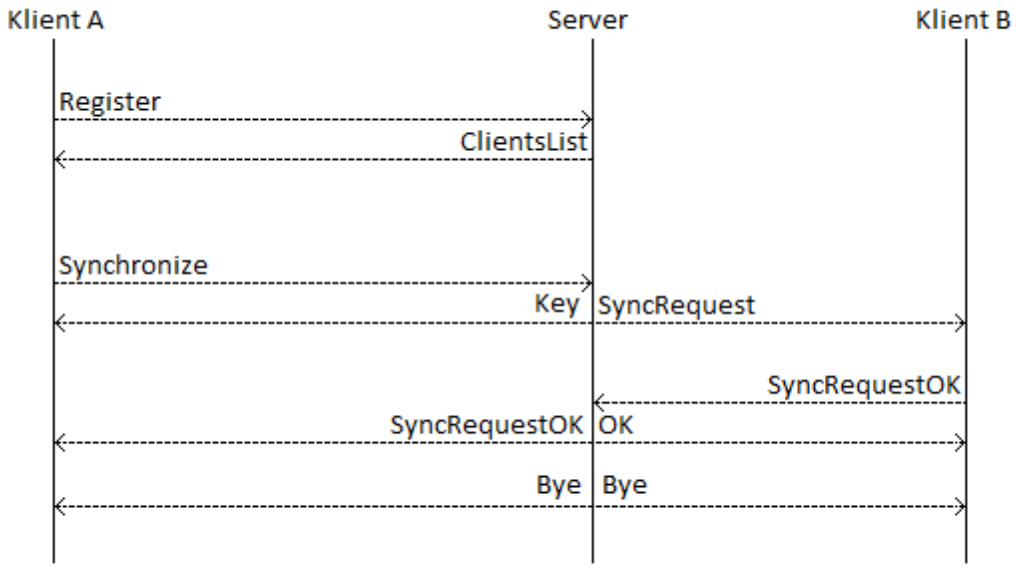
Délka zprávy (4B)	Message		Data
	MessageType	DataLength	

Tabulka 4.3 Formát zprávy

Délka zprávy uvádí délku serializované zprávy `Message`. Zpráva `Message` obsahuje typ zprávy a délku dat. Data mohou být buďto jedna ze serializovaných zpráv typu data, nebo obsah souboru. V příloze C je blíže specifikováno, jaký typ zprávy je asociován s jakými daty.

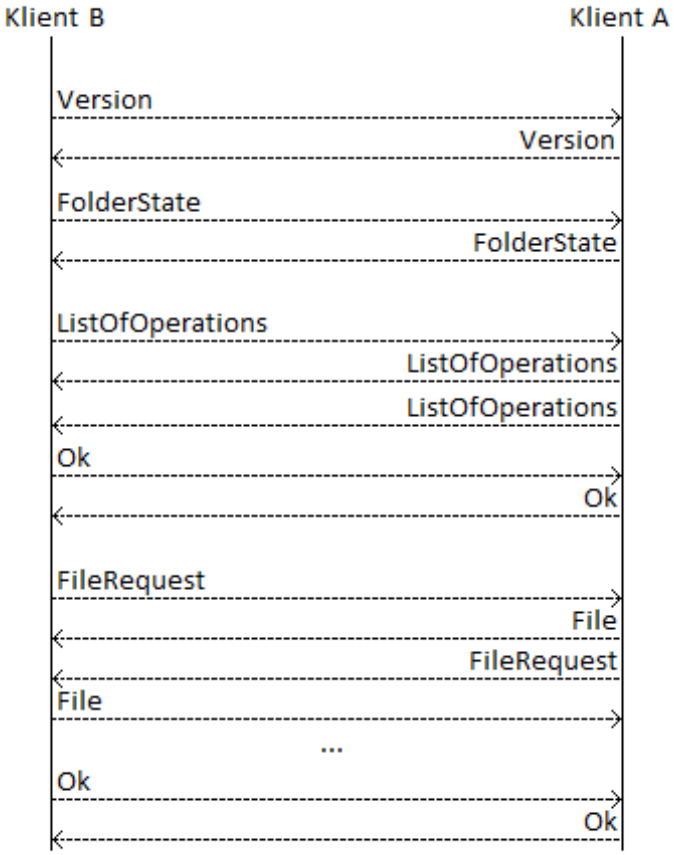
Komunikace mezi klientem a serverem by měla být pokud možno minimální. Klient se serverem otevře spojení a odešle své registrační údaje (zpráva `Register`). Server na to odpoví seznamem dostupných klientů v jeho skupině obsahujícím i seznamy jejich adres a portů (zpráva `ClientsList`). V tuto chvíli je proces registrace považován za dokončený. Klient může dále kdykoli poslat zprávu `UpdateInfo`, pokud se změní verze jeho repliky. Server na ni odpoví zprávou `ClientsList`.

