České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Vladimír Pokorný**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Framework pro synchronizaci obecných verzovaných entit v distribuovaném prostředí**

Pokyny pro vypracování:

Navrhněte a implementujte systém s architekturou client-server pro synchronizaci obecných verzovaných entit. Implementujte framework, který řeší pokračování synchronizace a výměnu dat přímým spojením mezi klienty. Vemte v potaz dynamicky se měnící stav klientů (přibývající - ubývající). Implementujte vzorové použití frameworku na systému pro synchronizaci dat a změn jednotlivých souborů.

Seznam odborné literatury:

RESTfull .NET, Jon Flanders, ISBN: 978-0-596-51920-9, O"Reilly 2009

Troelsen, Andrew, 2012, Pro C# 5.0 and the .NET 4.5 Framework. Apress, ISBN: 1430242337

Robinson, Simon, 2003, Professional C# 5.0 and .NET 4.5.1, ISBN: 1118833031

Introduction to Windows Communication Foundation, URL: https://msdn.microsoft.com/en-us/library/dd936243.aspx

Vedoucí: Ing. Martin Mudra

Platnost zadání: do konce zimního semestru 2016/2017

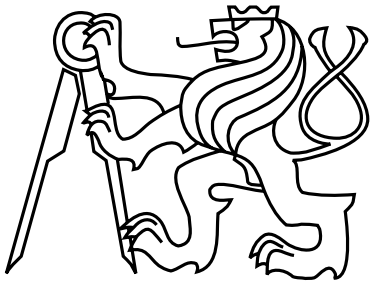
[redacted]
doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

L.S.

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

V Praze dne 14. 10. 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Framework pro synchronizaci obecných verzovaných entit v distribuovaném prostředí

Bc. Vladimír Pokorný

Vedoucí práce: Ing. Martin Mudra

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

25. května 2016

Poděkování

Chtěl bych poděkovat svému vedoucímu práce Ing. Martinu Mudrovi za všechny poskytnuté rady a připomínky.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. 5. 2016 .....

Abstract

This thesis deals with design and implementation of the framework for general data entity synchronization including creating and maintaining of theirs version history. The framework uses client-server architecture. The emphasis is on the continuation of synchronization in case of server connection failure, using direct client connection (peer-to-peer). This thesis also deals with conflict detection and resolution in both of the communication modes (client-server and peer-to-peer). The sample implementation of the framework usage for simplified file synchronization is also included.

Abstrakt

Tato práce se zabývá návrhem a implementací frameworku umožňujícího synchronizaci obecných datových entit včetně jejich verzování. Framework používá architekturu klient-server a důraz je kladen na schopnost pokračování synchronizace přímým spojením mezi klienty (peer-to-peer) v případě výpadku spojení se serverem. Zároveň se tato práce zabývá detekcí a způsobem řešení konfliktů v obou komunikačních režimech (klient-server i peer-to-peer). Součástí této práce je ukázková implementace použití tohoto frameworku pro zjednodušenou synchronizaci souborů.

Obsah

1 Úvod **1**
1.1 Cíl práce . . . 1

2 Analýza **3**
2.1 Požadavky . . . 3
2.1.1 Funkční požadavky frameworku . . . 3
2.1.2 Nefunkční požadavky frameworku . . . 4
2.1.3 Funkční požadavky na ukázkovou implementaci . . . 4
2.1.4 Nefunkční požadavky na ukázkovou implementaci . . . 4
2.1.5 Negativní vymezení . . . 5
2.2 Existující řešení . . . 5
2.2.1 Dropbox LAN sync . . . 5
2.2.2 BitTorrent Sync . . . 5
2.2.2.1 BitTorrent . . . 6
2.2.3 Syncthing . . . 6
2.2.4 Další podobné služby . . . 6
2.2.5 Souhrn . . . 7
2.3 Nalezení klientů . . . 7
2.3.1 WCF Discovery . . . 8
2.4 Komunikace v distribuovaném prostředí . . . 8
2.4.1 Režim komunikace peer-to-peer . . . 9
2.4.2 Režim komunikace klient v roli serveru . . . 9
2.4.3 WCF Self-hosting . . . 10
2.4.4 Algoritmy shody . . . 10
2.5 Synchronizace entit a konflikty . . . 11
2.5.1 Vznik konfliktů . . . 11
2.5.2 Řešení konfliktů verzí . . . 12
2.5.2.1 Poslední nahraná je aktuální . . . 12
2.5.2.2 Automatické řešení . . . 13
2.5.2.3 Řešení konfliktu uživatelem . . . 13
2.5.3 Konflikty entit . . . 14
2.5.3.1 Řešení konfliktů . . . 14
2.6 Verzování v distribuovaném prostředí . . . 15
2.6.1 Verzovací vektor . . . 15
2.6.1.1 Princip fungování . . . 15

2.6.1.2 Pravidla fungování . . . 15
2.6.1.3 Výhody a nevýhody . . . 16
2.6.2 Konflikty v distribuovaném prostředí . . . 17
2.6.2.1 Ruční řešení konfliktu . . . 17
2.6.2.2 Automatické řešení konfliktu . . . 18
2.7 Technická řešení . . . 18
2.7.1 Inversion of Control framework . . . 18
2.7.2 Databáze . . . 19
2.7.3 Logovací nástroj . . . 19
2.7.4 Řešení pro uživatelské rozhraní . . . 20

3 Návrh **21**
3.1 Základní popis architektury . . . 21
3.1.1 Datové úložiště pro synchronizaci souborů . . . 22
3.2 Hledání klientů . . . 22
3.3 Mechanismy pro synchronizaci dat . . . 23
3.3.1 Identifikace entit . . . 23
3.3.2 Verzování entit . . . 24
3.3.3 Skupiny entit . . . 25
3.3.4 Model komunikace mezi klienty . . . 25
3.3.5 Komunikace . . . 25
3.3.6 Přepínání mezi režimy komunikace klient-server a peer-to-peer . . . 26
3.3.7 Volba klientů pro komunikaci peer-to-peer . . . 26
3.3.8 Řešení konfliktů . . . 27
3.4 Komunikační protokol . . . 27
3.4.1 Komunikační rozhraní . . . 27
3.5 Architektura systému . . . 28
3.5.1 Architektura serveru . . . 28
3.5.2 Architektura klienta – back-end . . . 29
3.5.3 Architektura klienta – front-end . . . 30
3.6 Popis základních operací synchronizace . . . 31
3.6.1 Vytváření nové entity nebo verze na klientovi . . . 31
3.6.2 Vytváření nové entity nebo verze na serveru . . . 32
3.6.3 Získání nových entit na klientovi . . . 33
3.6.4 Získání entity na základě požadavku od klienta . . . 33
3.7 Databáze . . . 34
3.7.1 Serverová databáze . . . 34
3.7.1.1 Tabulky pro konkrétní implementaci frameworku . . . 36
3.7.2 Klientská databáze . . . 36
3.7.2.1 Tabulka pro konkrétní implementaci frameworku . . . 37
3.8 Uživatelské rozhraní . . . 37

4 Implementace **39**
4.1 Software pro vývoj . . . 39
4.2 Software pro provozování systému . . . 39
4.3 Struktura projektu . . . 40

4.4 Oddělení frameworku od synchronizace konkrétních entit . . . 41
4.4.1 Serverová strana . . . 42
4.4.2 Klientská strana . . . 42
4.4.3 Další definovaná rozhraní . . . 42

5 Testování **43**
5.1 Testování přes synchronizaci souborů . . . 43
5.1.1 Testovací prostředí . . . 43
5.1.2 Testovací zařízení . . . 45
5.1.3 Testovací vybavení . . . 45
5.1.4 Testovací scénáře a výsledek testování . . . 45
5.2 Testování vyhledávání klientů . . . 46
5.2.1 Výsledek testu . . . 47
5.3 Testování ukázkové aplikace s uživateli . . . 47
5.3.1 Výsledek první iterace . . . 47
5.3.2 Výsledek druhé iterace . . . 48
5.3.3 Výsledek třetí iterace . . . 48
5.4 Test nasazení frameworku . . . 48
5.4.1 Výsledek testu . . . 48

6 Závěr **49**
6.1 Složitost přepínání mezi režimy komunikace . . . 49
6.2 Možnosti budoucího vylepšení systému . . . 50
6.2.1 Zabezpečení systému . . . 50
6.2.2 Optimalizace komunikace . . . 50
6.2.3 Komunikace mezi klienty . . . 51
6.2.4 Verzovací vektory . . . 51
6.2.5 Další vylepšení frameworku . . . 51
6.2.6 Rozšíření funkcionality ukázkové synchronizace souborů . . . 51

Literatura **53**

A Seznam použitých zkratek **57**

B Návod na použití frameworku **59**
B.1 Zprovoznění serveru . . . 59
B.2 Zprovoznění klienta . . . 60
B.3 Použití NHibernate a AutoMapper . . . 62

C Návod na spuštění systému pro synchronizaci souborů **63**
C.1 Nasazení serveru . . . 63
C.2 Spuštění klienta . . . 64

D Seznam kroků pro testování **65**
D.1 Inicializace aplikace . . . 65
D.2 Hledání klientů . . . 65
D.3 Základní synchronizace . . . 66

D.4 Přepínání mezi komunikačními režimy 67
D.5 Funkcionalita Client poolů 67

E Dokumentace komunikačního rozhraní **69**
E.1 Metody komunikačního rozhraní pro peer-to-peer 69
E.2 Metody serverového komunikačního rozhraní 70

F Snímky obrazovky ukázkové aplikace **71**

G Obsah přiloženého DVD **73**

Seznam obrázků

2.1 Diagram komunikace v režimu peer-to-peer 9
2.2 Diagram komunikace v režimu klient v roli serveru 10
2.3 Diagram nahrávání nové verze a její zařazení do historie 11
2.4 Diagram vzniku konfliktu z hlediska historie verzí 12
2.5 Diagram řešení konfliktu nastavením aktuální verze 12
2.6 Diagram řešení konfliktu vytvořením nové entity 13
2.7 Diagram řešení konfliktu zařazením do historie 13
2.8 Diagram aktualizace verzovacího vektoru 16

3.1 Základní rozdělení frameworku 21
3.2 Rozdělení částí frameworku 22
3.3 Hledání klientů 23
3.4 Diagram nahrávání historie entit na server 24
3.5 Diagram historie entit nahraných na server 24
3.6 Stahování entit v režimu komunikace peer-to-peer 26
3.7 Struktura SOAP zprávy s entitou obsahující soubor 28
3.8 Diagram architektury serveru 29
3.9 Diagram architektury klienta 29
3.10 Diagram architektury MVVM 31
3.11 Sekvenční diagram vytváření nové verze entity na klientovi 31
3.12 Sekvenční diagram vytváření nové verze entity na serveru 32
3.13 Sekvenční diagram získávání nových entit na klientovi 33
3.14 Sekvenční diagram získání entity na základě požadavku od klienta 34
3.15 Schéma serverové databáze 35
3.16 Schéma klientské databáze 37
3.17 Návrh uživatelského rozhraní hlavního okna aplikace 38

4.1 Diagram komponent serverové části 41
4.2 Diagram komponent klientské části 41

5.1 Konfigurace zapojení zařízení pro testování 44
5.2 Konfigurace zapojení zařízení pro testování obou režimů komunikace zároveň 44
5.3 Konfigurace síťového zapojení zařízení pro testování hledání klientů 46

F.1 Obrazovka pro první inicializaci aplikace 71
F.2 Hlavní obrazovka aplikace 72

F.3 Notifikační zelená ikona s otevřeným vyskakovacím oknem 72
F.4 Notifikační modrá ikona 72

G.1 Obsah přiloženého DVD 73

Seznam tabulek

2.1 Porovnání existujících řešení 7
2.2 Porovnání způsobů hledání klientů 8

5.1 Seznam zařízení pro testování 45
5.2 Seznam nástrojů pro testování 45

E.1 Seznam metod komunikačního rozhraní pro peer-to-peer komunikaci 69
E.2 Seznam metod serverového komunikačního rozhraní 70

Kapitola 1

Úvod

Datová synchronizace je běžnými uživateli poslední dobou využívána stále častěji. Používá se zejména v režimu komunikace klient-server. Hlavním důvodem může být rozšiřující se nabídka cloudových úložišť, která jsou využívána jednak k ukládání souborů, ale např. v případě Microsoftu nebo Googlu také např. k ukládání kontaktů, kalendářů apod.

Pro uživatele z tohoto přístupu plynou dvě zásadní výhody. Spousta uživatelů totiž vlastní více než jedno zařízení a jednou z výhod synchronizace je zajištění dostupnosti stejných dat na všech těchto zařízeních. Může se třeba jednat o různé stolní počítače a notebooky, nebo chytré telefony a tablety.

Druhou výhodou synchronizace je zajištění zálohování dat, které uživatel ocení např. v případě poškození disku, ztráty zařízení, nebo prostě jenom při upgradu starého telefonu za nový model. V těchto případech nejenže uživatel o tato data nepřijde, ale při dokončení synchronizace na novém zařízení budou všechna tato data k dispozici.

Celá tato synchronizace probíhá zcela automaticky, takže se uživatel nemusí o nic starat. Synchronizace může probíhat mezi klientským zařízením a serverem (např. s cloudovým úložištěm) nebo může probíhat přímým spojením mezi jednotlivými zařízeními (peer-to-peer) bez použití žádného centrálního prvku. Oba tyto způsoby mají své výhody a nevýhody.

1.1 Cíl práce

Účelem této práce je navrhnout a implementovat framework, který umožní kombinovat oba režimy komunikace (klient-server a peer-to-peer) podle potřeby, a tedy využít výhody obou dvou. Typ synchronizovaných entit bude záležet na konkrétním použití tohoto frameworku, ale podporované budou všechny entity, které lze přenášet po síti pomocí streamu dat. Ukázková implementace použití tohoto frameworku bude synchronizovat soubory ve složce. Vzhledem ke komplexnosti tohoto způsobu synchronizace bude zaručena možnost použití frameworku i pro jednodušší entity, jako jsou třeba telefonní kontakty.

Použití frameworku bude spočívat v implementaci přístupu ke konkrétním entitám (čtení i zápis). Tyto entity mohou být uloženy např. v databázi. O samotnou synchronizační logiku včetně verzování jednotlivých entit a včetně komunikace v obou režimech se postará sám framework. Pro správné fungování frameworku bude potřeba ještě provést nastavení v konfiguračních souborech.

Kapitola 2

Analýza

V této kapitole jsou nejdříve specifikovány funkční a nefunkční požadavky. Po této části následuje popis existujících řešení, která mají alespoň trochu podobnou funkcionalitu s tou, která je řešena v rámci této práce. Dále jsou zde části popisující způsoby hledání klientů pro komunikaci v distribuovaném prostředí a možné způsoby komunikace v tomto prostředí.

Podstatná část analýzy se věnuje popisu synchronizace obecných entit včetně vzniku a řešení konfliktů. Důležitou částí je popis způsobu verzování v distribuovaném prostředí, protože rozhodování o tom, která verze je novější, nemusí být úplně jednoduché.

Závěr analýzy se věnuje technickým řešením, které mohou být vhodné pro vývoj tohoto systému.

2.1 Požadavky

V této podkapitole je uveden seznam funkčních a nefunkčních požadavků rozdělených na požadavky týkající se frameworku (synchronizační logika) a požadavky týkající se ukázkové implementace použití tohoto frameworku (synchronizace souborů). Požadavky se týkají systému jako celku, tzn. serverové i klientské strany.

2.1.1 Funkční požadavky frameworku

- Systém bude schopen synchronizovat entity v režimu komunikace klient-server.
  - Systém bude umožňovat na serveru vytvářet nové entity, nové verze entit, nebo mazat entity.
  - Klienti si budou ze serveru průběžně stahovat aktuální entity.
- Systém bude schopen synchronizovat entity v režimu komunikace peer-to-peer.
  - Klient bude umožňovat lokálně vytvářet nové entity, nové verze entit, nebo mazat entity.
  - Klienti si budou průběžně stahovat aktuální entity z některého jiného klienta.
- Systém bude schopen entity verzovat (vytvářet jejich historii).

- Systém bude umožňovat při výpadku spojení automaticky přejít do režimu synchronizace peer-to-peer.
- Systém bude schopen při obnovení spojení po výpadku automaticky přejít do režimu synchronizace klient-server a zároveň nahrát historii vytvořených verzí entit na server.
- Systém bude ukládat informace o ostatních klientech pro potřeby navázání komunikace v režimu peer-to-peer.
- Systém bude umožňovat entity shlukovat do skupin, kdy každá entita bude moct patřit do více těchto skupin.
- Systém bude schopen detekovat a řešit konflikty.
- Klient bude schopen vyhledávat dostupné klienty.

2.1.2 Nefunkční požadavky frameworku

- Systém bude vytvořen formou frameworku. Neboli framework bude obstarávat synchronizační logiku včetně komunikace mezi všemi zúčastněnými zařízeními. Konkrétní implementace používající framework bude sloužit k přístupu ke konkrétním typům entit (např. čtení a zápis do souboru).
- Systém bude spustitelný na operačním systému Microsoft Windows 7.
- Zdrojové kódy a jejich dokumentace budou v anglickém jazyce.
- Nebudou použity žádné knihovny s licencí, které by vyžadovaly publikování celé aplikace pod licencí této knihovny.

2.1.3 Funkční požadavky na ukázkovou implementaci

- Klient bude umožňovat synchronizovat soubory ve vybrané složce na pevném disku.
  - V této složce bude klient zachytávat události vytvoření, úpravy, smazání a přejmenování souboru a následně vytvářet příslušné nové verze entity.
  - Aktuální soubory ze serveru nebo z jiného klienta se budou stahovat také do této složky.
- Klient bude umožňovat řešit konflikty vzniklé v režimu komunikace klient-server.
- Klient bude umožňovat přidávání a odebírání vybraných souborů do skupin.

2.1.4 Nefunkční požadavky na ukázkovou implementaci

- Klient bude používat grafické uživatelské rozhraní.
- Klient bude v anglickém jazyce.

2.1.5 Negativní vymezení

V rámci této práce nebude řešena jakákoliv forma zabezpečení systému. Takže ukládaná data a ani síťová komunikace se nebude nijak šifrovat. V systému se nebude počítat s existencí uživatelských účtů, takže nebude probíhat žádná autentizace ani autorizace a synchronizace správných entit tedy bude probíhat pouze na základě konkrétních skupin entit.

2.2 Existující řešení

Při průzkumu existujících řešení se ukázalo, že podobnou funkcionalitu některé služby nabízejí, ale v žádném z těchto případů se nejednalo o knihovnu, ale vždy o celý produkt. Nejbližší funkčnost poskytuje Dropbox, který obsahuje funkcionalitu LAN sync, která ale bez funkčního internetového připojení nefunguje. Další dvě služby BitTorrent Sync a Syncthing naopak fungují bez jakéhokoliv ukládání dat na server. Detailněji jsou tyto služby včetně jejich funkcionality popsány v následujících podkapitolách.

2.2.1 Dropbox LAN sync

Dropbox je jedno z nejznámějších cloudových úložišť souborů. Pro jeho používání lze použít webové rozhraní, nebo některého ze spousty klientů jak pro desktop, tak pro mobilní zařízení. Klienti pro desktop umožňují kompletní synchronizaci souborů. Jako volitelnou funkci nabízejí LAN sync, která umožňuje synchronizovat soubory přímo s počítači v lokální síti[1], což zrychluje přenos dat. Přestože všichni klienti jsou ve stejné síti, tak je vyžadováno připojení k internetu, protože informaci o změněných datech klienti získávají ze serveru a teprve následně zkouší stažení souboru od nějakého zařízení v lokální síti. Aby LAN sync fungoval, musí být všichni klienti na stejné podsíti nebo broadcast adrese, protože jinak se klienti nebudou schopní nalézt.

Výsledkem je, že tato funkce je určená pouze pro zrychlení přenosu dat, kdy komunikace po lokální síti je rychlejší než přes internet. Tato funkce rozhodně neřeší možnost synchronizace dat při nedostupném serveru nebo internetovém připojení.

2.2.2 BitTorrent Sync

BitTorrent Sync k synchronizaci přistupuje úplně jinak než běžná cloudová úložiště, protože žádný cloud nepoužívá, ale ke svému fungování využívá peer-to-peer spojení a server tudíž vynechává[2]. Z toho důvodu tedy neexistuje žádný centrální prvek, který by rozhodoval o tom, která verze souboru je jediná aktuální, ale musí si tuto informaci každý jednotlivý klient udržovat sám. Tím pádem je vysoká šance na vznik konfliktů, se kterými se opět každý klient musí vypořádat sám.

Tento program je založený na protokolu BitTorrent, nad kterým je vybudováno kompletní synchronizační prostřední mezi klienty včetně pokročilých funkcí jako je sdílení souborů nebo verzování. Jelikož se jedná o proprietární software, tak nejsou dostupné informace, jak přesně funguje.

2.2.2.1 BitTorrent

BitTorrent je komunikační protokol pro sdílení souborů v sítích typu peer-to-peer[3]. Převážně je používán pro stahování velkých souborů. Pro zahájení přenosu je potřeba některý BitTorrentový klient a torrent soubor obsahující metadata o konkrétních souborech pro přenos včetně adresy trackeru. Následně se klient připojí do swarmu (swarm jsou všichni peeři sdílející daný torrent).

Samotné stahování pak probíhá tak, že soubor je rozdělen na mnoho malých částí, kdy každá může být stahována od jiného peera ve swarmu a tím být dosaženo co nejvyšší přenosové rychlosti. Jakmile má peer některé části stažené, může je nabídnout ostatním peerům ke stažení.

Vyhledávání klientů může probíhat několika způsoby:

- Local discovery – Vyhledání peerů ve stejné lokální síti pomocí multicast požadavku.
- Tracker – Jedná se o server obsahující informace, kde je jaký soubor uložen (tedy IP adresy peerů).
- PEX (Peer exchange) – Peeři připojení do stejného swarmu si přímo mezi sebou vyměňují informace o ostatních peerech ve swarmu.
- DHT (Distributed hash table) – Informace o umístění souborů na jednotlivých klientech není uložena centrálně, ale distribuovaně na různých uzlech. Z každého DHT uzlu je pak možné se dostat na několik jiných DHT uzlů. Pro připojení k nějakému prvnímu DHT uzlu je ale nutné znát jeho adresu.

2.2.3 Syncthing

Podobně jako BitTorrent Sync k synchronizaci souborů přistupuje také Syncthing[4]. Funkcionalita těchto dvou konkurenčních programů je velice podobná. Významný rozdíl je v tom, že Syncthing používá vlastní protokoly, které má na webových stránkách zdokumentovány. Druhým důležitým rozdílem je, že tento program má otevřené zdrojové kódy pod licencí Mozilla Public License Version 2.0.

Vyhledávání klientů probíhá dvěma způsoby: buď pomocí Global Discovery Protocol, kdy se klient dotazuje serveru každých 30 minut, nebo pomocí Local Discovery Protocol, kdy klient posílá každých 30 sekund broadcast požadavek do lokální sítě. Obě tyto možnosti se dají vypnout.

