F3 Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Framework pro synchronizaci datových struktur v distribuovaném prostředí

Bc. Tomáš Hrabáček

Vedoucí práce: Ing. Martin Mudra
Studijní program: Otevřená informatika
Obor: Softwarové inženýrství
Květen 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hrabáček** Jméno: **Tomáš** Osobní číslo: **466354**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Framework pro synchronizaci datových struktur v distribuovaném prostředí

Název diplomové práce anglicky:

Framework for data synchronization in the distributed deployment model

Pokyny pro vypracování:

Analyzujte koncensus algoritmy RAFT [1] a PAXOS [2] popsané v rámci vědeckých prací. Na základě analýzy navrhněte framework pro obecnou synchronizaci dat v distribuovaném prostředí obsahující jednotky až nižší desítky samostatných instancí synchronizované entity. V rámci analýzy využijte vhodných principů optimistické replikace dat z vědeckého článku [3].
V rámci analýzy důsledně popište situace a stavy, ke kterým může v distribuovaném prostředí dojít (odpojení instancí, samostatné ostrovy atp.) Navrhněte synchronizační algoritmus, který bude umět tyto stavy detekovat a případně řešit.
V rámci návrhu
vycházejte z principů výše zmíněných koncensus algoritmů RAFT a PAXOS. Navrhněte případné modifikace či užití těchto algoritmů v rámci synchronizace. Implementujte framework na platformě .NET [4] pro synchronizaci dat v distribuovaném prostředí. Implementujte prototyp aplikace využívající synchronizační framework. Framework otestujte na alespoň 4 simulovaných samostatných instancích. Vytvořte simulaci jednotlivých stavů, ke kterým může v průběhu synchronizace dojít a otestujete je pomocí integračního testování. Simulujte synchronizaci většího (~10 000) počtu entit a simulujte výpadky jednotlivých instancí v průběhu synchronizace. Důsledně otestujte jejich obnovení do plně synchronizovaného stavu při obnovení připojení. K systému implementujte jednoduché GUI vizualizující stav dané instance v průběhu synchronizace a stav synchronizovaných entit.

Seznam doporučené literatury:

[1] Diego Ongaro. RAFT algoritmus - https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14
[2] Leslie Lamport. Paxos Made Simple http://lamport.azurewebsites.net/pubs/paxos-simple.pdf
[3] Optimistic Replication https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2003-60.pdf
[4] https://www.packtpub.com/product/c-8-and-net-core-3-1-recipes-2nd-edition-second-edition-video/9781838986728
[5] Maarten van Steen, Andrew S. Tanenbaum. Distributed Systems

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Martin Mudra, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **10.02.2021** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2022**

Ing. Martin Mudra podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce Ing. Martinu Mudrovi, za cenné rady, které pomohly k jejímu dokončení. Dále děkuji rodině za podporu v průběhu studia i při vypracování této práce.

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, 14. května 2021

Abstrakt

Tato práce se zabývá analýzou možností synchronizace dat v distribuovaném prostředí. Zaměřuje se zejména na způsoby optimistické replikace dat a možností využití konsenzus algoritmů pro tyto účely. Práce dále obsahuje návrh a popis implementovaného frameworku, který se zaměřuje na optimistickou výměnu záznamů mezi uzly v distribuovaném prostředím s využitím modifikovaného algoritmu RAFT.

Klíčová slova: synchronizace datových struktur, distribuované prostředí, optimistická replikace dat

Vedoucí práce: Ing. Martin Mudra

Abstract

This work analyzes the possibilities of data synchronization in a distributed environment. It focuses mainly on optimistic data replication and the possibility of using consensus algorithms for these purposes. The work also contains a design and description of the implemented framework, which focuses on the optimistic exchange of records between nodes in the distributed deployment model with the use of a modified RAFT algorithm.

Keywords: data synchronization, distributed deployment model, optimistic replication

Title translation: Framework for data synchronization in the distributed deployment model

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

Synchronizace dat je jedním z klíčových problémů v distribuovaných systémech. Distribuované systémy se typicky skládají z několika uzlů, které spolu komunikují pomocí zpráv. Data mohou být distribuovány mezi uzly s cílem provést výpočet nad daty rychleji, díky většímu výpočetnímu výkonu. Dalším využitím je zvýšení dostupnosti systému jako celku pomocí replikace dat na jednotlivé uzly. Uživatelé poté mohou přistupovat paralelně ke stejným datům pomocí různých uzlů. V případě jakékoliv změny dat, je nutné tuto změnu propagovat na všechny repliky v daném systému, což bude cílem této práce.

1.1 Motivace

K synchronizaci dat v distribuovaných systémech lze přistoupit ze dvou směrů. První směr je pesimistický – blokování zápisu dat, pokud není dostupný dostatečný počet uzlů systému, případně je zvolen pouze jeden uzel, na kterém lze provádět zápis. Druhý směr je optimistický, kdy jsou data zapsána nejprve lokálně na uzlu a následně distribuována ostatním uzlům. V tuto chvíli ale vzniká riziko, že na dvou odlišných uzlech dojde ke změně stejné datové entity. Tím vzniká mezi změnami konflikt, který musí být řešen. Výhodou optimistického přístupu je, že uživateli je umožněn zápis i čtení dat kdykoliv, i když dojde k selhání ostatních uzlů, nebo spojení mezi nimi.

1.2 Cíl práce

Cílem práce je vyvinout framework, který usnadní programátorům práci při vývoji aplikací, které musí umožnit uživatelům manipulaci s daty i v případě

nedostupnosti připojení k ostatním uzlům, ať už kvůli jejich selhání, nebo selhání sítě. Následně musí dojít k synchronizaci dat a tím vzniká riziko konfliktních změn mezi daty. Framework zajistí propagaci změn dat na všechny uzly a umožní programátorovi implementovat automatické vyřešení konfliktu, případně postoupení vyřešení konfliktu uživateli.

Kapitola 2

Analýza

Tato kapitola nejprve uvádí do problematiky distribuovaných systémů a datových úložišť, následně specifikuje požadavky na framework, který je cílem práce. Dále je v kapitole popsána optimistická a pesimistická replikace dat. V případě pesimistické replikace jsou popsány i algoritmy Paxos a Raft, které jsou jejími představiteli. Na konci kapitoly jsou uvedeny stavy, do kterých se distribuovaný systém může dostat z pohledu topologie sítě.

2.1 Distribuované systémy

Definic pro distribuovaný systém je mnoho, tato práce uvažuje definici, kdy distribuovaný systém je kolekce autonomních výpočetních uzlů, které se jeví uživateli jako jeden celek [47].

2.1.1 Vlastnosti distribuovaných systémů

U distribuovaných systémů existují dvě zásadní vlastnosti, které se sledují. První vlastností je živost (liveness), ta garantuje, že v distribuovaném systému dojde časem k něčemu dobrému, tedy bude dosažen žádoucí stav [47]. Příkladem může být, že na každý dotaz od klienta bude odpovězeno. Druhá vlastnost je bezpečnost (safety). Bezpečnost garantuje, že v distribuovaném systému nikdy nedojde k něčemu špatnému, tedy nebude dosažen nežádoucí stav. Bezpečnost si lze představit tak, že klientovi se na dotaz nikdy nevrátí chybná odpověď [47].

Distribuované systémy můžeme rozdělit z pohledu času [47]:

- Asynchronní.
- Synchronní.
- Částečně synchronní.

Asynchronní distribuované systémy nemají limit na odchýlení lokálních hodin uzlu od hodin ostatních uzlů. Nemají žádné časové limity na rychlost vykonávání práce na uzlech. Nejsou dány časové limity na maximální trvání přenosu zprávy mezi dvěma uzly. Tím pádem je obtížné rozeznat, zda se doručení zprávy zpozdilo, nebo došlo k její ztrátě [47].

U synchronních systémů jsou naopak tyto časové limity známy. Tedy známe maximální odchylku lokálních hodin vůči hodinám ostatních uzlů, známe horní časové limity pro vykonání určité činnosti na uzlu, známe maximální dobu, kterou bude přenesení zprávy trvat [47].

Bohužel synchronní distribuované systémy existují pouze v teoretické rovině [47]. Na druhou stranu předpoklad, že každý systém je asynchronní, je příliš pesimistický a ztížilo by to návrh distribuovaných algoritmů. Pro reálná použití je vhodné uvažovat o částečně synchronních systémech – většinu času se systém chová jako synchronní, dle definovaných časových limitů, ale mohou nastat situace, kdy jsou tyto limity překročeny. V tomto případě uvažujeme že zpráva byla ztracena, resp. uzel přestal pracovat, ale jelikož jsou limity definovány uměle pro asynchronní systém, je možné, že přenesení zprávy opravdu pouze trvá déle resp. provedení výpočtu na uzlu trvá déle. Tedy chybně jsme detekovali ztracenou zprávou, resp. selhání uzlu [47].

2.1.2 Standardní problémy

Problémy, které jsou řešeny distribuovanými algoritmy lze rozdělit do následujících kategorií [35]:

1. **Alokování zdrojů** (Resource allocation) – algoritmy pro optimalizaci přidělování typicky hardwarových zdrojů (např. operační paměť, výpočetní čas) pro jednotlivé procesy.
2. **Atomická činnost** (Atomic commit) – činnost v rámci níž je provedeno několik změn jako jedna nedělitelná operace. Pokud je každá jednotlivá změna v rámci operace úspěšná, pak je úspěšná i celá operace. Pokud jedna ze změn selže, zruší se aplikování všech v rámci dané operace a

systém se uvede do původního stavu před zahájení atomické činnosti. Algoritmy řešící tento problém jsou například dvoufázový potvrzovací protokol a třífázový potvrzovací protokol.

3. **Distribuované hledání** (Distributed search) – typicky grafové úlohy, jako je hledání nejkratší cesty v grafu, hledání nejmenší kostry v grafu.

4. **Konsenzus** (Consensus) – nalezení konsenzu neboli shody je proces při kterém se několik výpočetních uzlů domlouvá na společné hodnotě, či změně. Známými algoritmy pro řešení konsenzu jsou algoritmy Paxos a Raft, které jsou detailněji popsány v kapitole 2.6.

5. **Neblokující datové struktury** (Non-blocking data structures) – řešení problému, kdy selhání jednoho či více vláken neovlivní práci ostatních pracujících vláken.

6. **Replikace** (Replication) – replikace stavu výpočetního uzlu na ostatní výpočetní uzly v síti. Replikace se dělí na dva typy – pesimistickou a optimistickou, detailnější popis tohoto problému se nachází v kapitole 2.4.

7. **Spolehlivý přenos** (Reliable broadcast) – problém přenesení zprávy od jednoho výpočetního uzlu k ostatním uzlům za předpokladu, že jakýkoliv uzel může během komunikace selhat.

8. **Volba vůdce** (Leader election) – proces, při kterém je zvolen jeden z výpočetních uzlů jako vůdce (leader), který bude následně zodpovědný za určitou činnost, např. přijímání požadavků od klientů, přidělování práce ostatním uzlům apod.

9. **Vzájemné vyloučení** (Mutal exclusion) – řešení problému přístupu více výpočetních uzlů ke sdílenému zdroji ve stejný okamžik. Řeší se například pomocí zámků, monitorů, semaforů.

Výše zmíněné problémy jsou mezi sebou úzce propojeny. Například konsenzus algoritmy Paxos [30] a Raft [40] se dají využít i pro volbu vůdce a díky tomu, že se dokážou dohodnout na jedné hodnotě, dokáží zajistit replikaci této hodnoty i na ostatní uzly v síti.

Popsané problémy většinou pramení z toho, že jeden uzel se nemůže spolehnout na stav sítě, či uzlů. Stav sítě a uzlů se v průběhu času může měnit a nemusí být vždy zřejmé, k jakému problému došlo. Zde je výčet neznámých a nejistot, které přináší práce v distribuovaném prostředí [35]:

- Neznámá topologie sítě.
- Nejisté pořadí doručení zpráv.

- Nejistý čas doručení zpráv.
- Možnost ztráty zpráv.
- Neznámý počet uzlů v síti.
- Rozdílná rychlost práce uzlů.
- Potencionální selhání uzlů.

Algoritmy řešící výše uvedené standardní problémy mají, kvůli nejistotám, kompromis v určitých parametrech. Typicky to bývá spolehlivost oproti rychlosti, nebo konzistence oproti dostupnosti.

2.1.3 Škálování distribuovaných systémů

Distribuované systémy přináší nové možnosti škálování – horizontální škálování (scaling out). Systémy běžící pouze na jednom výpočetní uzlu lze škálovat pouze vertikálně (scaling up) – zvyšováním výkonu daného výpočetního uzlu (rozšíření operační paměti, vylepšení procesorů apod.). Naopak horizontální škálování umožňuje přidat nové výpočetní uzly a tím zvýšit výpočetní výkon celého systému [47].

Další možností je zvýšení dostupnosti díky horizontálnímu škálování. V případě sytému běžícího na jednom výpočetním uzlu, je tento uzel místem, na kterém závisí celý systém – selže uzel, selže systém, v literatuře označováno jako single point of failure. Pokud je systém distribuován na více uzlech, lze ho nakonfigurovat tak, že i při výpadku části uzlů, bude systém dále fungovat [47].

Na druhou stranu distribuce vyžaduje úsilí navíc, v podobě přenosu dat mezi jednotlivými uzly. Dále přináší sebou i nová rizika – selhání komunikačního kanálu, případně výpočetního uzlu. To má za následek větší složitost algoritmů, které s těmito stavy musí umět pracovat [47].

2.2 Distribuovaná datová úložiště

Synchronizace datových struktur je velmi rozšířena v oboru databázových systémů a distribuovaných datových úložišť. V případě datových úložišť se lze setkat zejména s pojmy ACID, CAP a BASE, jenž reprezentují různé vlastnosti datových úložišť.

2.2.1 ACID

Zkratka ACID vyjadřuje, jaké vlastnosti by měla splňovat databázová transakce [25]. Tato zkratka se nejčastěji používá u relační databázových systémů (RDBMS). Jednotlivá písmena zkratky znamenají následující [25]:

- Atomicity (atomicita) – soubor několika změn se provede jako jedna operace. Pokud se jedna ze změn nepovede, neprovede se žádná změna (obdobně jako u atomické činnosti zmíněné v kapitole 2.1.2).
- Consistency (konzistence) – transakce převede stav databáze z jednoho konzistentní stavu do jiného konzistentní stavu, tj. nebudou porušeny podmínky, které byly stanoveny na databázové úrovni (např. nebudou porušeny cizí klíče, omezení na unikátnost apod.).
- Isolation (izolace) – v případě více současně běžících vláken, se vlákna navzájem neovlivňují.
- Durability (trvalost) – v případě, že se transakce dokončí, je její výsledek trvalý, nedojde k jeho ztrátě např. výpadkem napájení.

ACID vlastnosti splňují nejčastěji relační databáze jako jsou Microsoft SQL Server, PostgreSQL a další.

2.2.2 CAP

CAP teorém (Brewerův teorém) udává, že distribuované datové úložiště dokáže garantovat pouze dvě z následujících tří vlastností [13]:

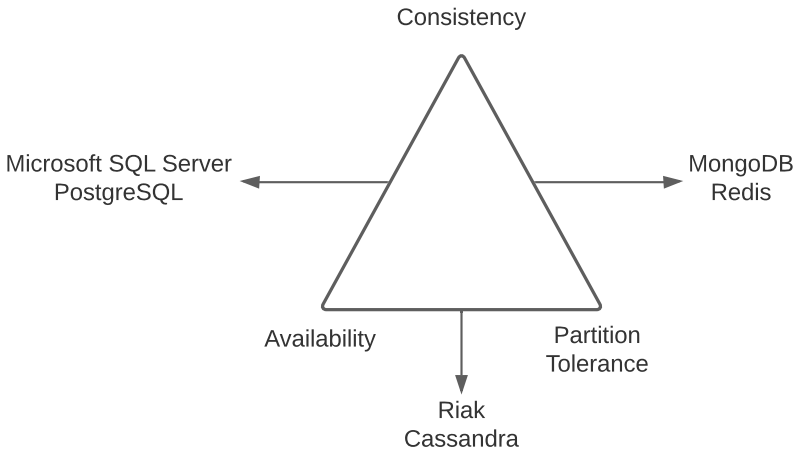
- Consistency (konzistence).
- Availability (dostupnost).
- Partition tolerance (odolnost vůči přerušení).

Konzistence je zde brána z jiného pohledu, než je tomu u pojmu ACID. Zde se jedná pohled na konzistence dat mezi jednotlivými uzly sítě. V podstatě všechny operace čtení i zápisu se musí provést atomicky v rámci celé topologie. To má za následek, že po jakékoliv operaci zápisu, všechny operace čtení na všech uzlech v síti vrátí stejná data (v případě stejné dotazu) [22].

Dostupnost znamená, že pokud korektně běžící uzel přijme dotaz, musí na něj odpovědět – nesmí ho zamítnout [22].

Odolnost vůči přerušení stanovuje, že systém pokračuje v práci i v případě, že se jeden či více uzlů izoluje od ostatních. To znamená, že dojde k přerušení komunikačního kanálu mezi uzly a ty se rozpadnou do jednotlivých ostrovů [22].

V případě, že lze vybrat pouze zajištění dvou vlastností ze tří, vznikají tři možné kombinace [22] označované jako CA, CP, AP. Výběr těchto vlastností lze také ilustrovat jako trojúhelník (obrázek 2.1), kde můžeme vybrat pouze dva vrcholy.



Obrázek 2.1: CAP teorém – vizualizace pomocí trojúhelník, kde lze zvolit pouze dva vrcholy

Popis kombinací

Kombinaci CA (konzistence a dostupnost) představují již zmíněné tradiční relační databázové systémy v předchozí kapitole. Každý systém, který má pouze jeden uzel, splňuje podmínky CA. V případě sítě s více uzly, se musí operace provést atomicky na všech uzlech, toho lze dosáhnout pomocí již zmíněných dvoufázových zamykacích protokolů. Tyto protokoly ale potřebují pro svou činnost konektivitu ke všem, resp. většině uzlů. Pokud by uzel nebyl ve skupině uzlů tvořící většinu, tak nemůže přijmout požadavky na čtení a zápis, jelikož si nemůže být jistý, že jeho data jsou aktuální vzhledem ke většině uzlů – mohl by tím porušit konzistenci. V případě ale, že neodpoví, porušuje tím předpokládanou dostupnost. Jelikož výpadku konektivity mezi uzly nelze zabránit, rozhodujeme se u distribuovaných systémů především mezi kombinacemi CP a AP [22].

Představitelem kombinace CP (konzistence a odolnost vůči přerušení) je NoSQL databáze MongoDB [37]. Databáze MongoDB řeší konzistenci tím způsobem, že dovolí čtení a zápis dat pouze na jednom uzlu v síti. Na ostatní

uzly poté data replikuje[1]. V případě že dojde k rozpadu sítě, MongoDB je schopné pracovat, dokud se v jednom ostrovu nachází více jak polovina uzlů. Uzly nenacházející se v tomto ostrovu odmítají požadavky na čtení a zápis, mohou si to dovolit, jelikož dostupnost není garantována [37].

Kombinace AP preferuje dostupnost a odolnost vůči přerušení nad konzistencí. V případě rozpadu sítě, musí každý uzel umožňovat čtení i zápis dat. Z toho vyplývá, že data poskytovaná v rámci jednoho ostrova budou odlišná od těch, které budou poskytována v jiném ostrově, v případě že po rozpojení sítě došlo k zápisu dat. V tuto chvíli také vzniká možnost, že dojde k odlišným změnám nad stejnými daty v různých ostrovech, tedy vzniká konflikt mezi změnami [22]. Jako představitele lze uvést databázové systémy Riak [8] a Cassandra [2]. Riak k detekci a řešení konfliktních změn používá verzovací vektory [8]. Zatímco systém Cassandra se vydává cestou minimalizace rizika konfliktů tím, že jednotlivé změny dat, neprovádí pro celý datový objekt najednou, ale po jednotlivých vlastnostech objektu (každá vlastnost je uložena v jednom sloupci) [2]. Pro tento přístup, kdy konzistence není klíčová se vytvořil nový pojem BASE, který je podrobněji popsán níže.

2.2.3 BASE

Koncept vlastností BASE je protiklad ke konceptu ACID [42]. Zahrnuje následující vlastnosti [42]:

- Basically Available – systém je v podstatě dostupný po celou dobu, pouze částečné výpadky mohou nastat, celý systém neselže najednou.
- Soft State – stav není stabilní, mění se v průběhu času i bez vstupů od uživatele (dochází k doplňování dat z jiných uzlů, probíhá oprava konfliktů).
- Eventual Consitency – systém se stane konzistentní v průběhu času, když budou všechny uzly dostupné mezi sebou a systém neobdrží žádný vstup.

2.2.4 Shrnutí

Koncepty ACID a BASE jdou proti sobě. Relační databáze podporují ACID, což odpovídám zajištění konzistence a dostupnosti z pohledu CAP teorému. Koncept BASE je doménou některých NoSQL databází, kde postačuje pouze eventuální konzistence [42].

[1] MongoDB lze nastavit tak, že lze číst i z replik, v tu chvíli ale negarantuje konzistenci.

V případě eventuální konzistence je umožněno zapisovat a číst data, i když je část uzlů v rámci sítě izolovaná [42]. Tím pádem může dojít ke konfliktním změnám nad daty. Je tedy nutný mechanismus, který po obnovení spojení zajistí výměnu chybějících dat mezi uzly a také mechanismus, který zajistí detekci a řešení konfliktní změn mezi daty. Tyto mechanismy bude poskytovat framework vytvořený v rámci této práce. Příklady těchto mechanismů se nacházejí v další kapitole 2.4 Replikace dat.