Ve chvíli, kdy chce klient `A` navázat spojení s jiným klientem `B`, zašle serveru žádost o toto spojení (`Synchronize`). Je-li klient `B` dostupný, server vygeneruje nový klíč, pro šifrování komunikace mezi klienty, který odešle zpět klientovy `A` (zpráva `Key`), a také klientovy `B` současně s přeposláním žádosti o spojení (zpráva `SyncRequest`). Klient `B` informuje server o tom, že žádost přijímá (zpráva `SyncRequestOk`), odešle zprávu Bye a odpojí se od serveru a začne odesílat pakety na veřejnou adresu a port klienta `A`, které obdržel od serveru, a naslouchat příchozím spojení na portech, které serveru odeslal při registraci. Server potvrzení přeposílá klientovi `A`, který se následně také odpojí od serveru a snaží se spojit s klientem `B` odesíláním paketů na jeho adresy a porty. Při odesílání paketů na veřejnou adresu a port musí oba klienti jako zdrojový port použít ten, který předtím použili pro první spojení se serverem. I v případě, že jsou oba klienti za různými zařízeními (EIM) NAT, měli by být v tuto chvíli spojeni.



Obrázek 4.1 Příklad komunikace mezi serverem a klienty

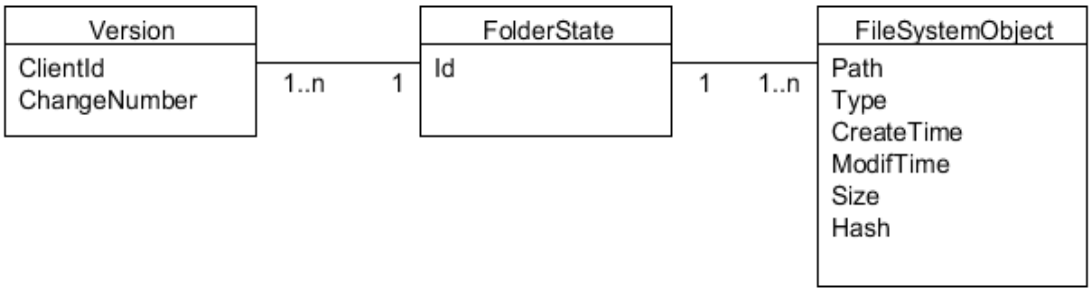
Klient, který zaslal žádost o synchronizaci, dále klient `A`, bude zodpovědný za provedení samotného algoritmu pro usmíření. Druhý klient bude označen jako klient `B`. Jsou-li klienti spojeni, zašlou si navzájem zprávu se svojí aktuální verzí (zpráva `Version`) a nejnovější otisk stavu repliky (zpráva `FolderState`), jehož verze není novější než kterákoli s aktuálních verzí obou klientů. Klienti porovnají obdržený otisk s tím, který obdrželi a v případě, že se liší, budou jako poslední synchronizovaný stav uvažovat prázdnou repliku. Pokud by klienti použili chybné nebo rozdílné stavy pro následné odvození sekvence provedených operací a proces usmíření by uspěl, vygenerované operace pro usmíření by mohly vést ke ztrátě uživatelových dat. Tento krok zde tedy slouží jako určitá pojistka. Klienti se nyní shodly na stavu v minulosti, ze které vygenerují minimální sekvence operací. Klient `B` odešle svůj seznam klientovi `A` (zpráva `ListOfOperations`). Klient `A` provede algoritmus pro usmíření replik, jehož výsledkem budou dvě dvojice seznamů operací pro oba klienty – seznam operací reflektující změny provedené při řešení konfliktů a seznam operací druhého klienta, které mají být propagovány. Klient `A` odešle klientovi `B` seznam operací (zpráva `ListOfOperations`), které je třeba provést na jeho replice. Klienti začnou provádět operace na svých replikách. V případě operace vytvoření nebo úpravy souboru, si klient obsah tohoto souboru vyžádá od druhého klienta (zpráva `FileRequest`), který mu obsah tohoto souboru pošle (zpráva `File`). Klienti se o dokončení operací informují (zprávou `Ok`) a znovu si pro kontrolu vymění otisk stavu svých replik.



Obrázek 4.2 Příklad komunikace mezi klienty

4.5 Uložení dat

Klient musí ukládat otisky minulých verzí repliky, otisk současného stavu repliky a nastavení, zahrnující mimo jiné uživatelské jméno a heslo, jednoznačný klíč identifikující klienta a cestu k synchronizované složce. Otisky minulých verzí repliky by měli být ukládány do databáze.



Obrázek 4.3 ER diagram databáze

5 Implementace

Aplikace klienta i serveru byla implementována v jazyce C# pro platformu .NET verze 4.5. Dále byly použity tyto knihovny:

- UDT.Net – knihovna pro komunikaci pomocí UDT protokolu.
- protobuf-net – knihovna pro serializování zpráv pomocí ProtocolBuffers.

Výsledné řešení se skládá z těchto knihoven a aplikací:

- `PunchCommonTypes` – knihovna definující třídy používané klientskou i serverovou částí aplikace.
- `Protocol` – knihovna definující třídy implementující navazování spojení, zabezpečenou komunikaci a přenos zpráv.
- `Synchro` – knihovna implementující samotný synchronizační algoritmus.
- `PunchSync` – knihovna implementující chování klienta.
- `MiddleMan` – konzolová aplikace serveru.
- `Client` – Windows Forms aplikace klienta.