Výměna dat probíhá spojením typu peer-to-peer. Za předpokladu, že dvě běžící zařízení na sebe nevidí (např. nacházejí se za NAT), tak Syncthing používá Relay Protocol a relay servery, které výměnu dat umožní. Samotná výměna dat je specifikována v Block Exchange Protocol.

2.2.4 Další podobné služby

Kromě výše uvedených služeb existují i některé další, nabízející podobnou funkcionalitu. Třeba takový SpiderOak (cloudové úložiště zaměřující se hlavně na ochranu soukromí dat)

obsahuje, anebo obsahoval funkci LAN Sync[5], která by měla stejně jako v případě LAN sync u Dropboxu sloužit pro zrychlení přenosu dat na lokální síti. Tato funkce, ale není nikde zdokumentována, pouze o ní existují zmínky[6].

Další zajímavou službou je Cubby, která je podobně jako Dropbox normálním cloudovým úložištěm, ale nabízí funkci nazvanou DirectSync[7], která umožňuje vypnout synchronizaci některých složek do cloudu a synchronizovat si její data pouze v režimu komunikace peer-to-peer. V tomto případě si ale uživatel musí vždy vybrat, ve kterém režimu bude Cubby pracovat.

2.2.5 Souhrn

Jak už bylo zmíněno, tak žádné ze zkoumaných existujících řešení nenabízí stejnou funkcionalitu, které se věnuje tato práce. Pro přehlednost je výsledné porovnání klíčové funkcionality těchto řešení uvedeno v tabulce 2.1.

Tabulka 2.1: Porovnání existujících řešení

	Dropbox LAN sync	BitTorrent Sync	Syncthing	SpiderOak	Cubby
Umí synchronizovat v režimu klient-server?	Ano	Ne	Ne	Ano	Ano
Umí synchronizovat v režimu peer-to-peer?	Částečně[1]	Ano	Ano	Není známo	Ano
Umí automaticky přepínat synchronizaci mezi klient-server a P2P?	Ne	Ne	Ne	Není známo	Ne
Jedná se o knihovnu nebo framework?	Ne	Ne	Ne	Ne	Ne
Jedná se o open-source řešení?	Ne	Ne	Ano	Ne	Ne

2.3 Nalezení klientů

Pro přímé vytvoření komunikace mezi jednotlivými klienty je v první řadě důležité tyto klienty najít. Na základě analýzy existujících řešení bylo zjištěno, že existují dvě základní metody hledání klientů.

První metodou jsou různé discovery protokoly, pomocí kterých se klienti pokoušejí najít se svépomocí, nejčastěji pomocí nějakého multicast nebo broadcast požadavku. Nevýhodou této metody je, že není schopna najít všechny klienty z důvodu různých zařízeních, která zablokují posílání multicast a broadcast požadavků.

Druhá metoda se spoléhá na server, který všichni klienti znají a hlavně má kompletní databázi klientů včetně jejich adres. Z tohoto serveru si klienti mohou stáhnout aktuální

[1] Jenom pro výměnu samotných dat. Seznam souborů se vždy získává ze serveru.

seznam dostupných klientů. U takto nalezených klientů není zaručeno, že se s nimi bude možné spojit, protože se mohou zacházet za zařízeními se zapnutou funkcí NAT.

Klienty nalezené pomocí těchto dvou metod je možné uložit do paměti a v budoucnu při hledání klientů využít i tuto paměť. Nevýhodou dat v této paměti je, že nemusí být aktuální. Tato metoda může být vhodná v případě, že nebyli nalezeni klienti pomocí předchozích dvou metod. Porovnání těchto třech způsobů hledání je uvedeno v tabulce 2.2.

Tabulka 2.2: Porovnání způsobů hledání klientů

	Server	Multicast	Lokální databáze
Najde všechny klienty?	Ano	Ne	Ne
Pracuje bez připojení k serveru?	Ne	Ano	Ano
Získá aktuální data o klientech?	Ano[2]	Ano	Ne

2.3.1 WCF Discovery

Framework WCF přímo pro discovery obsahuje podporu[8]. Konkrétně využívá protokol WS-Discovery, který používá standardy webových služeb[9], konkrétně SOAP-over-UDP. Tento protokol využívá multicast adresu 239.255.255.250 a komunikuje přes UDP na portu 3702.

Výhodou je objevení nejbližších klientů bez použití jakékoliv třetí strany. Nevýhodou jsou omezené schopnosti objevení dalších klientů vyplývající z použití mulicast zpráv, které nemusí projít přes router, takže i když jsou dvě zařízení přímo vedle sebe a na stejné síti, neznamená to, že jsou schopná se navzájem najít. Dalším problémem, který se může objevit, je blokování komunikace firewallem.

Pro vyřešení těchto nevýhod tento framework obsahuje možnost vystavení služby zvané Discovery Proxy, která slouží jako repozitář všech aktivních služeb (klientů). Výhodou je, že metody pro ukládání a vyžádání si klientů je potřeba naimplementovat ručně, takže nic nebrání např. použití databáze. Další důležitou vlastností je, že použití této proxy se silně prováže s vyhledáváním pomocí multicastu, takže vše probíhá zcela automaticky. Konkrétně to znamená, že když klient vystaví službu, tak to oznámí službě proxy. Dále to znamená, že když klient zavolá metodu pro hledání služeb, tak framework to zkusí jak pomocí multicastu, tak pomocí proxy služby.

Velkou nevýhodou tohoto přístupu je, že se vyhledají všechny služby, což by vzhledem k tomu, že každý klient jednu službu vystavuje, znamenalo najití úplně všech aktivních klientů v internetu. Pro synchronizaci souborů je ale potřeba najít jenom ty klienty, kteří mají stejné soubory.

2.4 Komunikace v distribuovaném prostředí

Ve standardním režimu komunikace klient-server jsou role všech zařízení pevně daná. Komunikaci pokaždé iniciuje klient, který posílá požadavek na server. Server má známou, pevně danou URL adresu.

[2] Za předpokladu, že klient od výpadku spojení se serverem nezměnil síť

V distribuovaném prostředí se nabízejí dva možné způsoby komunikace[10][11]:

- Každý klient navazuje spojení s ostatními klienty a snaží se s nimi dostat do synchronního stavu.
- Jeden klient je zvolen za server a ostatní klienti potom komunikují pouze s tímto dočasným serverem.

V obou případech je ale důležité mít znalost alespoň o nějakých dalších klientech dostupných pro komunikaci.

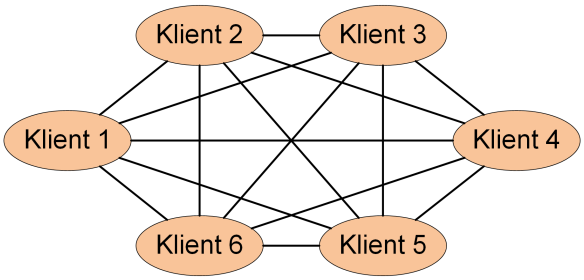
2.4.1 Režim komunikace peer-to-peer

V režimu komunikace peer-to-peer se tedy všichni klienti snaží dostat do synchronního stavu pomocí komunikace s ostatními klienty (obrázek 2.1). K tomuto se dá přistoupit následujícími dvěma způsoby:

- Request-response
- Publish-subscribe

Při použití způsobu request-response je komunikační kanál vytvářen jen na dobu vyřízení jednoho požadavku a následně je uzavřen[12]. Výhodou tohoto způsobu je malé množství otevřených konexí, které záleží pouze na počtu aktuálně zpracovávaných požadavků. Pro získání nových dat je potřeba se periodicky dotazovat ostatních klientů na změny.

Druhý způsob funguje na principu publish-subscribe. Při navázání komunikace mezi dvěma klienty se tito klienti k sobě navzájem zaregistrují a pokud dojde k nějaké změně entity, je tato informace rozeslána všem registrovaným klientům[13]. Takto registrovaní klienti mají neustále otevřený komunikační kanál, takže jejich počet je potřeba nějakým způsobem limitovat, protože jinak by existovalo tolik konexí, kolik by bylo klientů zapojených do synchronizace.



Obrázek 2.1: Diagram komunikace v režimu peer-to-peer

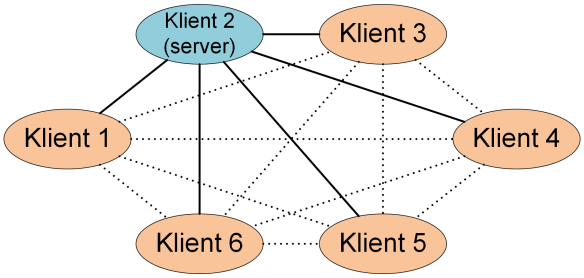
2.4.2 Režim komunikace klient v roli serveru

V případě použití tohoto režimu budou všichni klienti komunikovat úplně stejným způsobem i po přechodu z režimu komunikace klient-server do režimu v distribuovaném prostředí.

Jediné, co budou muset udělat, je domluvit se na volbě dočasného serveru a následně přesměrovat komunikaci na adresu tohoto nově zvoleného serveru (obrázek 2.2).

Samotná volba některého klienta za server je na tomto režimu to nejsložitější. Vzniknout musí pouze jeden server, a tudíž se na tom musí všichni klienti jednoznačně dohodnout. Podstatné je, aby takto zvolený server měl dostatečné síťové připojení a dostatečně výkonný hardware.

Problematické je, že jednoznačná volba serveru není zaručena. Každý dočasně zvolený server se může kdykoliv odpojit a pak je nutné zvolit nový. Při následném obnovení spojení s hlavním serverem se o nahrání entit na server musí postarat všichni klienti, kteří vystupovali v roli serveru.



Obrázek 2.2: Diagram komunikace v režimu klient v roli serveru

2.4.3 WCF Self-hosting

Standardně jsou služby nasazovány na nějaký webový server, jako je například server Internet Information Services. WCF služby jsou navrženy tak, aby je bylo možné kromě serveru IIS provozovat i přímo v desktopových aplikacích nebo Windows službách[14].

Nasazení WCF služby přímo v aplikaci se nazývá self-hosting a je použitelné pro zprovoznění peer-to-peer komunikace.

2.4.4 Algoritmy shody

U systémů pracujících v distribuovaném prostředí je v některých případech potřeba zajistit konsensus (shodu) mezi všemi zařízeními[15]. K tomuto účelu slouží algoritmy shody, mezi které patří například Paxos[15]. Princip těchto algoritmů spočívá v tom, že některé zařízení odešle ostatním návrh a ty ho mohou přijmout nebo odmítnout. V případě většinového přijetí je tento návrh prohlášen za řešení.

V případě navrhovaného systému by bylo možné použití některého z těchto algoritmů pro řešení konfliktů při komunikaci bez dostupného serveru. Tyto algoritmy ale vyžadují schopnost komunikace mezi všemi zařízeními a příliš nepočítají s jejich dynamickým přibýváním a ubýváním, takže může docházet ke vzniku konfliktu v řešení konfliktu. Vzhledem k těmto vlastnostem není použití těchto algoritmů příliš vhodné.

2.5 Synchronizace entit a konflikty

Obecně synchronizace je situace, kdy více událostí probíhá zároveň a je potřeba zajistit, aby všechny části systému byly ve stejném stavu, nebo aby provedly určité operace ve správný čas. Existuje tedy více druhů jako například v případě počítačů synchronizace procesů nebo vláken. V případě entit se bude jednat o datovou synchronizaci.

Cílem datové synchronizace je dosažení synchronního stavu entit[16][17], neboli zajištění konzistentních dat jednotlivých entit na všech zařízeních. Takže při změně entity na jednom zařízení se tato změna musí projevit na všech ostatních zapojených do synchronizace. Synchronního stavu by měla zařízení dosáhnout, jak nejrychleji to bude možné.

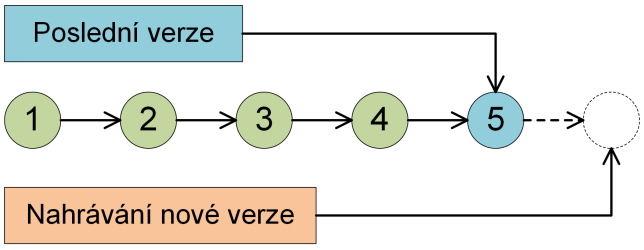
Jednodušší situace nastává, pokud se jedná o jednosměrnou synchronizaci, tedy že změny vznikají pouze v jednom zařízení a tyto změny se pak odesílají do libovolného množství jiných zařízení.

Složitější situace je v případě obousměrné synchronizace, kde vzhledem k tomu, že dosáhnutí synchronního stavu v reálném čase často bývá nemožné ať už z hlediska rychlosti nebo nedostupnosti síťového připojení, mohou vznikat konflikty. Tyto konflikty nastávají při modifikaci jedné entity na dvou nebo více místech současně (nebo po dobu nedostupného připojení). V tomto případě je potom náročné rozhodnout, která verze je aktuální.

Tato práce se zabývá obousměrnou datovou synchronizací, takže následující podkapitoly popisují vznik a řešení konfliktů.

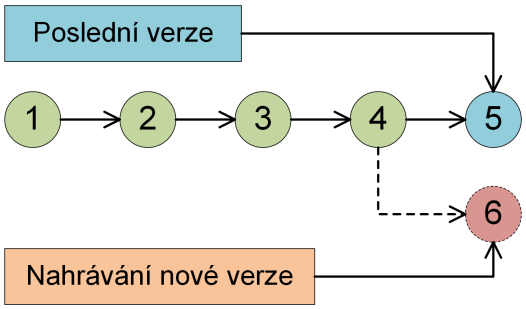
2.5.1 Vznik konfliktů

V první řadě je konflikty potřeba nějakým způsobem detekovat. To se dá udělat tím, že si každá entita bude držet svoji historii verzí a při nahrávání nové verze entity na server bude rovnou řečeno, na jakou verzi entity se tato data nahrávají (obrázek 2.3). Když se potom dvě a více verzí nahrává na jednu konkrétní verzi, znamená to vznik konfliktu (obrázek 2.4).



Obrázek 2.3: Diagram nahrávání nové verze a její zařazení do historie

Dalším typem, složitějším na detekci, je konflikt mezi různými entitami. Tyto konflikty vznikají opět nahráním nové verze, ale nevznikají tím, že by se dvě verze nahrávaly na jednu konkrétní, ale vznikají nějakým dalším omezením daného systému. Jako příklad lze uvést případ, že nemohou existovat dvě auta se stejnou registrační značkou. Dalším typickým příkladem je souborový systém, kdy ve stejné složce nemohou existovat dva soubory se stejným názvem. V případě souborů se do konfliktu tedy lze dostat pouhým přejmenováním, nebo přesunutím souboru.



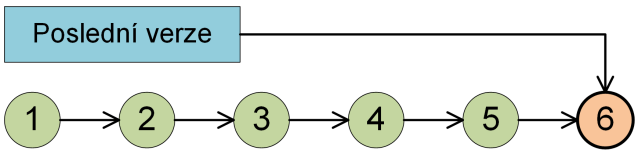
Obrázek 2.4: Diagram vzniku konfliktu z hlediska historie verzí

2.5.2 Řešení konfliktů verzí

V prostředí klient-server rozhoduje o aktuální verzi server, který k tomu může přistoupit několika způsoby[16]. Při komunikaci peer-to-peer je rozhodování o aktuální verzi náročnější, protože v tu chvíli zde nevystupuje žádný nadřazený prvek, který by měl konečné slovo. V této práci se ale počítá s existujícím serverem v pozadí, takže klienti mohou počkat, až bude server dostupný a nechat výsledné rozhodnutí na něm.

Jak už bylo zmíněno, pro detekci konfliktu lze použít historii verzí (ID verze entity) a následně pro řešení konfliktu je možné uchovávat čas vytvoření entity. Pokud server udržuje pouze aktuální verzi entity, tak možná řešení jsou následující:

- Konfliktní verzi nastavit jako aktuální (obrázek 2.5)
- Pokusit se o sloučení obou verzí
- Vytvořit novou entitu a tím zachovat obě verze (obrázek 2.6)
- Konfliktní verzi zahodit

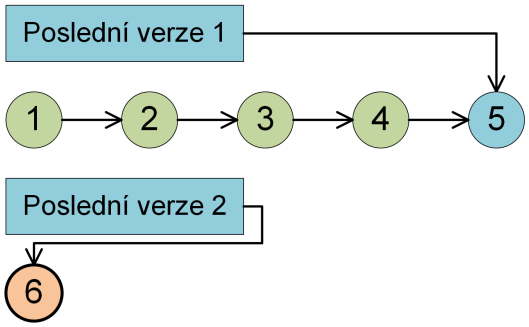


Obrázek 2.5: Diagram řešení konfliktu nastavením aktuální verze

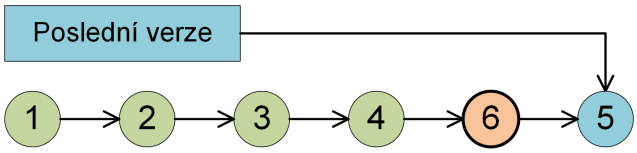
Jestliže ale server udržuje historii entit (verzování), tak mu přibyde ještě jedna možnost řešení a to zařazení konfliktní verze do historie (obrázek 2.7). Tím uživatel o žádná data nepřijde a může se k ni kdykoliv vrátit.

2.5.2.1 Poslední nahraná je aktuální

Nejjednodušší a zároveň nejhloupější způsob řešení je, že se server bude tvářit, jako že žádné konflikty neexistují, a když mu někdo nahraje novou verzi, tak tato bude brána jako aktuální. Problémem tohoto přístupu je jednak ztráta dat, ale významnější problém je, že uživatel se o této ztrátě žádným způsobem nedozví.



Obrázek 2.6: Diagram řešení konfliktu vytvořením nové entity



Obrázek 2.7: Diagram řešení konfliktu zařazením do historie

2.5.2.2 Automatické řešení

V některých případech může server jednoduše rozhodnout sám. Nejjednodušší je porovnání, jestli jsou obě entity stejné, což lze na souborech provést například porovnáním jejich hash kódu. Pokud jsou tedy obě verze shodné, je následně jedno, která entita bude vybrána jako aktuální.

Za předpokladu, že obě verze nejsou stejné, tak by pro uživatele bylo ideálním řešením sloučení změn z obou entit, což lze provést pouze v některých případech. Obvykle je to možné pouze nad textovými soubory, ale pouze za předpokladu, že se nejedná o konfliktní změnu uvnitř tohoto souboru (např. v každé entitě jiná úprava stejného řádku). Přesně pro tyto účely existují hotové algoritmy a jsou použity například ve verzovacím systému Git.

Dalším možným řešením je využití data modifikace entity, kdy se jako aktuální verze bere ta s nejvyšším časem. V tomto případě ale nastává problém nesynchronního času mezi klienty a serverem. Na tento údaj se tedy dá spolehnout pouze v případě, že se tyto časy shodují. Pokud se ale rozcházejí (jiné časové pásmo, letní čas, špatně seřízené minuty), tak nelze jednoznačně rozhodnout, která verze vznikla později. Pro zaručení funkčnosti by bylo potřeba zařídit synchronizaci hodin na všech zařízeních.

Když není zařízen synchronní čas na všech zařízeních, tak jediný spolehlivý čas je ten na serveru. Sice se podle něho dá řídit a určovat, která entita je aktuálnější, ale nemusí to být rozhodnuto správně. Není totiž zaručeno, že ta entita, která přijde na server jako poslední, je opravdu ta nejnovější. Tato entita mohla vzniknout kdysi v minulosti, ale zařízení do aktuální chvíle nebylo připojeno k internetu. Takže toto řešení není rozhodně vhodné.

2.5.2.3 Řešení konfliktu uživatelem

V případě řešení konfliktů uživatelem si server nechá nahrát všechny verze a při detekci konfliktu nějaké verze je tato verze označena jako konfliktní a uživatel je vyzván, aby tuto

situaci vyřešil. Nabízená řešení mohou být následující:

- Nahrazení aktuální verze
- Zařazení do historie
- Zahození
- Zachování obou verzí jako aktuálních (duplikace entity)

Způsob řešení konfliktu pomocí nahrazení je označení právě příchozí verze jako aktuální. Původní verze je tímto přesunuta do historie. Zcela opačně funguje způsob zařazení entity do historie, kdy serverová aktuální verze zůstává nezměněna, ale nově příchozí verze je zařazena do historie, jako kdyby vznikla před aktuální serverovou verzí.

Pokud to daný systém povoluje, tak uživatel může zvolit řešení konfliktu zahozením nově příchozí verze. Tím je konflikt vyřešen velice snadno, ale za cenu toho, že uživatel tím přichází o kompletní historii změn entit.

Zajímavou další možností řešení konfliktu je zachování obou entit jako aktuálních. Toho je docíleno tím, že na serveru je vytvořena úplně nová entita, která s tou původní nemá nic společného. Pouhá duplikace ale nemusí stačit kvůli možnému konfliktu mezi entitami, takže například v případě souborů bude nutné nově vzniklý soubor přejmenovat (běžně se používá původní název s přidaným postfixem např. číslem nebo názvem zařízení).

2.5.3 Konflikty entit

Obecně se nedá říct, v jakém případě vznikají konflikty mezi entitami. Tyto konflikty silně závisí na typu entit a na omezení daného systému. V některých případech třeba tyto konflikty nemusí vůbec existovat, protože se entity vůbec do konfliktu dostat nemohou, což může být například galerie obrázků, kde je úplně jedno, že se dvě entity jmenují stejně.

Pokud se tedy ale entity mohou dostat do konfliktu (dva stejně pojmenované soubory) je potřeba to řešit. Tyto konflikty mohou nastat kdykoliv při nahrání nové entity nebo nové verze.

2.5.3.1 Řešení konfliktů

Pro řešení konfliktu dvou entit existují opět dva možné přístupy, kdy situaci vyřeší buď server sám automaticky, nebo nechá rozhodnutí na uživateli.

V tomto případě mohou automatická řešení zafungovat velice dobře, ale je nutné o tom informovat uživatele. Nelze jednoznačně určit, jaké řešení bude pro danou entitu a systém nejvhodnější. V některých situacích může být nejlepší jednoduše nahrání této verze odmítnout jako chybné. V jiných případech se může do dané entity nějak zasáhnout, aby nebyla konfliktní (např. již zmíněné přejmenování souboru).

Výše popsané způsoby mohou proběhnut buď zcela automaticky, nebo je uživatel vyzván, aby si to rozhodl podle sebe.

Dalším trochu drastickým způsobem je, že nově příchozí verze zůstane nezměněna, ale s původní se tedy musí něco udělat. Nabízená řešení jsou například původní verzi přejmenovat, nebo ji rovnou smazat (operace nahrazení). Tento způsob by ve většině případů měl být dostupný pouze jako uživatelova volba.

2.6 Verzování v distribuovaném prostředí

V distribuovaném prostředí neexistuje žádný centrální rozhodovací prvek[17], takže každý klient musí být schopen rozhodnout pořadí vzniku verzí a zároveň detekovat vznik konfliktů. Tento princip se nazývá kauzalita.