Distribuce dat na různé uzly v rámci sítě (horizontální škálování) přináší možnosti, jak zvýšit dostupnost a spolehlivost. Na druhou stranu distribuce dat přináší nové překážky (potencionální výpadek části sítě nebo uzlu) v případě, že chceme udržet data konzistentní na všech uzlech [22].

2.3 Analýza požadavků

Soupis funkčních a nefunkčních požadavků na systém, který bude navržen a implementován v rámci této práce.

2.3.1 Funkční požadavky

Seznam funkčních požadavků, které musí systém splňovat:

1. Systém bude podporovat uložení entit.
2. Systém bude podporovat verzování uložených entit.
3. Systém bude podporovat synchronizaci uložených entit napříč uzly systému.
4. Jednotlivé uzly systému v síti budou schopny ukládat data trvale, i v případě výpadku připojení k ostatním uzlům.
5. Systém bude schopen detekovat nedostupnost ostatních uzlů.
6. V případě rozpadu sítě na menší části, se uzly budou nadále synchronizovat s dostupnými uzly v dané části.
7. Po obnovení spojení mezi uzly systému, bude systéme schopen synchronizovat chybějící data mezi jednotlivými uzly.
8. Uzel, který selže, je schopen si po obnovení synchronizovat chybějící data.
9. Systém bude schopen identifikovat konfliktní změny instance dané entity.

10. Systém bude umožňovat uživateli ruční řešení konfliktních změn.
11. Systém bude umožňovat programátorovi implementovat automatické sémantické a syntaktické řešení konfliktních změn.

2.3.2 Nefunkční požadavky

Seznam nefunkčních požadavků, které musí systém splňovat:

1. Systém bude komunikovat po síti.
2. Jednotlivé uzly systému budou komunikovat pomocí API.
3. Systém bude implementován formou knihovny.
4. Systém bude postaven na technologii .NET [34].
5. Zdrojové kódy budou v anglickém jazyce.

2.4 Replikace dat

Replikace dat se zabývá udržováním kopií dat na vícero uzlech (nazývané repliky) v síti. Jedná se o klíčovou techniku, jak v distribuovaném systému zajistit lepší dostupnost a výkonost. Dostupnost je zlepšena tím, že data jsou dostupná i když některá z replik není dostupná. Lepší výkonost systému je zajištěna tím, že uživatelé mohou přistupovat k replikám, které jsou k nim geograficky blíže a tím dochází ke snížení latence. Další vylepšení výkonu je z pohledu zvýšení propustnosti dat. Jelikož data jsou poskytována více uzly, lze tudíž obsloužit větší množství požadavků [43].

Replikaci dat můžeme rozdělit do dvou směrů. Prvním směrem je tradiční pesimistická replikace dat, též označována jako eager replication. Druhým směrem je optimistická replikace dat, označována jako lazy replication [43].

Případ pesimistické replikace dat odpovídá zajištění konzistence z CAP teorému. Systém se snaží udržet jednu kopii dat na všech replikách [12]. Toho může být dosaženo různými způsoby, avšak základní koncept zůstává vždy stejný – zablokování přístupu k datům, dokud není zajištěno, že jsou aktuální [44]. Jeden z využívaných algoritmů k tomuto účelu se nazývá primary-copy algoritmus. Tento algoritmus pracuje tak, že vybere jednu primární repliku, která bude zodpovědná za přijímání a zpracování požadavků od klientů. Ostatní repliky pouze čekají na změny, které jim zašle primární replika. Dalším algoritmem je dvoufázový potvrzovací protokol, ten nejprve

uzamkne všechny repliky a poté jim najednou aktualizuje data. Lze také využít konsenzus algoritmy využívající kvórum a nechat hlasovat uzly o každé jednotlivé změně [44].

Jednoznačnou výhodou je již zmíněná konzistence, na druhou stranu pesimistická replikace neposkytuje příliš dobrou dostupnost a výkonost v sítích, které jsou pomalé či nespolehlivé, ve kterých může docházet k výpadkům komunikačních kanálů. V případě primary-copy přístupu je rozpad sítě fatální pro uzly, které ztratí spojení s primární replikou – přestanou získávat aktualizace. Výkonnost pro zápis dat může klesat se zvyšujícím se počtem uzlů v systému, jelikož se i zvyšuje doba potřebná pro replikaci dat na všechny uzly. Nejvíce se to dotýká dvoufázového potvrzovacího protokolu, který musí provést uzamčení na více uzlech, tak i konsenzus algoritmů, které potřebují získat hlas od více uzlů [44].

Optimistická replikace naopak zajišťuje pouze eventuální konzistenci dat, ale poskytuje dostupnost a odolnost vůči přerušení dle CAP teorému. Umožňuje uživatelům přistupovat k datům z jakékoliv repliky a to kdykoliv. Základní předpoklad pro tento způsob je, že konfliktní změny dat se vyskytují pouze vzácně a data na jednotlivých replikách jsou téměř konzistentní. Hlavní rozdíl mezi optimistickou a pesimistickou variantou replikace je ve způsobu přístupu k práci s daty, zejména v propagování jejich změn. Pesimistické algoritmy blokují přístup k replikám, dokud není změna propagována všude. Optimistické algoritmy naopak nejdříve data uloží u sebe a poté je na pozadí asynchronně posílají ostatní replikám. Mezitím je umožněn zápis dat jak na dané replice, tak i na všech ostatních. Tento přístup dělá z optimistické replikace více dostupný a výkonný způsob pro sdílení dat, ale za cenu, že je zajištěna pouze eventuální konzistence dat a může dojít tedy ke konfliktům, které bude potřeba vyřešit.

Vzhledem k povaze zadání se budeme nadále primárně zabývat optimistickou replikací dat, jelikož odpovídá definovaným požadavkům – uzly jsou schopny ukládat data, i když došlo k výpadku spojení k ostatním uzlům. U systémů, které využívají optimistickou replikaci se sleduje několik vlastností [43]:

- Počet primárních replik – replik, které mohou zapisovat data. Systémy dělíme na single-master a multi-master.
- Dle úrovně definování operací – state-transfer a operation-transfer systémy.
- Plánování – jakým způsobem systém přistupuje k řazení operací.
- Správa konfliktů – jakým způsobem systém přistupuje k identifikaci a řešení konfliktů.
- Propagace dat – jaký způsobem jsou data propagována na ostatní repliky.

- Garance konzistence – jakou maximální odlišnost dat na jednotlivých replikách systém garantuje.

2.4.1 Single-master a multi-master systémy

Systémy lze dělit dle toho, kolik může být v jednu chvíli primárních replik. Pokud to může být pouze jedna replika, hovoříme o single-master systémech. Tyto systémy pracují na stejném principu jako pesimistické primary-copy algoritmy popsané výše, s tím rozdílem, že data na ostatní repliky mohou přenášet na pozadí, až po tom, co odpoví klientovi, že zápis dat uspěl. Další odlišnost spočívá ve způsobu čtení dat. Čtení je umožněno ze všech replik. Single-master systémy mají tedy i podobné vlastnosti jako primary-copy algoritmy, zejména nejsou příliš efektivní v případě systému zapisující velké množství dat v krátkém časovém úseku. Systémy využívající pouze jednu primární repliku je vhodné použít v případě, že většina dotazů na data je pouze čtecích [43].

Práce s konflikty v single-master systémech je triviální. Primární replika vždy dokáže snadno identifikovat konflikty – vznikají pouze v případě, že dva uživatelé ve stejnou chvíli upraví stejná data. Řešení je též snadné, požadavek od klienta, který přišel později se zamítne a klient je nucen si načíst nová data [44].

Pokud primárních replik může být více, jedná se o multi-master systémy. V těchto systémech je uživatelům umožněno provádět změny dat z vícero uzlů. Změny se poté propagují na ostatní repliky asynchronně. Proto v těchto systémech může dojít ke vzniku konfliktů mezi již uloženými daty [43]. Identifikace a řešení těchto složitějších konfliktů je nastíněna v další kapitole 2.4.4.

Multi-master systémy jsou lépe škálovatelné z pohledu dostupnosti zápisu dat oproti single-master systémům. S přidáním nových replik umožníme více klientům zapisovat data, ale s větším počtem replik stoupá riziko konfliktů, které budou muset být řešeny [43].

2.4.2 Operation-transfer a state-transfer systémy

Dělit replikační systémy můžeme i z pohledu operací, které umožňují data číst a manipulovat s nimi. State-transfer systémy umožňují danou entitu pouze číst, nebo ji celou přepsat. Operation-transfer systémy naopak umožňují definovat operace z vyššího pohledu – ze sémantického významu pro jednotlivé typy entit [43].

State-transfer systémy jsou jednodušší na implementaci, jelikož se jejich práce skládá pouze z posílání posledního stavu entity ostatním replikám, nejčastěji určené pomocí časové značky. V případě, že se bude jednat o velké entity, posílání jejich celého obsahu při každé změně může být značně neefektivní. Pro snížení zátěže sítě posíláním nadbytečných informací, lze zavést mechanismus, který bude zajišťovat, že si repliky budou udržovat malou historii posledních změn. V případě že replika, která má obdržet entitu, není příliš pozadu, stačí jí poslat pouze poslední změny. Pokud je replika příliš pozadu, tj. změny by ji neuvedly do stavu stejného jako má replika odesílající entitu, pošle se celá entita. Tato krátká historie nedosahuje stále složitosti operation-transfer systémů, jelikož každá změna obsahuje časovou značku, tudíž změny lze jednoznačně seřadit [44].

Oproti tomu operation-transfer systémy jsou složitější, což je jejich hlavní nevýhoda – složitější implementace. Musí si udržovat celou historii operací, které byly provedeny nad entitou a jednotlivé repliky se musí shodnout na jejich pořadí. Výhodou je, že se nepřenáší všechna data, jak tomu je u základní verze state-trasnfer systémů. Další výhodou jsou širší a flexibilnější možnosti řešení konfliktu, jelikož historie, jak se entita měnila v průběhu času je známa [43].

2.4.3 Plánování

Plánování se zabývá uspořádáním jednotlivých operací. Cílem je uspořádat operace tak, aby všechny repliky konvergovaly k jednotnému stavu. Plánování můžeme rozdělit do dvou skupin, na syntaktické a sémantické. Plánování se uplatňuje pouze u operation-transfer systémů [43].

Syntaktické plánovaní si zakládá na obecnosti a jednoduchosti. K řazení využívá často metadata daných operací, např. časovou značku vytvoření operace nebo prioritu operace. Díky tomu lze použít tento způsob na jakoukoliv operaci, pokud budou potřebná metadata zaznamenána, není potřeba vědět, co daná operace dělá. Tato jednoduchost sebou přináší i nevýhody v podobě zbytečných konfliktů [43]. Pokud budou například současně odeslány 3 požadavky: uživatel A požaduje 1 kus produktu X, uživatel B požaduje též jeden kus produktu X, uživatel C přidá na sklad 5 kusů produktu X. Na počátku je ve skladu pouze jeden produkt X. Pokud budou požadavky seřazeny tak, jak jsou uvedeny v textu, požadavek uživatele B bude zamítnut z důvodu nedostatečného množství produktu X na skladě.

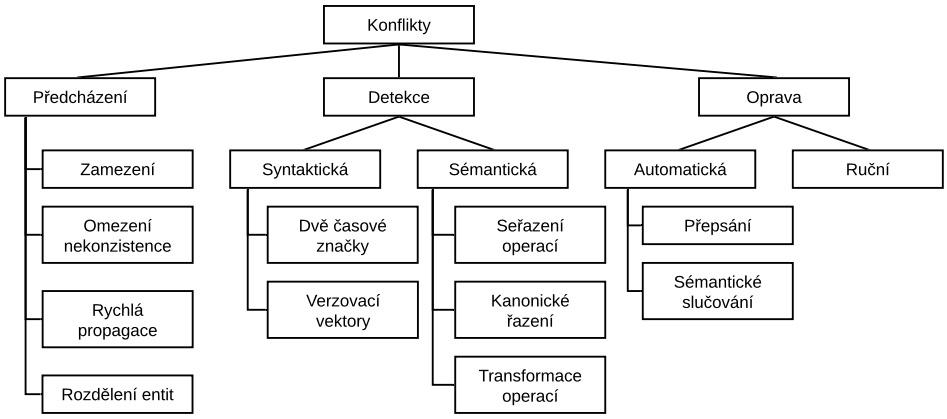
V případě sémantického plánování lze využít významu jednotlivých operací k jejich seřazení. Operace z předchozího příkladu lze uspořádat tak, že všechny požadavky budou uspokojeny. Stačí na základě znalosti aktuálního množství produktů na skladě vyhodnotit, že požadavek uživatele B bude zamítnut a

pokusit se operace přeuspořádat tak, že před dalším požadavkem na vyjmutí produktu X ze skladu, se provede požadavek na přidání produktu X na sklad.

Sémantické plánování tedy umožňuje podrobnější seřazení operací a tím přispívá k redukování množství konfliktů. Je tomu ale za cenu výrazně vyšší složitosti oproti syntaktickému plánování. Plánovače musí být konstruovány zvlášť pro každý druh entity a jejich různé kombinace operací [43].

2.4.4 Správa konfliktů

Jakými způsoby lze přistoupit ke konfliktům je vyjádřeno pomocí obrázku 2.2. Konflikt vzniká, jestliže operace nedokáže splnit definované předpoklady. V případě operace aktualizace dat to je typicky podmínka, že nová data nahrazují předchozí verzi dat a ta se od posledního čtení nezměnila. Porušení této podmínky může být způsobeno například tak, že vícero uživatelů manipuluje paralelně se stejnými daty, případně některý uživatel manipuluje s daty, která již nejsou aktuální [43].



Obrázek 2.2: Klasifikace práce s konflikty [44]

V ideálním případě je nejlepší se vzniku konfliktům vyhnout. Tento způsob je využit u pesimistických algoritmů a single-master algoritmů. Jak je ale výše popsáno, tyto algoritmy nedosahují vysoké dostupnosti pro zápis dat. Dalšími způsoby je rozdělení entity na co nejmenší části tak, aby změna jedné části, neovlivňovala další části. Zvýšením rychlosti, jakou se data propagují na ostatní repliky, též dojde ke snížení rizika vzniku konfliktů, jelikož je zde menší šance, že uživatel pracuje nad neaktuálními daty. Pokud dojde k odpojení jedné repliky od ostatních, lze omezit divergenci jejích dat např. omezení počtu operací, které mohou být vykonány v odpojeném stavu. Případně lze omezit dobu, po kterou lze data upravovat – např. data lze měnit pouze prvních 30 minut v odpojeném stavu, pak již nelze, dokud se spojení neobnoví a nedojde k synchronizaci replik [43].

Jestliže ke konfliktům dochází, je nutné je detekovat. Detekce se dělí obdobně jako plánování na sémantický a syntaktický přístup. Celkově plánovaní a detekce jsou si velice blízké. Syntaktická detekce též spoléhá na metadata operací v podobě časových značek či verzovacích vektorů, kdežto sémantická detekce spoléhá na význam jednotlivých operací. Syntaktický přístup je snazší a použitelný obecně, ale některé operace může označit zbytečně za konfliktní, i když ze svého významu nejsou. Sémantický způsob může detekovat tedy opravdu jen konflikty, které má smysl řešit, ale je nutné vytvořit zvlášť algoritmy pro jednotlivé druhy operací na základě jejich významu.

Pro detekované konflikty se musí najít řešení – opravit je. Jedním ze způsobů je přenechat opravu na uživateli. Druhým způsobem je konflikt vyřešit automaticky – programově. Nejjednodušším způsobem je přepsání – vybere se jedna z operací, která vyhraje a přepíše změny v té druhé, tudíž změny z druhé operace jsou navždy ztraceny (lost updates). Pokud nechceme, aby k těmto ztrátám docházelo je nutné se zabývat sémantikou jednotlivých operací a sloučit je v jednu. To ale vyžaduje opět konstrukci mechanismů unikátních pro jednotlivé typy entit stejně jako tomu je u sémantického plánování a sémantické detekce konfliktů. V některých případech ale ani při použití sémantického slučovaní nelze rozhodnout, jak má sloučená operace vypadat a je nutné rozhodnutí přenechat na uživateli [43].

2.4.5 Propagace dat

Operace provedené na lokální replice musí být přeneseny i na ostatní repliky. Propagaci lze klasifikovat ze dvou pohledů – topologie sítě, kterou jsou data přenášeny a způsob, jakým si mezi sebou uzly vyměňují data.

Sítě typu mesh jsou pro replikaci dat vhodnější, než topologie jako jsou strom, či hvězda, jelikož mezi jednotlivými uzly bývá více spojů, a tudíž i při výpadku jednoho ze spojů, či uzlů, lze využít ostatních cest k propagaci dat. Algoritmy, které se snaží využít všech možný cest k tomu, aby doručily data se nazývají epidemické – data epidemicky šíří celou sítí, tak aby byla co největší pravděpodobnost, že se dostanou k cílovému uzlu. Naopak u topologií strom, hvězda apod. stačí aby jediný spoj či uzel selhal a dojde ke ztrátě spojení se všemi uzly, které spoléhaly na tuto jedinou cestu či uzel [44].

Existuje několik způsobů, jakým zajistit výměnu dat mezi jednotlivými replikami. Repliky mohou dle vlastní iniciativy posílat ostatním replikám nové operace a to okamžitě, jakmile je operace provedena, případně periodicky po dávkách. Opačný způsob je, když se repliky periodicky dotazují ostatních replik, zda u sebe nemají nějaké nově provedené operace. V neposlední řadě je též možné využít kombinace zmíněných způsobů [43].

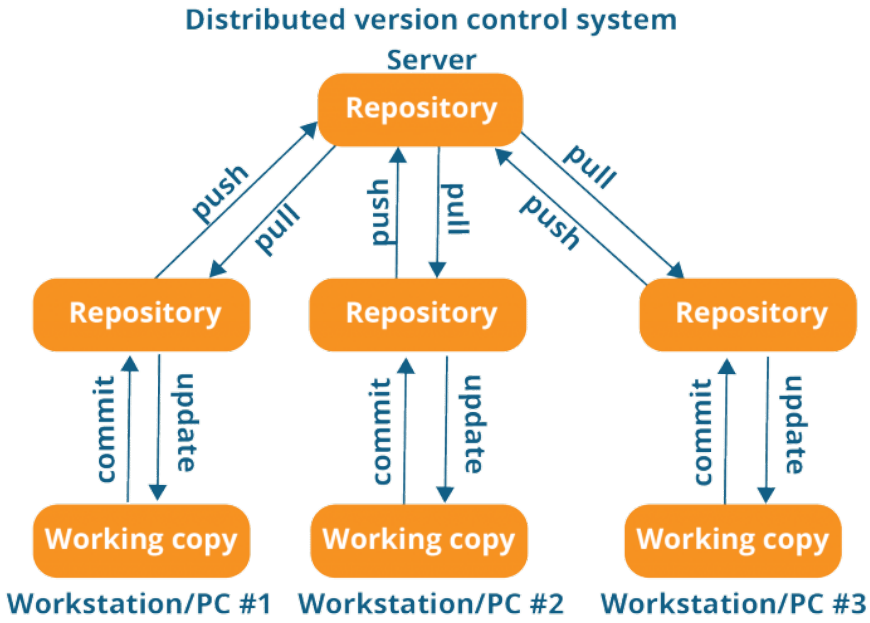
2.4.6 Garance konzistence

Systémy s optimistickou replikací dat nejčastěji poskytují pouze eventuální konzistenci – stavy jednotlivých replik eventuálně konvergují k jednomu společnému. O něco větší konzistenci poskytuje koncept zvaný ohraničená divergence. Ta nastavuje limity a podmínky, jak příliš může stav repliky divergovat vzhledem k ostatním [43]. Takovými limity mohou být počty operací, či doba odpojení zmíněné v kapitole 2.4.4.

2.5 Synchronizace pomocí verzování

Nástroje pro distribuovanou správu verzí (distributed version control system), např. CVS [3], či novější Git [4] jsou ve své podstatě optimistické replikační systémy [43]. Umožňují kolaborativní spolupráci více programátorů nad jednou společnou bází zdrojových kódů. Základem je centrální server, který obsahuje repozitář (viz obrázek 2.3). V repozitáři jsou uloženy zdrojové kódy po jednotlivých souborech, ve stejné podobě jako by je měl programátor uložen v souborovém systému vlastního počítače. Spolu s aktuální verzí jednotlivých souborů repozitář také uchovává celou jejich historii. Při práci s verzovacími nástroji si programátor nejprve naklonuje repozitář z centrálního serveru do svého počítače. Následně již může pracovat nad staženými soubory obsahující zdrojové kódy. Jakmile upraví jakýkoliv ze souborů, vzniká tím na jeho počítači modifikovaná privátní kopie (working copy). Programátor průběžně při své práci vytváří ucelené sady změn nad zdrojovými kódy. Tyto změny ukládá do verzovacího systému, vytváří novou verzi (případně revizi) dat, nad kterou následně provede commit, neboli potvrzení, že se jedná o ucelenou sadu změn. Verze obsahuje informace o řádcích souborů, které byly změněny od předchozí verze. Tato verze je uložena v rámci naklonovaného repozitáře na počítači programátora. Pokud změny považuje za finální a chce je sdílet s ostatními programátory, musí provést příkaz push, tedy odeslání verze na centrální repozitář [15].