5.1 Klient

Klient musí umožnit interakci s uživatelem, sledovat změny v synchronizované složce, zahájit synchronizaci s jiným klientem a reagovat na požadavky na synchronizaci od jiných klientů. Interakci s uživatelem zajišťuje hlavní vlákno aplikace. Aplikace vytvoří další dvě vlákna. Jedno pro komunikaci se serverem a ostatními klienty (komunikační vlákno), další pro monitorování změn v synchronizované složce (monitorovací vlákno). Po spuštění aplikace je zaregistrována notifikační ikona s kontextovým menu, z kterého je přístupné hlavní okno aplikace. Pokud je aplikace spuštěna poprvé, je vygenerováno nové GUID pro identifikaci klienta a otevře se okno pro nastavení uživatelského jména, hesla a cesty k synchronizované složce. Hlavní vlákno aplikace vytvoří komunikační vlákno, které potom vytvoří monitorovací vlákno. Komunikační vlákno informuje hlavní vlákno o svém stavu pomocí událostí. Stejně tak informuje monitorovací vlákno synchronizační vlákno v případě, že detekuje změnu ve složce.

.NET Framework od verze 4.0 nabízí strukturu `CancellationToken`, již lze použít pro ukončení asynchronních operací. Objektem třídy `CancellationTokenSource` je vytvořen objekt `CancellationToken`, který lze předávat nově vytvořeným vláknům. Vlákno, které tento objekt obdrží, ho může kopírováním předávat dál. Původním objektem třídy `CancellationTokenSource` lze signalizovat, že mají být operace ukončeny. Je v režii kódu každého vlákna, který `CancellationToken` obdržel, aby kontroloval, zda nebyla signalizována operace zrušení. `CancellationToken` bude použit pro ukončení činnosti ostatních vláken z hlavního vlákna aplikace.

Komunikační vlákno je implementováno třídou **`SyncPeer`** v knihovně `PunchSync`. Pro komunikaci se serverem i klienty je použita třída `Protocol.Socket` z knihovny `Protocol`. Algoritmus tohoto vlákna se chová tak, jak bylo uvedeno dříve v části věnované protokolu, s jednou výjimkou. Z důvodu snazší implementace není implementováno obousměrné posílání obsahu souborů. Klient, který byl iniciátorem synchronizace, začne o soubory žádat jako první. Jakmile skončí, informujíce o tom druhého klienta zprávou `Ok`, začne o soubory žádat druhý klient. Aby byly operace

pro usmíření monitorovacím vláknem ignorovány, monitorovací vlákno je po dobu samotné synchronizace pozastaveno.

Nastavení a verze repliky jsou ukládána jako uživatelská nastavení pomocí Application Settings. Application Settings je funkce Visual Studia pro ukládání aplikačních a uživatelských nastavení aplikace. Jednotlivým nastavením Application Settings je nutné definovat jméno a typ. Typem může být kterýkoli datový typ, který lze serializovat pomocí XML. Tyto nastavení jsou ukládána do dvou XML souborů ve složce AppData uživatele.

Komunikační vlákno o svém stavu informuje vlákno hlavní pomocí událostí. Funkce, které je obsluha těchto událostí delegována, je prováděna v komunikačním vlákně. Na tyto události je potřeba reagovat aktualizováním vlastností vizuálních komponent v hlavním okně aplikace, ale k těmto vlastnostem lze přistupovat pouze z vlákna, v kterém byly vytvořeny. To je vyřešeno voláním metody `BeginInvoke` vizuální komponenty, která provede asynchronní vykonání funkce ve vlákně, v kterém byla komponenta vytvořena.

5.1.1 Monitorování složky

Pro sledování změn ve složce byla použita třída .NET Frameworku `FileSystemWatcher`. Třída implementuje veřejnou metodu `MonitorFolder`, která je spuštěna v monitorovacím vlákně. Aby bylo možné určit, zda od posledního ukončení aplikace nedošlo v synchronizované složce ke změně, algoritmus této metody nejdříve načte poslední otisk stavu složky ze souboru. Data v tomto souboru jsou serializována pomocí ProtocolBuffers. Poté projde strukturou synchronizované složky a vytvoří její otisk, který porovná s otiskem načteným ze souboru. Pokud se otisky liší, informuje o tom komunikační vlákno. Při vytváření instance této třídy může být definován časový interval, po jehož uplynutí má být vytvořena událost informující komunikační vlákno o změně ve složce, je-li taková změna detekována. Zároveň při každém vyvolání této události je otisk složky uložen.

5.1.2 Zabezpečení přístupu k proměnným z více vláken a signalizace

Zde budou popsány některé z mechanizmů, použitých pro zabezpečení správné manipulace s proměnnými, ke kterým bude přistupováno z více vláken, a signalizaci. Funkce `OnChange`, která obsluhuje události informující o operaci provedené ve sledované složce, je volána z vlákna systému, proto je zde zapotřebí použít některý mechanizmus pro signalizaci této události monitorovacímu vláknu. V tomto případě byla použita třída `SemaphoreSlim`, což je efektivnější varianta třídy `Semaphore` dostupná v .NET Frameworku od verze 4.0 nevyužívající semaforů jádra operačního systému. Monitorovací vlákno vyčkává na uvolnění semaforu a následně událost obslouží. Funkce pro čekání, například funkce `Wait` třídy `SemaphoreSlim`, mohou přijmout strukturu `CancellationToken` jako parametr a čekání tak může být přerušeno.