Použití jednoduchých časových značek nestačí. Tento mechanismus sice dokáže rozhodnout, která verze vznikla dříve a která později, ale nedokáže zjistit, jestli některé verze vznikly současně, a tudíž jsou mezi sebou v konfliktu. Další nevýhodou je nutnost zajistit přesnou synchronizaci času na všech zúčastněných klientech.

2.6.1 Verzovací vektor

Mechanismus verzovacích vektorů řeší oba zmíněné problémy[18][19]. Neboli je schopný rozhodovat o pořadí vzniku jednotlivých verzí a zároveň i detekovat současný vznik dvou verzí (konflikt). Výhodou je, že ke svému fungování nepotřebuje čas, takže není potřeba zajistit synchronizované hodiny na klientech.

2.6.1.1 Princip fungování

Verzovací vektor obsahuje tolik prvků, kolik replik je zapojeno do synchronizace. Každá pozice v tomto vektoru odpovídá právě jedné konkrétní replice. Číselná hodnota na každé pozici odpovídá verzi vytvořené příslušnou replikou.

Každá replika si udržuje čítač, který reprezentuje poslední číslo verze vytvořené danou replikou.

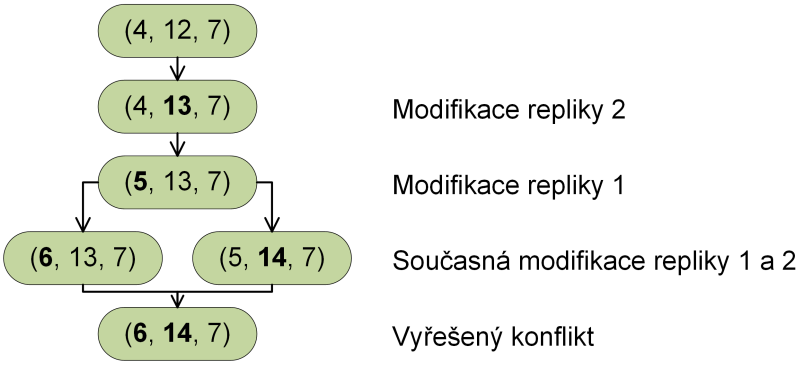
2.6.1.2 Pravidla fungování

- Při první inicializaci jsou všechny čítače nastaveny na nulu a vektory jsou také nastaveny jako nulové.
- Při lokální změně repliky se čítač inkrementuje o jedna a ve verzovacím vektoru se příslušná pozice dané repliky nastaví na aktuální hodnotu čítače.
- Při synchronizaci dvou replik si obě repliky zaktualizují verzovací vektor: pro každou pozici (i) ve vektoru platí, že se vezme ta větší hodnota z obou původních vektorů (V_A a V_B) a ta se použije. Vzorec výpočtu je následující $V_A[i] = V_B[i] = \max(V_A[i], V_B[i])$

Porovnáním dvou verzovacích vektorů lze bezpečně zjistit, jestli jsou dvě verze identické, současné (konfliktní) anebo rozhodnout pořadí jejich vzniku (obrázek 2.8).

- Identické verze – oba vektory musí mít na každé pozici stejné hodnoty. Např. $(5, 8, 9) = (5, 8, 9)$
- Vektor V_A vznikl dříve než vektor V_B – pro oba vektory musí na každé pozici platit, že $V_A[i] \leq V_B[i]$ a minimálně v jednom případě musí platit, že $V_A[i] < V_B[i]$. Např. $(5, 8, 9) < (5, 41, 10)$

- Vektor V_B vznikl dříve než vektor V_A – platí stejná pravidla jako v předchozím bodě, jenom vektory jsou prohozené. Např. $(5, 41, 10) > (5, 8, 9)$
- Současné (konfliktní) verze – platí pokud vektory nesplňují předchozí pravidla (alespoň na jedné pozici platí $V_A[i] < V_B[i]$ a na nějaké jiné $V_A[j] > V_B[j]$). Např. $(5, 8, 53) \parallel (45, 8, 9)$



Obrázek 2.8: Diagram aktualizace verzovacího vektoru. Tučná čísla reprezentují provedenou aktualizaci ve verzovacím vektoru.

Princip verzovacích vektorů je podobný principu verzovacích hodin[20], přičemž oba mechanismy využívají vektorů celých čísel, kde tato čísla vždy reprezentují jeden zdroj. Podstatný rozdíl je v tom, že verzovací hodiny slouží k určování pořadí vzniku událostí, kdežto verzovací vektory se spoléhají pouze na stav replik a neřeší, že došlo k synchronizační události.

Mechanismus verzovacího vektoru tak, jak je popsán výše, počítal s pevným počtem replik, což by byl problém. Tento problém jde snadno vyřešit tím, že vektor bude umožňovat dynamické zvětšování velikosti a jeho jednotlivé hodnoty budou místo čísla verze obsahovat dvojici hodnot obsahující identifikaci klienta a číslo verze. Díky této změně už nebude záležet na konkrétním pořadí prvků uvnitř vektoru. Vektor bude vypadat např. takto: $\Big([ClientA, 44], \quad [ClientD, 81], \quad [ClientC, 6]\Big)$

2.6.1.3 Výhody a nevýhody

Výhody:

- Vždy dokáže bezpečně rozhodnout, která verze vznikla dříve a případně, že jsou obě verze v konfliktu.
- Schopnost zapojit do synchronizace i další nové klienty.

Nevýhody:

- Datová náročnost. Při rostoucím počtu klientů zároveň roste celková délka vektoru. Tato délka se už následně nezkracuje, protože by následně nebylo možné bezpečně detekovat pořadí a konflikty.

Tato nevýhoda se dá vyřešit tím, že staré hodnoty některých klientů se z verzovacího vektoru odeberou. Tímto způsobem se tedy sníží velikost vektoru, ale zvýší se riziko, že budou detekovány falešné konflikty.

Nevýhody verzovacích vektorů řeší například mechanismus Dotted Version Vector, jehož cílem je zefektivnění způsobu sledování kauzality v distribuovaném úložišti[21].

2.6.2 Konflikty v distribuovaném prostředí

Jako mechanismus pro detekci konfliktů v distribuovaném tedy může posloužit verzovaný vektor. V okamžiku, kdy je konflikt detekován, vědí o něm pouze ti dva klienti, mezi kterými byl detekován. V tuto chvíli tedy nastávají dvě možnosti. První variantou je nechat všechny klienty synchronizovat se dále a tedy nechat tento konflikt postupně distribuovat mezi všechny dostupné klienty. Druhá možnost je okamžité automatické vyřešení konfliktu. V tomto druhém případě se ostatní klienti o vzniklém konfliktu vůbec nedovědí a synchronizují si už pouze novou nekonfliktní verzi.

2.6.2.1 Ruční řešení konfliktu

Výhodou tohoto řešení je, že uživatel je upozorněn, že ke konfliktu došlo a jsou mu nabídnuty možnosti jak tento konflikt vyřešit:

- Nahradit původní entitu příchozí entitou.
- Příchozí entitu zahodit.
- Vytvořit úplně novou entitu a tím zachovat obě.

Nevýhodou je, že o vzniklém konfliktu je potřeba lokálně udržovat různé informace, mezi které patří:

- Na jaké verzi entity tento konflikt vznikl.
- S jakou verzí entity je v konfliktu.
- Jakým způsobem byl tento konflikt vyřešen.

Další vlastností tohoto způsobu je, že informace o konfliktu jsou rozdistribuovány mezi dostupné klienty. Z toho sice plyne uživatelská výhoda, že konflikt se dá vyřešit z libovolného místa, ale nevýhodou je, že dva klienti mohou v jeden okamžik každý zvolit jiné řešení konfliktu a vytvořit tím konflikt v řešení konfliktu. Dále je pak potřeba toto řešení rozdistribuovat mezi ostatní klienty.

2.6.2.2 Automatické řešení konfliktu

Automatický způsob řešení konfliktu je jednodušší, protože není potřeba vytvářet další mechanismy pro distribuci konfliktů a jejich řešení, ale využije se již existující mechanismus pro vytváření nových verzí entit a jejich synchronizaci.

Pro fungování tohoto mechanismu je potřeba provést rozhodnutí, jakým způsobem bude konflikt automaticky řešen. Nahrazení původní entity příchozí entitou nebo zahození této příchozí entity není vhodné řešení, protože uživatel takto může přijít o důležitá data. Nejvhodnějším řešením je zachování obou entit, přičemž jedna musí být pozměněna, aby nebyla s druhou v konfliktu (např. přejmenovat soubor).

2.7 Technická řešení

Pro celý tento systém bylo potřeba zvolit nějaké prostředí nebo framework, který by umožňoval následující body:

- Byl použitelný pro klientskou i pro serverovou část systému.
- Umožňoval jednoduché navázání komunikace v režimech klient-server i peer-to-peer.
- Umožňoval jednoduchou změnu typu entit vyměňovaných v komunikačním protokolu.

Jako nejzásadnější bod se ukázal požadavek na jednoduchou změnu typu entit, který by měl zaručit co nejjednodušší použití tohoto systému jako frameworku. Ve většině případů je totiž při definici komunikačního rozhraní potřeba přímo specifikovat všechny datové typy parametrů i návratových hodnot.

V tomto případě se jako výhodný ukázal framework WCF, který je součástí .NET Frameworku od verze 3.0[22] a který v definici rozhraní umožňuje použít genericitu[23], pomocí které je možné definovat konkrétní typ entity až při konkrétní implementaci použití synchronizačního frameworku. Tento generický interface je generický pouze do doby kompilace, protože výsledné vygenerované komunikační rozhraní už musí obsahovat pevně specifikované datové typy. Pro účely tohoto systému je tento interface ale plně dostačující.

Framework WCF dále umožňuje jednoduché vytváření komunikace v režimu klient-server (vytvoření serverové i klientské části) i v režimu peer-to-peer[24]. Další výhodou je že WCF obsahuje discovery protokol, který umožní vyhledávání klientů pro komunikaci P2P.

Z výše uvedených důvodů byl pro implementaci tedy zvolen .NET Framework 4.5 a programovací jazyk C#[25][26]. Vzhledem k této volbě byl pro uživatelské rozhraní zvolen WPF framework. Tento framework umožňuje snadné vytváření uživatelského rozhraní a odděluje od sebe vzhled (definovaný v XAML souboru) od funkčnosti[27] (kód v C#).

2.7.1 Inversion of Control framework

Inversion of Control (zkráceně IoC) je návrhový princip pro vytváření aplikací[28]. Základním principem je otočení řízení vykonávání programu, kdy je uživatelský kód volán z frameworku a tudíž tento framework musí mít o tomto kódu nějakou základní znalost.

Jedná se o opačný princip než u klasických knihoven, kdy uživatelský kód musí znát konkrétní metody uvnitř knihovny, aby je mohl zavolat.

Součástí IoC frameworku je Container, který se stará o správu tříd, kdy zajišťuje jejich vytváření, správu životního cyklu a jejich konfiguraci. Takovýto Container také využívá návrhový vzor Dependency injection pro vyřešení závislostí mezi třídami[29]. Tímto je umožněno snižování závislostí mezi jednotlivými komponentami a zvyšování modularity programu. Výhodou je, že se programátor tedy nemusí starat odkud se tyto komponenty vezmou, protože o to se postará IoC framework.

Princip IoC je vhodný i pro tvorbu synchronizačního frameworku, protože ten bude obsluhovat všechnu synchronizační logiku a zároveň volat uživatelský kód pro vykonávání jenom některých akcí (převážně čtení a ukládání entit).

Pro používání IoC Containeru v .NET Frameworku je možné použít knihovnu Castle Windsor[30] 3.3.0. Tento Container umožňuje konfiguraci pomocí XML souboru, takže není nutná rekompilace. Tato knihovna je publikována pod licencí Apache License Version 2.0, která umožňuje volné použití[31].

2.7.2 Databáze

Pro ukládání dat je vhodné použít relační databázi. Pro přístup k databázi je možné použít framework pro objektově relační mapování (ORM), který mapuje databázové záznamy na objekty a opačně. Součástí těchto frameworků bývá i jiný způsob vytváření dotazů, který odstraňuje závislost na konkrétním SQL dialektu a tedy umožňuje snadnější výměnu relační databáze.

Na serveru je tedy možné použít třeba Microsoft SQL Server. Na klientovi je ale potřeba použít nějakou lokální databázi, která bude sloužit pouze pro danou aplikaci a nebude nutné ji instalovat. Tyto požadavky splňuje databáze SQLite[32], která ukládá data do jednoho zvoleného souboru. Obsluha databáze probíhá přes knihovnu SQLite, která je distribuovaná pod licencí Public domain umožňující libovolné použití[33].

V prostředí .NET Framework se pro ORM nabízí pokročilý nástroj NHibernate[34] 4.0.4, který je portem frameworku Hibernate, určeným pro jazyk Java. NHibernate je publikován pod licencí GNU Lesser General Public License Version 2.1. V případě dynamického linkování knihovny tato licence umožňuje používání v aplikaci s libovolnou licencí[35].

Transakce v NHibernate lze buď používat ručně zavoláním příslušných metod, nebo lze využít výhod integrace NHibernate do Castle Windsor a nechat si volat příslušné metody pro používání transakcí automaticky. Pro toto automatické volání je potřeba v kódu označit metody přistupující k databázi atributem Transaction.

2.7.3 Logovací nástroj

Vzhledem k funkcionalitě navrhovaného systému, kde bude probíhat různá síťová komunikace a zároveň bude běžet více vláken pro zpracování různých požadavků, bude potřeba pro účely ladění programu použít nějaký logovací nástroj.

V .NET Frameworku je možné pro logování použít Apache log4net[36] 1.2.13, který je portem Apache log4j z jazyku Java. Výhodou této knihovny je možnost logovat do velkého

množství různých výstupů včetně možnosti vytvořit si vlastní výstup. Tato knihovna je distribuována pod licencí Apache License Version 2.0.

2.7.4 Řešení pro uživatelské rozhraní

Vzhledem k tomu, že součástí této práce je i jednoduchá ukázková aplikace pro synchronizaci souborů a tato aplikace má mít uživatelské rozhraní, je vhodné zvolit Microsoftem vytvořený architektonický vzor Model-View-ViewModel[37] (zkráceně MVVM). Hlavním účelem tohoto vzoru je oddělení vývoje uživatelského rozhraní od vývoje business logiky.

Pro lepší dodržení tohoto vzoru je možné použít nástroj MVVM Light Toolkit 5.2.0, jehož účelem je urychlit vývoj MVVM aplikací[38]. Tato knihovna je distribuována pod licencí MIT, která umožňuje volné použití[39].

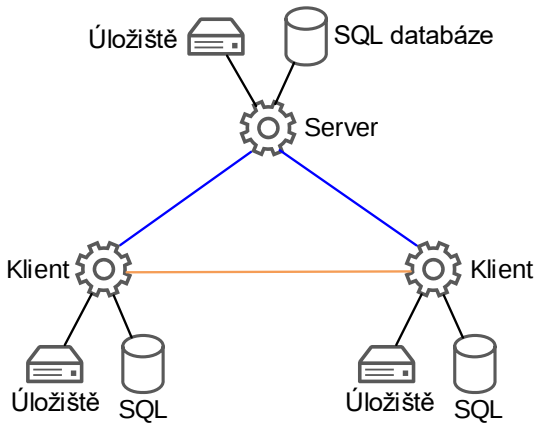
Kapitola 3

Návrh

Návrh řešení se zabývá několika důležitými částmi. Popsán zde je základ architektury, zvolený způsob hledání ostatních klientů, mechanismy potřebné pro synchronizaci, zvolený komunikační protokol, architektura systému, základní operace pro synchronizaci, návrh databáze a uživatelské rozhraní klientské aplikace pro synchronizaci souborů.

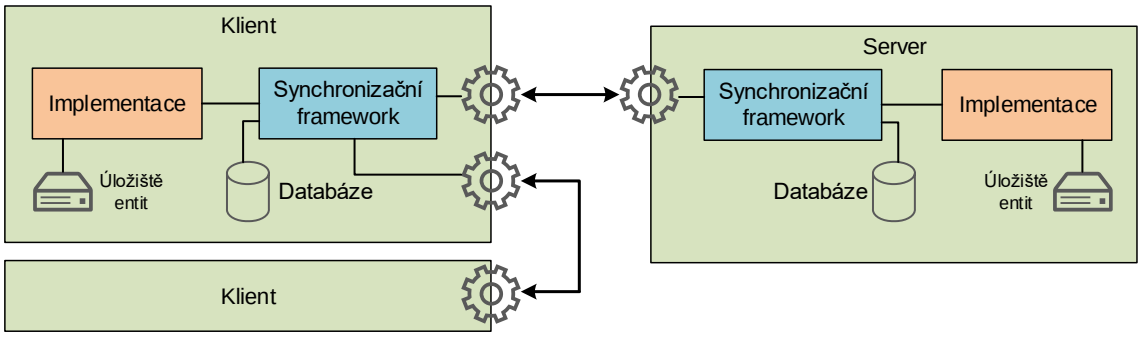
3.1 Základní popis architektury

Framework se bude skládat z několika samostatných částí. Nejabstraktnější rozdělení je na serverovou a klientskou část (obrázek 3.1). Klientská část ve výsledku bude mnohem složitější, protože bude vystupovat ve více rolích a to konkrétně jako klient v komunikaci se serverem a jiným klientem, nebo jako server pro jiného klienta.



Obrázek 3.1: Základní rozdělení frameworku. Ozubená kola reprezentují klietskou nebo serverovou aplikaci včetně komunikačních rozhraní. Modrá spojnice znázorňuje spojení v režimu komunikace klient-server a oranžová spojnice znárorňuje přímé spojení mezi klienty.

Další rozdělení jak na serveru, tak v klientovi nastane v oddělení frameworku pro synchronizaci obecných entit a v samotném použití tohoto frameworku pro synchronizaci souborů.



Obrázek 3.2: Rozdělení částí frameworku. Ozubená kola reprezentují vystavená komunikační rozhraní.

O celou komunikaci mezi dvěma uzly (klient-server, nebo peer-to-peer) se bude starat sám framework, takže použití tohoto frameworku bude spočívat v tom, naimplementovat samotné ukládání a čtení dat včetně možnosti volby libovolného úložiště (obrázek 3.2). Veškerá data týkající se frameworku a jeho konfigurace budou uložena v relační databázi (toto platí pro serverovou i klientskou část).

3.1.1 Datové úložiště pro synchronizaci souborů

Pro ukládání souborů na serveru i na klientovi bude použita kombinace dvou úložišť a to relační databáze pro ukládání metadat souborů a souborový systém pro konkrétní data souborů.

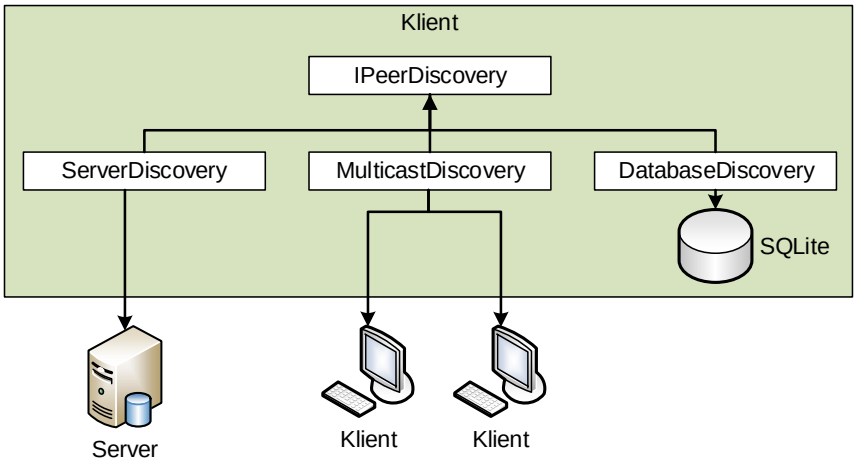
Na serverové straně bude vytvořeno jednoduché úložiště. Data v tomto úložišti se budou fyzicky nacházet ve vybrané složce na disku. Jejich jména budou nějaké unikátní identifikátory. Každý soubor bude mít tento identifikátor uložený v příslušném záznamu mezi metadaty v databázi.

Na klientské straně budou soubory také uloženy přímo ve vybrané složce na disku, ale s tím rozdílem, že se bude jednat běžnou složku, takže soubory zde budou pod svým jménem a měla by se tady normálně tvořit stromová struktura složek. Zároveň tato složka bude monitorována, aby při změně souboru byla tato změna synchronizována. Každý soubor bude opět mít příslušný záznam v databázi, který bude obsahovat cestu k souboru.

3.2 Hledání klientů

Mechanismus pro hledání klientů bude využívat celkem tři způsoby vyhledávání (obrázek 3.3). Prvním způsobem bude přímé vyhledání pomocí multicast v síti LAN, které bude probíhat pomocí WCF Discovery, který bude schopný nalézt jenom nejbližší klienty na síti LAN. Nevýhodou je, že ne vždy se může podařit nalézt všechna dostupná zařízení na stejné síti, protože v cestě může být router nebo nějaké jiné zařízení, které multicast nepropustí.

Z toho důvodu bude k dispozici druhý mechanismus, který bude využívat databázi na serveru obsahující IP adresy klientů. Toto řešení umožní vyhledat i klienty, kteří spolu mohou jednoduše navázat P2P spojení, ale nacházejí se v jiné síti.



Obrázek 3.3: Hledání klientů znázorněné z hlediska použitých tříd a rozhraní.

Třetím způsobem vyhledávání bude malá databáze klientů uložená přímo u klienta, která bude obsahovat pouze relevantní záznamy (IP adresy klientů se stejnými soubory). Tato klientská databáze bude aktualizována pomocí předchozích dvou mechanismů. Díky této databázi klienti získají širší znalost IP adres jednotlivých zařízení i při výpadku připojení k serveru. Zároveň si budou moct při navázání spojení s jinými klienty tyto znalosti vyměňovat.

3.3 Mechanismy pro synchronizaci dat

V této podkapitole jsou popsány důležité mechanismy pro fungování synchronizace včetně režimů synchronizace klient-server a peer-to-peer i jejich přepínání.

3.3.1 Identifikace entit

Synchronizace mezi klientem a serverem je ten jednodušší způsob výměny dat, protože to, co bude na serveru, bude vždy ta správná aktuální verze a klienti si svoje entity postupně zaktualizují.

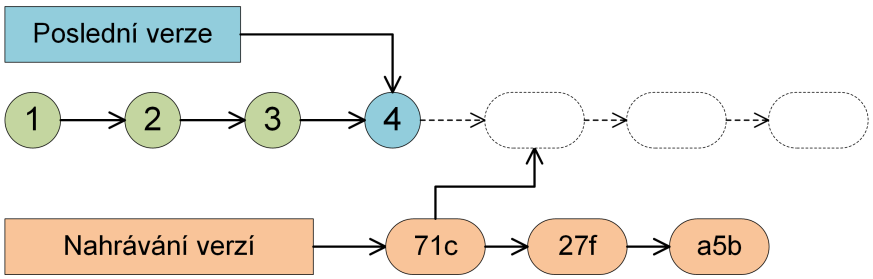
Pro jednoznačnou identifikaci entit bude sloužit serverové ID, které vygeneruje serverová databáze[40]. Klienti si toto ID budou ke každé entitě ve své databázi poznamenávat jako ServerID.