V případě, že centrální repozitář neobsahuje žádné nové změny, tedy předchozí verze, na kterou se nahrávaná verze odkazuje, je aktuální, změny se okamžitě uloží. Pokud ale již došlo ke změnám – poslední platná verze je odlišná od té, na kterou se odkazuje aktuálně nahrávaná, musí se přistoupit k detekování, zda nenastal konflikt mezi verzemi. Pokud změněné řádky pro každý soubor jsou v jednotlivých verzí v disjunktních množinách (žádný řádek nebyl změněn v obou verzích), konflikt nenastává a změny se automaticky sloučí a vytvoří se nová verze, která obsahuje změny z obou předešlých verzí. V opačném případě, kdy je detekován konflikt mezi změnami, je programátor vyzván, aby jej manuálně vyřešil [15].



Obrázek 2.3: Model architektury distribuovaných verzovacích systémů [9]

Poté co jsou změny uloženy na centrálním repozitáři, mohou si je ostatní programátoři stáhnout do svého lokálního repozitáře pomocí příkazu pull – ten se provádí i automaticky před příkazem push, aby mohlo dojít k detekci konfliktů. Změny jsou následně automaticky aplikovány na privátní kopii dat (update).

Distribuované verzovací systémy umožňují tzv. větvení (branching). Vytvoření jednotlivých větví slouží programátorům k nerušenému vytváření verzí, jejich commitování a následnému odesílání na centrální repozitář, aniž by zatím museli řešit konfliktní změny, které vytvořili ostatní programátoři v jiných větví. Konfliktní změny se řeší až ve chvíli, kdy dochází ke sloučení (merge) dvou větví. Při sloučení větví se všechny změny ze všech verzí v každé větvi sloučí do jedné a tyto výsledné verze se mezi sebou porovnávají, tak jako tomu bylo při sloučení dvou verzí, jenž je popsáno výše. Vytváření větví sebou nese další výhody pro vývoj softwaru – např. uchování stabilní verze, postupné přidávání nové otestované funkcionality [15].

Z pohledu dělení z předchozí kapitoly se jedná o multi-master operation-transfer systémy, které jsou zapojeny do hvězdy a komunikují mezi sebou přes centrální server. Detekce konfliktů a jejich řešení je distribuováno, jelikož se provádí na jednotlivých replikách (lokálních repozitářích). Objekt, nad kterým synchronizační systém umožňuje pracovat je vždy celý řádek souboru, definuje nad nimi následující operace: přidání, smazání, editace. Detekce konfliktů se provádí na úrovni jednotlivých řádků souborů, pokud konflikt nastane, řeší se manuálně. Verzovací nástroje tedy nepodporují sémantické řešení konfliktů a jsou silně zaměřeny na textové soubory [43].

2.6 Analýza konsenzus algoritmů

Nalezení konsenzu je jedním z typických problémů, které se řeší v distribuovaných systémech. Algoritmy řešící nalezení konsenzu musí splňovat následující formální vlastnosti [47]:

- Konečnost – každý korektně běžící výpočetní uzel rozhodne pro nějakou hodnotu.
- Platnost – pokud všechny uzly navrhnou hodnotu h, tak každý korektně běžící uzel se rozhodne pro hodnotu h.
- Integrita – každý korektně běžící výpočetní uzel se rozhodne nejvýše pro jednu hodnotu a pokud se rozhodne pro hodnotu h, tak hodnota h musela být navržena nějakým výpočetním uzlem.
- Dohoda – pokud se korektně běžící uzel rozhodne pro hodnotu h, tak všechny korektně běžící výpočetní uzly se rozhodnou pro hodnotu h.

Ke konsenzu lze dojít v synchronních distribuovaných systémech. Naopak v asynchronních distribuovaných systémech ke konsenzu dojít nelze [47]. Vyplývá to z FLP teorému (Fisher, Lynch, and Paterson Impossibility). Ten udává, že v asynchronním distribuovaném systému nelze dosáhnout zároveň živosti i bezpečnosti distribuovaného výpočtu, pokud v něm může docházet k selháním – byť i jediného uzlu [19]. Teorémy CAP a FLP jsou si do jisté míry podobné. Oba udávají, že v síti, kde může dojít k selhání spojů či uzlů, nelze zajisti zároveň živost i bezpečnost. V případě CAP teorému je bezpečností myšlena konzistence a živostí je myšlena dostupnost [21].

Jak již víme, synchronní distribuované systémy jsou pouze teoretické, proto se budeme dále zabývat pouze částečně synchronními systémy. V těchto systémech lze konsenzus nalézt, jelikož dle stanovených limitů pro přenos zpráv mezi uzly, dokážeme říci, zda cesta, případně uzel, selhaly.

Dále je v této kapitole obsažen popis porovnání dvou hlavních představitelů z řad algoritmů pro nalezení konsenzu. Algoritmy se nazývají Paxos a Raft. U algoritmu Paxos je uvedeno více jeho variant. Oba algoritmy využívají kvór a principu replikace stavového automatu [45] na jednotlivých uzlech systému. Replikací stavového automatu dochází k pesimistické replikaci dat mezi jednotlivými uzly [46].

2.6.1 Algoritmus Paxos

Algoritmus Paxos vytvořil matematik a informatik Leslie Lamport [30]. V současné době se již jedná o rodinu algoritmů, které se mírně liší ve svých vlastnostech. Příklady variant algoritmu jsou: základní Paxos, Multi-Paxos, Fast Paxos, Byzantine Paxos. Všechny varianty mají stále stejný cíl – dojít ke konsenzu.

Aby Paxos fungoval korektně, obsahuje několik požadavků a předpokladů na systém, ve kterém má běžet. Tyto požadavky většinou platí, až na výjimky, pro všechny jeho varianty. Jedním ze základních požadavků je počet uzlů potřebných pro nalezení konsenzu. Počet potřebných uzlů lze vyjádřit rovnicí $n = 2F + 1$, kde n je počet hledaných uzlů a F je počet uzlů v chybovém (failure) stavu. Z toho vyplývá, že počet správně pracujících uzlů musí být vždy ostře větší, než počet chybujících uzlů [31].

Algoritmus Paxos předpokládá, že uzly systému mají následující vlastnosti:

- Každý uzel pracuje libovolnou rychlostí.
- Uzel může selhat – přestat pracovat.
- Uzel s perzistentní pamětí se může po selhání připojit zpět do systému.
- Na uzlech nedochází k Byzantským selháním [32] (neplatí pro Byzantine Paxos [14]).

Dále Paxos obsahuje předpoklady pro síť, ve které komunikuje:

- Síť je plně propojena – uzly mohou posílat zprávy kterémukoliv jinému uzlu.
- Může dojít ke ztrátě zpráv, změny jejich pořadí, nebo k jejich duplikaci.
- Zprávy jsou odesílány asynchronně a jejich doručení může trvat libovolnou dobu.
- Zprávy jsou doručeny bez poškození (neplatí pro Byzantine Paxos [14]).
- Konfigurace sítě je statická [2].

V rámci algoritmu je využíváno několika rolí, které jsou přiřazeny uzlům v systému. Role určují chování uzlu. Každý uzel může v danou chvíli mít přiřazeno vícero rolí. Role jsou následující:

[2] Existují varianty podporující dynamické změnu uzlů – např. Elastic Paxos [11].

- *Proposer* – uzel, který obsluhuje požadavky (např. nahrání souboru, aktualizace dat) přijaté od klientů. Uzel dále tyto požadavky navrhne uzlům s rolí Acceptor.
- *Acceptor* (Voter) – uzel, který se podílí na schválení nebo zamítnutí návrhů na změnu stavu od Proposer uzlů.
- *Learner* – uzel který provádí změnu stavu, která byla schválena uzly s rolí Acceptor.

Aby došlo ke konsenzu, je nutné, aby došlo ke schválení jednoho návrhu od uzlů Proposer nadpoloviční většinou uzlů typu Acceptor. V následujícím popisu bude uvažováno, že všechny uzly v síti mají přiřazeny role Proposer, Acceptor a Learner zároveň, jelikož se jedná o nejběžnější užití. Z toho vyplývá, že pro schválení návrhu je nutný souhlas nadpoloviční většiny uzlů v dané síti. Správnost konsenzu vyplývá z následujících podmínek [33]:

- Pouze změna, která byla navržena může být schválena.
- Pouze jediná změna je schválena.
- Uzel nikdy neaplikuje navrženou hodnotu, dokud není schválena.

Základní Paxos

Základní varianta algoritmu Paxos je výchozí pro všechny ostatní varianty, které jsou od ní odvozeny [30]. Algoritmus pracuje s číslováním návrhů, každý nový návrh musí mít větší číslo než předchozí. Zároveň uzly musí vybírat čísla návrhů z navzájem disjunktních množin, tak aby nedocházelo k tomu, že se v síti objeví dva různé návrhy se stejným číslem. To může být například zajištěno pomocí postupně zvyšující se ho čísla pro návrhy v kombinaci s číslem uzlu (např. $X.Y$ – kde X je číslo o jedna vyšší, než je číslo předchozího návrhu a Y je číslo uzlu), čísla uzlů musí být samozřejmě unikátní. Samotné schválení změny, která se má aplikovat probíhá ve dvou následujících fázích [33]:

1. (a) Proposer uzel si vybere číslo n a odešle zprávu s tímto číslem uzlům typu Acceptor. Tuto zprávu označujeme jako prepare request. Aby mohlo dojít k potencionálnímu schválení, musí být zpráva odeslána alespoň nadpoloviční většině uzlů Acceptor.
   (b) Uzel Acceptor přijímá zprávu prepare request. Pokud tato zpráva obsahuje číslo n, které je menší, než číslo předchozí zprávy, odmítá požadavek. V rámci odmítnutí pošle zpět uzlu nejvyšší číslo zprávy, které zatím obdržel a číslo poslední schválené změny, díky

kterému si Proposer uzel je schopen doplnit chybějící schválené změny. V případě, že číslo zprávy n je vyšší než doposud známé číslo předchozích zpráv, odpoví uzel zpět zprávou promise, čímž uzlu slibuje, že neodpoví žádnému jinému uzlu zprávou promise na žádost (prepare request) s číslem, které by bylo nižší. Zároveň do zprávy přiloží návrh na změnu stavu, který není schválen a dostal jej v předchozích zprávách typu prepare request. Pokud žádný návrh předtím nepřijal, nepřikládá ho do zprávy.

2. (a) V případě, že uzel Proposer získá od nadpoloviční většiny Acceptor uzlů odpověď typu promise, odesílá jim zprávu accept request a vkládá do ní změnu stavu, kterou obdržel v odpovědi promise. Pokud se v žádné z odpovědí návrh nenacházel, vloží do zprávy svůj návrh. Zprávu označí opět číslem n, tedy číslem, který označil i původní prepare request.

   (b) Pokud Acceptor přijme accept request s číslem n, pak ho schválí pouze v případě, že mezitím neodeslal jinému uzlu promise na zprávu s číslem m větším než n, tedy že neslíbil, že neschválí žádný jiný návrh, který bude obsahovat menší číslo, než je nové nejvyšší číslo m.

Uzly změnu aplikují v případě, že byla schválena nadpoloviční většinou uzlů typu Acceptor. Realizace může být například provedena tak, že kdykoliv Acceptor schválí změnu, pošle ji všem Learner uzlům. Ti ji aplikují, jakmile dostanou schválení od většiny uzlů Acceptor.

U výše popsaného algoritmu není zaručeno, že skončí. Může dojít k situaci, kdy dva uzly navrhují změnu a neustále se předbíhají, ale žádný z nich nedokončí úspěšně obě fáze. Tento problém může být eliminován přidáním role vůdce (leader) mezi Proposer uzly. Vůdce je poté jediný uzel, který může navrhnout změnu. Algoritmus Paxos sám o sobě může být využit k volbě vůdce. Vůdcem se stane uzel, jehož návrh bude schválen. Aby nedocházelo k předbíhání uvedenému výše, je nutné dostatečně randomizovat dobu, za kterou uzel zahájí navržení změny, pokud mu předchozí návrh nebyl schválen.

Multi-Paxos

Varianta Multi-Paxos je určená pro dosažení konsenzu nad sérií návrhů. Oproti základní variantě výrazně redukuje množství zpráv, které musí být přeneseny v průběhu běhu algoritmu. Základem pro tuto variantu je předpoklad, že se vůdce nemusí pro každé schválení návrhu měnit, ale může po určitou dobu zůstat stejný [30].

Začátek běhu algoritmu je stejný jako u základní varianty – pro schválení návrhu je nutné, aby prošel oběma fázemi a schválila ho nadpoloviční většina

Acceptor uzlů. Ten, kdo vznesl schválený návrh, se stává vůdcem a ten jediný může vznášet další návrhy. Vůdcem zůstává tak dlouho, dokud jsou platné podmínky, za nichž byl zvolen, tj. vůdce je sám aktivní (nedošlo k výpadku) a má přístup alespoň k většině uzlů typu Acceptor. Pokud jsou tyto podmínky splněny je možné, po úspěšném zvolení vůdce vypustit první fázi, tedy zprávy prepare request a odpovědi na ně – promise. Vůdce rovnou posílá zprávy typu accept request. Tím se snižuje počet přenesených zpráv, nutných ke schválení nové změny, zároveň se tím i snižuje čas, za který je nová změna schválena.

Kromě čísla zprávy se přidává i číslo instance. Číslo instance se na počátku rovná číslu zprávy, poté se s každým návrhem zvyšuje, zatímco číslo zprávy zůstává stále stejné, dokud je vůdce platný.

Ostatní varianty

Existuje mnoho dalších variant a optimalizací základní varianty algoritmu Paxos. Příkladem je Fast Paxos, který umožňuje v určitých případech klientům navrhovat změny přímo uzlům Acceptor [29]. Tím se opět snižuje počet zpráv, který je potřebný ke schválení změny. Další variantou je Byzantine Paxos, který naopak přidává nový typ zprávy – verify, díky které dokáže detekovat a překonat Byzantské chyby [14].

2.6.2 Algoritmus Raft

Algoritmus Raft navrhl Diego Ongaro v roce 2014 jako alternativu k variantám algoritmu Paxos. Cílem bylo vytvořit konsenzus algoritmus, který bude snadněji pochopitelný a lépe použitelný pro reálné systémy [39].

Raft pracuje nad stejnými požadavky jako algoritmus Paxos (viz kapitola 2.6.1). Základní varianta opět nepočítá s byzantskými chybami, ale existuje rozšíření algoritmu, které tyto chyby zvládá, v práci je popsána pouze základní varianta [16]. I podmínka počtu uzlů je stejná, tedy je nutno n uzlů a jejich počet lze získat pomocí stejné rovnice $n = 2F + 1$, kde n je počet hledaných uzlů a F je počet uzlů v chybovém (failure) stavu. Algoritmus dekomponuje problém dosažení konsenzu na dvě části [39]:

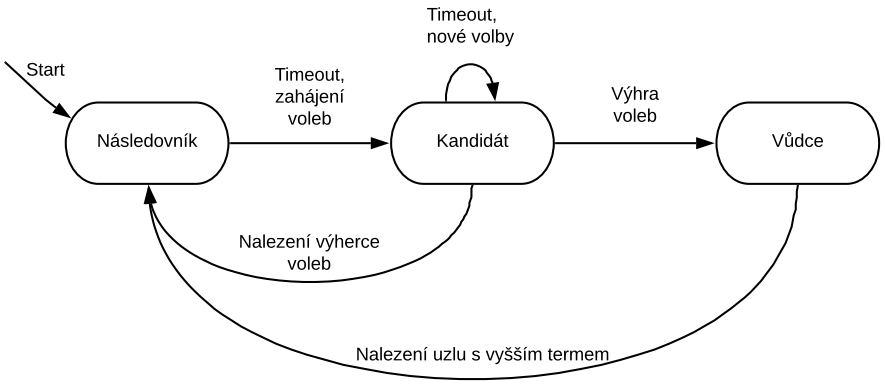
1. Běžný chod – replikace logu.
2. Změna vůdce.

Na počátku běhu algoritmu je nutné zvolit vůdce, ten pak řídí replikaci logu a celý chod algoritmu, dokud jsou platné podmínky, za jakých byl zvolen.

K volbě i k běžnému chodu je využito tří rolí, každý uzel v danou chvíli musí mít právě jednu roli. Role jsou následující [39]:

- Vůdce (leader) – pouze jediný uzel, který řídí chod.
- Kandidát (candidate) – dočasná role, než je zvolen vůdce.
- Následovník (follower) – podřízený vůdci, přijímá příkazy od vůdce.

Přechody mezi jednotlivými rolemi jsou zobrazeny pomocí stavového diagramu na obrázku 2.4. Na začátku se uzel dostává do role následovník. Vůdce v pravidelných časových intervalech posílá svým následovníkům zprávy, že je stále aktivní a platný vůdce. V případě, že následovníkovi nepřijde delší dobu tato zpráva – vyprší stanovený časový limit (timeout), přesouvá se následovník do stavu kandidát a zahajuje nové volby. V případě, že zjistí, že volby vyhrál jiný uzel, vrací se do stavu následovník. Pokud ve stanoveném časovém limitu nevyhraje volby on, ani nedostane informace od jiného uzlu, že je vyhrál, zahájí nové volby. Pokud volby vyhraje, postupuje do role vůdce. Z té se může vrátit pouze do role následovník, a to pouze v případě, že dostane zprávu od jiného vůdce, který má vyšší číslo termu [39].



Obrázek 2.4: Diagram stavů pro algoritmus Raft [26]

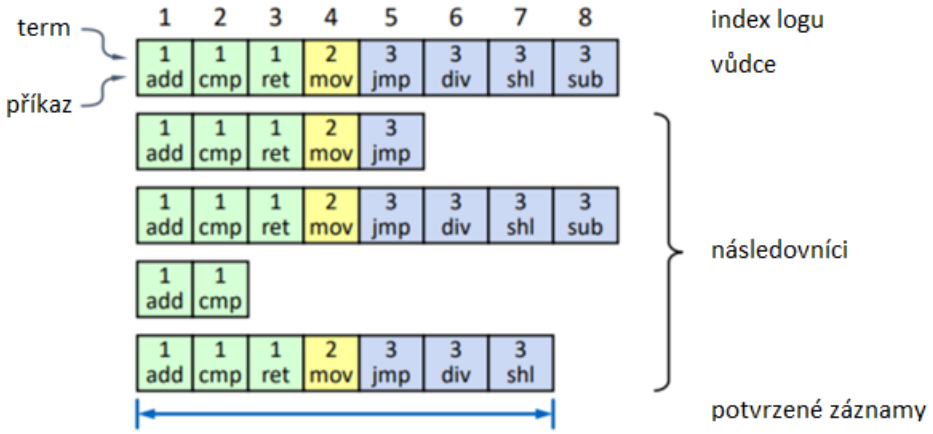
Term je číslo volebního období. Pokud se zahajují nové volby, term se zvyšuje. V každém termu může být zvolen maximálně jeden vůdce. Když uzel zahájí volby a přesune se do stavu kandidát, zároveň zvýší i číslo termu a dá hlas sám sobě. Následně pošle žádost o hlas (*RequestVote*) všem ostatním uzlům a čeká na jejich odpověď. Uzly, kteří mají nižší číslo termu, si ho lokálně zvýší na stejnou hodnotu, která se nachází v přijaté zprávě o hlas, pokud nejsou ve stavu následovník, přejdou do něj. Pokud uzel má stejně nebo více potvrzených zpráv v logu, vrátí kladnou odpověď – hlasují pro něj, v opačném případě žádost zamítnou. Pokud hlasují pro kandidáta, potvrzují

tím i, že nebudou hlasovat pro žádný jiný uzel ve volbách s tímto nebo nižším číslem termu. V případě, že uzel, který přijal žádost o hlas má vyšší číslo termu, než je číslo termu nacházející se v žádosti, vrátí negativní odpověď [39].

Kandidát vyhrává volby, jakmile získá nadpoloviční většinu hlasů, nečeká na odpovědi od ostatních uzlů, jelikož jeho vítězství už neovlivní. Pokud by volby nevyhrál a ani by se ve stanoveném časovém limitu neozval jiný uzel, že vyhrál volby, zvýší číslo termu a volby se opakují. Volby nemusí vyhrát žádný uzel, pokud ani jeden nezíská nadpoloviční většinu hlasů, to se například může stát v případě sudého počtu uzlů v síti, kdy se hlasy rozdělí přesně na polovinu mezi dva kandidáty. O výhře voleb a přesunu do role vůdce, uzel informuje ostatní pomocí periodické zprávy – prázdné zprávy *AppendEntry* (heartbeat) [39].

Zvolený vůdce provádí replikaci svého logu na všechny ostatní uzly. Od klientů přijímá nové příkazy, ty si uloží do svého logu. Vůdce je zároveň jediný uzel, který příkazy od klientů může přijímat, následovníci musí klienty přesměrovat na vůdce. Pokud je uzel kandidát, odmítne požadavek a klient musí vyčkat, než proběhne volba nového vůdce. Vůdce následně operace posílá svým následovníkům pomocí zpráv *AppendEntry*. Tato zpráva kromě samotných operací obsahuje i informace o předchozím záznamu – na jakém indexu logu se nachází a v jakém termu byl přijat, dále index záznamu, který je poslední potvrzený. Informace o předchozích záznamech se používají k synchronizaci logu, pokud došlo k nějakému výpadku a logy jsou rozdílné. Index posledního potvrzeného záznamu slouží pro aplikaci příkazů z logu. Stavové automaty aplikují pouze příkazy, které jsou v potvrzených záznamech. Pokud přijatý index je vyšší než lokální, aplikují se všechny záznamy z logu až po tento a hodnota indexu se aktualizuje. Následovníci odpovídají na zprávy *AppendEntry* číslem – indexem posledního záznamu, který mají. Vůdce si tyto indexy ukládá zvlášť pro každý uzel, tím ví, komu jaké záznamy scházejí. Pokud vůdce zjistí, že více jak polovina uzlů má záznamy až do indexu i, uloží hodnotu i jako poslední potvrzený index, až po potvrzení záznamu odpovídá klientovi, že jeho příkaz byl vykonán. Při další poslání zprávy *AppendEntry* si i následovníci nastaví jako poslední potvrzený záznam na index i [39]. Logy a jejich stav je ilustrován na obrázku 2.5.