Dalším mechanismem použitým pro zabezpečení přístupu k proměnným z více vláken je použití konstrukce `lock`. Konstrukce `lock` umožňuje označit blok příkazů jako kritickou sekci, kterou smí v určitém okamžiku provádět pouze jedno vlákno. Konstrukci `lock` je předán objekt, který je uzamčen podobně jako v případě mutexů.

5.1.3 Další třídy

Knihovna `Synchro` definuje třídu **`Reconciler`**, která implementuje algoritmus pro usmíření replik a řešení kolizí. Třída `Reconciler` implementuje také statickou metodu pro vytvoření minimálního seznamu operací ze dvou stavů replik. Třída **`ObjectPath`** definovaná v knihovně `PunchCommonTypes` slouží pro uložení cesty k objektu v synchronizované složce. Cesta je uložena

v normalizované formě – jednotlivé objekty v cestě jsou oddělovány znakem lomítka (například: /cesta/k/souboru.dat). Třída obsahuje metodu pro obdržení skutečné cesty (například: C:\sync_slozka\cesta\k\souboru.dat).

Z důvodu snadnější implementace nebudou otisky replik ukládány do databáze. Namísto toho budou serializovány do souboru pomocí ProtocolBuffers.

5.2 Knihovna `Protocol`

Knihovna definuje tyto třídy:

- **`Socket`**
- **`TlsStream`**
- **`AesStream`**

Třída `Socket` zajišťuje navázání spojení se serverem nebo klientem, přijímání příchozích spojení, odesílání a přijímání zpráv, jejich serializaci a deserializaci a zabezpečení celé komunikace. Třída `TlsStream` vytváří objekt typu `SslStream` a iniciuje TLS spojení mezi serverem a klientem. Cestu k PFX souboru, který obsahuje certifikát a privátní klíč, předá server objektu třídy `Socket` při zavolání metody `Listen`. Třída `AesStream` dědí od třídy `Stream` a používá třídu `AesCryptoServiceProvider`. Její konstruktor obdrží jiný objekt typu `Stream`, do/z kterého bude data zapisovat/číst, a klíč, který bude použit pro šifrování. Třída `AesStream` zajišťuje šifrování a dešifrování dat do ní zapisovaných respektive z ní čtených a je použita pro zabezpečení komunikace mezi klienty. Jako vnitřní objekt typu `Stream` objektů `TlsStream` a `AesStream` je použit objekt typu `Udt.NetworkStream`, který zapisuje/čte data do/z objektu typu `Udt.Socket`. Knihovna UDT.Net obsahuje obě posledně jmenované třídy.

Klienti se při navazování spojení potřebují zároveň zkusit připojit pomocí všech svých adres. Pro každé naslouchání/připojování je proto vytvořeno samostatné vlákno. Z úspěšných spojení je potom vybráno jedno. Spojení s největší prioritou bude spojení pomocí IPv4, dále IPv6 a potom spojení pomocí veřejných adres obdržených od serveru.

5.3 Server

Server je implementován jako konzolová aplikace. Jako parametr je aplikaci předána cesta k PFX souboru s certifikátem a privátním klíčem, heslo k otevření PFX souboru a port, na kterém bude server naslouchat příchozím spojením. Z důvodu snadnější implementace je server implementován jako iterativní. Vzhledem k tomu, že komunikace mezi klientem a serverem probíhá formou požadavek – odpověď, server by měl i při obsluhování většího množství klientů zůstat responsivní. Server čeká na příchod zprávy od některého z klientů nebo na nové spojení. Pokud přijde žádost na nové spojení, spojení je přijato. Pokud přijde požadavek od některého z klientů, klientovi je zaslána odpověď. Samotná komunikace je realizována použitím třídy `Protocol.Socket`.

6 Testování

Pro ověření funkčnosti výsledné aplikace bylo provedeno testování. Výsledky testování budou dále diskutovány. Testovány byly tyto aspekty:

- Správnost synchronizačního algoritmu.
- Ustanovení spojení meze klienty za NAT.
- Efektivita přenosu dat.

6.1 Testování synchronizačního algoritmu

Cílem této fáze bylo otestování správného chování synchronizačního algoritmu a řešení kolizí. Testována byla synchronizace mezi dvěma instancemi aplikace na jediném počítači. Aplikace serveru byla spuštěna na stejném počítači. Tato fáze testování probíhala v tomto prostředí:

- Microsoft Windows 7 SP1 32-bit
- Microsoft .NET Framework 4.5.1

Cílem tohoto testování bylo především ověřit správnou funkčnost řešení kolizí. Byla testována každá z možných dvojic konfliktních operací. Zde bude pro ilustraci uvedeno konkrétní testování pro dvojici konfliktních operací `Edit(π/π', X); Edit(π, File(Y))`.

Nejdříve byla připravena složka s těmito adresáři a soubory:

```
[s1_edit_edit]
     │
     ├[d]
     └[s2]
         │
         ├[d]
         └[s3]
             │
             ├[d]
             └[s4]
                 │
                 └stejny.txt
```

Obě instance aplikace byly spuštěny a pro obě byly nastaveny nové prázdné složky pro synchronizaci. Do složky přidělené jedné z nich byla připravená složka zkopírována. Klient správně detekoval změnu ve složce a inicioval synchronizaci s klientem náhodně vybraným ze seznamu dostupných klientů, který obdržel od serveru. V tomto případě s druhou instanci aplikace klienta na lokálním počítači. Po skončení synchronizace byly obsahy synchronizovaných složek porovnány. Obsahy obou složek se shodovaly. Tímto byla ověřena základní funkčnost synchronizačního algoritmu.