V případě synchronizace mezi dvěma klienty je situace komplikovanější. Za předpokladu, že klient bude znát serverové ID a pouze si vyměňovat data s jiným klientem, bude to celkem bez problému, protože se bude jednat o správná data (potvrzená serverem).

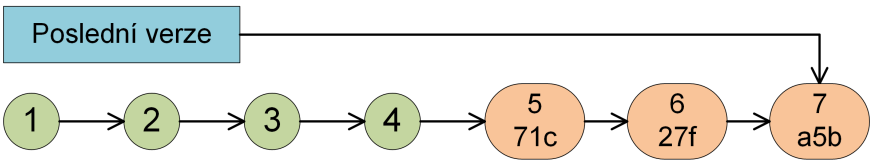
Komplikovanější situace to bude v případě, když server přestane být dostupný a data na klientovi se změní. Za normálních okolností by klient tato data odeslal na server a ten mu vrátil vygenerované ID. V tomto případě tomu ale tak nebude, takže si klient bude muset vygenerovat nějaký vlastní globálně unikátní identifikátor[40] (GUID), na základě kterého si bude vyměňovat data s ostatními klienty. Generátor tohoto identifikátoru je obsažen

v .NET Frameworku a je navržen tak, aby pravděpodobnost, že budou vygenerovány dva stejné identifikátory, byla statisticky zanedbatelná[41].

Z tohoto důvodu bude potřeba, aby si klienti kromě serverového ID u entit udržovali také jejich GUID, pokud to tedy zrovna bude potřeba.



Obrázek 3.4: Diagram nahrávání historie entit na server. Čísla reprezentují serverová ID a řetězce 71c, 27f a a5b reprezentují identifikátory GUID.



Obrázek 3.5: Diagram historie entit nahraných na server. Čísla reprezentují serverová ID a řetězce 71c, 27f a a5b reprezentují identifikátory GUID.

Při následném obnovení spojení se serverem bude potřeba tato data nahrát na server (obrázek 3.4), protože jak už bylo zmíněno, to je to jediné místo určitě obsahující správné verze. Jelikož se o nahrání stejné verze může pokusit více klientů najednou, bude si toto GUID muset udržovat i server (obrázek 3.5) a díky tomu tedy bude schopný detekovat, že se daná verze už na serveru nachází. Jakmile tedy bude klient mít u všech entit vyplněná serverová ID, tak budou tato data kompletně synchronní s daty na serveru.

3.3.2 Verzování entit

Pro správné vytváření historie entit poslouží časová značka CreateDate, která bude určovat čas vzniku dané verze entity. Pro následnou rekonstrukci historie bude tedy stačit seřadit záznamy podle této značky.

Každé zařízení si k vytváření této časové posloupnosti bude používat svoje vlastní hodiny, protože na časy ostatních zařízení (serveru nebo klientů) se nelze spolehnout z důvodu nesynchronizovaných hodin.

Při komunikaci klient-server i peer-to-peer bude potřeba porovnávat dvě verze a zjišťovat, která z nich je novější, případně jestli jsou v konfliktu. Časové značky se k tomuto účelu nehodí právě z důvodu rozdílně nastavených hodin na různých zařízeních.

Při nahrávání entit na server bude stačit použít údaj obsahující, která verze entity byla předchozí (a tudíž kam se má nahrávaná verze zařadit). Aby stejná verze nebyla nahrána vícekrát, bude potřeba podle GUID kontrolovat, zda už nahraná nebyla.

V přímé komunikaci mezi klienty bude potřeba použít verzovací vektory, jejichž porovnáním lze jednoznačně rozhodnout, která verze je novější, nebo zda jsou v konfliktu.

3.3.3 Skupiny entit

Každá entita bude patřit do nějaké skupiny. Každá tato skupina se bude nazývat Client pool. Účelem těchto Client poolů bude seskupit k sobě entity, které k sobě nějakým způsobem patří a naopak oddělit od sebe entity, které k sobě nepatří. Díky tomu bude možné na klientovi synchronizovat jenom entity ve zvolených Client poolech.

Mechanismus Client poolů bude také umožňovat sdílení entit tím, že každou entitu bude možné přiřadit do více Client poolů.

Zároveň díky tomuto mechanismu bude možné filtrovat vyhledávání klientů dostupných pro peer-to-peer komunikaci.

3.3.4 Model komunikace mezi klienty

V distribuovaném prostředí (při nedostupném serveru) bude použit režim komunikace peer-to-peer. Žádný klient nebude přebírat roli dočasného serveru, protože by nebyla zaručena jeho jednoznačná volba. Problém by byl hlavně v tom, že každý klient může patřit do různé množiny Client poolů (do různých skupin), ale i každá entita může patřit do různé množiny Client poolů. Takže by buď musel existovat klient, který by v roli serveru obsloužil všechny klienty (musel by obsluhovat i Client pooly, o kterých nic neví), nebo by musel pro každý Client pool být zvolen dočasný server (v tomto případě by mohlo nastat, že se o stejnou entitu bude starat více serverů).

Pro jednoduché přepínání mezi režimy komunikace klient-server a peer-to-peer bude vhodné zajistit podobnost komunikačních rozhraní. Z toho tedy vyplývá, že v režimu peer-to-peer budou klienti fungovat na principu request-response, a tedy že se budou dotazovat ostatních klientů na nové entity.

3.3.5 Komunikace

Pokud vznikne nějaká nová entita nebo nová verze, bude potřeba zavolat příslušnou metodu frameworku, který se již postará o její synchronizaci. V obou komunikačních režimech se klient nejdříve pokusí danou entitu nahrát na server. Teprve po selhání nebo úspěšném dokončení nahrávání bude tato entita klientem nabídnuta pro synchronizaci v komunikačním režimu peer-to-peer.

Nové entity bude framework získávat pomocí periodického dotazování. V režimu komunikace klient-server se bude klient dotazovat serveru a v režimu komunikace peer-to-peer bude při dotazu nejdříve zvolen některý jiný aktivní klient a následně na něm bude tento dotaz proveden.

Toto získávání nových entit bude provedeno ve dvou fázích. V první fázi bude proveden dotaz na seznam nových entit (nebo verzí). Ve druhé fázi budou tyto entity postupně stahovány. Tento postup je potřebný pro synchronizaci složitějších entit jako jsou třeba soubory.

V režimu komunikace peer-to-peer bude funkčnost frameworku omezená pouze na synchronizaci entit. Řešení konfliktů, vytváření Client poolu nebo přiřazování entit do Client poolu bude fungovat pouze při fungujícím spojení na server.

3.3.6 Přepínání mezi režimy komunikace klient-server a peer-to-peer

Za normálních okolností, kdy bude dostupné připojení k serveru, budou klienti operovat v režimu komunikace klient-server. Při detekci výpadku spojení se serverem přejdou klienti do režimu synchronizace peer-to-peer. Při následné detekci obnovení spojení se serverem přejdou klienti zpět do režimu komunikace klient-server a pokusí se na server nahrát všechny entity, které vznikly v režimu peer-to-peer.

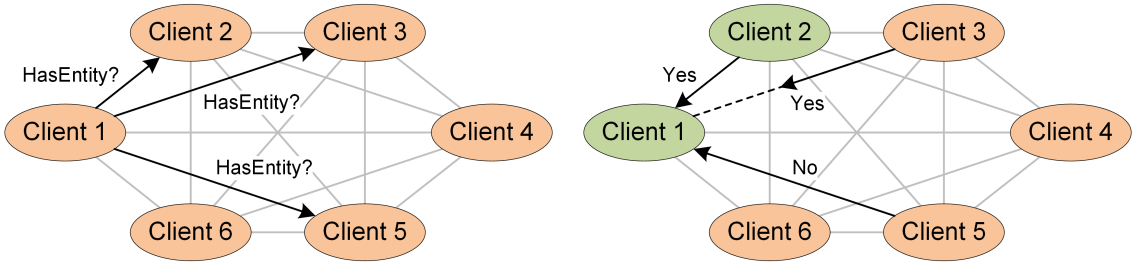
Detekce ztráty spojení se serverem bude probíhat jednoduše tím, že libovolný požadavek na server selže z důvodu vypršení časového limitu (timeout) nebo kvůli chybě v komunikaci.

Detekce obnovení spojení se serverem bude probíhat periodickým dotazováním serveru, jestli už je aktivní (spojení je obnoveno, pokud tento dotaz neselže).

3.3.7 Volba klientů pro komunikaci peer-to-peer

Každý klient si bude udržovat seznam klientů připojených do stejného Client poolu. Tento seznam klient získá při hledání klientů (viz. podkapitola 3.2 Hledání klientů).

Při získávání seznamu nových entit bude použit seřazený seznam aktivních klientů, ze kterého se budou postupně brát jednotliví klienti. Po dokončení iterace tohoto seznamu bude vytvořen nový seznam – budou dotazování známí klienti, jestli jsou aktivní a pokud ano, tak budou přidány do tohoto nového seznamu. Tento seznam bude následně náhodně promíchán. Těmito kroky bude zaručeno, že budou postupně dotazováni všichni aktivní klienti a že jejich pořadí bude alespoň trochu náhodné.



(a) Dotaz který klient má entitu

(b) Různě rychlá odpověď

Obrázek 3.6: Stahování entit v režimu komunikace peer-to-peer

Při stahování bude použit seznam aktivních klientů. Před samotným zahájením stahování se vyberou náhodní klienti ze seznamu a těmto klientům bude najednou odeslán dotaz (obrázek 3.6a), jestli se u nich daná entita nachází[1]. Stahování bude zahájeno od toho klienta, který nejrychleji odpoví pozitivně (obrázek 3.6b). V případě samých negativních odpovědí se zkusí požadavek zopakovat s dalšími klienty. Pokud se entita nebude nacházet u žádného aktivního klienta, tak se celá operace stahování entity zopakuje s časovým odstupem znovu.

3.3.8 Řešení konfliktů

U konfliktů, které vzniknou v komunikaci klient-server, bude řešení nabídnuto uživateli. Možná řešení budou následující:

- Konfliktní verzi nastavit jako aktuální
- Konfliktní verzi zařadit na příslušné místo v historii (značka CreateDate bude uměle vytvořen tak, aby verze byla v historii správně umístěna)
- Zachovat obě verze (pro konfliktní verzi vytvořit novou entitu)

Konflikty, které vzniknou v režimu komunikace peer-to-peer, budou vyřešeny automaticky zachováním obou verzí.

3.4 Komunikační protokol

Vzhledem k analýze byl tedy pro komunikaci zvolen framework WCF z .NET Frameworku. Pro využití výhod zmíněných v analýze budou komunikační rozhraní vystavena přes basicHttpBinding. Konkrétně to tedy znamená, že jako komunikační protokol bude použit SOAP a pro aktuálně vystavené rozhraní bude vygenerován WSDL soubor popisující toto rozhraní.

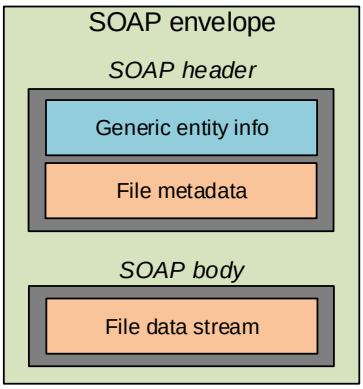
Pro komunikaci bylo také zvažováno použití architektury REST, která je frameworkem WCF také dobře podporována[42]. Tato architektura ale použita nebude, protože pro potřeby navrhovaného systému bude vhodnější použití protokolu SOAP, který lépe odpovídá požadavkům na komunikační rozhraní. Z architektonického stylu REST budou použity jenom některé principy, např. bezestavovost nebo model komunikace klient-server.

Díky možnosti použití self-hostingu v klientovi, který u vystavené služby umí také použít basicHttpBinding, bude komunikace mezi dvěma klienty velice podobná komunikaci mezi klientem a serverem. Díky této podobnosti bude přepínání mezi režimy synchronizace klient-server a peer-to-peer jednodušší.

SOAP protokol používá pro výměnu zpráv formát XML a nejčastěji se provozuje přes transportní protokol HTTP[43]. Díky tomu je snadné komunikovat mezi různými platformami. Díky podpoře ze strany WCF se tento způsob komunikace používá velice snadno včetně možnosti posílat stream dat. Nevýhodou může být zapouzdření zpráv do XML formátu, kvůli kterému roste celková velikost zpráv (např. kvůli párování tagů, ale hlavně kvůli kódování streamu pomocí Base64).

3.4.1 Komunikační rozhraní

Pro přenos dat bude framework používat generický interface, kterému stačí v konfiguračním souboru naspecifikovat typ přenášené entity a podle toho se vygeneruje konkrétní rozhraní pracující s touto konkrétní entitou. V ukázkové implementaci se bude jednat o stream dat souboru včetně jeho metadat. Pro fungování synchronizace si bude muset framework k těmto datům přibalit další informace, mezi kterými budou identifikátory entity a identifikátory její verze (obrázek 3.7).



Obrázek 3.7: Struktura SOAP zprávy s entitou obsahující soubor

Z tohoto důvodu bude existovat abstraktní třída specifikující kontrakt pro generický interface obsahující všechny potřebné informace pro framework. Konkrétní implementace tohoto kontraktu bude obsahovat data konkrétní entity (metadata souboru a stream dat). K tomuto účelu se nejvíce hodí serializace pomocí MessageContract[44], která umožňuje kompletní kontrolu nad SOAP zprávou pomocí atributů. Díky tomu je možné posílat stream dat v těle SOAP zprávy pomocí atributu MessageBodyMember a doplňující data potom mohou být serializována do hlavičky této SOAP zprávy pomocí atributu MessageHeader.

3.5 Architektura systému

Popis architektury je z důvodu přehlednosti rozdělen na serverovou a klientskou stranu, která je dále rozdělena na front-end a back-end.

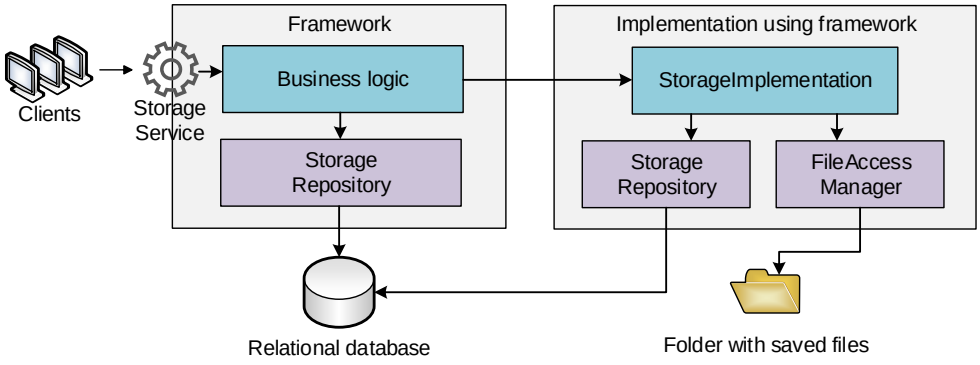
3.5.1 Architektura serveru

Architektura serveru je znázorněna na obrázku 3.8. Server bude vystavovat rozhraní StorageService pro připojování klientů. Přes toto rozhraní budou k dispozici veškeré dostupné serverové operace. Všechny požadavky budou předávány frameworku ke zpracování, který k tomu bude používat přístup k databázi přes StorageRepository nebo přístup ke konkrétním entitám přes interface IStorageImplementation. Implementaci tohoto interface bude muset dodat programátor používající tento framework.

Ukázková implementace používající framework pro synchronizaci souborů bude obsahovat business logiku převážně řídící ukládání nebo čtení dat. StorageRepository zprostředkuje přístup k databázi pro ukládání metadat a FileAccessManager přístup k ukládání samotných dat souborů.

Vstupní třída StorageService bude jednotlivé požadavky předávat zodpovědným třídám. Konkrétně se bude jednat o třídy obsluhující následující funkcionalitu:

- Ukládání a čtení entit.
- Řešení vzniklých konfliktů.



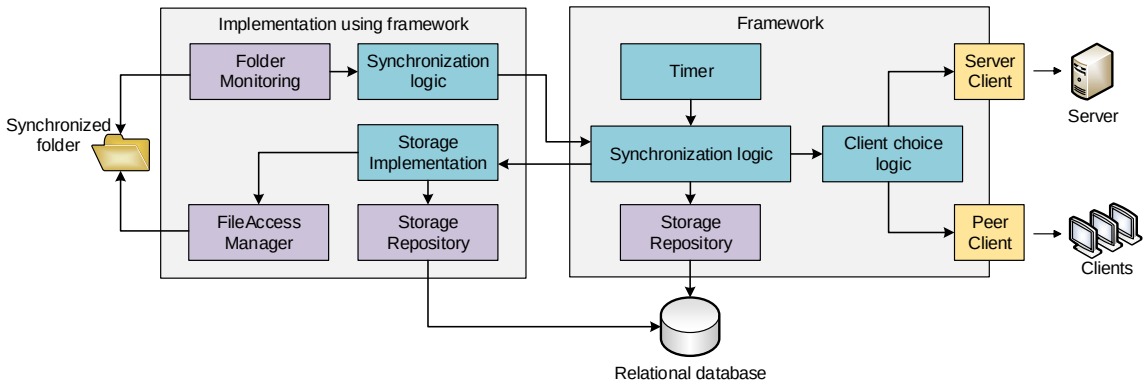
Obrázek 3.8: Diagram architektury serveru

- Správa Client poolů včetně přiřazování a odebírání entit z vybraných poolů.
- Správa informací o klientských zařízeních (hlavně pro potřeby vyhledávání klientů pro režim komunikace P2P).

3.5.2 Architektura klienta – back-end

Architektura back-endu klientské strany se bude skládat ze dvou částí. Hlavní část bude vystupovat v roli klient a pomocí komunikace se serverem nebo s jiným klientem bude jejím účelem dostat se do synchronního stavu. Účelem druhé části bude obsluhovat požadavky od jiných klientů. Z důvodu přehlednosti jsou obě části architektury popsány zvlášť.

Architektura části sloužící pro vyřizování požadavků od jiných klientů bude stejná jako v případě serveru (na obrázku 3.8). Výsledný rozdíl bude jenom v dostupnosti jiných operací na komunikačním rozhraní a tedy i v použití jiných tříd, které budou mít trochu jinou logiku.



Obrázek 3.9: Diagram architektury klienta

Architektura části vystupující roli klient je zobrazena na obrázku 3.9. Synchronizační operace budou spouštěny dvěma komponentami. Jedna z komponent se bude nacházet v implementaci používající framework a konkrétně v tomto případě to bude komponenta monitorující změny obsahu složky na disku. Druhá komponenta spouštějící synchronizační logiku

bude časovač, který bude opakovaně ve zvoleném intervalu stahovat nové entity a provádět další potřebné operace.

Stejně jako v případě serveru bude framework ukládat data do databáze přes StorageRepository. Manipulace s konkrétními entitami bude probíhat přes interface IStorageReaderImplementation nebo IStorageWriterImplementation. Implementace těchto interface musí být opět dodány programátorem používajícím tento framework.

Synchronizační logika bude mít na starosti vše, co se nějakým způsobem týká synchronizace entit. Hlavní funkcionalita bude následující:

- Lokálně si aktualizovat entity, aby byly synchronní s ostatními zařízeními.
- Nahrávat nové entity a nové verze na server. Toto nahrávání bude nastávat vždy po vytvoření nové entity nebo verze. Při výpadku serveru budou tyto entity nahrané až při obnovení spojení se serverem.
- Detekovat a řešit konflikty.
- Přidávat a odebírat entity z Client poolů.
- Vytvářet Client pooly a připojovat klienta do Client poolů.

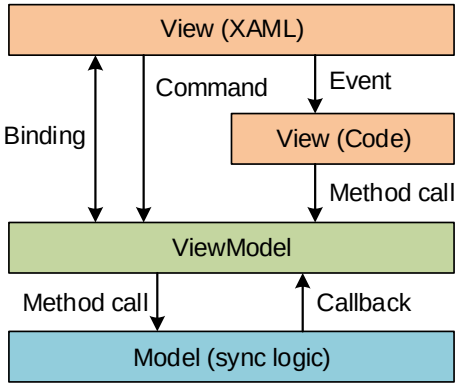
Součástí frameworku bude také logika volby klienta pro komunikaci (v tomto případě se klientem myslí třída zajišťující komunikaci). Účelem této části tedy bude přepínání mezi režimy synchronizace klient-server a peer-to-peer, takže bude muset umět detekovat selhání spojení se serverem i jeho následné obnovení. V režimu komunikace peer-to-peer bude tato část také zodpovědná za výběr různých klientů (neboli peerů) pro komunikaci.

3.5.3 Architektura klienta – front-end

Tento popis architektury se týká klientské aplikace s ukázkovou implementací synchronizace souborů. Vzhledem k tomu, že tato aplikace bude používat grafické uživatelské rozhraní vytvořené pomocí frameworku WPF, bude architektura této aplikace používat architektonický vzor MVVM (obrázek 3.10). Tento vzor rozdělí aplikaci na tři části: Model, View a ViewModel.

Jako Model bude sloužit veškerá naimplementovaná synchronizační logika. Tato část tedy bude obsahovat celý klientský synchronizační framework (takže veškerou logiku pro synchronizaci, řešení konfliktů, komunikaci v obou režimech apod.). Zároveň zde bude obsažena implementace používající tento framework (tedy manipulace se soubory a monitorování změn vybrané složky na souborovém systému). Architektura této části je detailně popsána v podkapitole 3.5.2 Architektura klienta – back-end.

View se v tomto případě bude definovat značkovacím jazykem XAML. Ke každému View bude příslušet jeden ViewModel. Vlastnosti komponent z View se následně budou bindovat do vybraných propert ve ViewModelu (dojde k provázání property a vlastnosti dané komponenty). ViewModel dále bude zprostředkovávat přístup k Modelu, neboli k ovládání synchronizační logiky.