Raft zajišťuje bezpečnost distribuovaného systému tím, že zajistí vykonání potvrzených záznamů na všech uzlech ve stejném pořadí. To je zajištěno díky vůdci, který může být v danou chvíli pouze jeden platný. Živost algoritmu je docílena pomocí volby náhodného časového limitu pro timeout voleb z intervalu <T, 2T>, kde T je minimální doba pro timeout voleb definovaná uživatelem systému [39].



Obrázek 2.5: Logy jednotlivých uzlů v algoritmu Raft [27]

2.6.3 Porovnání

Jak dokládá autor algoritmu Raft ve své doktorské práci, je Raft snadnější pro pochopení i pro implementaci oproti algoritmu Paxos [39]. Toho je docíleno dekompozicí na dva problémy – volba vůdce, replikace logu a dále tím, že se uzly vždy nachází pouze v jednom stavu. Je také podrobněji popsán, než algoritmus Paxos, zejména části, které se zabývají zotavením z výpadků sítě. U algoritmu Paxos je toto detailně popsáno až v dalších vědeckých článcích [26].

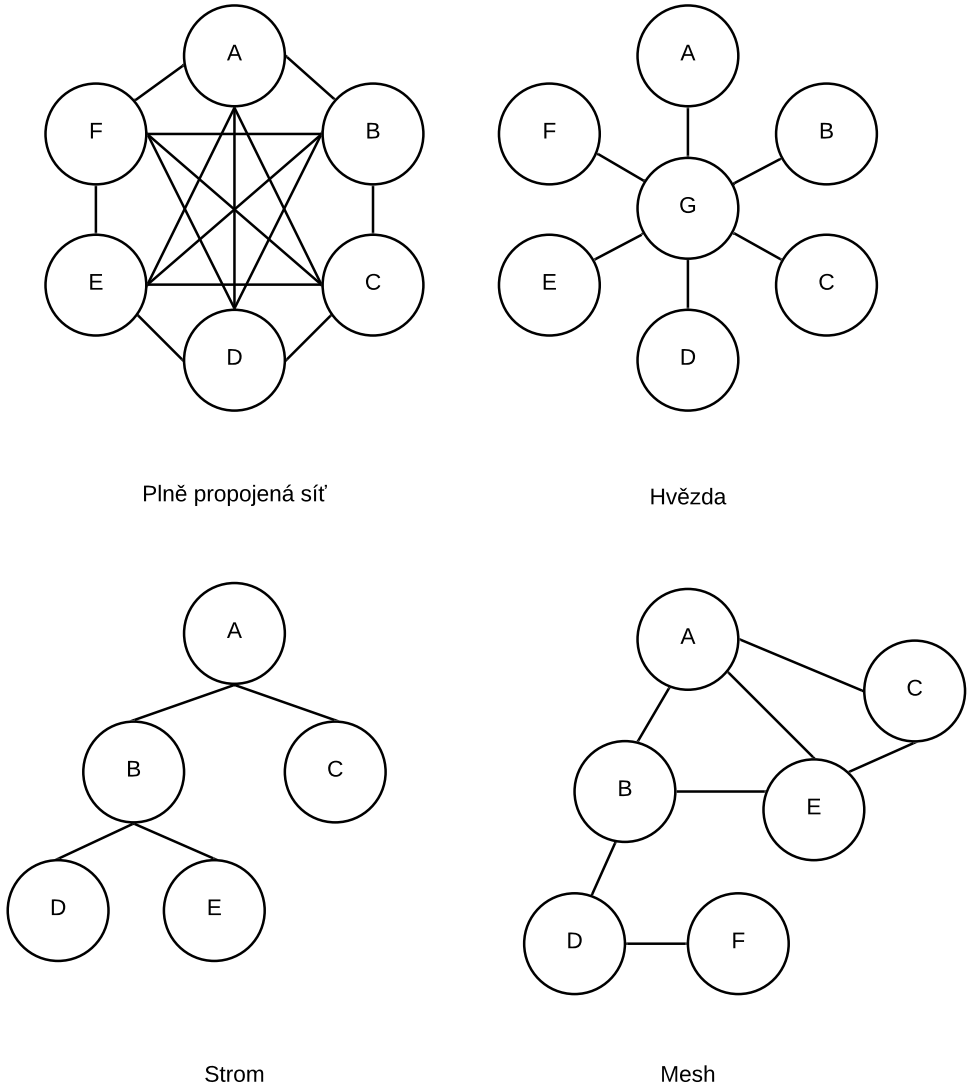
Oba jsou to pesimistické algoritmy a potřebují pro své fungování – nalezení konsenzu minimálně nadpoloviční většinu uzlů v korektním stavu [26]. Tudíž jejich základní verzi nelze použít pro splnění požadavků na cílený framework, ale lze použít jejich myšlenky pro konstrukci algoritmu, který bude replikaci zajišťovat optimisticky.

2.7 Analýza stavů distribuovaných systémů

V distribuovaných systémech může dojít k několik různým stavům z hlediska selhání uzlů, nebo kanálů propojujících jednotlivé uzly. V případě selhání uzlů můžeme hledět na jejich počet – kolik uzlů v dané chvíli je v chybovém stavu. Pokud je jich např. polovina, pro zmíněné algoritmy Raft a Paxos je to kritické, nedokážou dosáhnout konsenzu [47].

Z hlediska selhání kanálů je možné provést podrobnější klasifikaci. Pokud při výpadku kanálu dojde k rozpadu na dvě části, hovoříme o single-partitioning. Pokud dojde k rozdělení na více částí, jedná se multiple-partitioning [41].

Můžeme se též zajímat o to, kolik v jednotlivých osamostatněných částech – ostrovech, zůstalo uzlů. Pokud například dojde k rozpadu sítě na tři ostrovy, ale jeden ostrov bude stále obsahovat více jak polovinu uzlů, algoritmy typu Raft a Paxos budou v tomto ostrovy stále schopny pracovat. K jakým případům může dojít záleží velice na topologii sítě – ukázkové topologie jsou vyobrazeny na obrázku 2.6. Vybrané topologie dostačují k popisu všech stavů rozpadu sítě, ke kterým dochází v distribuovaných systémech.



Obrázek 2.6: Ukázkové topologie sítí

V případě plně propojené sítě je potřeba selhání n - 1 cest vedoucích od jednoho uzlu, aby došlo k jeho izolování. Pokud chceme rozdělit část na rovnoměrné poloviny, bude potřeba přerušit $p^2/4$ cest. Naopak odstraněním pouze jedné cesty nedojde v sítí k významným změnám, pouze se musí zvolit náhradní cesta přes sousední uzel. U ostatních zmíněných topologií stačí odstranění jediné cesty k izolování uzlu. V případě hvězdy dojde přerušením

cesty vždy k izolaci jednoho uzlu. V topologiích strom a mesh záleží na konkrétní přerušené cestě. V případě stromu může dojít k odpojení i více uzlů, které se ocitnou v samostatném ostrově (cesta mezi A a B). U topologie mesh může dojít ke všem již zmíněným stavům. Přerušenou cestu mezi A a E lze nahradit pomocí A-C-E, případně A-B-E. Přerušení cesty mezi D a F by znamenalo izolaci uzlu F. Přerušení cesty mezi B a D by vyústilo k rozpadu na ostrovy A-B-C-E, který by měl stále nadpoloviční většinu uzlů a D-F [28].

Je vidět, že plně propojené sítě jsou odolnější vůči selhání jednotlivých cest, ale jejich konstrukce je nákladnější oproti ostatním typům topologií, jelikož vyžadují více cest. Dále se v práci budeme zabývat topologiemi typu mesh, jelikož jsou dostatečně obecné a může v nich docházet ke stejným stavů jako v ostatních [28].

Kapitola 3

Návrh

Kapitola návrh se zprvu zabývá přizpůsobením algoritmu RAFT pro účely optimistické replikace a následně i celkovou architekturou synchronizačního frameworku. Následuje popis jednotlivých typů konfliktů, jejich detekce a řešení pomocí verzování. Obsažen je i návrh databázového modelu, či komunikace mezi třídami zajišťující manipulaci se synchronizovanými entitami.

3.1 Přizpůsobení algoritmu RAFT pro účely optimistické replikace

V předchozích kapitolách byl představen algoritmus RAFT, jenž slouží k pesimistické replikaci dat a volbě vůdce [39]. Cílem práce je však optimistický replikační algoritmus, tudíž je nutné RAFT upravit. Změny jsou typicky relaxací původních podmínek algoritmu, které zaručovaly jeho pesimistický přístup k replikaci dat.

Komunikační model zůstane zachován, uzly komunikují pouze s vůdcem, mezi sebou pouze v případě voleb [39]. Z tohoto důvodu je nutné umožnit, aby v případě, že se síť rozpadne na více částí, byl v každé části zvolen nový vůdce, který bude zajišťovat synchronizaci mezi uzly (splnění funkčního požadavku číslo 6). Toho bude docíleno relaxací podmínky, že pro výhru voleb je nutná polovina hlasů všech uzlů v síti. Z toho vyplývá, že i pro potvrzení záznamu nebude potřebná polovina všech hlasů.

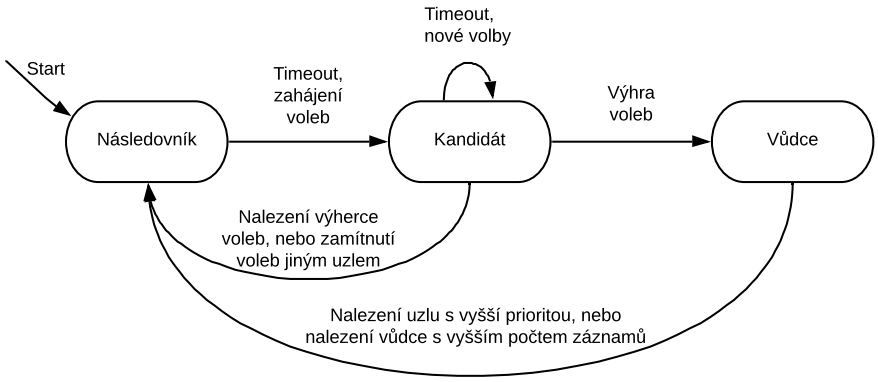
Pro dosažení optimistické replikace, si bude každý uzel data ukládat nejdříve u sebe a až poté je propagovat dále. K trvalému uložení nepotřebuje souhlas jiných uzlů, ani nemusí být v danou chvíli zvolen v síti vůdce. Pro dosažení tohoto chování je nutné upravit stávající log algoritmu RAFT, který je

společný pro všechny uzly, na tzv. multi-log, jenž obsahuje log pro každý uzel [44].

Z výše uvedených změn je zřejmé, že původní log ztrácí svou schopnost zabraňovat konfliktům, či je případně řešit zahození nepotvrzených změn. Konflikty mezi entitami je nutné detekovat a řešit na jiné úrovni. Detekce a řešení konfliktů bude zajištěno pomocí verzování jednotlivých entit. Algoritmus bude ale stále zajišťovat korektní replikaci multi-logů mezi uzly v síti.

3.1.1 Změny ve volbě vůdce

Algoritmus bude využívat stejné tři role, jako algoritmus RAFT [39]. Rozdíl bude v podmínkách pro akce, které budou vyvolávat přechody mezi stavy. Diagram stavů algoritmu i s přechody se nachází na obrázku 3.1. Ze stavu následovník se do stavu kandidát dostane stejným způsobem – do daného časového limitu nepřijde heartbeat zpráva od vůdce.



Obrázek 3.1: Diagram stavů pro modifikovaný algoritmus RAFT

Při přechodu do stavu kandidát, si uzel uloží aktuální stav multi-logu který má. Vyhlásí nové volby, a kromě termu posílá v žádostech o hlas i uložený stav multi-logu. Ostatní uzly mohou odpovědět nejen udělením či neudělením hlasu, ale i zamítnutím vyhlášených voleb. V případě obdržení odpovědi s tímto příznakem kandidát okamžitě zastavuje vyhlášené volby a přesouvá se do stavu následovník. Z tohoto důvodu se uzel nepřesouvá do stavu vůdce ihned poté, co obdrží potřebnou polovinu hlasů, ale čeká až mu odpoví všechny dostupné uzly. Výhra nebo prohra voleb se nepočítá na základě počtu získaných hlasů z celkového počtu uzlů, ale z počtu dostupných uzlů v dané síti – tím je zajištěno, že v případě rozpadu sítě bude zvolen v každé části nový vůdce. Na žádost o hlas reagují ostatní uzly následovně – v případě, že kandidát není validní, zamítni jeho volby. V opačném případě

uděl hlas, pokud je term žádosti větší, resp. neuděl hlas, pokud je term žádosti menší. Zda je kandidát validní se posuzuje následovně:

- Kandidát má vyšší prioritu – je validní.
- Kandidát má nižší prioritu – není validní.
- Kandidát má stejnou prioritu:
  - Kandidát má více nebo stejně záznamů v multi-logu – je validní.
  - Kandidát má méně záznamů v multi-logu – není validní.

Priorita je manuálně přiřazená hodnota uzlu. V případě porovnávání multi-logů se využívá přijatý multi-log kandidáta ze zprávy s aktuálním multi-logem daného uzlu. To neplatí v případě, že uzel, kterému přišla žádost o hlas, je též ve stavu kandidát. Tento uzel použije pro porovnání přijatý multi-log kandidáta z žádosti o hlas, se svým stavem multi-logu, který si uložil před začátkem voleb. Tím dojde k eliminaci, že by se uzly neustále předbíhaly v tom, kdo z nich má více záznamů v multi-logu, jelikož oproti algoritmu RAFT, lze záznamy do multi-logu přidávat, i když není zvolen v síti vůdce [39]. V případě, že kandidát udělil hlas jinému kandidátovi (jiný kandidát byl validní a má vyšší číslo termu), zastaví své volby a přechází do stavu následovník, jelikož kvůli nižšímu termu oproti druhému kandidátu jsou jeho volby již neplatné.

Jestliže kandidát vyhraje volby a stane se vůdcem, existuje několik případů, ve kterých přejde do stavu následovník. Prvním případem je přijmutí heartbeat zprávy od jiného vůdce [39]. Pokud je jiný vůdce z jeho pohledu platný, tj. má vyšší nebo stejnou prioritu, ale má více záznamů v multi-logu, přejde do stavu následovník. V opačném případě na zprávu odpovídá s tím, že do odpovědi uvede jako vůdce sebe. Druhým případem, ve kterém vůdce přechází do stavu kandidát je, když na svou heartbeat zprávu obdrží odpověď, ve které není uveden jako vůdce. Odpovědět tímto způsobem může jiný vůdce, jak je popsáno výše, nebo i uzly, které jsou ve stavu následovník nebo kandidát. V obou stavech uzel kontroluje při přijetí heartbeat zprávy, zda vůdce, který ji poslal, je platný. Zde se pohlíží na platnost vůdce pouze z výše priority. Pokud je priorita vůdce stejná nebo vyšší, je považován za platného. Následně odpovídá vůdci a přikládá do odpovědi jeho unikátní ID a uznává ho tedy jako platného. V případě, že uzel detekoval, že vůdce není platný, tak do odpovědi uvede aktuálního vůdce, kterého má uloženého (platí pouze pro stav následovník), případně neuvede žádného vůdce, pokud ještě nebyl zvolen (platí pro oba stavy – následovník i kandidát). Tímto mechanismem je docíleno, že v síti nebude po výměně heartbeat zpráv více platných vůdců.

Term při zkoumání platnosti vůdců nehraje již žádnou roli, jelikož není navázán na platnost záznamů v logu (všechny jsou platné). Platnost se tedy zkoumá pouze z pohledu priority uzlů a počtu záznamů v multi-logu.

3.1.2 Transformace logu na multi-log

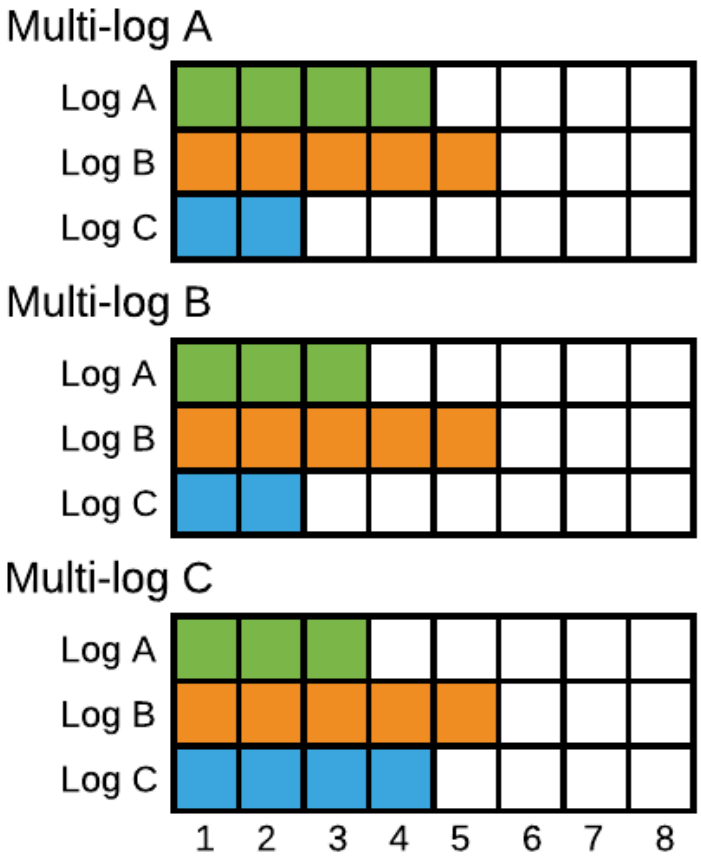
Další změnou v algoritmu RAFT je transformace log na multi-log, jelikož funkční požadavek číslo 4 požaduje uložení dat trvale, i když nebude dostupné spojení s ostatními uzly. Nesmí tedy docházet k tomu, že by záznamy v logu byly přepsány jinými, tak jako se to děje u nepotvrzených záznamů v rámci původního algoritmu RAFT [39].

Nově použitý multi-log bude mít následující strukturu – pro každý uzel v síti se bude držet jeden klasický log (viz obrázek 3.2). Při vytváření nového záznamu, musí uzel zapisovat pouze do logu se svým ID. Naopak při přijmutí již vytvořených záznamů od jiného uzlu, se tyto záznamy musí vkládat pouze do logů s ID ostatních uzlů, ne do vlastního logu.

Obrázek 3.2 uvádí příklad, kdy se synchronizují mezi sebou tři uzly pojmenované A, B, C. V každém multi-logu je plně replikován log B, zatím co uzel A má ve svém logu nový záznam na indexu 4, který nereplikoval do multi-logů ostatních uzlů. Stejně tak C má nové nereplikované záznamy na indexech 3 a 4.

Jelikož každý uzel může vkládat záznamy do svého logu, i když není vůdce, musí být zajištěna propagace těchto záznamů ostatním uzlům (v případě RAFT posílá pouze vůdce záznamy svým následovníkům) [39]. Uzly ve stavu následovník budou kontrolovat, zda vůdci neschází některé záznamy z multi-logu. V případě, že vůdci budou záznamy chybět, začne mu je následovník posílat zprávami *AppendEntry*. Kontrolu bude provádět ve dvou případech. Prvním je uložení záznamu do svého logu. V tu chvíli je zřejmé, že vůdci tento záznam musí chybět, a tudíž musí být poslán. Druhým případem je změna vůdce. Jakmile dojde ke změně vůdce – ať už ze stávajícího na jiného nebo z žádného vůdce na nově zvoleného, uzel provede kontrolu stavu multi-logu vůdce oproti svému. Došlo-li totiž k rozpadu sítě a nyní k následnému propojení s tím, že nový vůdce je ze sítě, která byla odpojena, budou mu scházet záznamy, které uzel vytvořil v době, kdy byla síť rozpadlá. Následovník získává informace u stavu multi-logu vůdce z heartbeat zpráv, které mu jsou vůdcem zasílány.