Následovalo samotné ověření správné detekce a řešení dvojice kolizních operací. Oba klienti byli pozastaveni. Ve složce jednoho z klientu byl pozměněn obsah souboru `stejny.txt`. Ve složce druhého klienta byla celá složka `s1_edit_edit` odstraněna a místo ní byl vytvořen soubor s názvem `s1_edit_edit`. Klienti byli znovu spuštěni, správně detekovali změny ve složce a spustili proces synchronizace. Po dokončení synchronizace byly obě složky znovu porovnány. Očekávaným výstupem

by mělo být, že obě synchronizované složky obsahují stejné soubory a složky a kolize by měli být vyřešeny. Po aplikování pravidel pro řešení kolizí, by měl být zachován soubor stejny.txt, jelikož byl na jedné z replik upraven, složky d by měli být odstraněny a k souboru s názvem `s1_edit_edit` by měl být připojen prefix upozorňující na kolizi.

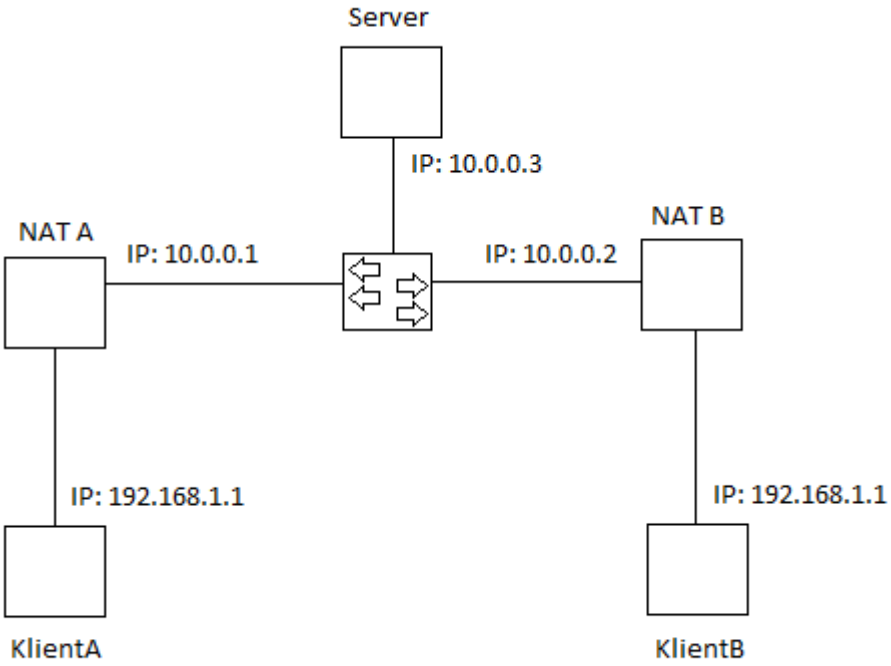
Takto vypadal obsah synchronizovaných složek obou klientů po dokončení synchronizace:

```
├[s1 edit edit]
│    │
│    └[s2]
│        │
│        └[s3]
│            │
│            └[s4]
│                │
│                └stejny.txt
└s1 edit edit_confl_a81a6d18-200d-4e4c-a055-697e70252744
```

Pro tento pár konfliktních operací tedy byla ověřena správná funkčnost. Obdobně probíhalo testování ostatních dvojic konfliktních operací. Tímto byla ověřena správná funkčnost samotné synchronizace.

6.2 Testování spojení klientů za NAT

Cílem této fáze testování bylo ověřit schopnost dvou klientů za různými zařízeními NAT spolu ustanovit spojení a zahájit proces synchronizace. Pro účel tohoto testování byl použit nástroj VirtualBox. Bylo vytvořeno pět virtuálních počítačů, jeden pro aplikaci serveru, dva pro klienty a dva pro zařízení NAT. Na všechny virtuální počítače byl nainstalován systém Windows Server 2008. V nástroji VirtualBox potom byly nastaveny síťové karty virtuální počítačů tak, že výsledná topologie virtuální sítě vypadala takto:



Obrázek 6.1 Topologie virtuální sítě

Pro funkcionalitu NAT byla na počítačích `NAT A` a `NAT B` použita služba systému Windows Server Routing and Remote Access Service. Na počítači server byla spuštěna konzolová aplikace serveru a na počítačích `KlientA`, `KlientB` aplikace klienta. Pro oba klienty byla nastavena adresa serveru 10.0.0.3. Klienti se spojili pomocí metody UDP hole punching, čímž bylo ověřeno její správné fungování.

6.3 Testování efektivity

V této fázi testování měla být sledována efektivita přenosu dat souborů během synchronizace. Testováno bylo přenesení souboru o velikosti 1GB. Doba přenosu mezi klienty byla měřena pomocí třídy `Stopwatch` knihovny tříd .NET Frameworku. Testování probíhalo v různých prostředích:

- Jeden počítač (loopback).
- Dva počítače se spojením s max. rychlostí přenosu 1Gb/s.