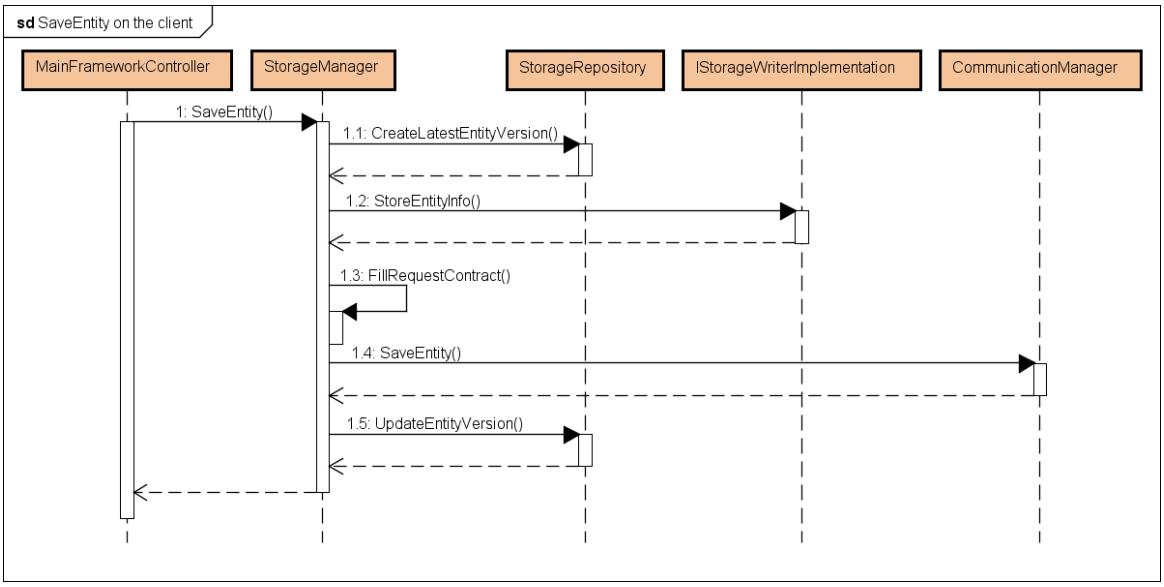


Obrázek 3.10: Diagram architektury MVVM. Zdroj: [45].

3.6 Popis základních operací synchronizace

V této části je popsána posloupnost kroků pro provedení základních synchronizačních operací na serverové i klientské straně.

3.6.1 Vytváření nové entity nebo verze na klientovi



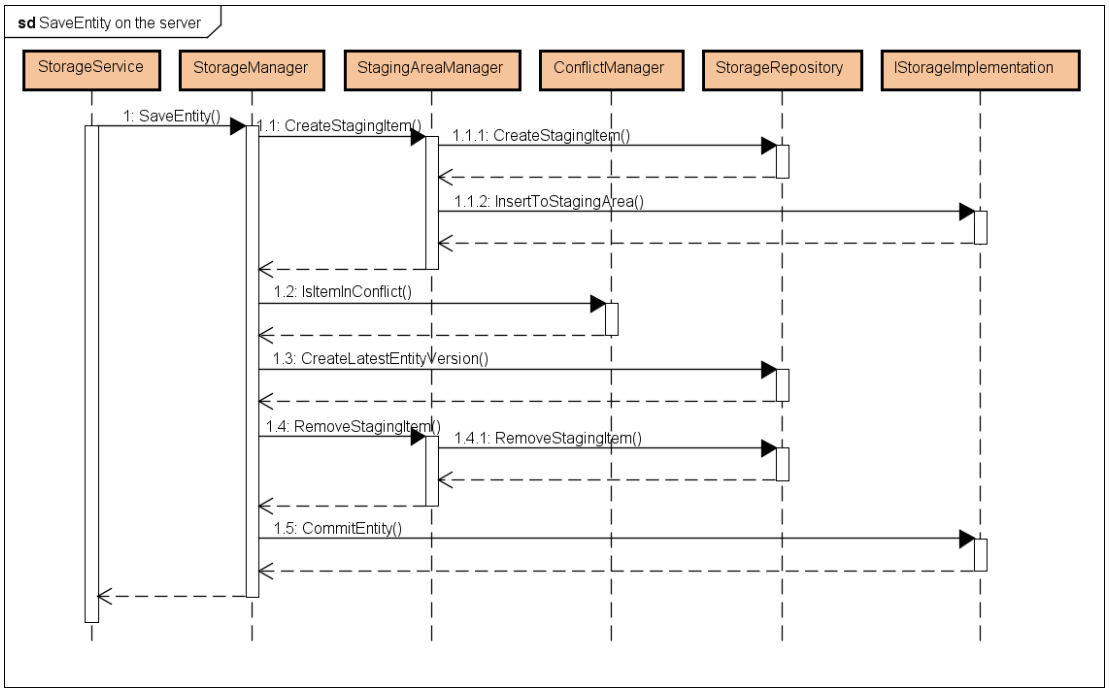
Obrázek 3.11: Sekvenční diagram vytváření nové verze entity na klientovi

Postup při vzniku nové entity nebo nové verze (znázorněno na obrázku 3.11) bude následující:

1. Vytvoření nového záznamu v tabulce EntityVersion a v případě nové entity i v tabulce Entity.

2. Uložení konkrétních informací o entitě (přes IStorageWriterImplementation). V případě ukázkové implementace se bude jednat jenom o uložení metadat do databáze.
3. Vyplnění doplňujících údajů k entitě potřebných pro odeslání entity na server.
4. Odeslání entity přes CommunicationManager, který zajistí spojení se serverem.
5. Při úspěšném odeslání aktualizování záznamu v EntityVersion (doplnění serverového ID) a případně i záznamu v tabulce Entity.

3.6.2 Vytváření nové entity nebo verze na serveru

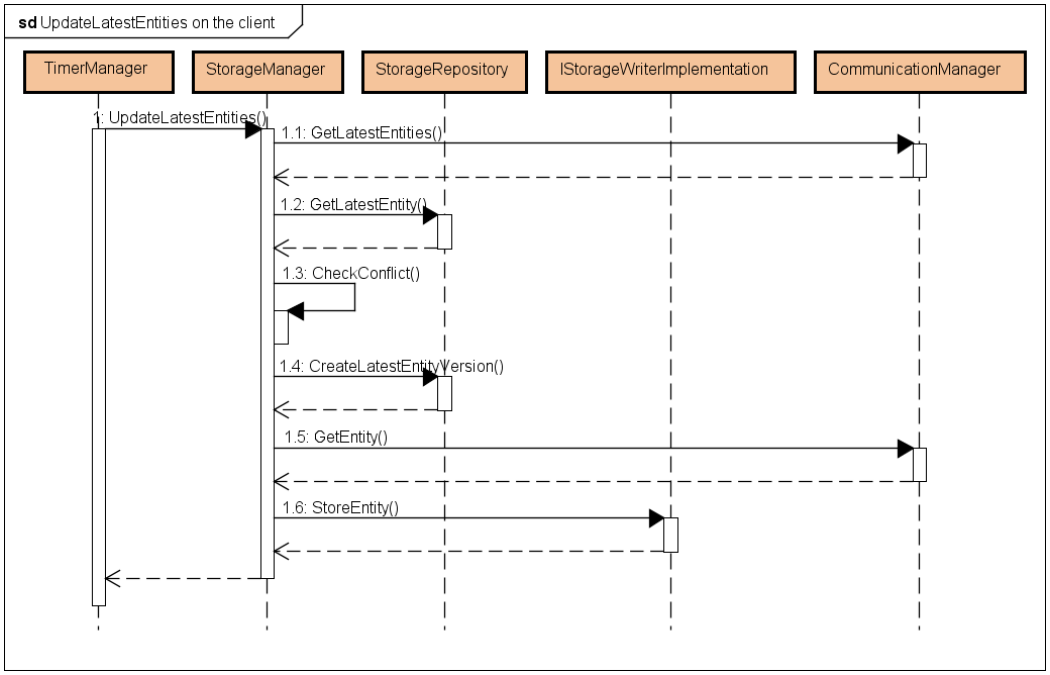


Obrázek 3.12: Sekvenční diagram vytváření nové verze entity na serveru

Postup při vkládání nových entit nebo nových verzí (znázorněno na obrázku 3.12) bude následující:

1. Vytvoření dočasné entity ve StagingArea.
   (a) Vytvoření záznamu v tabulce StagingArea.
   (b) Uložení konkrétní entity do dočasného úložiště.
2. Zkontrolování jestli entita není v konfliktu s jinou a pokud ano, vrátit příslušnou odpověď klientovi.
3. V případě úspěšného uložení entity vytvoření záznamu v tabulce EntityVersion.
4. Smazání záznamu z tabulky StagingArea.
5. Přesunutí entity do cílového úložiště (přes IStorageImplementation).

3.6.3 Získání nových entit na klientovi



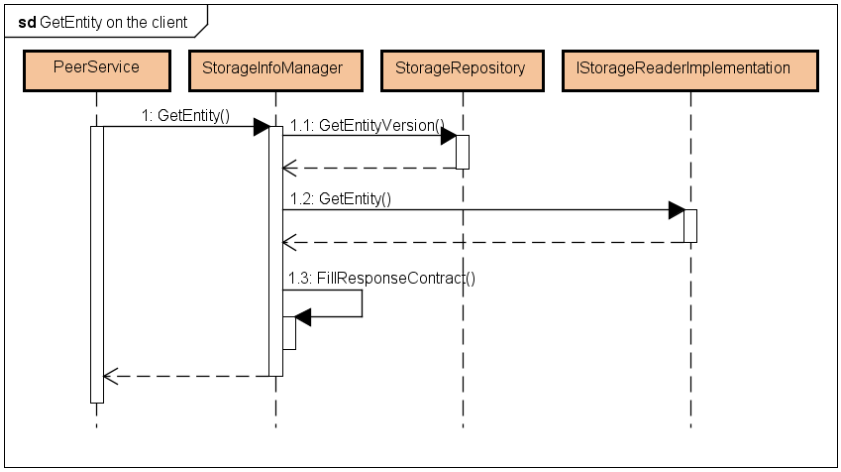
Obrázek 3.13: Sekvenční diagram získávání nových entit na klientovi

Postup pro získávání nových entit (znázorněno na obrázku 3.13) bude následující:

1. Získání seznamu nových entit přes CommunicationManager zajišťující spojení se serverem nebo s vybraným jiným klientem.
2. Získání seznamu lokálních entit.
3. Porovnání seznamu lokálních a nově příchozích entit pro rozhodnutí, které entity jsou novější a které entity se tedy mají stahovat. Zároveň bude kontrolováno, jestli entity nejsou v konfliktu.
4. Vytvoření záznamů v tabulce EntityVersion (pro příchozí entity, které jsou novější než lokální).
5. Stažení jednotlivých entit přes CommunicationManager, který opět zajistí dostupné spojení.
6. Uložení konkrétních entit přes IStorageWriterImplementation.

3.6.4 Získání entity na základě požadavku od klienta

Při vyřizování požadavku od nějakého klienta bude probíhat čtení entity na serveru i na klientovi stejným způsobem. Proto tu bude tento postup popsán pouze jednou. Postup pro čtení entit (znázorněno na obrázku 3.14) bude tedy následující:



Obrázek 3.14: Sekvenční diagram získání entity na základě požadavku od klienta

1. Vyhledání příslušného záznamu v tabulce EntityVersion.
2. Přečtení konkrétní entity (přes IStorageReaderImplementation).
3. Vyplnění doplňujících údajů k přečtené entitě pro potřeby frameworku.
4. Odeslání entity klientovi.

3.7 Databáze

Obě části systému (klient i server) budou potřebovat ukládat data do relační databáze. Obě strany budou pracovat se stejným typem entit, takže databázová schémata budou podobná. Rozdíl mezi těmito schématy bude spočívat hlavně v tom, že serverová část bude navíc obsahovat StagingArea pro nahrávání entit a klientská část bude navíc obsahovat DownloadArea pro doposud nestažené entity. Další rozdíl mezi schématy bude způsoben rozdílnými primárními klíči na serveru a na klientech.

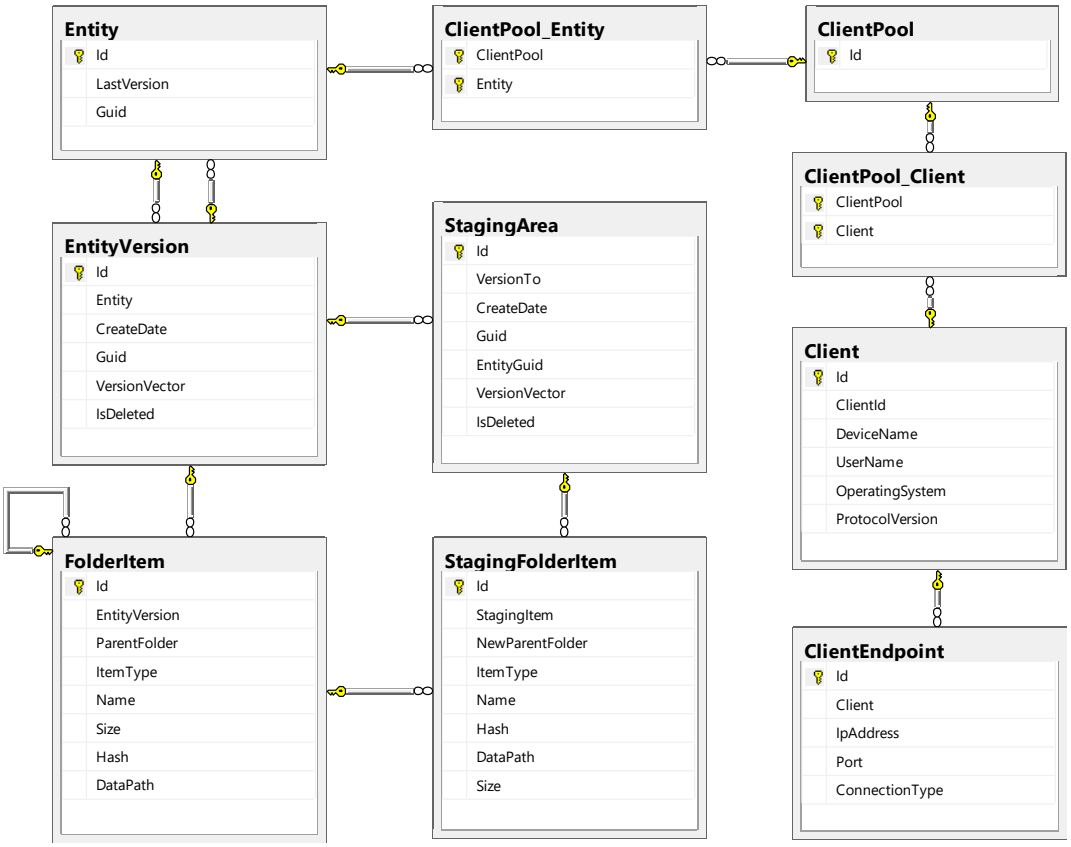
Serverová strana bude používat Microsoft SQL Server, který bude sloužit k ukládání informací o veškerých entitách a klientech. Klientská strana bude používat databázi SQLite, která bude ukládat jenom data potřebná pro daného klienta. Udržovat tuto lokální databázi bude důležité zejména pro správné fungování synchronizace (korektní verzování entit a řešení konfliktů) včetně fungování v komunikačním režimu peer-to-peer (lokální verzování, zprostředkování entit dalším klientům a nahrávání historie entit na server).

3.7.1 Serverová databáze

Schéma serverové databáze se nachází na obrázku 3.15. První, co bude potřeba uchovávat, jsou entity (hlavně jejich identifikace pomocí ID a případně GUID) a historie jejich verzí. Význam použitých identifikátorů je popsán v podkapitole 3.3.1 Identifikace entit. Entity budou uloženy v tabulce Entity a jednotlivé záznamy historie entit budou v tabulce

EntityVersion. Pro zjednodušení vytváření dotazů nad databází tabulka Entity bude obsahovat referenci na nejnovější (aktuální) verzi v tabulce EntityVersion.

Tabulka EntityVersion opět bude obsahovat sloupce sloužící k identifikaci (ID a případně GUID). Dále bude obsahovat čas vytvoření, který bude sloužit k rekonstrukci historie verzí zvolené entity. Pro označení, že je entita smazána, zde bude příznak IsDeleted. Poslední významný sloupec v této tabulce bude verzovací vektor sloužící klientům pro porovnávání verzí mezi sebou (která je novější, nebo že jsou v konfliktu).



Obrázek 3.15: Schéma serverové databáze

Pro skupiny entit bude v databázi existovat tabulka ClientPool. Vzhledem k tomu, že každý Client pool může obsahovat více entit a zároveň každá entita může patřit do několika Client poolů, tato tabulka bude propojena s tabulkou Entity vztahem M:N.

Pro potřeby vyhledávání klientů budou v databázi uloženy jejich detailní informace (tabulka Client). Každý klient musí být nějakým jedinečným způsobem identifikován, takže z toho důvodu si každý klient po prvním spuštění vygeneruje své GUID sloužící právě k identifikaci. Kromě tohoto sloupce bude tabulka obsahovat také doplňující informace jako je například název zařízení, nebo verze komunikačního protokolu.

Každý klient na sebe bude mít navázáno několik koncových bodů (tabulka ClientEndpoint), které jednak mohou reprezentovat více aktuálních síťových rozhraní např. WiFi a mobilní internet, nebo mohou zaznamenávat historii připojení např. internet doma a v práci.

Koncovým bodem bude myšlena IP adresa, port a typ připojení.

Tabulka Client bude navázána na tabulku ClientPool vztahem M:N, který umožní vyhledávání pouze těch klientů, kteří obsahují potřebné entity pro synchronizaci.

Nahrávání entity může zabrat nějaký čas, nebo po dokončeném nahrávání může být detekován konflikt entity. Z toho důvodu bude v databázi tabulka StagingArea k poznamenání si dočasných dat včetně informace, na kterou verzi se entita nahrává. Po úspěšném nahrání entity bude vytvořen záznam v tabulce EntityVersion a z tabulky StagingArea bude příslušný záznam smazán. Pokud bude detekován konflikt, záznam v tabulce StagingArea zůstane, dokud tento konflikt nebude vyřešen.

3.7.1.1 Tabulky pro konkrétní implementaci frameworku

Následující dvě tabulky již nebudou sloužit pro potřeby frameworku, ale pro konkrétní implementaci synchronizace souborů.

Tabulka FolderItem bude sloužit pro ukládání metadat složek a souborů a bude ve vztahu 1:1 k tabulce EntityVersion. Jelikož EntityVersion reprezentuje obecnou entitu, bude jednodušší uchovávat data pro složky i soubory v jedné tabulce a typ rozlišovat pomocí sloupce s hodnotou enum. Dále tato tabulka musí obsahovat referenci na rodičovskou složku (opět na tabulku FolderItem), což vyřeší stromovou strukturu složek i příslušnost souborů do konkrétní složky. Soubory budou ještě obsahovat hash a cestu k souboru, což může být například cesta na disku, nebo klíč v nějakém blobovém úložišti.

Tabulka StagingFolderItem bude mít obdobné vlastnosti jako FolderItem jenom s tím rozdílem, že bude sloužit pro potřeby tabulky StagingArea, se kterou bude ve vztahu 1:1 a zároveň reference na rodičovskou složku bude směřovat na tabulku FolderItem.

3.7.2 Klientská databáze

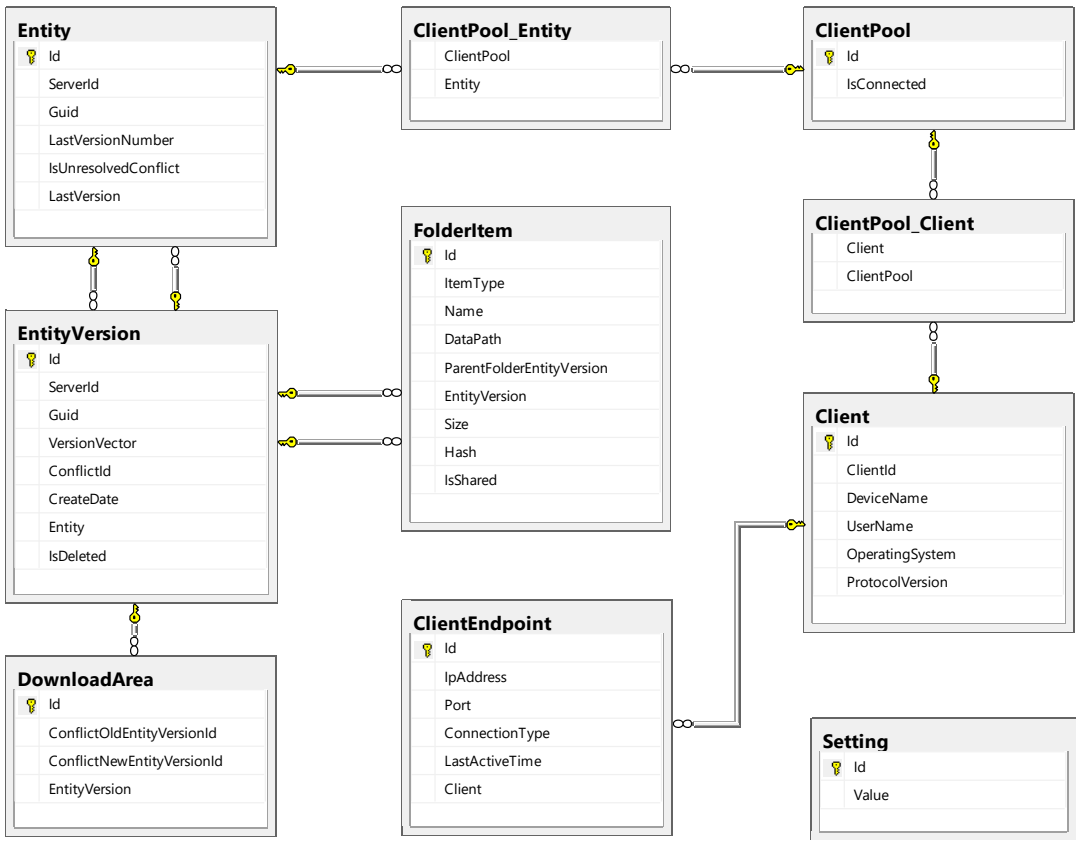
Databázové schéma na klientské straně (obrázek 3.16) bude velice podobné tomu na serverové straně. Jak už bylo zmíněno, rozdíl bude spočívat v některých přidaných nebo odebraných tabulkách. Tento rozdíl bude způsoben hlavně tím, že server bude čekat na požadavek od klienta, a tedy bude očekávat příjem nových entit (tyto entity budou nejdříve nahrány do StagingArea), kdežto klient naopak bude muset všechny operace vyvolávat sám.

Vzhledem k dvoufázovému stahování entit, kdy v první fázi bude jenom získán seznam nových entit, bude potřeba ukládat informaci o nestažených entitách do tabulky DownloadArea. Po úspěšném stažení bude potřeba příslušný záznam z této tabulky smazat.

Tabulka Entity bude oproti serverové databázi rozšířena o ServerId obsahující identifikátory vygenerované serverem, pomocný příznak označující entity s nevyřešeným konfliktem a LastVersionNumber obsahující poslední lokální číslo verze sloužící pro generování nových verzovacích vektorů.

Tabulka EntityVersion bude opět rozšířena o ServerId a dále potom o ConflictId obsahující číslo konfliktu na serveru (pokud tedy daná verze bude v konfliktu).

Kromě výše uvedených změn zde také přibude tabulka Setting sloužící k ukládání nastavení nebo nějakých doplňujících informací, např. GUID klienta.