Při posílání záznamů z multi-logu je nutné zajistit jejich správné pořadí, aby byla došlo na každém uzlu k replikování správného stavu. Jednou z možností, jak docílit správného pořadí, je globální řazení záznamů. Z předchozích kapitol je známo, že zajistit synchronizovaný čas v síti, kde může docházet k výpadkům je nemožné [47]. Ze stejného důvodu nelze uvažovat nad čítačem či mechanismem stejným z algoritmu RAFT – uložení záznamu až po potvrzení polovinou uzlů [39]. Tudíž nebude cíleno na zachování přesného globálního pořadí záznamů, ale na pořadím takovém, aby po odeslání všech záznamů, došlo k replikaci správného stavu na všech uzlech. Tohoto způsobu řazení bude docíleno použitím indexů logů a času jejich vytvoření v rámci daného uzlu,

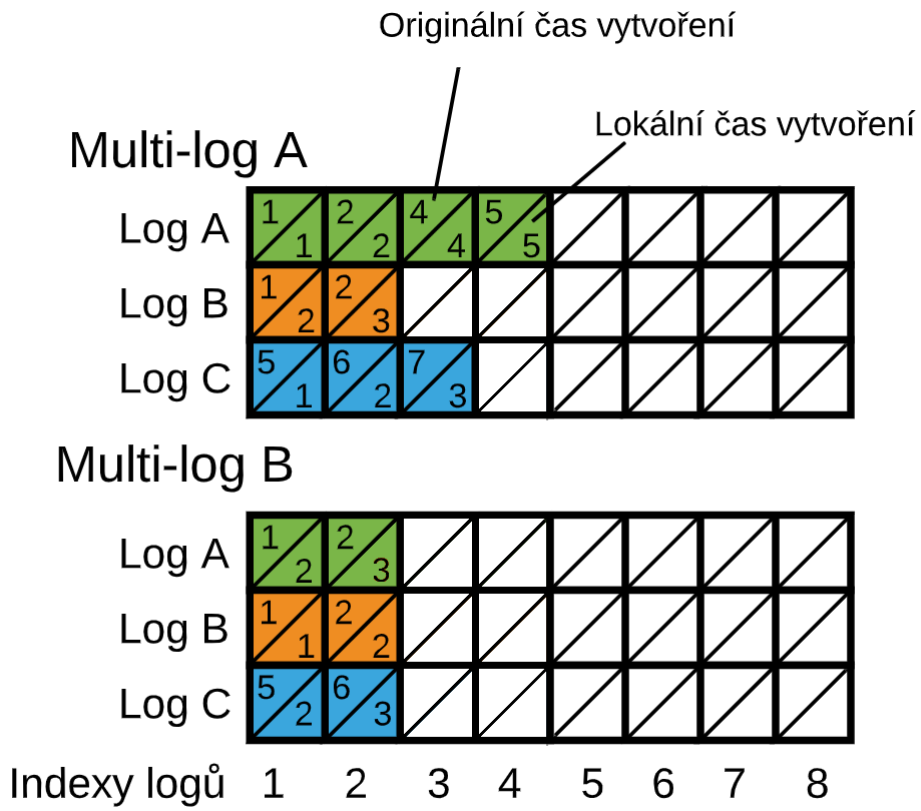


Obrázek 3.2: Příklad multi-logů pro síť s uzly A, B, C

spolu s využitím verzování entit. Pokud tedy uzel bude posílat záznamy z multi-logu jinému uzlu, nejprve nalezne všechny záznamy, ze všech jednotlivých logů, které druhý uzel nemá. Tyto záznamy pozná díky znalosti stavu multi-logu druhého uzlu. Stav je reprezentován dvojicí hodnot – klíče, jenž je název logu a hodnotou, která reprezentuje, na jakém indexu se nachází poslední záznam. Po nalezení chybějících záznamů, seřadí uzel tyto záznamy podle data vytvoření záznamu v rámci daného uzlu (tedy založení vlastního záznamu, nebo čas přijetí od jiného uzlu).

Příkladem mohou být uzly A a B, kde uzel A je vůdcem a uzel B je následníkem. Stav jejich multi-logů je ilustrován na obrázku 3.3. Vůdce A zná aktuální stav svého následovníka B. Tedy ví, že mu schází záznamy v logu A na indexech 4 a 5 a záznam z logu C na indexu 3.

Při propojení uzlů trvá synchronizace záznamů 1 časovou jednotku. Pokud uzel vytváří záznam ve svém logu, je originální i lokální čas vytvoření stejný. Pro první dva záznamy v logu A je vidět, že uzel A je vytvořil a uzel B je přijal se zpožděním jedné časové jednotky. Zároveň docházelo i k synchronizaci



Obrázek 3.3: Příklad synchronizace multi-logů mezi uzly A a B

záznamů v logu C. Jelikož originální čas vytvoření záznamů v logu C je vyšší než lokální, znamená to, že uzel C má oproti uzlům A a B posunutý čas napřed – konkrétně o 5 časových jednotek (4 jednotky jsou rozdíl mezi časy vytvoření + 1 jednotka za čas potřebný k synchronizaci, tedy než záznam dorazil k uzlu A). Následně přijaté záznamy z logu C, vůdce A přeposlal uzlu B.

Nyní vůdce A zjišťuje pořadí chybějících záznamů pro následovníka B. Kdyby byl brán v potaz originální čas vytvoření, bylo by pořadí záznamů následující: A3, A4, C3. Toto pořadí není ale správné, jelikož záznam C3 byl přijat ještě před vytvořením záznamů A3 a A4. V případě použití řazení podle lokálních časů vytvoření je již pořadí záznamů správné: C3, A3, A4.

3.1.3 Shrnutí změn

Pro účely frameworku pro optimistickou replikaci dat, musejí být provedeny v algoritmu RAFT změny, které byly popsány a odůvodněny výše [39]. Zde

následuje stručný přehled změn v algoritmu a v jeho chování:

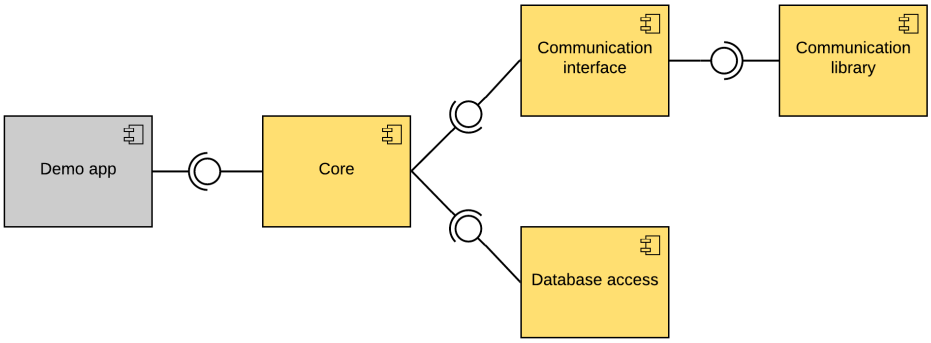
- Umožnění volby vůdce v každé části síti, pokud dojde k jejímu rozpadu – stačí získat polovinu hlasů od dostupných uzlů.
- Zneplatnění voleb jiným uzlem – zabránění, aby se stal vůdcem uzel s nízkou prioritou či s malým počtem záznamů.
- Rozhodování o udělení hlasu kandidátu nejprve podle priority, následně až dle počtu záznamů.
- Využití termu pouze pro účely voleb – aby uzel nehlasoval pro více kandidátů v jednom kolem.
- Trvalé uložení záznamů i bez jejich potvrzení ostatními uzly.
- Trvalé uložení záznamů na jakémkoliv uzlu, nejenom na vůdci.
- Použití multi-logu místo logu.
- Řazení záznamů dle data přijetí při doplňování multi-logu jiného uzlu.
- Následovník posílá zprávy *AppendEntry* vůdci, v případě, že má pro některý log více záznamů než vůdce.

3.2 Architektura synchronizačního frameworku

Synchronizační framework bude rozdělen do několika komponent, které budou mezi sebou spolupracovat na zajištění celkové synchronizace mezi jednotlivými uzly v daném clusteru. Náhled celé architektury je na obrázku 3.4. Cílem architektury je od sebe oddělit komunikační vrstvu, synchronizační logiku a přístup do databáze. Jednotlivé komponenty jsou navrženy tak, aby bylo možné každou z nich implementovat jako dynamicky linkovanou knihovnu (DLL), jež nabízí platforma .NET [34]. DLL jsou snadno přenositelné a lehce nahraditelné za jiné [36].

Jádrem celého synchronizačního frameworku bude komponenta Core. Komponenta Core bude obsahovat mechanismy pro volbu vůdce a pro zajištění synchronizace s ostatními uzly. K přístupu do databáze bude využívat komponentu Database access. Tato komponenta bude definovat databázové entity s jejich mapováním, třídy pro manipulaci s nimi a vytvářet spojení do databáze.

Dále bude Core komponenta využívat komunikační knihovnu pro odesílání a přijímání zpráv od ostatních uzlů. Komunikační knihovna bude mít vlastní definované rozhraní pro komunikaci s ostatními uzly, i datové kontrakty, které



Obrázek 3.4: Náhled architektury synchronizačního frameworku

bude přenášet. Zároveň bude knihovna detekovat výpadky (nedostupnost) jednotlivých uzlů a případně i přeposílat zprávy skrze dostupné uzly, v případě, že nedošlo k výpadku uzlu, ale pouze k výpadku přímého spojení mezi uzly. Využití knihovny bude realizováno přes definované rozhraní (komponenta Communication interface).

Samotná Core komponenta bude vystavovat rozhraní pro použití frameworku. Toto rozhraní bude použito demo aplikací, která bude sloužit k testování a zároveň k prezentaci možného použití frameworku.

3.3 Detekce konfliktů pomocí verzování

Po úpravě algoritmu RAFT do verze, kdy umožňuje optimistickou replikaci dat, nastává riziko, že dojde ke konfliktu mezi provedenými změnami. Dojít k němu může, jelikož je umožněno ukládat na každém uzlu data, bez jakéhokoliv potvrzení ostatními uzly. Tedy dva rozdílné uzly mohou upravit stejná data jiným způsobem. Tyto konflikty je nutné detekovat, k tomu bude využito verzování entit, které bude framework synchronizovat.

Při každé změně entity bude vytvořena její nová verze, která bude odkazovat na původní entitu, jako na svého rodiče. Kromě odkazu na rodiče bude verze entity obsahovat i další metadata. Obsahovat bude i časový údaj, od kdy, do kdy verze platí. Samozřejmostí je i číslo dané verze, to bude růst s každou novou verzí. Čísla verzí sama o sobě nebudou unikátní, unikátní budou až v kombinaci s unikátním ID uzlu, který entitu s daným číslem verze vytvořil. Proto dále v textu budou využívány zápisy verzí např. A1, B2, kde písmeno bude unikátní ID uzlu, nasledované číslem verze na daném uzlu. Tedy verze A1 je první verze vytvořená uzlem A, zatímco B2 je již druhá verze, kterou uzel B vytvořil. V souvislosti s konflikty, zde budou dva příznaky, jeden pro označení, že entita je v konfliktu s jinou entitou (i s vícero entitami),

druhý bude označovat, že tento konflikt nelze automaticky vyřešit a je nutný manuální zásah uživatele.

Pokud by bylo umožněno mazání verzí, mohlo by dojít k tomu, že jeden uzel smaže všechny verze, zatímco jiný novou verzi vytvoří. Pak by došlo k tomu, že by se na všechny uzly replikovaly znovu všechny verze (pokud by předtím byly smazány) a přidala se k nim nově vytvořená verze. Tudíž by se ztratila informace o smazání, z tohoto důvodu bude fyzické mazání verzí zakázáno. Mazat entitu bude možné pouze pomocí tzv. logického smazání – nastavení příznaku, že je entita smazána a bude se tedy jednat o vytvoření nové verze [48]. Příznak, zda je verze lokální, bude sloužit k rozlišení, která z konfliktních verzí se má zobrazovat uživatelům. Posledním z metadat bude odkaz na předchozího verzi, kterou mají verze v konfliktu jako první společnou. Tento odkaz bude nazýván synchronní rodič.

Synchronní rodič bude tedy vytvářet plochou strukturu konfliktních verzí, zatímco klasický rodič bude udržovat původní vztahy mezi verzemi. Plochá struktura je využita z toho důvodu, že u konfliktních verzí není jednoznačná logická souvislost. Výhodou ploché struktury také je, že každá entita, bude mít vždy maximálně jeden konflikt (v konfliktu můžou být více než dvě verze).

Zároveň s vytvořením nové verze se v logu daného uzlu vytvoří nový záznam, jenž bude odkazovat na tuto verzi. Záznamu se nastaví aktuální datum a čas jako čas vytvoření. Index záznamu bude o jedna vyšší, než má aktuální poslední záznam.

Jestliže dva uzly vytvoří ve stejný čas novou verzi ke stejné entitě, dojde k rozvětvení původní jednotné linie verzí na dvě větve, obdobně jako u verzovacích systémů [15]. Díky tomuto rozvětvení lze programově nalézt konflikt. Po jeho nalezení dojde k označení konfliktních verzí příznakem a nastavení společného synchronního rodiče. Konflikt lze následně vyřešit pomocí založení nové verze, která bude odkazovat na obě konfliktní verze – tato verze bude v textu označována jako sloučená verze. Z tohoto důvodu je nutné, aby odkazy rodič a synchronní rodič byly kolekce a ne proměnné s jednou hodnotou.

Vytvoření verze spolu se založením do logu a detekcí konfliktu bude prováděno v jedné databázové transakci. Pokud bude konflikt detekován, přistoupí se k jeho řešení v nové databázové transakci, která bude vytvořena po úspěšném dokončení předchozí transakce. Využitím databázových transakcí relační databáze bude zajištěna integrita dat [25].

V průběhu synchronizace může dojít i ke složitějším konfliktům. Jednotlivé typy konfliktů jsou rozepsány v následující části, spolu se způsobem detekce i nastavením správného synchronního rodiče. U každého typu konfliktu bude přiložen obrázek, kde v levé polovině bude strom rodičů z pohledu popisovaného uzlu. K němu se bude přidávat nová verze z jiného uzlu, která bude

naznačena rudě červenou barvou. Verze, které budou touto verzí ovlivněny budou podbarveny růžovou barvou. V pravé polovině bude již vyobrazen strom synchronních rodičů po nastavení správného synchronního rodiče pro všechny konfliktní verze, tedy i pro nově přidanou.

3.3.1 Konflikt dvou verzí

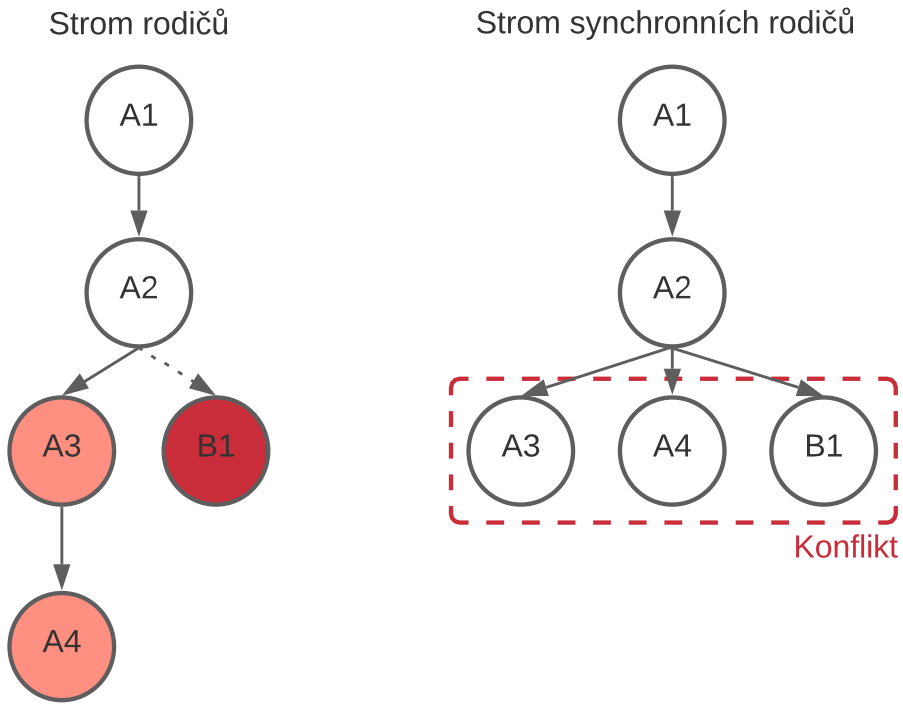
Základním a nejjednodušším konfliktem je konflikt dvou verzí. V tomto případě, mohou mít konfliktní verze maximálně jednoho potomka (následující verze), který ale není společný pro obě verze – tedy nemají sloučenou verzi, která by konflikt řešila. Na obrázku 3.5 je zachycen příklad tohoto konfliktu. Uzel B vytvořil verzi B1, která se odkazuje na verzi A2, ale na tuto verzi se již odkazuje verze A3, na kterou se váže dále verze A4. Obrázek zachycuje pohled uzlu A, kdy mu přichází verze B1 od jiného uzlu. Strom synchronních rodičů byl původně pro entitu linie verzí A1 až A4 (stejný jako strom rodičů, před přidáním B1). Ze stromu rodičů je vidět ,že pro verzi A3 a B1 (a jejich následné verze) je společný rodič verze A2 a proto je pro všechny zmíněné verze nastaven jako synchronní rodič, viz druhá polovina obrázku.

Výsledný konflikt je mezi verzemi A3, A4 a B1. Z původní stromové struktury není zřejmá logická souvislost mezi konfliktními verzemi – verze B1 může patřit před verze A3, A4, za ně či případně mezi ně. Uvedení konfliktů do ploché struktury zjednodušuje jeho možné řešení. Zároveň usnadňuje detekci i případných dalších konfliktních verzí, které by se mohli odkazovat na verze A3, A4 nebo B1.

Z pohledu B se jedná zprvu o stejný konflikt, nejprve bude v konfliktu verze A3 s B1, ale následně do tohoto konfliktu přibude verze A4. Tento druh konfliktu je popsán v následující části textu.

3.3.2 Konflikt s verzemi v konfliktu

Pokud bude přidána verze, která odkazuje na verzi, jenž je v konfliktu, je tato nová verze automaticky též v konfliktu. Její synchronní rodič je stejný jako synchronní rodič verze, na níž se odkazuje. Následující obrázek 3.6 vyjadřuje stejný případ, který byl popsán v předchozím typu konfliktu, ale nyní je z pohledu uzlu B. Uzel B již přijal verzi A3 a detekoval konflikt dvou verzí (3.3.1). Nyní přijímá verzi A4. Strom synchronních rodičů je stejný jako strom rodičů (zatím bez verze A4). Verze A4 odkazuje na verzi A3, která je již v konfliktu, proto se jako synchronní rodič nastaví verze A2, která je již synchronní rodič pro verzi A3. Nyní lze vidět, že po doplnění všech verzí jsou



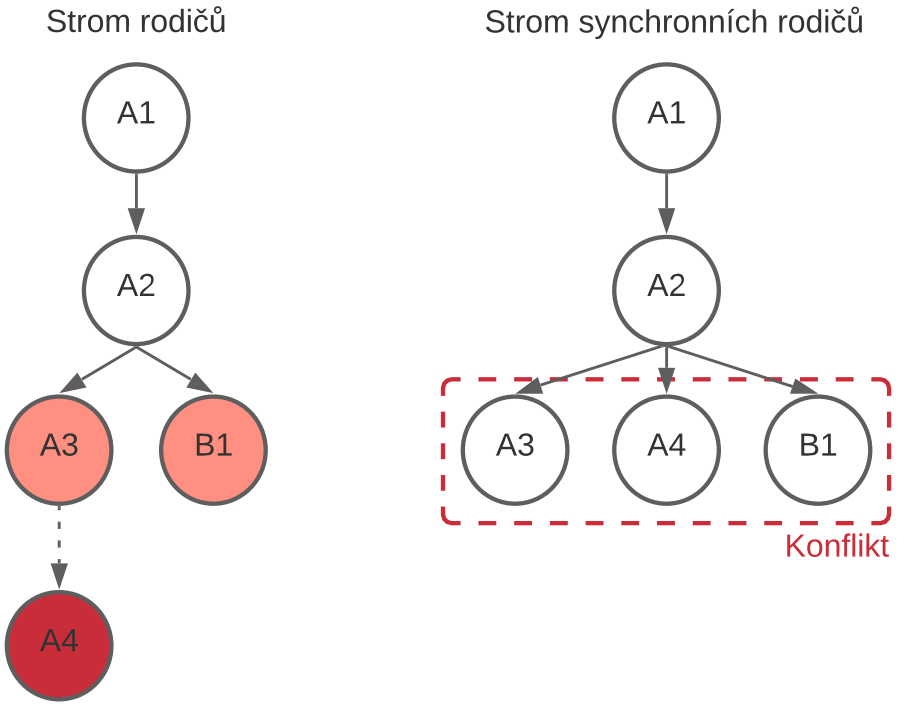
Obrázek 3.5: Příklad konfliktu dvou verzí – po detekci nové verze z jiného uzlu (B1) je konflikt mezi B1 a A3, tudíž i mezi A4

stromy synchronních rodičů na uzlech A a B stejné (obrázek 3.5 z pohledu uzlu A a obrázek 3.6 z pohledu uzlu B).

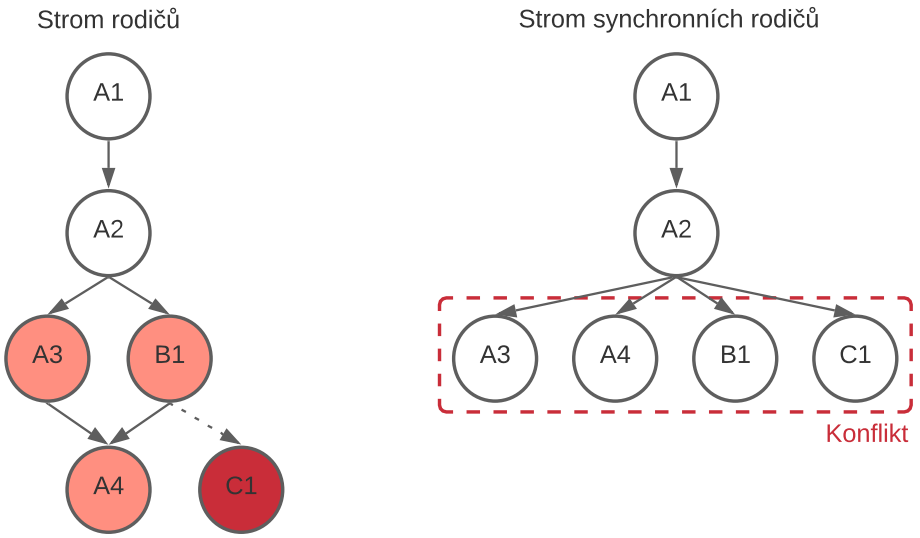
3.3.3 Konflikt nové verze se sloučenou verzí

Konflikt nové verze se sloučenou verzí spočívá v tom, že poté co je jeden konflikt vyřešen, např. konflikt dvou verzí (3.3.1), přijde uzlu nová verze, která se odkazuje na verzi, která byla v konfliktu původně, viz obrázek 3.7. Zde původní konflikt mezi verzemi A3 a B1 vyřešila sloučená verze A4. Poté uzlu přišla verze C1, odkazující se B1. V tomto případě je společný rodič verze A2 – prarodič C1. Proto se ve stromě synchronních rodičů na verzi A2 přímo navážou všechny verze, které jsou v klasickém stromě pod ní, v tomto případě verze A3, A4, B1 a C1.