Rychlost přenosu souboru mezi klienty (UDT.Net) byla porovnána s přenosem souboru pomocí samotné knihovny UDT a s přenosem pomocí protokolu Teredo. V případě testování přenosu mezi dvěma počítači byla zaznamenána i rychlost přenosu pomocí protokolu SMB[17]. Výsledky měření jsou v tabulce 6.1.

Test č.	Spojení	UDT (MB/s)	UDT.Net (MB/s)	Teredo (MB/s)	SMB (MB/s)
1	loopback	6,32	10,24	25,02	-
2	ether (1Gb/s)	21,62	5,63	62,19	67

Tabulka 6.1 Výsledky měření rychlosti přenosu souboru (1GB)

[17] Microsoft SMB Protocol je protokol pro sdílení souborů po síti.

Protokol UDT byl navržen pro přenos velkých objemů dat ve vysokorychlostních sítích WAN. [18] Z výsledků měření je patrné, že použití protokolu UDT v testované síťové konfiguraci není efektivní. Z tohoto důvodu byla do výsledné aplikace přidána podpora pro spojení pomocí protokolu Teredo. To bylo realizováno implementací třídy `TeredoSocket` v knihovně `Protocol`. V případě, že oba klienti (systém, na kterém jsou spuštěni) umožňují spojení pomocí Teredo, klienti toto spojení upřednostní. Podpora lze v okně nastavení klienta vypnout.
Následně bylo provedeno testování efektivity přenosu dat použitím protokolu UDT a Teredo. Testován byl přenos souboru o velikosti 869MB mezi dvěma počítači se spojením s max. rychlostí přenosu 1G/s. Výsledky měření jsou v tabulce 6.2.

	Doba přenosu (s)	Rychlost přenosu (MB/s)
Teredo	87,37	9,94
UDT.Net	129,564	6,7

Tabulka 6.2 Výsledky měření rychlosti přenosu souboru (869MB)

Z měření vyplývá, že použití protokolu Teredo je efektivnější. Při porovnání s výsledky předchozího měření je patrné, že klienti nevyužívají dostupné přenosové pásmo. Důvodem je neefektivní implementace šifrování a dešifrování zpráv pomocí protokolu AES.

7 Závěr

Cílem této práce bylo navrhnout a implementovat nástroj pro jednoduchou synchronizaci souborů v běžném síťovém prostředí. Testováním výsledného řešení bylo ověřeno očekávané chování synchronizačního algoritmu, včetně jeho části pro řešení kolizí. Dále byla ověřena schopnost klientů ustanovit spojení, pokud se oba nacházejí za zařízeními (EIM) NAT. Z výsledků testování efektivity výsledného nástroje při přenosu větších objemů dat vyplývá, že původní výběr protokolu UDT nebyl pro cílené síťové prostředí vhodný. Tento problém byl řešen rozšířením výsledné aplikace o podporu spojení mezi klienty pomocí protokolu Teredo. Dalším možným řešením by bylo implementovat vlastní mechanizmus pro spolehlivý přenos dat nebo nahradit algoritmus pro předcházení zahlcení v UDT vlastní implementací.

Z výsledků testování dále vyplývá, že implementace šifrování a dešifrování zpráv pomocí AES ve výsledném řešení je neefektivní a brání plnému využití přenosového pásma. Dalším problémem je nepoužití databáze pro ukládání otisků předešlých stavů synchronizované složky.

Rozšířením této práce by mohla být efektivnější implementace zabezpečení komunikace mezi klienty. Dále by mohl být nástroj rozšířen o použití databáze pro ukládání otisků složek s uloženými procedurami pro vyhledání potřebného otisku. Možným dalším rozšířením by mohlo být šíření informací o umístění (adresách) klientů pomocí distribuovaných hashovacích tabulek. Takové řešení by se mohlo obejít bez serveru, pokud by úlohu serveru převzali jednotlivé uzly vzniklé peer-to-peer sítě.

Literatura

[1] Tridgell, A.; Mackerras, P.: *The rsync algorithm*, [Online], 1998, [cit. 2014-05-14]. URL http://rsync.samba.org/tech_report/

[2] Remote Differential Compression, Microsoft MSDN Library. [Online]. URL http://msdn.microsoft.com/en-us/library/aa373254%28v=vs.85%29.aspx

[3] Drago, I.; Mellia, M.; Munafò, M.; Sperotto, A.; Sadre, R.; Pras, A.: Inside Dropbox: Understanding Personal Cloud Storage Services, In *IMC '12 Proceedings of the 2012 ACM conference on Internet measurement conference*, 2012, ISBN 978-1-4503-1705-4.

[4] Dropbox, [Online]. URL https://www.dropbox.com/.

[5] BitTorrent Technology, [Online]. URL http://www.bittorrent.com/sync/technology

[6] Cohen, B.: The BitTorrent Protocol Specification, 2008, [Online], [cit. 2014-05-14]. URL http://www.bittorrent.org/beps/bep_0003.html

[7] Loewenstern, A.; Norberg, A.: DHT Protocol, [Online], 2008, [cit. 2014-05-14.] URL http://www.bittorrent.org/beps/bep_0005.html.

[8] Marťák, A.: Synchronizace a zálohování dat pod OS Linux, bakalářská práce, Brno, FIT VUT v Brně, 2013.