Obrázek 3.16: Schéma klientské databáze

3.7.2.1 Tabulka pro konkrétní implementaci frameworku

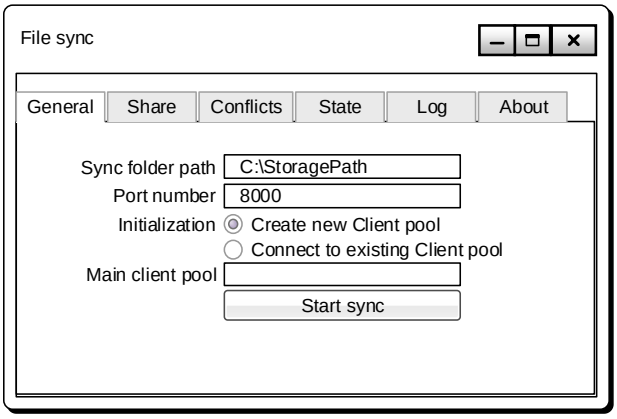
Vzhledem k tomu, že hodnoty primárních klíčů v této databázi nebudou mít kromě provázání záznamů žádnou vypovídající hodnotu (pro identifikaci bude sloužit pouze ServerId, nebo GUID), bude vytváření stromové struktury souborů složitější. Takže tabulka FolderItem pro rodičovský prvek nebude mít referenci sama na sebe, ale místo toho bude mít referenci na rodičovský EntityVersion, který už bude obsahovat platné identifikátory ServerId a GUID.

Druhá reference z FolderItem na EntityVersion má stejnou funkci jako na serverové straně, neboli prováže konkrétní entitu s obecnou reprezentací entity ve vztahu 1:1.

3.8 Uživatelské rozhraní

Hlavním účelem této práce je vytvoření frameworku pro synchronizaci entit a z tohoto důvodu bude vytvořeno jenom jednoduché uživatelské rozhraní pro demonstraci funkčnosti frameworku.

Aplikace se bude skládat ze dvou částí. Hlavní okno aplikace (obrázek 3.17) bude obsahovat záložky a jednotlivé záložky budou následující:



Obrázek 3.17: Návrh uživatelského rozhraní hlavního okna aplikace

- Inicializace aplikace – bude sloužit pro nastavení potřebná k zahájení synchronizace (výběr synchronizované složky nebo nastavení hlavního Client poolu).
- Sdílení – bude sloužit pro prezentaci funkčnosti týkající se Client poolů, neboli připojení klienta do Client poolu nebo přiřazení entity do Client poolu.
- Konflikty – bude sloužit pro vypsání všech nevyřešených konfliktů, které nastaly při komunikaci se serverem. Po kliknutí na konflikt se otevře dialogové okno s nabídkou možných řešení.
- Stav aplikace – bude zobrazovat informace o tom, co framework právě provádí za operace.
- Výpis logu – bude vypisovat všechny logy z aplikace a frameworku (hlavně pro debugovací účely)

Druhou částí aplikace bude notifikační ikona na hlavním panelu Windows v oznamovací oblasti. Účelem této ikony bude zobrazovat, v jakém režimu synchronizace se aplikace nachází (klient-server, peer-to-peer, nebo synchronizace nespuštěna). Po kliknutí na tuto ikonu se zobrazí seznam naposled synchronizovaných souborů.

Kapitola 4

Implementace

Tato kapitola se zabývá hlavně technickými záležitostmi. Popsán je zde tedy software potřebný pro vývoj a provozování systému, struktura celého projektu (ve Visual Studiu nazývána jako Solution) a způsob oddělení části reprezentující synchronizační framework od části obsahující implementaci přístupu ke konkrétnímu typu entit.

4.1 Software pro vývoj

Vzhledem k volbě platformy .NET Framework 4.5 a programovacího jazyka C# vyplynula volba dalšího software. Vzhledem k tomu, že vývoj serverové i klientské strany probíhal na jednom počítači, bylo potřeba nainstalovat i software pro provozování serverové strany.

V první řadě bylo potřeba zvolit vývojové prostředí a to konkrétně Microsoft Visual Studio Enterprise 2015. Součástí tohoto prostředí je také remote debugger, který se hodil v případě, že nastávala chyba na nějakém zařízení bez nainstalovaného Visual Studia.

Jako operační systém pro vývoj byl zvolen Microsoft Windows 10 Pro x64. Databázový systém pro serverovou stranu systému byl zvolen Microsoft SQL Server 2014 Developer Edition.

Všechny výše zmíněné programy jsou studentům dostupné z programu DreamSpark Premium.

Pro nasazování a testování serverové strany bylo vzhledem k použitému software a vzhledem k použití frameworku WCF potřeba použít webový server IIS ve verzi 10. Tento server lze do zvolené edice Windows bezproblémově doinstalovat.

4.2 Software pro provozování systému

Klientská i serverová strana je spustitelná na operačním systému Microsoft Windows s nainstalovaným prostředím .NET Framework 4.5. Serverová strana systému ještě vyžaduje nainstalovaný webový server IIS minimálně ve verzi 7. Minimální podporované verze Windows jsou tedy následující:

- Microsoft Windows Vista SP2
- Microsoft Windows Server 2008 SP2

4.3 Struktura projektu

Celý systém se dělí celkem do tří částí. Jedná se o implementaci serveru, implementaci klienta a sdílenou část. Serverová i klientská strana používají knihovny popsané v podkapitole 2.7 Technická řešení. Sdílenými projekty frameworku jsou:

- `InterClientCommunication.DataContracts` – obsahuje datové typy používané ve vystaveném komunikačním rozhraní.
- `InterClientCommunication.Shared` – obsahuje nezávislé sdílené třídy (např. pro verzovací vektory).
- `InterClientCommunication.UnitTest` – slouží pro testování nezávislých tříd (např. z projektu Shared).

Sdíleným projektem pro synchronizaci souborů je:

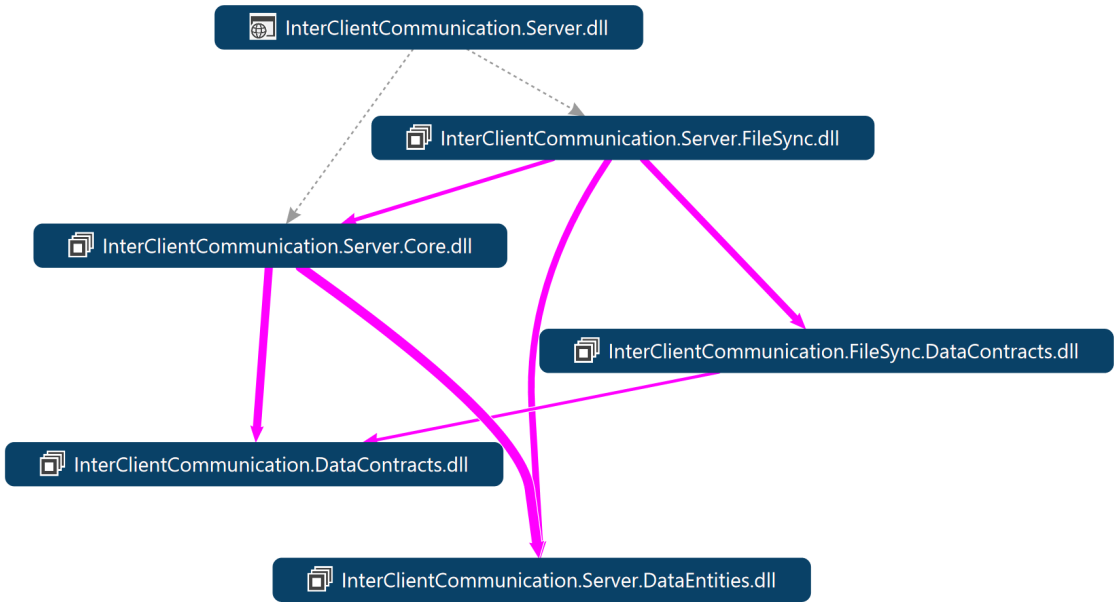
- `InterClientCommunication.FileSync.DataContracts` – obsahuje datové typy používané ve vystaveném komunikačním rozhraní sloužící pro potřeby synchronizace souborů.

Server se skládá z frameworkové části a části pro synchronizaci souborů (obrázek 4.1). Server je tvořen z následujících projektů:

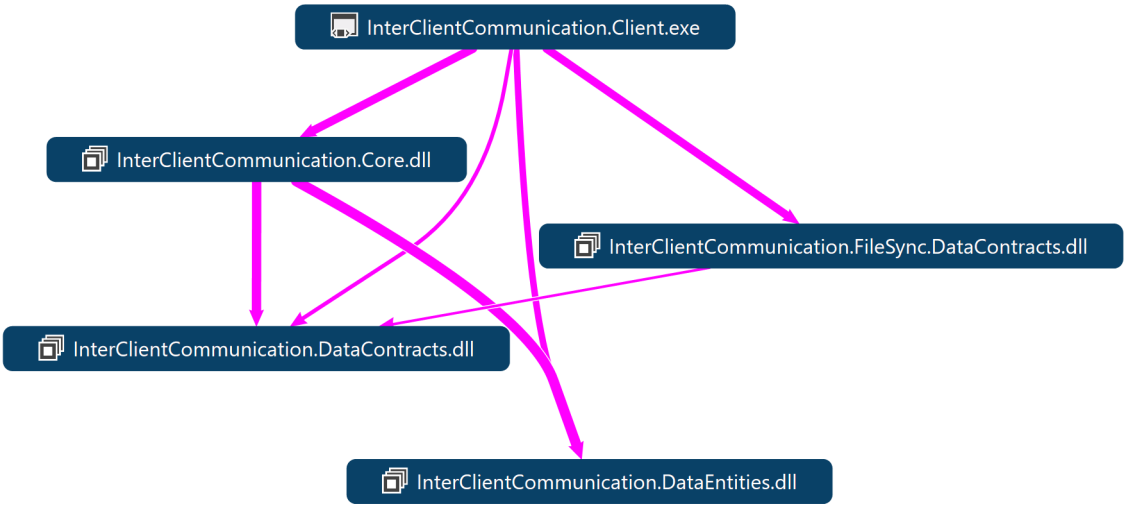
- `InterClientCommunication.Server` – slouží jenom k vystavení komunikačního rozhraní a konfiguraci serveru. Řízení je následně předáno do projektu Server.Core.
- `InterClientCommunication.Server.Core` – obsahuje veškerou synchronizační logiku frameworku.
- `InterClientCommunication.Server.DataEntities` – zprostředkovává přístup k databázi.
- `InterClientCommunication.Server.FileSync` – obsahuje funkcionalitu pro manipulaci s konkrétními entitami (čtení a ukládání souborů). Funkce jsou volány z projektu Server.Core.

Klient se opět skládá z frameworkové části a části pro synchronizaci souborů (obrázek 4.2). Klient je tvořen z následujících projektů:

- `InterClientCommunication.Client` – obsahuje uživatelské rozhraní aplikace, spouští synchronizační framework a zároveň mu zprostředkovává přístup ke konkrétním entitám (čtení a ukládání souborů).
- `InterClientCommunication.Core` – obsahuje synchronizační logiku frameworku včetně vyhledávání klientů a přepínání mezi režimy synchronizace klient-server a peer-to-peer. Zároveň tento projekt obsahuje podporu pro vystavení komunikačního rozhraní pro režim peer-to-peer.
- `InterClientCommunication.DataEntities` – zprostředkovává přístup k databázi.



Obrázek 4.1: Diagram komponent serverové části



Obrázek 4.2: Diagram komponent klientské části

4.4 Oddělení frameworku od synchronizace konkrétních entit

Framework je navržen tak, že obsahuje několik definovaných interface. Použití frameworku spočívá ve vytvoření tříd implementujících tyto interface a jejich následného zaregistrování do IoC Containeru. Tyto třídy (komponenty) musí být v Conatineru registrovány jako služby pod příslušným interface. Instance těchto tříd je potom vytvářena IoC Containerem a synchronizační framework jenom volá příslušné metody podle potřeby.

4.4.1 Serverová strana

Jak už bylo zmíněno, serverová část obsahuje projekt ICC.Server, který slouží pouze k vystavení komunikačního rozhraní. O vystavení této služby se nemůže postarat přímo framework, protože pro vystavení je nutné znát konkrétní typ entity přenášené během komunikace. Jakmile je znám tento typ, lze vytvořit komunikační rozhraní specifikované generickým interface IStorageService<T> a implementované generickou třídou StorageService<T> v projektu ICC.Server.Core. Z toho vyplývá, že StorageService zpřístupňuje všechny funkce serverové strany frameworku.

Volání metod konkrétní implementace probíhá pouze v případě řešení věcí týkajících se konkrétních entit (ukládání a čtení souborů). Toto volání probíhá přes interface IStorageImplementation. Pro správné fungování synchronizace je tedy nutné zaregistrovat třídu implementující tento interface do IoC Containeru serveru.

4.4.2 Klientská strana

Přístup k funkcím frameworku na klientské straně je zprostředkován přes třídu MainFrameworkController v projektu ICC.Core. V této třídě lze provádět konfiguraci, inicializaci a vkládání nových entit nebo verzí do synchronizace.

Volání metod pro řešení věcí týkajících se konkrétních entit (souborů) je prováděno přes interface IStorageReaderImplementation (pro čtení entit) a interface IStorageWriterImplementation (pro zápis entit). Pro správné fungování synchronizace je potřeba zaregistrovat třídy implementující tato rozhraní do IoC Containeru klienta.

Volání metod přes tyto interface je prováděno jednak pro dokončení lokálních synchronizačních operací (např. uložení nově stažené entity), ale také při zpracovávání požadavku od některého jiného klienta v režimu komunikace peer-to-peer.

4.4.3 Další definovaná rozhraní

Kromě již uvedených interface framework vyžaduje naimplementovat a registrovat interface IDatabaseSetup (na serverové i klientské straně) sloužící ke konfiguraci NHibernate pro zprovoznění přístupu k databázi. Tento interface také umožňuje nastavit mapování NHibernate pro některé další databázové entity a tím umožnit konkrétní implementaci synchronizace používat stejnou databázi jako používá framework. Výhodou je, že tímto lze použít již nakonfigurovaný NHibernate pro ukládání vlastních dat do databáze.

Kapitola 5

Testování

Vzhledem k povaze programu (synchronizační framework) bylo rozhodnuto hlavní testování provádět ukázkovým použitím (implementací) frameworku pro synchronizaci souborů. Tato synchronizace je pro testování vhodná, protože se jedná celkem o komplexní synchronizaci, kdy je potřeba počítat s náročným datovým přenosem, kdy entit může být velké množství a zároveň jejich velikost se může pohybovat od jednotek kilobajtů po stovky megabajtů. Dále zde může velice snadno vznikat mnoho různých konfliktů.

Tím, že framework vznikal zároveň s implementací synchronizace souborů, bylo také potřeba otestovat nasazení frameworku pro synchronizaci entit nějakého jiného typu. Z tohoto důvodu byla také naimplementována zjednodušená synchronizace telefonních kontaktů. Účelem tohoto testu bylo zjistit, jestli se systém chová jako framework, a tudíž jestli je jednoduché ho vzít a nasadit pro synchronizaci nějakých úplně jiných entit.

5.1 Testování přes synchronizaci souborů

Jak už bylo zmíněno, tak framework vznikal zároveň s jeho implementací nad synchronizací souborů. Proto také vznikala testovací verze desktopového klienta, která umožňovala ruční ovládání jednotlivých kroků synchronizačního frameworku. Výsledná verze klienta má toto ovládání vyřazené a framework se stará o synchronizaci sám automaticky.

Tento výsledný klient nemá naimplementovanou kompletní synchronizaci souborů, protože pro otestování frameworku stačilo tuto implementaci zjednodušit (např. synchronizovat soubory pouze z jedné úrovně složky a všechny podsložky tedy ignorovat).

5.1.1 Testovací prostředí

Pro testování bylo potřeba vytvořit nějaké testovací prostředí. Vzhledem k povaze systému, kdy probíhá synchronizace mezi serverem a více klienty, bylo nutné do testování zapojit více zařízení. Konkrétně bylo potřeba otestovat synchronizaci ve dvou režimech:

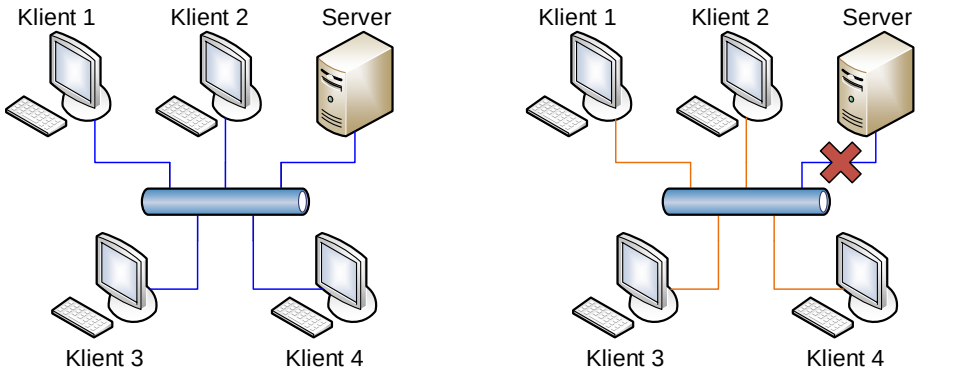
- Klient-server
- Peer-to-peer

Pro nejzákladnější otestování komunikace v režimu klient-server stačilo jedno zařízení s nasazeným serverem a spuštěným klientem. Pro testování synchronizace již bylo potřeba přidat alespoň jedno další zařízení, aby byla otestována propagace změn z jednoho klienta do druhého.

Testování synchronizace v režimu peer-to-peer vyžadovalo také alespoň dvě zařízení se spuštěnou klientskou aplikací.

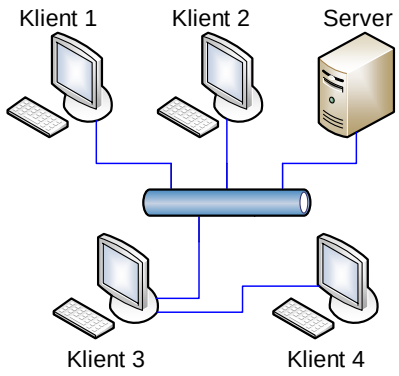
Pro testování kompletní funkcionality frameworku včetně automatického přepínání mezi oběma režimy komunikace byl tedy potřeba minimálně následující seznam zařízení:

- 1x server
- 2x klient



(a) Testování synchronizace klient-server (b) Testování synchronizace peer-to-peer

Obrázek 5.1: Konfigurace zapojení zařízení pro testování



Obrázek 5.2: Konfigurace zapojení zařízení pro testování obou režimů komunikace zároveň

Všechna tato zařízení bylo potřeba připojit do společné testovací sítě, kde na sebe zařízení vidí, a tedy spolu mohou komunikovat (obrázek 5.1a). V této síti bylo dále potřeba zajistit možnost odpojování serveru (obrázek 5.1b), čímž bylo možné testovat automatické přepínání mezi režimy komunikace klient-server a peer-to-peer při výpadku serveru.

Při vytváření testovací sítě byla vyzkoušena i varianta na obrázku 5.2, kde Klient 4 není připojen do sítě, ale je připojen ke Klientovi 3, se kterým tedy tvoří druhou síť, která s hlavní sítí není propojena. Vzhledem k této konfiguraci zařízení Klient 1, Klient 2 a Klient 3 pracují v režimu komunikace klient-server a zařízení Klient 4 v režimu peer-to-peer.

5.1.2 Testovací zařízení

Fyzicky k testování stačila pouze dvě fyzická zařízení, protože ostatní se dala zastoupit virtuálními počítači. Samotné testování tedy probíhalo na zařízeních s parametry uvedenými v tabulce 5.1.

Tabulka 5.1: Seznam zařízení pro testování

Zařízení	Procesor	Paměť	Operační systém
Stolní počítač	Intel Core i5-2400 3.1GHz	16 GB	Windows 10 Pro x64
Notebook	Intel Core i5-3427U 1.8GHz	4 GB	Windows 8.1 x64
Virtuální počítač	1 jádro	2 GB	Windows 7 Professional x64

5.1.3 Testovací vybavení

Pro monitorování činnosti aplikace a stavu databáze byly potřeba nástroje uvedené v tabulce 5.2.

Tabulka 5.2: Seznam nástrojů pro testování

Nástroj	Popis
log4net	Logovací nástroj pro vypisování všech důležitých událostí z programu.
DB Browser for SQLite	Nástroj pro vytváření, prohlížení a úpravu databázových souborů SQLite.
Microsoft SQL Server 2014 Management Studio	Nástroj pro správu Microsoft SQL Serveru obsahující také nástroje na prohlížení a úpravu databázových tabulek.
Oracle VM VirtualBox	Nástroj pro virtualizaci (provozování virtuálních počítačů).
WCF Test Client	Nástroj pro testování WCF služeb (vytváření a posílání požadavků na danou službu).

5.1.4 Testovací scénáře a výsledek testování

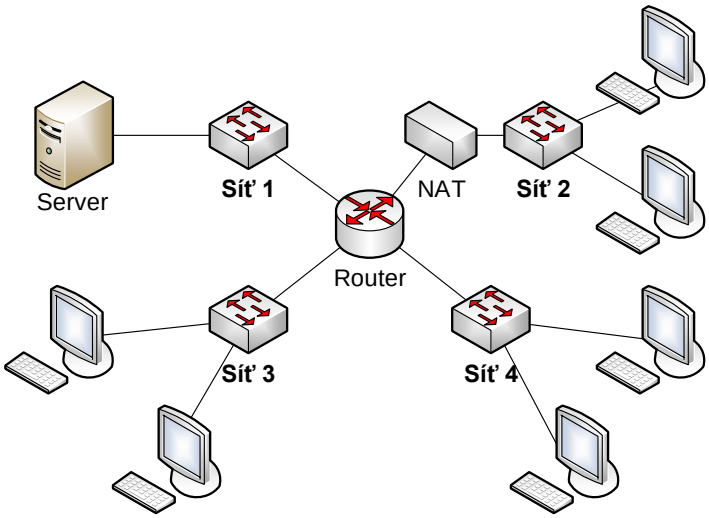
Když bylo testovací prostředí připraveno, bylo potřeba provést sérii testovacích úkonů, aby bylo ověřeno, že systém funguje spolehlivě a nevykazuje nějaké neočekávané chování. K tomuto účelu vznikl dokument se seznamem jednotlivých kroků, který se nachází v příloze D Seznam kroků pro testování.

V průběhu vývoje bylo tímto postupem objeveno hodně chyb, které byly následně opraveny. Při závěrečném testování pomocí tohoto seznamu nebyly objeveny žádné problémy.

Nevýhoda tohoto testování ale spočívala v tom, že bylo prováděno pouze jedním člověkem a tudíž systém nebyl pod příliš velkou zátěží a tedy nemusely být objeveny některé chyby týkající se paralelního provádění úkonů nad systémem.

5.2 Testování vyhledávání klientů

Otestovat vyhledávání klientů je celkem náročné, protože framework spouští všechny tři způsoby hledání zcela automaticky. Z toho důvodu bylo potřeba vytvořit takové testovací prostředí, které umožnilo testování jednotlivých mechanismů zvlášť. Toto prostředí tedy muselo být schopno vyřazovat jednotlivé vyhledávácí mechanismy.