Obrázek 3.7 vyjadřuje pohled uzlů A a B. Z pohledu uzlu C, který má verze A1, A2, B1, vytvořil verzi C1, se přijetí verze A3 rovná konfliktu dvou



Obrázek 3.6: Příklad konfliktu s verzemi v konfliktu – v konfliktu jsou verze A3 a B1, po detekci nové verze A4 je i tato verze zařazena do konfliktu s A3 a B1

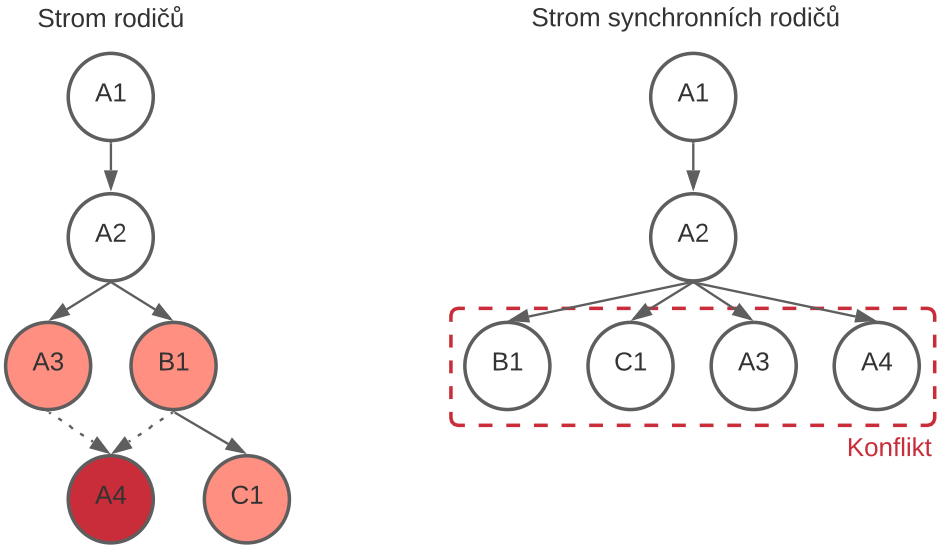


Obrázek 3.7: Příklad konfliktu nové verze se sloučenou verzí – konflikt mezi A3 a B1 byl vyřešen verzí A4, nová verze C1 toto řešení zneplatní a verze A3, A4, B1, C1 jsou uvedeny do konfliktu

verzí (3.3.1). Následně při přijetí verze A4 dojde ke konfliktu nekompletní sloučené verze (3.3.4), který je popsán níže.

3.3.4 Konflikt nekompletní sloučené verze

Při řešení konfliktů může dojít k situaci, kdy jeden uzel konflikt vyřeší, zatímco jiný uzel do původních konfliktních verzí přispěje novou verzí, která bude též v konfliktu. Pokud řešení konfliktu přijde uzlu, jenž přispěl do konfliktu, bude pro něj řešení konfliktu neplatné, protože nepokrývá všechny verze, které jsou aktuálně v konfliktu. Proto verzi s řešením zařadí do konfliktu mezi ostatní konfliktní verze. Tento případ nastává, pokud se podíváme na příklad z předchozího konfliktu z pohledu uzlu C. Na obrázku 3.8 je nyní uzel C ve stavu, kdy má v konfliktu verze A3, B1 a C1 a přichází mu sloučená verze A4. Tato verze ale neslučuje všechny verze, které jsou aktuálně v konfliktu, proto se zařadí též do konfliktu. Synchronní rodič je verze A2 – prarodič A4.



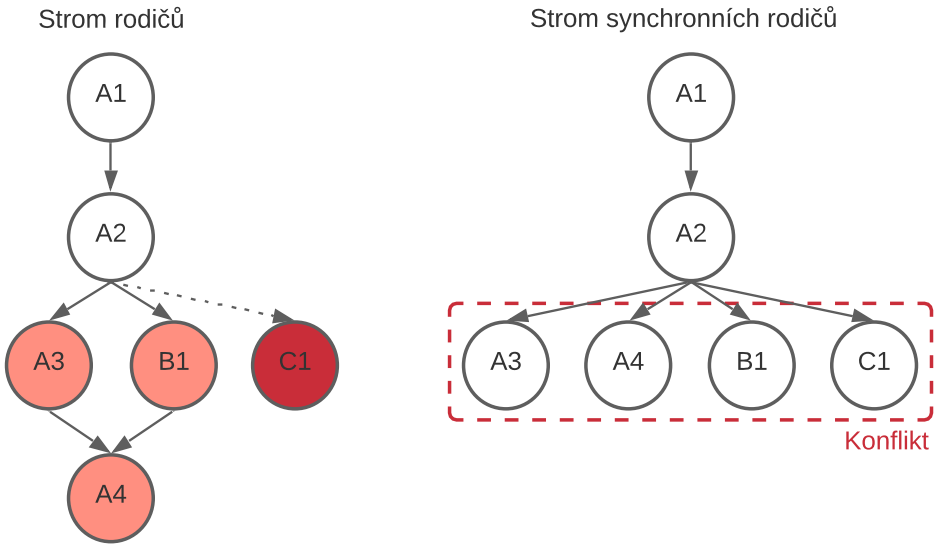
Obrázek 3.8: Příklad konfliktu nekompletní sloučené verze – v konfliktu jsou verze A3, B1, C1, nově detekovaná verze A4 řeší konflikt pouze mezi A3 a B1, proto je též uvedena do konfliktu

Opět, stejně jako u předchozí dvojice konfliktů je vidět, že po výměně všech verzí se stromy synchronních rodičů (obrázky 3.7 a 3.8) dostanou na všech uzlech do totožného stavu.

3.3.5 Konflikt s více verzemi

Konflikt s více verzemi je rozšíření základního konfliktu dvou verzí (3.3.1). V tomto typu konfliktu dochází k situaci, kdy má uzel 2 a více verzí v jednom konfliktu, tedy všechny odkazují na stejného synchronního rodiče. Tento

konflikt vyřeší, ale později přijímá uzel, který se odkazuje na verzi stejnou jako uzly, které byly v konfliktu. Tato nová verze zneplatní řešení původního konfliktu a dá ho do konfliktu spolu se všemi konfliktními verzemi, které řešilo. Synchronním rodičem se stává verze, která je společná pro původní konfliktní verze. Příklad tohoto případu se nachází na obrázku 3.9. Z pohledu uzlů A a B byl konflikt mezi verzemi A3 a B1 vyřešen pomocí verze A4, ale uzel C vytvořil verzi C1, která se odkazuje na verzi A2. Pro konfliktní verze se tedy nastaví synchronní rodič A2, jelikož je společný pro původní konfliktní verze a nově přidanou verzi.



Obrázek 3.9: Příklad konfliktu s více verzemi – konflikt mezi A3 a B1 vyřešila verze A4, nová verze C1 ale původní konflikt rozšiřuje, tudíž A4 je již neplatné řešení a verze A3, A4, B1, C1 jsou uvedeny do konfliktu

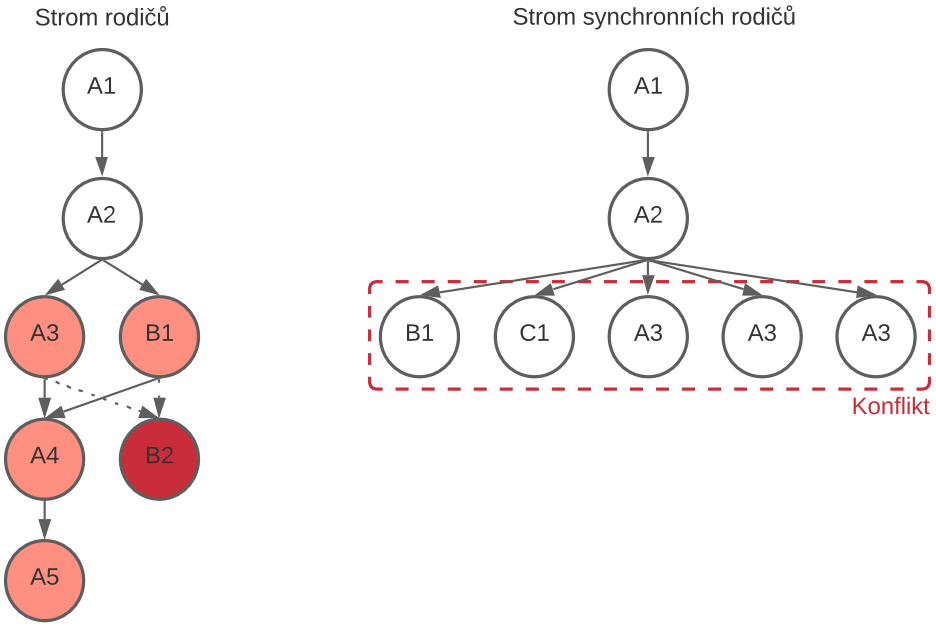
Z pohledu uzlu C by se nejprve při přidání verze A3 jednalo o konflikt mezi dvěma verzemi (3.3.1), následně přidání verze B1 by vytvořilo opět konflikt s více verzemi. Přidáním verze A4 by vznikl konflikt nekompletního sloučené verze, jelikož v konfliktu v tu chvíli uzel C bude mít verze A3, B1, C1, ale verze A4 bude řešit pouze konflikt mezi A3 a B1.

3.3.6 Konflikt dvou sloučených verzí

Posledním typem konfliktu je případ, kdy dva uzly vytvoří řešení pro stejný konflikt. V tomto případě se uvedou do konfliktu obě sloučené verze (řešení), jejich potomci a zároveň i verze, které byly řešeny sloučenými verzemi. Synchronní rodič pro všechny tyto verze bude rodič původních konfliktních verzí (prarodič sloučených verzí, obsahující řešení konfliktu). Tento případ

se nachází na obrázku 3.10, kde z pohledu uzlu A dochází k tomu, že mu přichází sloučená verze B2, jenž řeší konflikt verzí A3 a B1. Ten je ale již vyřešen pomocí verze A4, na kterou navazuje verze A5. Všechny vyjmenované verze se uvedou do konfliktu s tím, že synchronní rodič pro ně bude verze A2, prarodič verzí, jenž řeší stejný konflikt.

Z pohledu uzlu B se při přijetí verze A4 jedná o stejný typ konfliktu. Při přijetí verze A5 už má v konfliktu verze A3, A4, B1 a B2. Rodičovská verze A5 – verze A4 je již v konfliktu, jedná se tedy o konflikt s verzemi v konfliktu (3.3.2).

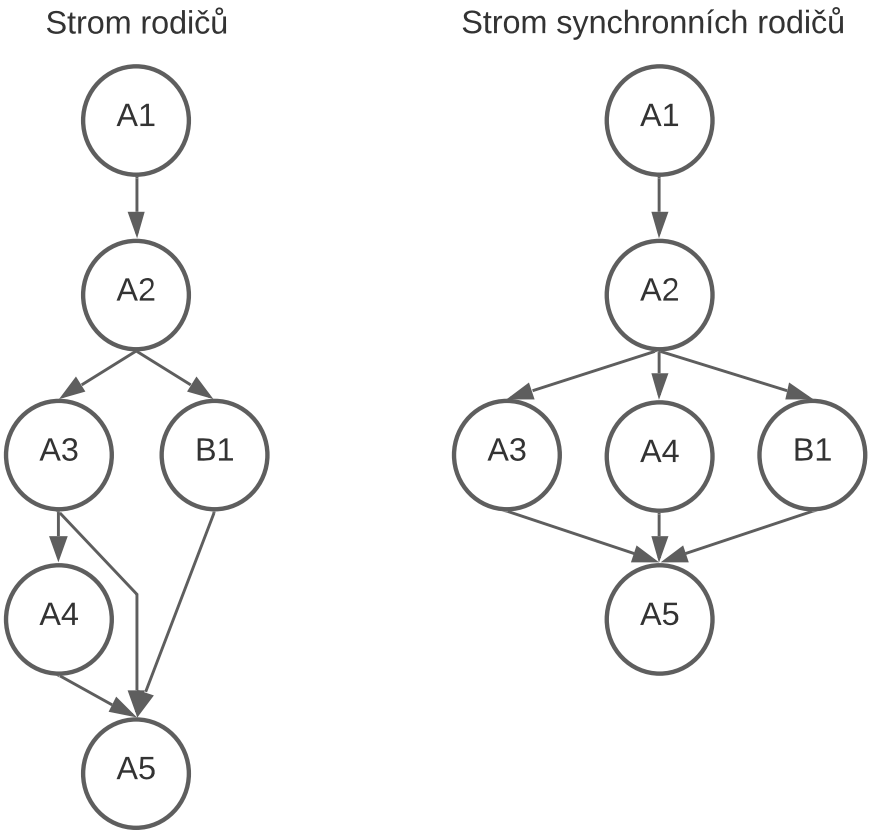


Obrázek 3.10: Příklad konfliktu dvou sloučených verzí – konflikt mezi A3 a B1 řeší verze A4, ale zároveň nově detekovaná verze B2, mezi řešeními vzniká konflikt a zároveň zaniká řešení původního konfliktu

3.4 Řešení konfliktů pomocí verzování

Řešení konfliktů bude zajištěno pomocí vytvoření již zmíněných sloučených verzí. Obrázek 3.11 vyjadřuje příklad řešení konfliktu, kdy sloučená verze A5 řeší konflikt mezi verzemi A3, A4 a B1. Verze řešící konflikt musí mít stejné rodiče ve stromě rodičů i ve stromě synchronních rodičů při založení a po celou dobu platnosti svého řešení. V případě, že by došlo konfliktu s touto verzí či jakoukoliv jinou verzí, která je ve stromě nad ní, tak se strom synchronních rodičů změní a nebudou tedy již synchronní rodiče odpovídat klasickým rodičům.

Řešení konfliktů bude probíhat dvěma způsoby: manuálně a automaticky. Manuální řešení budou provádět uživatelé aplikace, která bude využívat synchronizačního frameworku. Manuálně budou řešit konflikty, které nepůjdou vyřešit automaticky.



Obrázek 3.11: Příklad řešení konfliktu – konflikt mezi verzemi A3, A4, B1 řeší verze A5

Přidávání nové verze se bude dít spolu s detekcí konfliktu v jedné databázové transakci [25]. Pokud dojde k detekování konfliktu, všechny konfliktní verze se označí příznakem a dojde k nastavení příslušného synchronního rodiče. Pro zahájení automatického řešení konfliktu musí být splněny dvě podmínky. První podmínkou je, aby byl uzel ve stavu vůdce. Druhou podmínkou je, aby byly všechny verze ve finálním stavu. Verze je ve finálním stavu, pokud nemá nataven čas platnosti do, tudíž stále platí. Nebo verze má nastavený čas platnosti, což znamená, že existuje verze, která se na ni odkazuje a tato verze je již dostupná na daném uzlu. Tento stav lze ilustrovat na obrázku 3.11, kdy uzlu C, který žádnou z uvedených verzí zatím nemá, budou postupně přicházet v následujícím pořadí: A1, A2, A3, B1, A4, A5. Při přijetí verzí A1 až A3 nedochází ke konfliktu. Přijetí verze B1 vyvolá konflikt s verzí A3. Automatické řešení konfliktu nelze začít, jelikož verze A3 není ve finálním stavu, protože není již platná, jelikož má následující verzi A4, ale verze A4 není zatím na uzlu C dostupná. Po přijetí verze A4 nastává stejná situace,

nejedná se o finální verzi, tou je až verze A5. V tomto případě tedy k řešení konfliktu již došlo na jiném uzlu. Pokud by ale verze A5 neexistovala, mohl by uzel C zahájit řešení konfliktu po přijetí verze A4 v případě, že by byl v té chvíli ve stavu vůdce.

Řešení konfliktu se bude provádět v samostatné databázové transakci, která bude následovat až po detekci konfliktu. Pokud se nalezne řešení konfliktu, dojde k vytvoření nové sloučené verze, která bude mít za rodiče i synchronní rodiče všechny konfliktní verze. Opačně, pokud se řešení nenalezne, nastaví se všem verzím v konfliktu příznak, že konflikt nelze vyřešit automaticky a je tedy nutný manuální zásah uživatele.

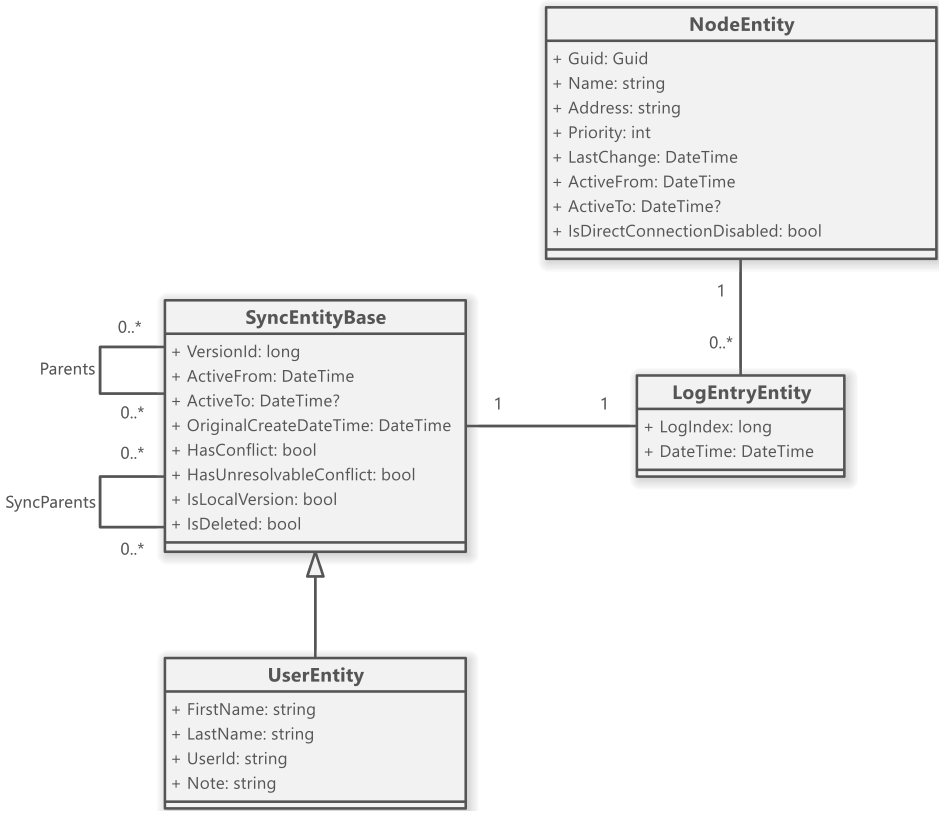
Automaticky řešit konflikty může uzel v roli vůdce. Pokud by mohl vytvářet řešení konfliktu každý uzel, docházelo by k neustálému vytváření konfliktních verzí. Konflikt dvou sloučených verzí (3.3.6) je ale stále možný z důvodu rozpadu sítě. Pokud se síť rozpadne ve chvíli, kdy v obou částech sítě mají uzly verze v konfliktu, vytvoří noví vůdci obou částí sítě své řešení. Jeden z uzlů ho vytvoří na základě toho, že byl vůdcem a zůstane i na dále platným vůdcem v dané části. Zatímco ve druhé části sítě dojde ke zvolení nového vůdce. Po zvolení vůdcem, uzel začne procházet konfliktní verze, které má u sebe a začíná řešit jejich konflikty. Stejně jako po detekci konfliktů i zde platí stejné podmínky pro zahájení automatického řešení konfliktu. Navíc přibývá podmínka, která vyžaduje, aby alespoň jedna z verzí neměla příznak, že ji nelze automaticky řešit. Tím se vyhne řešení konfliktů, které byly označeny za automaticky neřešitelné. Pokud by sestoupil z role vůdce, zastavil by procházení a řešení konfliktních verzí.

Mechanismus vyhledání a řešení konfliktů po zvolení do role vůdce je nutný z důvodu možného úplného selhání vůdce. Pokud by zvolený vůdce selhal, musí být nový uzel schopen uvést celou síť do konzistentního stavu.

3.5 Databázový model

Databázový model z obrázku 3.12 zachycuje databázové entity nejen pro samotný framework, ale i pro demo aplikaci. Frameworku patří entity: NodeEntity, LogEntryEntity a SyncEntityBase. Zatímco demo aplikaci patří zbývající entita UserEntity.