[9] Karmazín, J.: Synchronizace a zálohování dat pod OS Android, bakalářská práce, Brno, FIT VUT v Brně, 2013.

[10] Slováček, J.: Synchronizace a zálohování dat pod OS Windows, bakalářská práce, Brno, FIT VUT v Brně, 2013.

[11] Saito, Y.; Shapiro, M.: Optimistic replication, *ACM Computing Surveys (CSUR)*, sv. 37, č. 1, s. 42-81, 2005, [cit. 2014-05-14].

[12] Balasubramaniam, S.; Pierce, B.: What is a file synchronizer?, v *MobiCom '98 Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, New York, 1998, [cit. 2014-05-14].

[13] Ramsey, N.; Csirmaz: An algebraic approach to file synchronization, *ACM SIGSOFT Software Engineering Notes*, sv. 26, č. 5, s. 175-185, 2001, [cit. 2014-05-14].

[14] Srisuresh, P.; Jasmine Networks; Egevang, K.; Intel Corporation: Traditional IP Network Address Translator (Traditional NAT), RFC 3022, Internet Engineering Task Force, 2001.

[15] Srisuresh, P., Kazeon Systems; Ford, B.; M.I.T.; Kegel, D.: State of Peer-to-Peer (P2P) Communication across, RFC 5128, Internet Engineering Task Force, 2008, [cit. 2014-05-14].

[16] Ford, B.; Srisuresh, P.; Kegel, D.: Peer-to-Peer Communication Across Network Address Translators, v *USENIX Annual Technical Conference*, Anaheim, CA, 2005.

[17] Teredo, Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/windows/desktop/aa965909%28v=vs.85%29.aspx

[18] UDT: Breaking the Data Transfer Bottleneck, [Online]. URL http://udt.sourceforge.net/index.html.

[19] Getting Started with the .NET Framework, Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/hh425099%28v=vs.110%29.aspx

[20] Troelsen, A.: Pro C# 2010 and the .NET 4 Platform, 5th Edition, Apress, 2010, ISBN 978-1430225492.

[21] Advanced Encryption Standard (AES), [Online], FIPS 197, National Institute of Security and Technology, 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[22] Cryptography, Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/windows/desktop/aa380255%28v=vs.85%29.aspx

[23] Dierks, T.; Independent; Rescorla, E.; RTFM, Inc.:The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246, IETF, 2008, [cit. 2014-05-14]

[24] Transport Layer Security Protocol, Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/windows/desktop/aa380516%28v=vs.85%29.aspx

[25] Developer Guide - Protocol Buffers -- Google Developers, Google, [Online], 2012. URL https://developers.google.com/protocol-buffers/docs/overview.

[26] Obtaining Directory Change Notifications (Windows), Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/windows/desktop/aa365261%28v=vs.85%29.aspx

[27] FileSystemWatcher Class (System.IO), Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher%28v=vs.110%29.aspx

[28] DCE 1.1: Remote Procedure Call, [Online], The Open Group,1997, [cit. 2014-05-14]. URL http://pubs.opengroup.org/onlinepubs/9629399/apdxa.htm

[29] GUID structure, Microsoft, [Online]. URL http://msdn.microsoft.com/en-us/library/windows/desktop/aa373931%28v=vs.85%29.aspx

[30] UPnP Device Architecture Version 1.1, ISO/IEC 29341-1-1:2011, [Online], 2011, [cit. 2014-05-14]. URL http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=57494

Příloha A – Obsah CD

/Aplikace	Zdrojové kódy a další soubory Visual Studio „solution“
/Bin	Spustitelné soubory
/middleman	Aplikace serveru
/client	Aplikace klienta
/Cer	Certifikáty
/protocol.proto	Definice zpráv v ProtocolBuffers
/Manual.txt	Uživatelský manuál
/Práce.docx	Tato práce ve formátu docx
/xhella00.pdf	Tato práce ve formátu pdf

Příloha B – Uživatelský manuál

Pro samotný běh aplikace klienta i serveru je potřeba .NET Framework ve verzi 4.5.

Server

Aplikaci serveru lze spustit z příkazového řádku s parametry pro určení cesty k souboru PFX obsahujícím certifikát a privátní klíč, hesla k tomuto souboru a portu na kterém má server naslouchat příchozím spojení od klientů. Tyto parametry jsou nepovinné. V případě, že nejsou uvedeny, jako cesta k souboru PFX je zvolen soubor cer.pfx v adresáři, v kterém se nachází spustitelný soubor aplikace. Heslo k odemčení tohoto souboru je „puncCer“ a jako výchozí port je zvolen port 10011. Soubor cer.pfx je společně s aplikací uložen na přiloženém CD. Aplikace po spuštění vypisuje zprávy přijaté od připojených klientů.

Klient

Aplikace klienta po prvním spuštění nabídne okno pro vyplnění nastavení. Zde je potřeba uvést jméno synchronizovaného počítače, uživatelské jméno, a složku, která bude synchronizována. Stejné uživatelské jméno a heslo musí být použito u každého dalšího klienta, aby mezi nimi mohla probíhat synchronizace.