Obrázek 5.3: Konfigurace síťového zapojení zařízení pro testování hledání klientů

Testovací prostředí vycházelo ze schématu na obrázku 5.3, kde pro testování jednotlivých mechanismů hledání klientů stačilo vždy použít jenom některé části. Takto postavená síť umožňovala navazovat komunikaci mezi všemi dostupnými zařízeními s výjimkou situace, kdy se libovolný klient ze sítí 1, 3 a 4 pokoušel navázat spojení se zařízením v síti 2, což bylo zablokováno zapnutou funkcí NAT.

Vyřazení jednotlivých vyhledávací mechanismů:

- Hledání pomocí serveru bylo vyřazeno jednoduchým odpojením serveru od zbytku sítě.
- Hledání pomocí multicastu bylo vyřazeno použitím routeru se zapnutou funkcí NAT.
- Hledání pomocí lokální databáze bylo vyřazeno zneplatněním příslušných záznamů v databázi. K tomuto zneplatnění došlo změnou IP adres klientských zařízení zapojených do sítě.

Testování hledání klientů pomocí multicastu vyžadovalo:

- Použít síť 3.
- Vyřadit hledání pomocí serveru a lokální databáze.

Testování hledání klientů pomocí serveru vyžadovalo:

- Použít síť 1, síť 2 a síť 3.
- Vyřadit hledání pomocí multicastu a lokální databáze.

Testování hledání klientů pomocí lokální databáze vyžadovalo:

- Použít síť 2 a síť 3.
- Vyřadit hledání pomocí serveru a multicastu.
- Mít v lokální databázi uložené klienty nalezené pomocí předchozích dvou metod.

5.2.1 Výsledek testu

Zásadním pozorováním tohoto testu bylo, že kombinace těchto metod dokáže najít spoustu zařízení včetně těch, se kterými není možné komunikovat. Na druhou stranu díky obsáhlému seznamu nalezených zařízení je možné komunikovat i s těmi, co se nachází za různými routery (obrázek 5.3, komunikace mezi sítí 3 a sítí 4), a tedy je možné komunikovat například i přes internet.

5.3 Testování ukázkové aplikace s uživateli

Vzhledem k tomu, že jeden člověk je pro otestování systému pod zátěží nedostatečný, bylo provedeno i neformální testování s více uživateli. Účelem tohoto testu bylo objevit problémy týkající se paralelního vykonávání akcí a bylo tedy nutné, aby se všichni participanti tohoto testu zúčastnili najednou. Účelem tohoto testu nebylo objevení chyb týkajících se uživatelského rozhraní.

Testovací prostředí pro tento test vypadalo tak, jak je popsáno v podkapitole 5.1.1 Testovací prostředí.

Přestože účelem testu bylo objevit hlavně problémy frameworku, tak participanti objevili hlavně problémy v ukázkové implementaci synchronizace souborů. Některé problémy se týkaly funkcionality, ale spousta se týkala uživatelského rozhraní. Testování proběhlo v několika iteracích, mezi kterými byla prováděna oprava objevených chyb.

5.3.1 Výsledek první iterace

První iterace probíhala ještě nad rozpracovaným systémem, který ještě používal pouze debugovací uživatelské rozhraní určené pro vývoj. Díky tomu se participanti zaměřili hlavně na testování funkcionality. Během tohoto testu bylo objeveno hodně problémů týkajících se hlavně přechodu do režimu komunikace peer-to-peer, který přestával fungovat.

Kvůli tomuto problému bylo rozhodnuto testování v této iteraci ukončit a věnovat se dokončení systému včetně vytvoření nového uživatelského rozhraní.

5.3.2 Výsledek druhé iterace

V této iteraci již byla vytvořena první verze uživatelského rozhraní a framework prošel testem popsaným v podkapitole 5.1.4 Testovací scénáře a výsledek testování.

Během tohoto testu byla objevena spousta závažných problémů. Nejzásadnější dva problémy se týkaly ukázkové implementace synchronizace souborů. Konkrétně se jednalo o nemožnost smazat soubor a selhání aplikace v případě pokusu o synchronizaci souboru většího než 1,5 GB (způsobeno nehlídáním maximální velikosti).

Problémem byla také konfigurace komunikace v režimu peer-to-peer nastavená na jednovláknové vyřizování požadavků. Klient tedy vždy obsluhoval maximálně jeden požadavek a ostatní klienti s ním v tu chvíli nebyli schopni navázat spojení. Tímto docházelo k výraznému zpomalení synchronizace.

Mnoho objevů se ale nakonec týkalo uživatelského rozhraní ukázkové aplikace. Účelem tohoto rozhraní ale bylo hlavně demonstrovat funkcionalitu frameworku, takže nebylo k testování zamýšleno. Přesto některé tyto poznatky byly následně zapracovány.

5.3.3 Výsledek třetí iterace

Před touto iterací testování přibyla podpora pro mazání a přejmenování souborů. Dále přibyla možnost jednoduchého sdílení souborů pro demonstraci fungujícího mechanismu Client poolů. Ve frameworku byla zvýšena stabilita (hlavně odolnost proti chybám v uživatelské implementaci synchronizace a proti selhání při přenosu dat).

Uživatelské rozhraní ukázkové aplikace bylo vylepšeno pro snadnější inicializaci programu a ovládání bylo zpřehledněno. Pro lepší zobrazení aktuálního stavu přibyla notifikační ikona na hlavním panelu Windows.

Výsledkem testování této iterace byla zvýšená spokojenost participantů s uživatelským rozhraním i s funkčností synchronizace. Přesto bylo objeveno pár drobných chyb (např. špatná aktualizace některých informačních prvků). Jelikož se jednalo pouze o drobné chyby, byly opraveny bez dalších iterací testování s uživateli.

5.4 Test nasazení frameworku

Kvůli nasazení frameworku pro synchronizaci libovolných entit vznikl podrobný návod, který se nachází v příloze B Návod na použití frameworku. V rámci tohoto testu tedy bylo postupováno podle tohoto návodu a účelem bylo jednak zjistit, jestli je tento návod kompletní a zároveň jestli je framework použitelný pro synchronizaci i jiných entit než souborů.

5.4.1 Výsledek testu

Při nasazování frameworku pro synchronizaci nových entit nebyl zjištěn žádný závažný problém. Chyby byly objeveny jenom v samotném návodu, kde chyběly některé kroky, které byly ihned doplněny.

Kapitola 6

Závěr

V průběhu řešení této práce byl navržen a implementován systém s architekturou klient-server umožňující synchronizaci obecných datových entit. Všechny entity synchronizované tímto systémem jsou verzovány a tedy je vytvářena jejich historie.

Architektura tohoto systému je navržena tak, aby oddělila obecnou synchronizační logiku od manipulace s konkrétním typem entit, např. souborů. Neboli framework se stará o veškerou synchronizaci včetně zajištění komunikace a použití tohoto frameworku spočívá v implementaci přístupu ke konkrétnímu typu entit včetně vytvoření uživatelského rozhraní klientské aplikace.

Framework na klientské části systému umí detekovat ztrátu i obnovení spojení se serverem a na základě aktuálního stavu se starat o přepínání komunikace mezi režimy klient-server a peer-to-peer. Implementace používající tento framework se o aktuální režim komunikace nemusí vůbec starat, protože framework zajistí veškeré potřebné kroky sám automaticky. Z hlediska použití frameworku tedy mezi oběma režimy komunikace není rozdíl.

Měnící se stav klientů při komunikaci P2P je podchycen několika mechanismy. Noví klienti jsou opakovaně vyhledávání. Nalezení klienti jsou před zahájením komunikace testováni, jestli jsou dostupní. Při přerušení probíhající komunikace s jiným klientem je daná operace zopakována s jiným klientem.

Pro demonstraci funkcionality vznikl server i klient synchronizující soubory pomocí tohoto frameworku. Hlavním cílem této práce ale bylo vytvořit synchronizační framework, takže funkcionalita tohoto ukázkového klienta byla zjednodušena.

Výsledná funkcionalita systému tedy splňuje všechny definované cíle a požadavky.

6.1 Složitost přepínání mezi režimy komunikace

V průběhu práce se zjistilo, že navržení a vytvoření systému, který by umožňoval plynulé přecházení mezi režimy synchronizace klient-server a peer-to-peer, není jednoduchou záležitostí a je okolo toho potřeba řešit spoustu problémů.

V případě komunikace jenom v režimu klient-server, nebo jenom v režimu peer-to-peer jsou mechanismy pro synchronizaci jednoznačné a podle nich celý systém funguje. V případě kombinování obou komunikačních režimů je ale potřeba používat oba tyto mechanismy,

z nichž každý může fungovat jinak. Příkladem je třeba rozdílný způsob řešení konfliktů, nebo rozhodování o tom, která verze entity je aktuální (v případě komunikace v režimu klient-server má vždy pravdu server, ale v případě režimu peer-to-peer si to musí každý klient rozhodovat sám pomocí verzovacích vektorů).

Kromě řešení problémů týkajících se rozdílnosti těchto synchronizačních mechanismů je také potřeba řešit problémy týkající se přechodů mezi oběma komunikačními režimy. Příkladem je třeba přechod z režimu komunikace peer-to-peer do režimu klient-server, kdy je potřeba nahrát veškerá data, která vznikla během synchronizace v režimu peer-to-peer, na server. O toto nahrávání se musí pokusit všichni klienti, protože každý může mít jinou sadu dat. Tím na serveru může vzniknout mnoho konfliktů, které je potřeba nějak vyřešit (třeba zásahem uživatele).

Pravděpodobně z výše uvedených důvodů všechna existující synchronizační řešení popsaná v této práci fungují jenom v jednom zvoleném komunikačním režimu.

6.2 Možnosti budoucího vylepšení systému

Během analýzy, ale i v průběhu návrhu a implementace vyplynula spousta možností vylepšení tohoto systému. Všechna tato zlepšení by už ale byla nad rámec rozsahu této práce.

6.2.1 Zabezpečení systému

Celý systém by se dal rozšířit o různé formy zabezpečení. Konkrétně by se dala zavést správa uživatelů a s tím související jejich autentizace a autorizace. S tím by dále souviselo šifrování komunikace.

Zavedení autentizace a autorizace v režimu komunikace klient-server by bylo celkem přímočaré. Horší situace by nastala v režimu komunikace peer-to-peer, kdy by bylo potřeba autentizaci a autorizaci řešit i mezi jednotlivými klienty.

6.2.2 Optimalizace komunikace

Přenos dat by mohl být optimalizován několika způsoby. Důležitým prvkem pro optimalizaci by bylo hlavně v případě velkých entit rozdělení přenosu entity na více částí (toto by mělo význam třeba u velkých souborů, které by se přenášely po blocích např. o velikosti 4 MB). Tímto by mohl být zrychlen přenos v režimu komunikace peer-to-peer, kdy by se jednotlivé části entit mohly stahovat od různých klientů.

Výraznou optimalizací by mohl být mechanismus, který by se v komunikačním režimu klient-server nejdříve pokoušel stahovat entity od ostatních klientů a teprve v případě neúspěchu by stahoval data entit přímo ze serveru. Tímto by mohlo být sníženo vytížení serveru a zvýšení rychlosti přenosu dat mezi klienty na lokální síti.

Další mechanismus pro optimalizaci by detekoval, jestli entita se stejnými daty již nebyla nahrána na server v minulosti, protože tím by mohla odpadnout nutnost posílat celou entitu. To samé by mohlo platit pro opačný směr, tzn. pokud by se entita se stejnými daty již na klientovi nacházela, tak by nebylo nutné ji stahovat.

Pro zrychlení přenosu by bylo dobré operace stahování i nahrávání entit paralelizovat. K tomuto účelu by bylo nutné vytvořit nějakou frontu pro tyto operace, protože počet probíhajících operací by musel být nějak omezen.

V režimu komunikace peer-to-peer by bylo dobré zavést chytřejší mechanismus pro volbu různých klientů pro komunikaci. Někteří klienti by mohli být preferování na základě různých metrik, jako je způsob a kvalita síťového připojení.

6.2.3 Komunikace mezi klienty

V režimu komunikace peer-to-peer by bylo dobré vyřešit způsob výměny dat i se zařízeními, která se nacházejí za routerem se zapnutým NAT. V tomto případě by pravděpodobně bylo potřeba vytvořit nějaký kanál s možností duplexní komunikace.

Klienti v režimu komunikace peer-to-peer by mohli začít využívat další způsob hledání klientů a to konkrétně znalost ostatních klientů, se kterými bylo navázáno spojení. Tímto by se v některých případech mohl zvýšit počet nalezených klientů.

6.2.4 Verzovací vektory

V této práci jsou použity verzovací vektory v základní formě, protože splňují vše, co od nich bylo požadováno v rámci specifikace funkčních požadavků. Nevýhoda těchto vektorů ale spočívá v jejich rostoucí velikosti s rostoucím počtem klientů zapojených do synchronizace.

Pro zefektivnění práce s těmito vektory a pro úsporu paměti by bylo dobré zavést nějaký mechanismus, který by velikost těchto vektorů redukoval.

6.2.5 Další vylepšení frameworku

Mezi další způsoby vylepšení frameworku patří rozšíření funkcionality o správu historie verzí ať už na serverové nebo na klientské straně.

Další důležitou funkcionalitou by bylo zavedení mechanismu na pročištění lokální databáze od starých nebo neaktuálních záznamů.

6.2.6 Rozšíření funkcionality ukázkové synchronizace souborů

Funkcionalita ukázkového použití frameworku pro synchronizaci souborů by se dala rozšířit o následující body:

- Synchronizovat i soubory, které se nacházejí v podsložkách.
- Zlepšit mechanismus pro detekci změn souborů (správně detekovat všechny operace a odfiltrovat vícenásobnou detekci stejné události, která je vyvolána systémovou komponentou pro monitorování změn na souborovém systému).
- Pracovat se zámky souborů (jakýkoliv jiný program může kdykoliv uzamknout přístup k souboru a tudíž se tento soubor nepodaří synchronizovat).

- Zpětně analyzovat synchronizovanou složku na disku po spuštění aplikace (jestli nedošlo k nějaké změně, dokud aplikace neběžela).
- Průběžně rozšiřovat funkcionalitu aplikace o veškerou novou funkcionalitu podporovanou ze strany frameworku.

Literatura

[1] DEE, Matt. *Inside LAN Sync* [online]. Dropbox Tech Blog, 2015. [cit. 17. 1. 2016]. Dostupné z: <https://blogs.dropbox.com/tech/2015/10/inside-lan-sync/>.

[2] *BitTorrent Sync - Features* [online]. Sync, 2015. [cit. 17. 1. 2016]. Dostupné z: <https://www.getsync.com/features>.

[3] Wikipedia contributors. *BitTorrent* [online]. Wikipedia, The Free Encyclopedia, 2016. [cit. 17. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/BitTorrent>.

[4] *Syncthing's documentation* [online]. Syncthing v0.12 documentation, 2016. [cit. 17. 1. 2016]. Dostupné z: <http://docs.syncthing.net/index.html>.

[5] *SpiderOak Releasing LAN Sync* [online]. Cloud Storage Buzz, 2012. [cit. 17. 1. 2016]. Dostupné z: <https://cloudstoragebuzz.com/spideroak/spideroak-releasing-lan-sync/>.

[6] *Release Notes for 2012* [online]. SpiderOak, 2013. [cit. 17. 1. 2016]. Dostupné z: <https://spideroak.com/articles/release-notes-for-2012>.

[7] *What's DirectSync? Sync without the cloud!* [online]. Cubby Help Center, 2016. [cit. 17. 1. 2016]. Dostupné z: <http://help.cubby.com/knowledgebase/articles/143893-what-s-directsync-sync-without-the-cloud>.

[8] *WCF Discovery* [online]. Microsoft Developer Network, 2016. [cit. 17. 1. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/dd456782(v=vs.110).aspx>.

[9] BEATTY, John, et al. *Web Services Dynamic Discovery (WS-Discovery)* [online]. 2005. [cit. 17. 1. 2016]. Dostupné z: <http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf>.

[10] *Chapter 4: Distributed and Parallel Computing* [online]. University of California, Berkeley, 2016. [cit. 17. 5. 2016]. Dostupné z: <http://wla.berkeley.edu/~cs61a/fa11/lectures/communication.html>.

[11] KURKOVSKY, Stan. *Distributed Systems Architectures* [online]. Department of Computer Science, Central Connecticut State University, 2016. [cit. 17. 5. 2016]. Dostupné z: <http://www.cs.ccsu.edu/~stan/classes/cs530/slides/se-12.pdf>.

[12] Wikipedia contributors. *Request–response* [online]. Wikipedia, The Free Encyclopedia, 2016. [cit. 4. 5. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Request-response>.

[13] Wikipedia contributors. *Publish–subscribe pattern* [online]. Wikipedia, The Free Encyclopedia, 2016. [cit. 4. 5. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Publish-subscribe_pattern>.

[14] *Hosting Services* [online]. Microsoft Developer Network, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/ms730158(v=vs.110).aspx>.

[15] LAMPORT, Leslie. *Generalized Consensus and Paxos* [online]. Microsoft Research, 2005. [cit. 22. 5. 2016]. Dostupné z: <http://research.microsoft.com/apps/pubs/default.aspx?id=64631>.

[16] KIM, YounSoo a Hoon CHOI. *Data Synchronization and Conflict Resolution for Mobile Devices* [online]. Dept. of Computer Engineering, Chungnam National University, 2016. [cit. 17. 5. 2016]. Dostupné z: <http://strauss.cnu.ac.kr/research/sync/paper/11_SyncML_ConflictResolution.pdf>.

[17] MCCORMACK, Drew. *Data Synchronization* [online]. objc.io, 2014. [cit. 17. 5. 2016]. Dostupné z: <https://www.objc.io/issues/10-syncing-data/data-synchronization/>.

[18] PARKER, Douglas, et al. Detection of mutual inconsistency in distributed systems. *Transactions on Software Engineering*. 1983.

[19] Wikipedia contributors. *Version vector* [online]. Wikipedia, The Free Encyclopedia, 2015. [cit. 19. 2. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Version_vector>.

[20] BAQUERO, Carlos. *Version Vectors are not Vector Clocks* [online]. HASlab, 2011. [cit. 19. 2. 2016]. Dostupné z: <https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/>.

[21] PREGUIÇA, Nuno, et al. *Dotted Version Vectors: Efficient Causality Tracking for Distributed Key-Value Stores* [online]. Department of Informatics of the Universidade do Minho, 2012. [cit. 19. 2. 2016]. Dostupné z: <http://gsd.di.uminho.pt/members/vff/dotted-version-vectors-2012.pdf>.

[22] *Overview of the .NET Framework* [online]. Microsoft Developer Network, 2016. [cit. 17. 1. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx>.

[23] *Pattern for Creating Generic WCF Services* [online]. CodeProject, 2011. [cit. 17. 1. 2016]. Dostupné z: <http://www.codeproject.com/Articles/290148/Pattern-for-Creating-Generic-WCF-Services>.

[24] GREEN, Robert. *Introduction to Windows Communication Foundation* [online]. Microsoft Developer Network, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/dd936243.aspx>.

[25] NAGEL, Christian, Jay GLYNN a Morgan SKINNER. *Professional C# 5.0 and .NET 4.5.1*. Wrox, 2014. ISBN 1118833031.

[26] TROELSEN, Andrew. *Pro C# 5.0 and the .NET 4.5 Framework*. Apress, 2012. ISBN 1430242337.

[27] *Introduction to WPF* [online]. Microsoft Developer Network, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/aa970268(v=vs.100).aspx>.

[28] *Inversion of Control* [online]. Castle Project, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://github.com/castleproject/Windsor/blob/master/docs/ioc.md>.

[29] Wikipedia contributors. *Dependency injection* [online]. Wikipedia, The Free Encyclopedia, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Dependency_injection>.

[30] *Castle Windsor Documentation* [online]. Castle Project, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://github.com/castleproject/Windsor/blob/master/docs/README.md>.

[31] *Apache License Version 2.0* [online]. Apache Software Foundation, 2016. [cit. 6. 5. 2016]. Dostupné z: <http://www.apache.org/licenses/LICENSE-2.0>.

[32] *SQLite Documentation* [online]. SQLite, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://www.sqlite.org/docs.html>.

[33] Wikipedia contributors. *Public domain* [online]. Wikipedia, The Free Encyclopedia, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Public_domain>.

[34] *NHibernate Documentation* [online]. NHibernate, 2016. [cit. 6. 5. 2016]. Dostupné z: <http://nhibernate.info/doc/index.html>.

[35] Free Software Foundation. *GNU Lesser General Public License v2.1* [online]. GNU Project, 2016. [cit. 4. 5. 2016]. Dostupné z: <http://www.gnu.org/licenses/lgpl-2.1-standalone.html>.

[36] *Apache log4net Features* [online]. Apache Software Foundation, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://logging.apache.org/log4net/release/features.html>.

[37] *The MVVM Pattern* [online]. Microsoft Developer Network, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/hh848246.aspx>.

[38] *MVVM Light Toolkit* [online]. MVVM Light Toolkit, 2016. [cit. 6. 5. 2016]. Dostupné z: <http://www.mvvmlight.net/>.

[39] *The MIT License (MIT)* [online]. Open Source Initiative, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://opensource.org/licenses/MIT>.

[40] *Data Synchronization* [online]. TrailDevilsSync Wiki, 2016. [cit. 17. 5. 2016]. Dostupné z: <https://github.com/swissmanu/TrailDevilsSync/wiki/Data-Synchronization>.

[41] Wikipedia contributors. *Globally unique identifier* [online]. Wikipedia, The Free Encyclopedia, 2016. [cit. 17. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Globally_unique_identifier>.

[42] FLANDERS, Jon. *RESTfull .NET*. O'Reilly Media, 2009. ISBN 978-0-596-51920-9.

[43] SKONNARD, Aaron. *Understanding SOAP* [online]. Microsoft Developer Network, 2003. [cit. 11. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/ms995800.aspx>.

[44] *Using Message Contracts* [online]. Microsoft Developer Network, 2016. [cit. 6. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/library/ms730255(v=vs.110).aspx>.

[45] BUGNION, Laurent. *MVVM - Messenger and View Services in MVVM* [online]. Microsoft Developer Network, 2013. [cit. 12. 5. 2016]. Dostupné z: <https://msdn.microsoft.com/en-us/magazine/jj694937.aspx>.

Příloha A

Seznam použitých zkratek

GUID Globally unique identifier

HTTP Hypertext Transfer Protocol

IIS Internet Information Services

IoC Inversion of control

IP Internet Protocol

LAN Local Area Network

MVVM Model-View-ViewModel

NAT Network Address Translation

ORM Object-relational mapping

P2P Peer-to-peer

REST Representational state transfer

SOAP Simple Object Access Protocol

SQL Structured Query Language

UDP User Datagram Protocol

URL Uniform Resource Locator

WCF Windows Communication Foundation

WPF Windows Presentation Foundation

WSDL Web Services Description Language

XAML Extensible Application Markup Language

XML Extensible Markup Language

Příloha B

Návod na použití frameworku

Serverová i klientská část využívá Castle Windsor, takže obě části mají k dispozici IoC Container, který mohou využívat. Pro správné fungování frameworku je u některých komponent dokonce vyžadováno, aby byly do tohoto Containeru zaregistrovány. Framework má v Containeru zaregistrovány knihovny NHibernate pro přístup k databázi a AutoMapper pro mapování entit. Obě tyto knihovny jsou plně k dispozici i pro použití mimo framework.