NodeEntity uchovává informace o uzlech v síti. Framework si bude ukládat informace o sobě samém a o ostatních uzlech v síti. Každý uzel bude mít svůj unikátní identifikátor, podle kterého ho bude moci identifikovat jakýkoliv jiný uzel. Dalším parametrem bude jméno uzlu, které bude sloužit zejména uživatelům pro snadnější identifikaci uzlu. Parametr adresa, která bude využívána komunikačním API při synchronizaci uzlů, bude obsahovat URL nebo IP adresu uzlu. Následuje zmíněná priorita uzlu, která již slouží pro určení uzlů, které mají být přednostně vůdci clusteru. Parametr LastChange bude sloužit k uchovávaní informace, kdy naposledy došlo ke změně informací



Obrázek 3.12: ER diagram databáze

o uzlu. I po trvalém odpojením uzlu ze sítě, musí být uchováván záznam o jeho přítomnosti. Proto entita obsahuje ActiveFrom a ActiveTo, které značí, zda je uzel považován za aktivního. Poslední parametr IsDirectConnectionDisabled usnadní testování a umožní vizualizaci výpadků v demo aplikaci – bude vyjadřovat, zda je daný uzel propojen s aktuálním uzlem přímou cestou.

Entita LogEntryEntity vyjadřuje záznam v multi-logu. Bude navázána na NodeEntity a tímto vztahem bude vyjádřeno, do jakého z logů záznam patří. Samotný záznam bude obsahovat časový údaj o jeho vytvoření v rámci aktuálního uzlu. V neposlední řadě bude obsahovat i index, který bude určovat pořadí záznamu v daném logu.

Na LogEntryEntity bude již navázána samotná entita určená k synchronizaci. Entity určené k synchronizaci budou dědit od entity SyncEntityBase. Tato entita reprezentuje jednotlivé verze a obsahuje tedy číslo verze, které je unikátní spolu v kombinaci s unikátním identifikátorem NodeEntity. Rozsah, kdy je verze aktivní a čas, kdy byla původně entita vytvořena. Dalšími parametry jsou příznaky, zda je verze v konfliktu a zda případně lze řešit konflikt automaticky. Zda se jedná o lokální verzi či zda je verze označená za smazanou. Entita má též vztahy sama na sebe. Pro označení rodičovských verzí slouží vztah Parents a k označení synchronních rodičů slouží SyncParents.

Entita z demo aplikace UserEntity je určena k synchronizaci, proto dědí od SyncEntityBase. Obsahuje základní informace o uživateli – jméno, příjmení, unikátní identifikátor a poznámku.

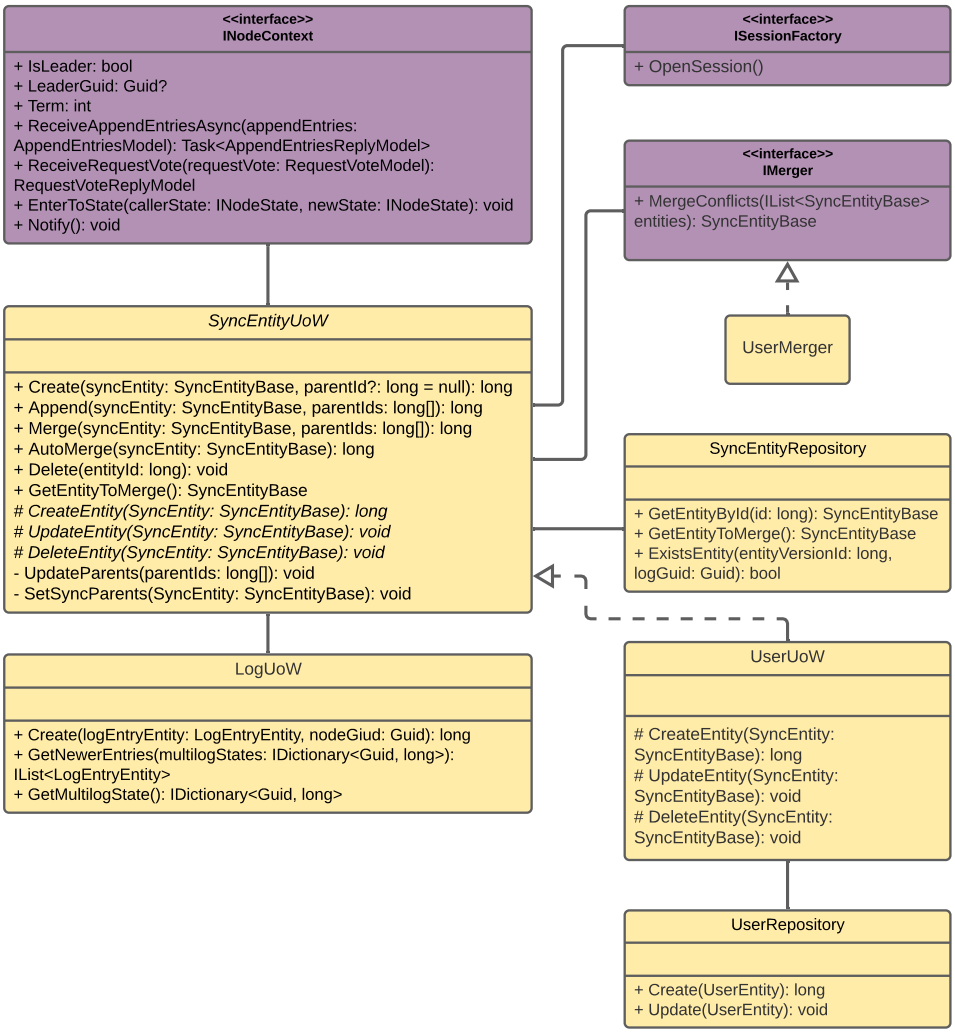
3.6 API pro manipulaci se synchronizovanými entitami

Pro obecnou manipulaci s databázovými entitami bude využito návrhových vzorů Repository a Unit of Work [20]. Vrstva repozitářů splňujících Repository vzor bude obsahovat metody pro vytváření, aktualizaci, mazání a získávání databázových entit. Vrstva repozitářů bude využívána vrstvou tříd splňujících vzor Unit of Work. Tyto třídy budou zajišťovat provedení manipulací s entitami v jedné databázové transakci [20]. Následující digram tříd 3.13 vyobrazuje třídy, které se zaobírají manipulací se synchronizovanými entitami.

Jádrem bude abstraktní třída *SyncEntityUoW*, od které budou dědit všechny Unit of Works (UoW), které budou pracovat se synchronizovanými entitami. Pro každou synchronizovanou entitu musí být vytvořen právě jeden UoW, který se bude o entitu starat. *SyncEntityUoW* bude definovat následující veřejné metody pro manipulaci se synchronizovanými entitami:

- **Create** – vytvoření nové verze entity.
- **Append** – přijetí verze od jiného uzlu.
- **Merge** – manuální vyřešení konfliktu mezi verzemi.
- **AutoMerge** – automatické vyřešení konfliktu mezi verzemi.
- **Delete** – označení verze za smazanou.
- **GetEntityToMerge** – získání verze, která je v konfliktu a je připravena pro automatické řešení.

Každý UoW pro synchronizovanou entitu bude muset implementovat abstraktní metody definované v *SyncEntityUoW* – *CreateEntity*, *UpdateEntity*, *DeleteEntity*. Metoda *CreateEntity* bude volána při každém založení nové verze entity. Bude mít na starost vytvoření entity v databázi a bude tedy muset volat vrstvu repozitářů. Z toho důvodu bude konkrétní UoW obsahovat i vlastní repozitář (viz *UserUoW* a *UserRepository* na obrázku 3.13). Tímto provoláním metody je programátorovi umožněno provést činnosti v konkrétním UoW, které jsou specifické pro založení konkrétního typu synchronizované entity. Metoda *UpdateEntity* bude volána v případě aktualizace metadat verze.



Obrázek 3.13: Diagram tříd manipulujících se synchronizovanými entitami

Například v případě nastavení platnosti rodiče či nastavení příznaků týkajících se konfliktů. Metoda *DeleteEntity* bude volána těsně před označením entity za smazanou. Bude též programátorům umožňovat provést v transakci úkony potřebné pro daný typ synchronizované entity, před jejím logickým smazáním [48]. Výsledkem logického smazání bude nová verze, která bude obsahovat již nastavený příznak o smazání.

Metoda **GetEntityToMerge** bude využívána v případě, že se uzel stane vůdcem. Pro každý typ synchronizované entity zkontroluje, že nemá verze v konfliktu, pokud ano, předá je k automatickému vyřešení.

Pro manipulaci s multi-logem bude sloužit UoW, který bude pro *SyncEntityUoW* zajišťovat vytvoření záznamu v logu s vygenerováním indexu pro lokální záznamy, případně vytvoření záznamu s příslušným indexem pro cizí logy. Dále bude obsahovat metody pro získávání stavů multi-logů jednotlivých uzlů a pro získávání novějších záznamů na základě poskytnutého stavu multi-logu. Tyto metody budou využívat třídy zajišťující synchronizaci multi-logů napříč uzly v síti.

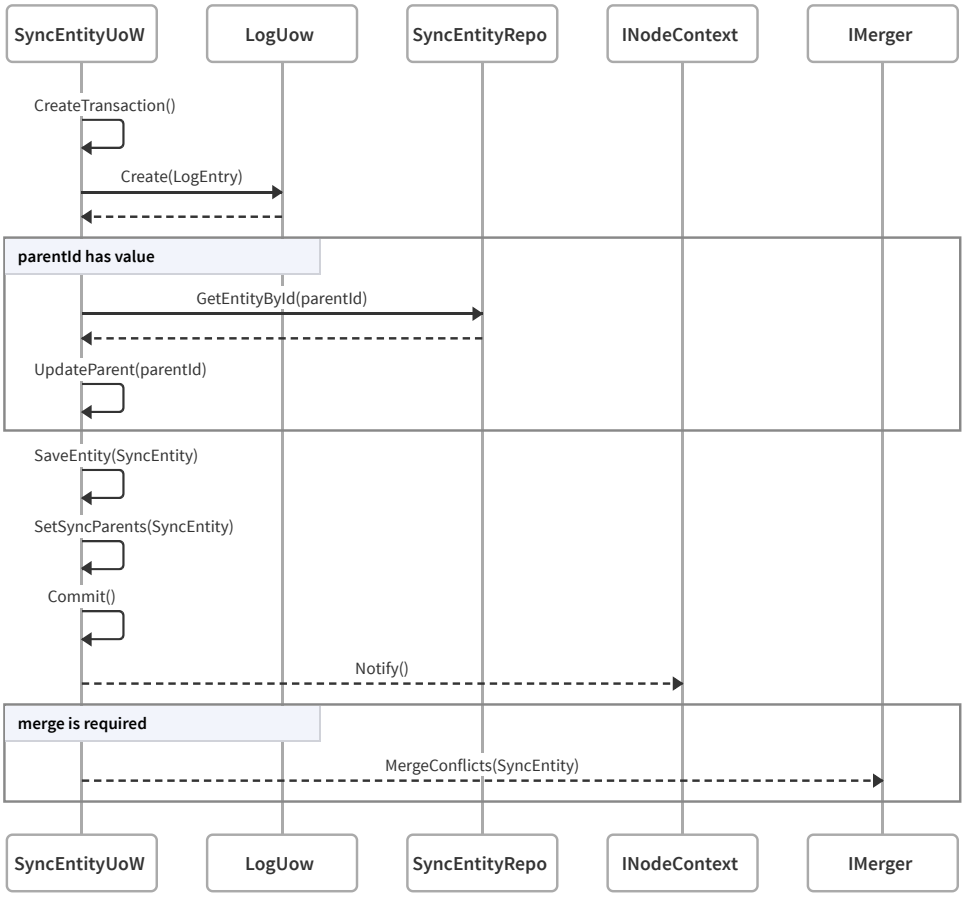
SyncEntityRepository bude repozitář, který bude zajišťovat získávání metadat o synchronizovaných entitách, tyto metadata bude využívat *SyncEntityUoW*, a to zejména při detekci konfliktů mezi verzemi.

Dále bude *SyncEntityUoW* využívat několik tříd splňujících rozhraní *INodeContext*, *ISessionFactory*, *IMerger*. *INodeContext* bude sloužit k získávání informací o stavu uzlu, v jaké roli se nachází, jaký je aktuální term apod. [40]. Z pohledu *SyncEntityUoW* je důležitá informace, zda je uzel vůdcem, aby mohlo být rozhodnuto, zda začne automatické řešení konfliktu. Dále skrze metodu *Notify* bude *SyncEntityUoW* notifikovat synchronizační mechanismus, že byla přidána nová verze, kterou má poslat ostatním uzlům síti v případě vůdce, resp. poslat vůdci v případě následovníka. *ISessionFactory* bude obstarávat vytvoření databázového spojení, ve kterém následně založí transakci [5]. Poslední třídou, kterou bude využívat *SyncEntityUoW*, je konkrétní implementace *IMerger* pro daný typ synchronizované entity. Implementace budou složit k automatickému vyřešení konfliktu mezi verzemi.

Proces vytvoření nové verze (volání metody **Create** na *SyncEntityUoW*) je zachycen na následujícím sekvenčním diagramu 3.14. Nejprve se založí databázová transakce, ve které se bude vytváření verze provádět. Založí se záznam v multi-logu, konkrétně v logu s ID aktuálního uzlu a index záznamu bude o 1 vyšší než index poslední záznam pro daný log. V případě, že byl specifikován rodič, načte se z databáze. Pokud je rodič aktivní, zneplatní se. Poté dojde k provolání abstraktní metody *CreateEntity*, která bude implementována v již konkrétním UoW pro daný typ synchronizované entity. Po uložení se volá metoda *SetSyncParents*, která detekuje případné konflikty. Pokud konflikt nalezne, nastaví všem konfliktním verzím příslušného synchronní rodiče. V opačném případě, kdy konflikt nedetekuje, nastaví klasického rodiče (pokud je specifikován), i jako synchronního rodiče. Po úspěšném uložení dat do databáze dojde k notifikování kontextu uzlu, že existuje nová verze, kterou je nutné synchronizovat. Nakonec započne automatické řešení konfliktu, pokud je vyžadováno a uzel ho může provést.

Proces přijetí verze od jiného uzlu, realizovaný skrze metodu **Append** se bude oproti vytvoření lišit tím, že na vstupu nebude přijímat pouze jednoho rodiče, ale list rodičů. List rodičů přijímá, jelikož ukládaná verze může být sloučená verze, která řeší konflikt – má tedy odkazy na více rodičů, na ty mezi kterými konflikt řeší. Zároveň před vložením verze do databáze zkontroluje, že

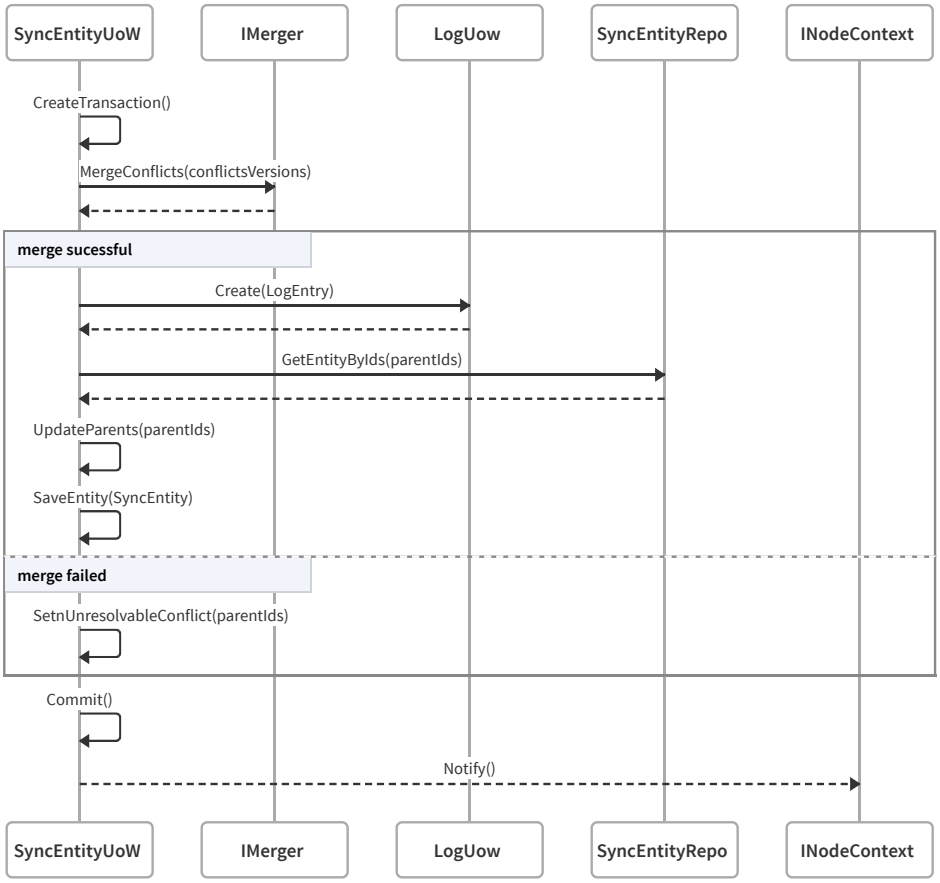
daná verze již neexistuje. Synchronizační proces uzlu je notifikován o vytvoření nové verze pouze v případě, že je uzel v roli vůdce. Jedná se tedy o případ, kdy následovník pošle vůdci nový záznam a ten ho následně posílá i ostatním následovníkům, kteří ho ještě nemají.



Obrázek 3.14: Sekvenční diagram - vytvoření synchronizované entity

Metoda **Merge** – proces ručního řešení konfliktu přijímá stejně jako *Append* na vstupu více rodičů. Na rozdíl od *Append* metody ale notifikuje o nutnosti synchronizace záznamu vždy, stejně jako metoda *Create*.

Sekvenční diagram 3.15 zobrazuje proces automatického řešení konfliktů (metoda **AutoMerge**). Nejprve opět dojde k založení databázové transakce. Poté se předá list konfliktních verzí konkrétní implementaci rozhraní *IMerger*, která se pokusí konflikt automaticky vyřešit. Pokud se konflikt podaří vyřešit, opět se aktualizují rodiče. V této chvíli rodiče (klasičtí i synchronní) nově vzniklé verze jsou všechny původně konfliktní verze. Následně proběhne uložení samotné nově vytvořené verze. Spolu s vytvořením záznamu v multi-logu, stejně jako v případě ukládáních verzí skrze metodu *Create*. Pokud se konflikt nepodaří vyřešit, jsou všechny tyto verze označeny příznakem, že je nelze vyřešit automaticky. Nakonec se provede uložení všech změn do databáze a notifikování synchronizačního procesu o nově vytvořené verzi.



Obrázek 3.15: Sekvenční diagram - automatické řešení konfliktu

3.7 Dynamická změna sítě

V průběhu synchronizace může dojít ke změně sítě uzlů. Jedním z důvodů může být nečekaný výpadek uzlu nebo spojení mezi uzly. Další změnu vyvolává připojení nového uzlu, případně jeho trvalé odpojení. Synchronizační framework bude tedy podporovat dynamickou změnu clusteru, aniž by muselo dojít k restartování všech uzlů [41].

3.7.1 Přidání uzlu

Informace o uzlech se budou přenášet ve zprávách *AppendEntry*, *RequestVote* a v jejich odpovědích [39]. Přesněji se bude přenášet list všech uzlů, který má odesílatel zprávy aktuálně uložené v databázi. Uzel přijímající zprávu zkontroluje pro každý uzel ze seznamu, že má o něm u sebe záznam, pokud ne, přidá si ho. Zároveň také kontroluje datum poslední aktualizace informací o uzlu, pokud je záznam z listu novější, aktualizuje svůj záznam. Datum poslední

změny u informací o uzlu může, kromě výjimky popsané níže v kapitole 3.7.2, nastavovat pouze uzel, kterému informace patří (netýká se vlastnosti, zda je uzel připojen přímo). Tím bude zajištěno, že nemohou vznikat konflikty mezi změnami informací o uzlu. Záznam s nejnovějším časovým údajem poslední změny je ten nejaktuálnější – uplatňuje se tedy pravidlo last-write-wins dle data [44].

Při spuštění aplikace se načtou z konfigurace informace o lokálním uzlu a provede se jejich aktualizace, případně vytvoření. Poté dojde k načtení sousedních uzlů, u nichž je vyžadován unikátní identifikátor a komunikační adresa (IP nebo URL). Do databáze se vloží záznamy o sousedních uzlech, které dle unikátního identifikátoru v databázi neexistují. Zbylé informace, např. jméno, platnost atd. získají následně od ostatních uzlů. Aby se uzel dokázal připojit do sítě, je nutné, aby měl v konfiguraci specifikovaného minimálně jednoho souseda. Díky němu následně zjistí informace o ostatních uzlech v síti. Po uplynutí času pro přijmutí *AppendEntry* zprávy od vůdce se uzel dostává do stavu kandidát a posílá sousedům zprávu *RequestVote*. V tuto chvíli se ostatní uzly dovídají o nově připojeném uzlu. Odpovídají a tím nově připojený uzel postupně získá informace o všech uzlech.