Po zadání těchto nastavení je zaregistrována ikona v notifikační oblasti a otevřeno stavové okno klienta. Skrze notifikační ikonu lze klienta ukončit nebo znovu otevřít zavřené stavové okno. Stavové okno zobrazuje současný stav klienta. Možné stavy:

- Idle – klient se připojuje k serveru nebo je připojen a vyčkává na žádost o synchronizaci od jiného klienta nebo na změnu v synchronizované složce.
- Syncing – klient se zrovna synchronizuje s jiným klientem. Prováděné operace jsou uváděny pod zobrazeným stavem.
- Stopped – klient byl pozastaven nebo bylo spojení se serverem přerušeno.

Aby mohl klient ověřit identitu serveru, musí být jeho certifikát v sytému importován mezi důvěryhodné certifikáty. Soubor s certifikátem (cer.cer) je uložen s aplikací klienta na přiloženém CD. V případě že server není dostupný (výchozí adresa je 127.0.0.1), je zobrazeno okno pro nastavení adresy, portu a názvu (tak jak bylo uvedeno při vytváření certifikátu) serveru. Pomocí volby Use Teredo lze povolit podporu pro spojení klientů pomocí protokolu Teredo. Tlačítko Pause/Start slouží k pozastavení nebo spuštění klienta. Tlačítko Settings otevře okno pro změnu nastavení údajů, které byly zadány při prvním spuštění aplikace. Povolením volby Show debug info je zobrazena aktuální verze repliky klienta, události o kterých informuje komunikační vlákno klienta a také tlačítko pro ukončení aplikace.

Nastavení aplikace je ukládáno pro přihlášeného uživatele. Proto není žádoucí dovolit více spuštěných instancí aplikace. Je-li spuštěna další instance, uživatel je na to upozorněn a aplikace je spuštěna v testovacím režimu, čímž je zabezpečeno, že nebude přistupovat k nastavením uživatele. Aplikace spuštěná v testovacím režimu se připojuje k serveru na lokálním počítači. Složka pro synchronizaci je pro ni vytvořena v cestě „C: \syncTest\guid”. Uživatelské jméno a heslo je „test“.

Příloha C – Protokol

Zde je uveden komunikační protokol v syntaxi ProtocolBuffers.

```
// typ zprávy
enum MessageType
{
    // client - server messages
    Register = 1;                    // data = RegisterData
    UpdateInfo = 3;                  // data = RegisterData
    ClientsList = 5;                 // data = ClientInfoListData
    Synchronize = 6;                 // data = ClientIdData
    SyncRequest = 7;                 // data = ClientKey
    SyncRequestOk = 8;               // data = ClientInfo
    Key = 9;                         // data = SessionKey
    Bye = 10;                        // no data

    // client - client messages
    Ok = 9;                               // no data
    No = 10;                         // no data
    ListOfOperations = 11;           // data = OperationList
    FileRequest = 12;                // data = ObjectPath
    File = 13;                       // data is file stream
    Version = 14;                    // data = ReplicaVersion
    FolderState = 15;                // data = FolderState
}

// samotná zpráva
message Message
{
    required MessageType Type = 1;
    required int64 DataLength = 2;
}

// následují zprávy nesoucí data

// registrační údaje klienta
message RegisterData
{
    required string User = 1;
    required ReplicaVersion Version = 2;
    required string ComputerName = 3;
    repeated Endpoint EndPoints = 4;
    required string ClientId = 5;
}
```

```
// identifikační číslo zařízení klienta
message ClientIdData
{
   required string ClientId = 1;
}

// seznam dostupných klientů
message ClientInfoListData
{
   repeated ClientInfo List = 1;
}

// informace o klientovy
message ClientInfo
{
   required string Id = 1;
   required ReplicaVersion Version = 2;
   required string ComputerName = 3;
   repeated Endpoint EndPoints = 4;
}

// typ objektu soubrového systému
enum ObjectType
{
   File = 1;
   Dir = 2;
   Any = 3;
}

// reprezentuje soubor/složku
message FileSystemObject
{
   required ObjectPath Path = 1;
   required ObjectType Type = 2;
   required ObjectMetadata Metadata = 3;
}

// otisk synchronizované složky
message FolderState
{
   repeated FileSystemObject ObjectList = 1;
}

// metadata souboru
message ObjectMetadata
```

```
  {
     required int64 Size = 1;
     required bytes Hash = 2;
  }

  // cesta k souboru
  message ObjectPath
  {
     required string Path = 1;
     required int32 Level = 2;  //  stupeň  zanoření  v  struktuře
synchronizované složky
  }

  // seznam operací
  message OperationList
  {
     repeated Operation OpList = 1;
  }

  // typ operace
  enum OperationType
  {
     Create = 1;
     Remove = 2;
     Edit = 3;
     Move = 4;
  }

  // reprezentuje operaci
  message Operation
  {
     required ObjectPath Path = 1;
     required ObjectPath TargetPath = 2;
     required OperationType Type = 3;
     required ObjectType ObjectType = 4;
     required bytes Hash = 5;
     required bool Aborted = 6;
  }

  // část verze repliky
  message ClientVersion
  {
     required string Id = 1;
     required int64 Version = 2;
  }
```

```
// verze repliky
message ReplicaVersion
{
   repeated ClientVersion VersionList = 1;
}

// klíč pro šifrování komunikace mezi klienty pomocí AES
message SessionKey
{
   required bytes Key = 1;
}

// data pro klienta žádaného o synchronizaci
message ClientKey
{
   required ClientInfo Client = 1;
   required bytes Key = 2;
}
```