B.1 Zprovoznění serveru

1. Na databázovém serveru založte novou databázi.
2. Ve složce Database se nacházejí SQL skripty pro vytvoření databáze. Tyto skripty postupně všechny spusťte podle jejich číselného označení.
3. Ve Visual Studiu vytvořte nový projekt, který bude obsahovat datové kontrakty (třídy, reprezentující data posílaná po síti).
   - V tomto projektu přidejte referenci na *System.ServiceModel*.
   - V tomto projektu založte třídu, která bude dědit od *InterClientCommunication.DataContracts.Shared.EntityContractBase* (tato třída bude reprezentovat přenášené entity pro synchronizaci).
   - Této nově vytvořené třídě přidejte atribut MessageContract a každé property, která má být serializována, přidejte atribut MessageHeader pro umístění do hlavičky zprávy nebo MessageBodyMember pro umístění do těla zprávy.
4. Ve Visual Studiu vytvořte nový projekt typu WCF Service Application (tento projekt bude sloužit k vystavování komunikačního rozhraní).
   - V tomto projektu přidejte referenci na *InterClientCommunication.Server.Core* a na projekt obsahující datové kontrakty (vytvořený v předchozím bodě).
   - Do tohoto projektu nakopírujte konfigurační soubory Web.config a Server.Container.config.

- V konfiguračním souboru Web.config se celkově na třech místech nachází tento, nebo podobný řetězec:
  - *InterClientCommunication.Server.Core.StorageService'1[[InterClientCommunication.FileSync.DataContracts.FolderItemEntityContract,InterClientCommunication.FileSync.DataContracts,Version=1.0.0.0,Culture=neutral,PublicKeyToken=null]]*
- Důležité je přepsat to, co se nachází uvnitř dvojitých hranatých závorek, na cestu k datovému kontraktu, který dědí od EntityContractBase. Pozor, na prvním místě se nachází úplná cesta k dané třídě a na druhém místě je cesta k assembly.
- Konfigurační soubor Server.Container.config slouží pro nastavování Castle Windsor Containeru, takže v tuto chvíli může zůstat nezměněn.

5. Ve Visual Studiu vytvořte nový projekt obsahující třídy pro ukládání a čtení synchronizovaných entit a konfiguraci databáze.
   - V tomto projektu přidejte referenci na *InterClientCommunication.Server.Core*, *InterClientCommunication.Server.DataContracts*, *InterClientCommunication.Shared*, na projekt obsahující nové datové kontrakty a na knihovnu NHibernate.
   - Do předchozího projektu (vystavujícího komunikační rozhraní) přidejte referenci na tento projekt.
   - Založte novou třídu, která bude implementovat interface *InterClientCommunication.Server.Core.Installer.IDatabaseSetup* (tato třída bude sloužit pro konfiguraci přístupu k databázi a přidání nových assembly s entitami, se kterými má pracovat NHibernate).
   - Založte novou třídu, která bude implementovat interface *InterClientCommunication.Server.Core.Manager.IStorageImplementation* (tato třída bude zajišťovat čtení entit a zapisování entit do nějakého úložiště). Jako generický typ použijte již vytvořený kontrakt (dědící od EntityContractBase).
   - Tyto dvě nové třídy je nutné zaregistrovat jako službu daného interface do Containeru (toto lze provést v konfiguračním souboru Server.Container.config).
6. Naimplementujte metody definované všemi použitými interface.
7. Server se bude spouštět přes projekt vytvořený jako WCF Service Application.
   - Ve vlastnostech projektu pod záložkou Web je vhodné nastavit nasazování serveru do Local IIS.
   - Při výchozím nastavení výsledná služba poběží na adrese: *http://localhost/{název_projektu}/StorageService.svc*

B.2 Zprovoznění klienta

Klient používá SQLite databázi, která se vytvoří sama, takže není potřeba konfigurovat connection string apod. a také není potřeba spouštět SQL skripty pro vytvoření tabulek.

Přístup k funkcím frameworku probíhá přes třídu *InterClientCommunication.Core.MainFrameworkController*. Klienta je nutné spouštět pod administrátorským oprávněním (kvůli vystavení komunikačního rozhraní pro režim peer-to-peer).

1. Ve Visual Studiu by už měl z předchozích kroků existovat projekt obsahující datové kontrakty s třídou dědící od EntityContractBase (vznikl při vytváření serveru).
2. Ve Visual Studiu vytvořte nový projekt, který bude reprezentovat výsledného klienta (např. WPF Application).
   - V tomto projektu přidejte referenci na *InterClientCommunication.Core* a na projekt obsahující datové kontrakty (zmíněný v předchozím bodě).
   - Přes NuGet Package Manager přidejte package *System.Data.SQLite.Core*.
   - Do tohoto projektu nakopírujte konfigurační soubor App.config (podstatné je to, co se nachází uvnitř tagu *<system.serviceModel>*).
   - V konfiguračním souboru App.config se celkově na čtyřech místech nachází tento, nebo podobný řetězec:
     - *InterClientCommunication.DataContracts.IPeerService‘1[[InterClientCommunication.FileSync.DataContracts.FolderItemEntityContract,InterClientCommunication.FileSync.DataContracts,Version=1.0.0.0,Culture=neutral,PublicKeyToken=null]]*
   - Důležité je přepsat to, co se nachází uvnitř dvojitých hranatých závorek, na cestu k datovému kontraktu, který dědí od EntityContractBase. Pozor, na prvním místě se nachází úplná cesta k dané třídě a na druhém místě je cesta k assembly.
   - V App.config je také potřeba provést nastavení URL adresy serveru. Konkrétně se jedná o nastavení atributu address v tagu *<endpoint name=“StorageService“>*, který se nachází uvnitř tagu *<client>*.
   - Dále do projektu přidejte app.manifest (tento krok není nezbytně nutný). Změňte atribut level uvnitř tagu *<requestedExecutionLevel>* na requireAdministrator, aby se aplikace spouštěla s administrátorským oprávněním.
3. V tomto projektu, nebo v nějakém novém bude potřeba vytvořit třídy implementující konfiguraci databáze, konfiguraci Castle Windsor Containeru a ukládání i čtení entit.
   - Tomuto vybranému projektu přidejte referenci na *InterClientCommunication.Core*, *InterClientCommunication.DataContracts*, *InterClientCommunication.Shared*, na projekt obsahující datové kontrakty a na knihovny NHibernate a Castle Windsor.
   - Založte novou třídu, která bude implementovat interface *InterClientCommunication.Core.Installer.IContainerInstaller* (tato třída bude zprostředkovávat konfiguraci Castle Windsor Containeru).
   - Založte novou třídu, která bude implementovat interface *InterClientCommunication.Core.Installer.IDatabaseSetup* (tato třída bude sloužit pro přidávání nových assembly s entitami, se kterými má pracovat NHibernate).

- Založte novou třídu, která bude implementovat interface *InterClientCommunication.Core.Manager.IStorageReaderImplementation* (tato třída bude zajišťovat čtení entit z nějakého úložiště).
- Založte novou třídu, která bude implementovat interface *InterClientCommunication.Core.Manager.IStorageWriterImplementation* (tato třída bude zajišťovat zapisování entit do nějakého úložiště).
- Třídy implementující interface IDatabaseSetup, IStorageReaderImplementation a IStorageWriterImplementation je nutné zaregistrovat jako službu daného interface do Containeru (toto lze provést ve třídě implementující interface IContainerInstaller).

4. Naimplementujte metody definované všemi použitými interface.
5. Při spuštění aplikace je nutné inicializovat Container, který se nachází v *InterClientCommunication.Core*. Inicializace se provádí zavoláním statické metody Init, která přebírá jako parametr objekt implementující interface IContainerInstaller.
6. Pro spuštění synchronizace je potřeba framework inicializovat zavoláním metody Init na objektu třídy MainFrameworkController. Samotná synchronizace se spouští metodou StartSync. Instanci této třídy je potřeba získat pomocí Containeru.

B.3 Použití NHibernate a AutoMapper

Pro konfiguraci AutoMapperu stačí vytvořit třídy dědící od třídy AutoMapper.Profile a v těchto třídách přepsat metodu (override) Configure. V těchto metodách lze následně provést konfiguraci. Takto vytvořené třídy je nutné zaregistrovat do Containeru jako službu pod AutoMapper.Profile.

Pro použití NHibernate stačí vytvořit třídu, která dědí od *InterClientCommunication.DataEntities.Daos.NHibernateTransactionalDao* nebo od *InterClientCommunication.Server.DataEntities.Daos.NHibernateTransactionalDao*. Instanci příslušné třídy je potřeba nechat si vytvořit IoC Containerem.

Příloha C

Návod na spuštění systému pro synchronizaci souborů

Tento návod popisuje nasazení serveru a spuštění aplikace pro synchronizaci souborů.

C.1 Nasazení serveru

Balíček pro nasazení serveru se nachází ve složce Binary\Server.

Minimální prerekvizity:

- Microsoft Windows Server 2008 SP2
- Microsoft SQL Server 2012
- .NET Framework 4.5
  - Povolený WCF HTTP Activation
- Webový server IIS 7
- Web Deploy (lze nainstalovat z Web Platform Installer)

Vytvoření databáze a konfigurace connection stringu:

- Vytvořte databázi s názvem ClientCommunicationDB.
- Spusťte postupně všechny skripty na vytvoření databázových tabulek ze složky Database.
- Pro nastavení connection stringu pro přístup k databázi upravte v souboru *InterClientCommunication.Server.SetParameters.xml* hodnotu parametru "MyConnectionString-Web.config Connection String".

Vytvoření úložiště souborů:

- Vytvořte složku ICCServerStorage na disku C.

Stručný postup nasazení aplikace do serveru IIS:

- Spusťte příkazovou řádku s oprávněním Administrátor.
- Spusťte *InterClientCommunication.Server.deploy.cmd* s parametrem /T pro otestování jestli je vše připraveno k nasazení.
- Spusťte *InterClientCommunication.Server.deploy.cmd* s parametrem /Y pro nasazení na lokální počítač.

Podrobný návod nasazení aplikace do serveru IIS se nachází v souboru readme.

C.2 Spuštění klienta

Klientská aplikace se nachází ve složce Binary\Client.

Minimální prerekvizity:

- Microsoft Windows Vista SP2
- .NET Framework 4.5

Spuštění aplikace:

- Nastavte adresu serveru v konfiguračním souboru *InterClientCommunication.Client.exe.config* uvnitř tagu *<endpoint name="StorageService">* v atributu address.
- Povolte ve firewallu příchozí komunikaci na portu 8000 (pro fungování komunikace v režimu peer-to-peer).
- Spusťte aplikaci s administrátorským oprávněním (pro fungování komunikace v režimu peer-to-peer).

Příloha D

Seznam kroků pro testování

D.1 Inicializace aplikace

- První spuštění
  - S běžícím serverem
  - Bez běžícího serveru
- Spuštění synchronizace
  - S běžícím serverem
  - Bez běžícího serveru
- Prvotní inicializace aplikace
  - Vytvořit nový Client pool
  - Připojit se do existujícího Client poolu
- Prvotní inicializace aplikace
  - S výchozími hodnotami portu a cesty k synchronizované složce
  - S vlastními hodnotami
    - Zkontrolovat, jestli zůstanou zapamatovány i po restartu aplikace

D.2 Hledání klientů

- Pomocí serveru
  - Spustit klientské aplikace s funkčním serverem
- Pomocí lokální databáze
  1. Vypnout server
  2. Zablokovat posílání multicastu po síti

3. Spustit klientské aplikace

- Pomocí multicast

1. Vypnout server
2. Povolit posílání mulitcastu po síti
3. Změnit IP adresy klientům
4. Spustit klientské aplikace

D.3 Základní synchronizace

- Připojit více klientů do jednoho Client poolu
- Přidat, upravit, přejmenovat a smazat soubor na jednom klientovi
  - Sledovat projevení se změn na ostatních klientech
  - V režimu komunikace klient-server
  - V režimu komunikace peer-to-peer
    - Sledovat, jestli zařízení střídá klienty, se kterými komunikuje
- Vytvořit konflikty
  - V režimu komunikace klient-server (alespoň 3)
  - V režimu komunikace peer-to-peer
    - Zkontrolovat automatické vyřešení na všech klientech
- Vyřešit konflikty vytvořené v režimu klient-server
  - Použít všechny možné způsoby řešení:
    - Nastavit jako aktuální
    - Zařadit do historie
    - Vytvořit novou entitu (zachovat obě verze)
  - Zkontrolovat korektnost řešení
    - Na serveru
    - Na všech klientech
- Synchronizace velkých souborů (cca. 1 GB)
  - S běžícím serverem
  - Bez běžícího serveru

D.4 Přepínání mezi komunikačními režimy

- Automatické přepnutí do režimu peer-to-peer
  1. Odpojit server
  2. Vytvořit/upravit soubory
  3. Nechat ostatní zařízení synchronizovat si změny mezi sebou
- Automatické přepnutí do režimu klient-server
  1. Odpojit některé klienty od sítě (někteří klienti ale musejí zůstat zapojení)
  2. Připojit server
  3. Sledovat nahrání historie souborů na server
     - Server získá všechna data
     - Všichni klienti si zaktualizují databázi novými serverovým ID
  4. Připojit odpojené klienty
  5. Sledovat nahrání zbytku dat na server a aktualizaci serverových ID
- Ruční přepínání do režimu peer-to-peer
  - Zkontrolovat, jestli se chová stejně jako v případě automatického přepnutí
- Jedno zařízení v režimu komunikace peer-to-peer a ostatní v režimu klient-server
  - Sledovat synchronizaci směrem od zařízení v režimu klient-server do zařízení v režimu peer-to-peer

D.5 Funkcionalita Client poolů

- Dvě skupiny klientů, každá připojená do jiného Client poolu
  - Sledovat nezasahování do synchronizace druhé skupiny
  - S běžícím serverem
  - Bez běžícího serveru
- Vytvoření nového Client poolu
  - S běžícím serverem
  - Bez běžícího serveru – zkontrolovat nefunkčnost
- Připojení vybraného klienta do nového Client poolu
  - S běžícím serverem
  - Bez běžícího serveru – zkontrolovat nefunkčnost
- Přidání a odebrání entit z Client poolu
  - S běžícím serverem – sledování synchronizace příslušných entit ze správných Client poolů (sdílení entit)
  - Bez běžícího serveru – zkontrolovat nefunkčnost

Příloha E

Dokumentace komunikačního rozhraní

Komunikační rozhraní použité na klientovi i na serveru používá komunikační protokol SOAP. Následující podkapitoly obsahují pouze seznam metod. Podrobnější popis metod a jejich datových typů se nachází v dokumentaci kódu na přiloženém DVD.

E.1 Metody komunikačního rozhraní pro peer-to-peer

Tabulka E.1: Seznam metod komunikačního rozhraní pro peer-to-peer komunikaci

Name	Description
GetEntity	Get concrete entity data.
GetLatestEntities	Get latest entity list.
GetPeerInfo	Get detail peer information (ID, endpoints, etc.).
GetUtcDateTime	Get current UTC time from device.
HasEntity	Check if entity data is present.
Ping	Empty method for checking connection.

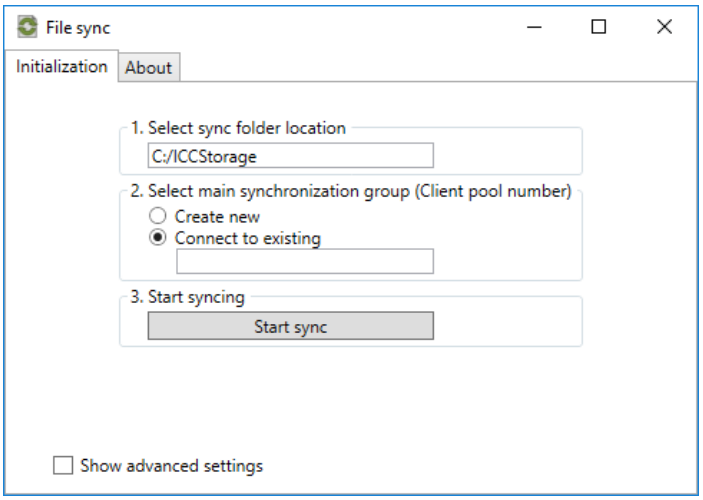
E.2 Metody serverového komunikačního rozhraní

Tabulka E.2: Seznam metod serverového komunikačního rozhraní

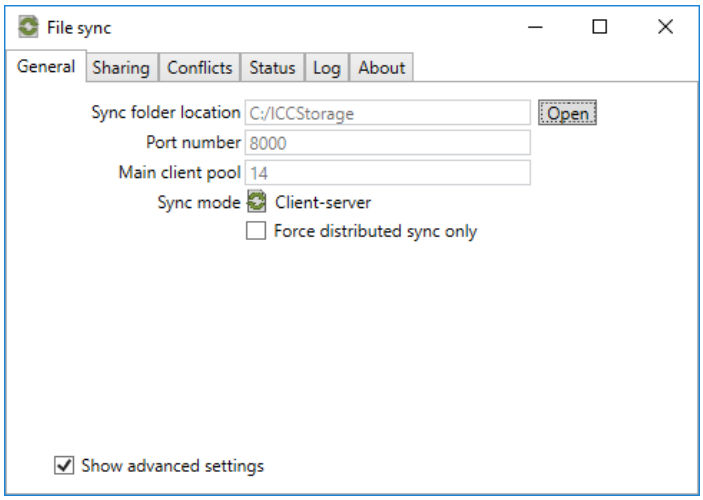
Name	Description
AddEntityToClientPool	Add specified entity to the specified Client pool.
AssignClientPool	Connect specified client to the specified Client pool.
CreateClientPool	Create new Client pool and add selected client to it.
CreateEntity	Create new entity.
GetClientPoolList	Get connected Client pool list for selected client.
GetClientsByPools	Get client info list filtered by list of Client pools (only clients connected to selected Client pools).
GetConflictEntityInfo	Get detail info about entity version which is already in version history and also is in specified conflict.
GetEntity	Get concrete entity data.
GetLatestEntities	Get latest entity list.
GetLatestEntity	Get the latest (or the newest) entity including concrete entity data.
IsEntityPresent	Check if entity and its versions are already present on the server.
Ping	Empty method for checking connection.
RemoveEntityFromClientPool	Remove specified entity from the specified Client pool.
ResolveConflict	Resolve specified entity conflict.
SaveEntity	Save new version of entity.
UpdateClient	Update client information details on the server.
UpdateClientEndpoints	Update client information including endpoints on the server.
UploadHistoryPack	Upload multiple entities to server as a history pack.

Příloha F

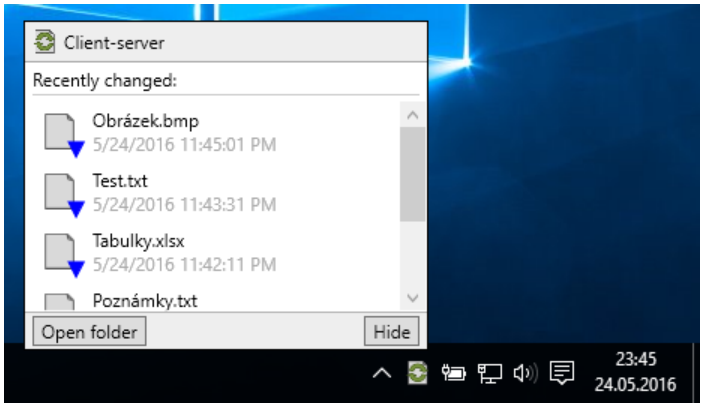
Snímky obrazovky ukázkové aplikace



Obrázek F.1: Obrazovka pro první inicializaci aplikace



Obrázek F.2: Hlavní obrazovka aplikace

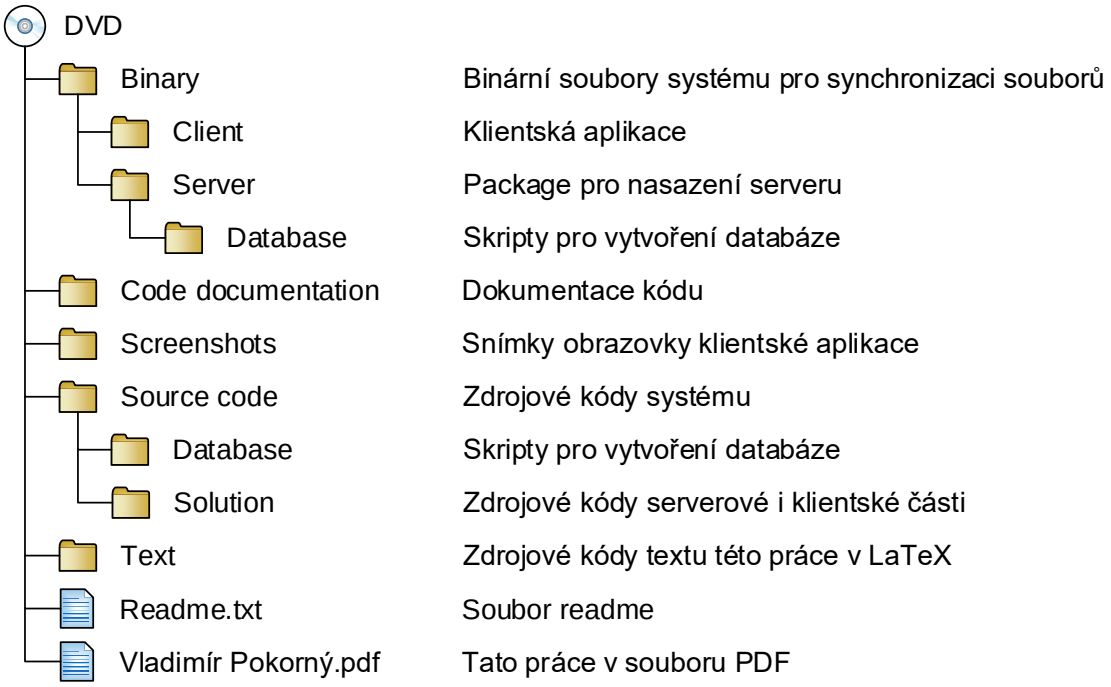


Obrázek F.3: Notifikační zelená ikona s otevřeným vyskakovacím oknem. Zelená barva symbolu znamená, že klient komunikuje v režimu klient-server.

Obrázek F.4: Notifikační modrá ikona. Modrá barva symbolu znamená, že klient komunikuje v režimu peer-to-peer.

Příloha G

Obsah přiloženého DVD



Obrázek G.1: Obsah přiloženého DVD