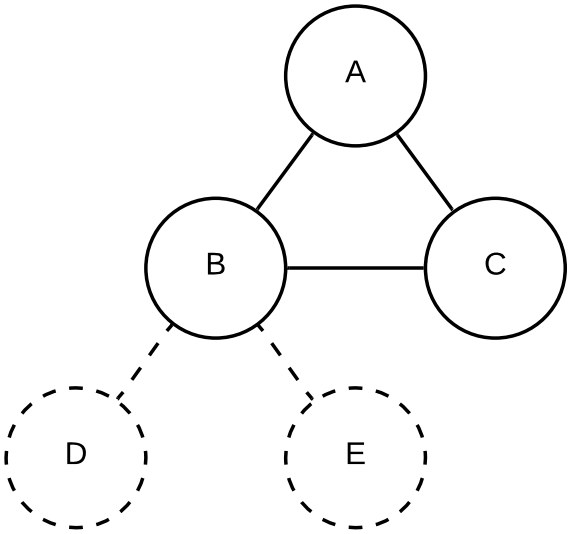
3.7.2 Odstranění uzlu

Výjimka z nastavování časového údaje o poslední změně slouží pro deaktivaci uzlu. Výjimkou lze nastavit pouze platnost „do“, tedy, že uzel již není aktivní. Tento údaj bude moci být nastaven i z jiných uzlů a bude nutné při nastavení dávat pozor, aby datum změny byl novější, než je poslední změna (z důvodu nesynchronizovaného času mezi uzly v distribuovaném prostředí). Tento mechanismus je nutný pro případ, že by došlo k selhání uzlu a již by nebylo možné ho opětovně zprovoznit.

3.7.3 Ovlivnění výsledku voleb vůdce

Dynamické přidávání uzlů by mohlo ovlivnit výsledky voleb tak, že by mohlo dojít ke zvolení dvou vůdců v jedné síti. Na obrázku 3.16 je topologie sítě stávajících uzlů A, B, C, do které se připojují uzly D a E skrze uzel B.

Prvním příkladem, kdy dojde k ovlivnění výsledku voleb a zvolení dvou vůdců je případ, kdy jsou např. uzly B a C rovnocenní kandidáti (mají stejný term, prioritu i počet záznamů). Připojující se uzly D a E, posílají *RequestVote* uzlu B, mají ale nižší prioritu než uzel B a proto jim uzel B volby zamítne. Ve stejnou chvíli by vyhlásili volby uzly B i C, C by získal hlas od A a od B ne, což mu stačí pro vítězství, jelikož získal 2/3 hlasů. O uzlech D a E se dozvídá



Obrázek 3.16: Topologie sítě uzlů A, B, C s dynamicky přidáním uzlů E a D

až z odpovědi na *RequestVote* od B. Zároveň uzel B získává hlasy od D a E, tudíž má 3/5 hlasů, což je také nadpoloviční většina a také se stává vůdcem.

Tomuto případu lze zabránit přidáním podmínky, která stanoví, že pokud počet aktivních uzlů v síti je před kontrolou výsledků voleb vyšší než před jejich zahájením, volby se budou považovat za prohrané a kandidát bude muset počkat na vypršení časového limitu, než bude moci vyhlásit nové. Ve výše uvedeném případě by došlo k situaci, že by uzel C své volby považoval za prohrané a čekal by na vypršení limitu, zatím by se mu ale ozval uzel B jako vůdce, který by volby vyhrál.

Druhým případem, kdy by mohlo dojít ke zvolení dvou vůdců je situace, kdy bude připojovaný uzel D mít vyšší prioritu než uzel B, ale zároveň nižší než aktuální vůdce C. Uzel B by uzlu D udělil hlas, tím by uzel D získal 2/2 hlasů a mohl se stát vítězem voleb. Této situaci zabrání výše zmíněná podmínka. Uzel D v odpovědi od B získá nové informace o dalších uzlech v síti, bude tedy mít více uzlů než před vyhlášením voleb a tím pádem poruší zmíněnou podmínku a své volby bude považovat za prohrané.

3.8 Komunikace mezi uzly

Pro komunikační rozhraní mezi uzly bude využito knihovny gRPC pro .NET [24]. Komunikační vrstva bude obsahovat i mechanismy pro detekci výpadků uzlů a cest. V případě detekování výpadku cesty se bude uzel snažit posílat zprávy cílovým uzlům skrze jiné mezilehlé uzly.

3.8.1 gRPC

Framework gRPC (gRPC Remote Procedure Calls) je open-source systém založený na vzdáleném volání procedur – tedy vykonání procedury na jiném stroji, než ze kterého je volána [24]. Pro transport dat využívá protokolu HTTP/2 [10] ve výchozím stavu [18]. Pro serializaci dat využívá mechanismus Protocol Buffers [24], jenž vyniká svou rychlou a úspornou serializací [7]. Mezi hlavní výhody gRPC patří jednoduchost definování API, snadná škálovatelnost a zároveň je i framework multiplatformní. Framework gRPC je dostupný pod licencí Apache License 2.0 [24], která umožňuje volné šíření softwaru [1]. Komunikační API uzlu bude vystavovat 3 metody, jež budou volány ostatními uzly:

- ***AppendEntry*** – bude sloužit pro poslání záznamů jinému uzlu, případně jako heartbeat notifikace, stejně jako tomu je u algoritmu RAFT [39].
- ***RequestVote*** – opět vychází z algoritmu RAFT, touto metodou bude uzel od ostatních žádat o hlas v průběhu voleb nového vůdce [39].
- ***Ping*** – metoda bude sloužit ke zjišťování stavu sítě. Metodou *Ping* bude uzel žádat jiný uzel o poskytnutí jeho informací o stavu sítě.

3.8.2 Detekce výpadků

Uzel si bude lokálně držet seznam uzlů, které jsou k němu připojeny přímou cestou. V periodických intervalech volat na všech aktivních uzlech proceduru *Ping*. Pokud se mu vrátí korektní odpověď, ponechá daný uzel v seznamu (případně přidá, pokud se tam nenachází). Pokud se mu nevrátí korektní odpověď ve stanoveném čase nebo mu přijde chybná odpověď, odstraní daný uzel ze seznamu. Tímto způsobem detekuje uzel výpadky přímých cest k ostatním uzlům.

V odpovědi na zavolání procedury *Ping* bude uzel odesílat seznam uzlů které jsou přes něj dostupné spolu s jejich vzdáleností. Pro určení vzdálenosti se bude používat metrika hop count – počet přeskoků [17]. Počet přeskoků se rovná počtu mezilehlých uzlů, přes které musí zpráva projít, než dorazí od zdroje k cíli. Dostupné uzly jsou jednak napřímo připojené (jejich cena se tedy rovná nule), ale i uzly které uzel získal v odpovědi na *Ping*. U takto získaných cest k uzlům se jejich vzdálenost zvýší o 1. U cest k uzlům, které nejsou přímo připojené, se ukládá i uzel, přes který musí být zpráva odeslána, aby mohla být doručena cíli. Pokud nebude možné se dostat k uzlu ani přes sousední uzly, bude ho lokální uzel považovat za nedostupný.

3.8.3 Přeposílání zpráv

Pokud zdrojový uzel nemá s cílovým uzlem přímé spojení, je využito mezilehlých uzlů k zajištění doručení zprávy (volání vzdálené procedury). Odesílající uzel vybere z přímo připojených uzlů ten, přes který vede nejkratší cesta do cílového uzlu. Pokud uzel přijme zprávu, která není pro něj, provede výběr ideálního uzlu, přes kterého zprávu poslat a pošle ji. Stejný proces funguje i pro odpovědi na zprávy.

Aby nedocházelo k nekonečnému přeposílání zpráv, bude každé zprávě nastaven maximální počet přeskoků (Time To Live – TTL), které bude moci provést. TTL se nastaví před odesláním zprávy na aktuální počet uzlů v síti, zmenšený o dva (zdrojový a cílové uzel se do přeskoků nepočítají). Na každém mezilehlém uzlu bude TTL sníženo o 1, až dosáhne hodnoty 0, bude zpráva zahozena a zdrojový uzel bude informován o expiraci TTL [17].

3.9 Technologie a knihovny včetně licencování

Synchronizační framework bude z důvodu požadavku v zadání implementován na platformě .NET [34]. Jedná se o multiplatformní technologii poskytující základní knihovny pro tvorbu počítačového softwaru. Kód bude psán v jazyce C#. Framework bude složen z několika projektů, jenž budou samostatně kompilovatelné. Demo aplikace bude vytvořena formou ASP.NET Core Web aplikace s MVC strukturou. Pro vytvoření GUI demo aplikace budou použity zejména technologie Razor pages, JavaScript, CSS [34].

Pro trvalé uložení dat bude využit relační databázový systém PostgreSQL. PostgreSQL je open-source databáze využívající jazyka SQL [23]. Pro abstrahování přístupu k databázi bude využito open-source ORM frameworku NHibernate [5].

Celkový přehled použitých knihoven v závislosti na implementaci se nachází v příloze D spolu s jejich licencemi a verzemi.

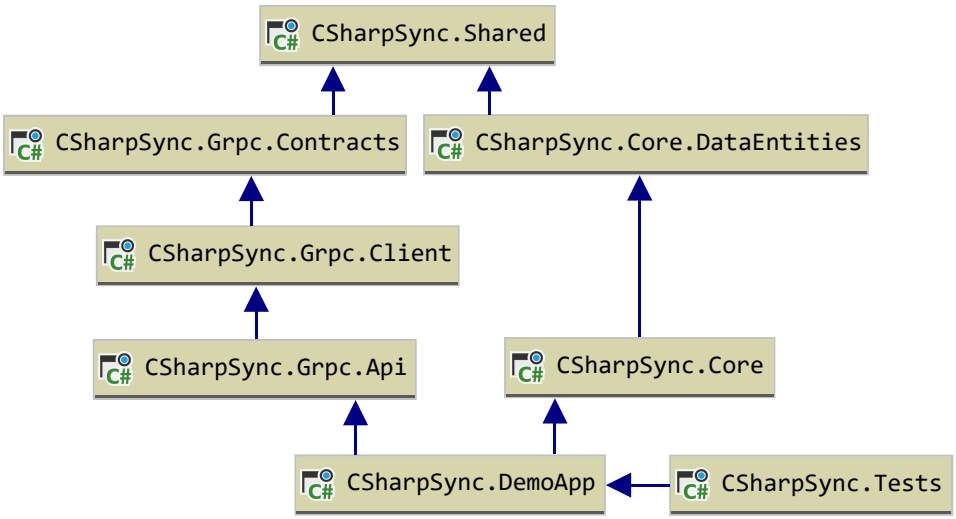
Kapitola 4

Realizace

V kapitole Realizace je popsán průběh implementace synchronizačního frameworku, včetně jeho výsledné podoby.

4.1 Struktura synchronizačního frameworku

Na následujícím obrázku 4.1 je výsledná struktura synchronizačního frameworku. Kromě *CSharpSync.DemoApp* a *CSharpSync.Tests* jsou všechny vyobrazené projekty typu Class Library. *CSharpSync.DemoApp* je typu ASP.NET Core Web Application a *CSharpSync.Tests* je testovací projekt využívající testovacího frameworku NUnit [6].



Obrázek 4.1: Struktura závislostí projektů implementovaného synchronizačního frameworku

K jakým účelům jednotlivé projekty slouží a co obsahují, je popsáno v následujícím seznamu:

- *CSharpSync.Shared* – sdílený projekt mezi všemi ostatními, obsahuje definice rozhraních a modelů.
- *CSharpSync.Grpc.Contracts* – definuje rozhraní komunikačního API uzlů skrze technologii gRPC.
- *CSharpSync.Grpc.Client* – obsahuje implementaci komunikačního klienta definovaného v *CSharpSync.Grpc.Contracts*. Zároveň obsahuje i třídy, které zjišťují a udržují informace o stavu sousedních uzlů a cest v celé síti.
- *CSharpSync.Grpc.Api* – implementuje serverovou část API definovaného v *CSharpSync.Grpc.Contracts*. Kontroluje, že přijatá zpráva je pro daný uzel, pokud ano, předává vyšší vrstvě. V opačném případě využije komunikačního klienta k přeposlání dále.
- *CSharpSync.Core.DataEntities* – definuje datové entity, které jsou uloženy v databázi. Zajišťuje vytváření spojení do databáze a poskytuje vrstvu repozitářů pro manipulaci s definovanými entitami.
- *CSharpSync.Core* – využívá poskytnutého komunikačního klienta a skrze něj zajišťuje synchronizaci s ostatními uzly v síti. Zároveň obsahuje API pro manipulaci se synchronizovanými entitami v rámci databázových transakcí skrze třídy Unit of Work.
- *CSharpSync.DemoApp* – webová aplikace která využívá synchronizačního frameworku a poskytuje GUI pro jeho vyzkoušení. Definuje ukázkovou entitu uživatele, která se synchronizuje napříč uzly v síti.
- *CSharpSync.Tests* – projekt obsahující integrační a jednotkové testy, jenž ověřují funkčnost synchronizačního frameworku.

4.2 Implementace stavového automatu za použití Inversion of Control

V aplikaci je využito principu Inversion of Control (IoC), který zajišťuje jednotlivým třídám poskytnutí závislostí, které potřebují pro svou práci [20]. Tento princip nelze použít všude, konkrétně ve třídách, které představují roli uzlu. Třídy představující roli uzlu využívají návrhové vzoru state (stav) [20], ale jelikož stavový automat zobrazený na obrázku 3.1 obsahuje cykly, IoC nedokáže z důvodu těchto cyklických závislostí vytvořit instance tříd reprezentující stav uzlu. Tento problém byl vyřešen pomocí využití návrhového vzoru

Factory method [20]. Instanciování jednotlivých stavů pak neprovádí IoC kontejner ale daná Factory metoda. Factory metody vyžadují ve svých argumentech závislosti, které jsou potřebné pro vytvoření instance daného stavu. Závislosti jsou stavům předány konstruktorem, čímž dochází k odstranění cyklických závislostí při konstrukci instancí tříd stavů.

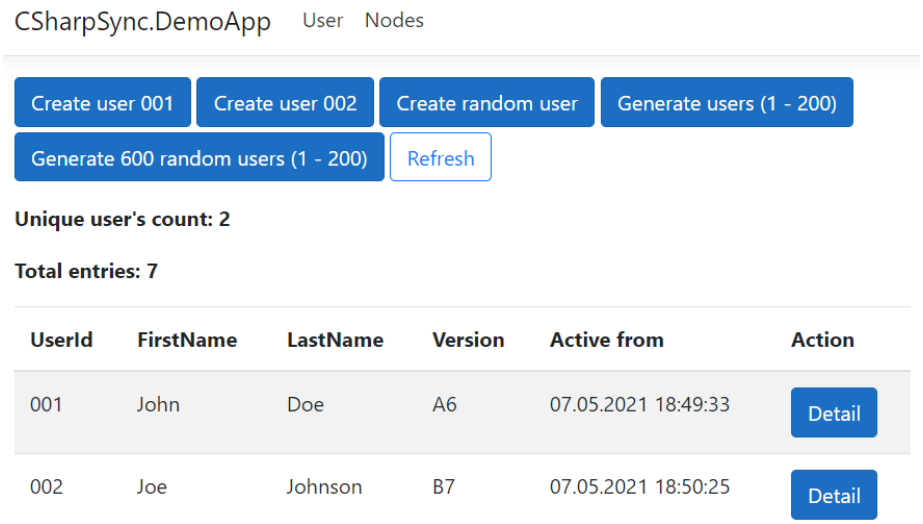
4.3 Manipulace se synchronizovanými entitami

Při manipulování s entitami dochází k uzamčení všech verzí v rámci daného typu entity. Je tak z důvodu možné manipulace se stromem synchronních rodičů. Pokud by došlo k současné modifikaci stromu více procesy, výsledný strom by nemusel být v korektním stavu. Tomu to případu musí být zabráněno, jelikož by byla porušena celková integrita dat.

Při založení verze se vytváří index záznamu v logu. Tento index je vytvářen v rámci aplikace a následně ukládán do databáze. Z tohoto důvodu nelze spustit více instancí aplikace oproti jedné databázi, protože by docházelo ke kolizím při vytváření indexu.

4.4 Implementované GUI

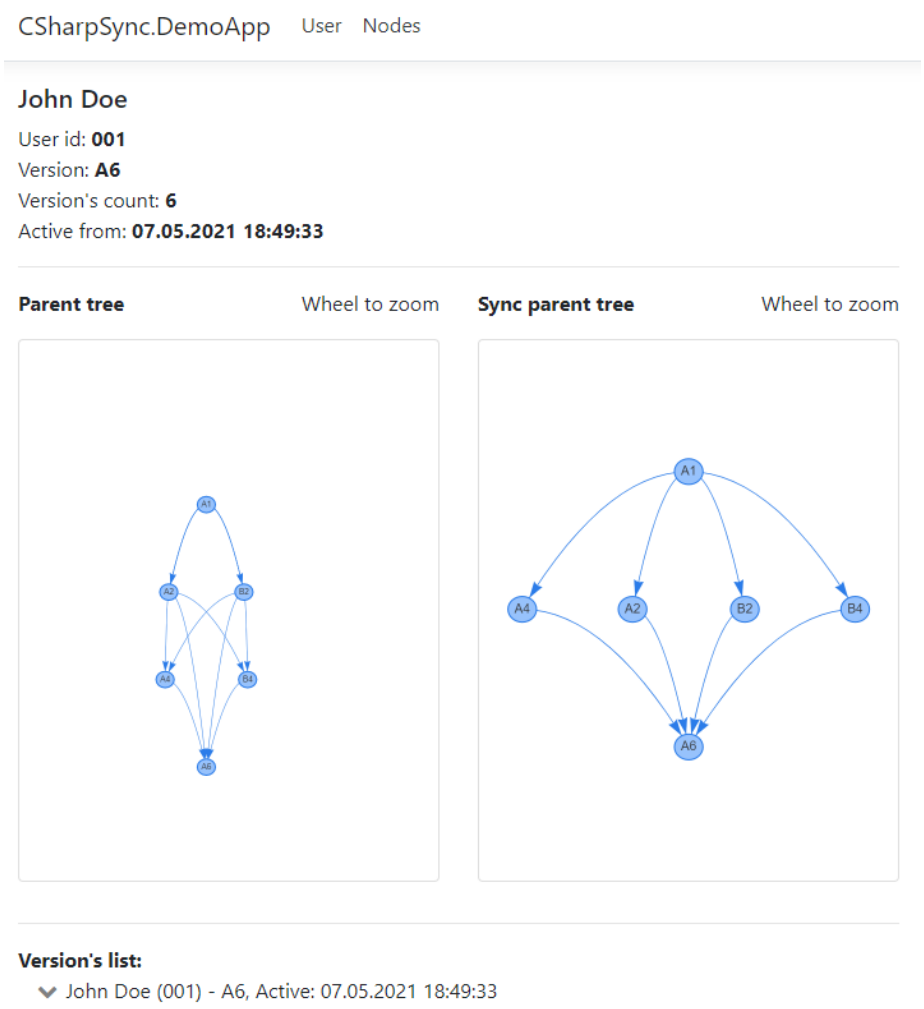
Demo aplikace obsahuje několik webových stránek na kterých lze zobrazit synchronizovanou entitu – uživatele nebo informace o sousedních uzlech. Na domovské obrazovce (obrázek 4.2) se nachází seznam uživatelů.



Obrázek 4.2: Domovská obrazovka se seznamem uživatelů

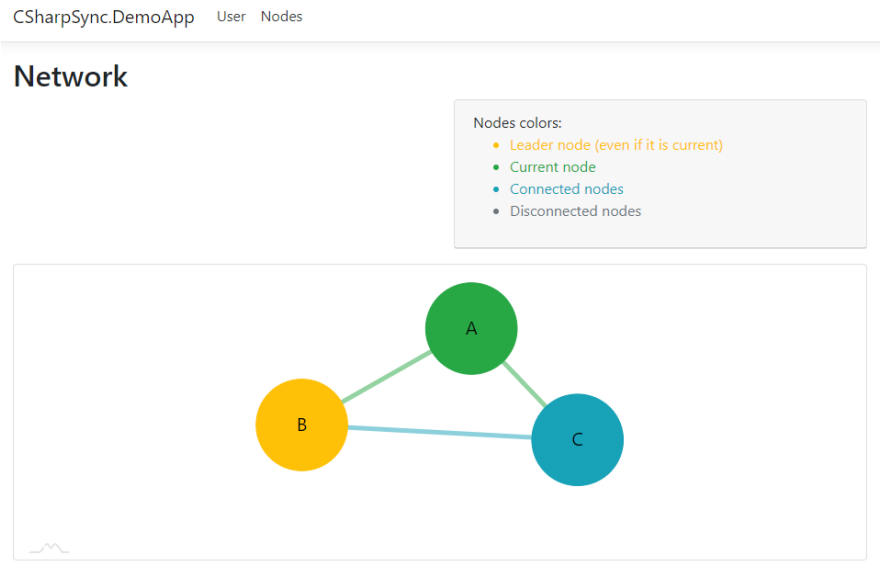
V záhlaví stránky se nacházejí tlačítka pro generování uživatelů. Níže se nachází počet unikátních uživatelů a celkový počet verzí. V tabulce jsou uvedeny základní informace o uživatelích – unikátní identifikátor uživatele, jméno, přímení, název aktuální verze, datum od kdy je verze aktivní. Pod detailem se zobrazí podrobnější informace o verzích daného uživatele.

Detail uživatele je na obrázku 4.3. Na začátku obrazovky se nacházejí opět základní informace o uživateli, které jsou následovány stromem rodičů a stromem synchronních rodičů. Ze stromu rodičů lze vidět, že verze A4 a B4 byly v konfliktu dvou sloučených verzí (3.3.6). Tím se dostaly do konfliktu i verze, jejichž konflikty řešily – A2, B2, viz strom synchronních rodičů. Tento konflikt vyřešila verze A, která je v tuto chvíli aktuální. Po najetí myší na jednotlivé verze se zobrazí podrobnější informace o nich. Ve spodní části obrazovky se nachází hierarchie verzí v podobě seznamu.



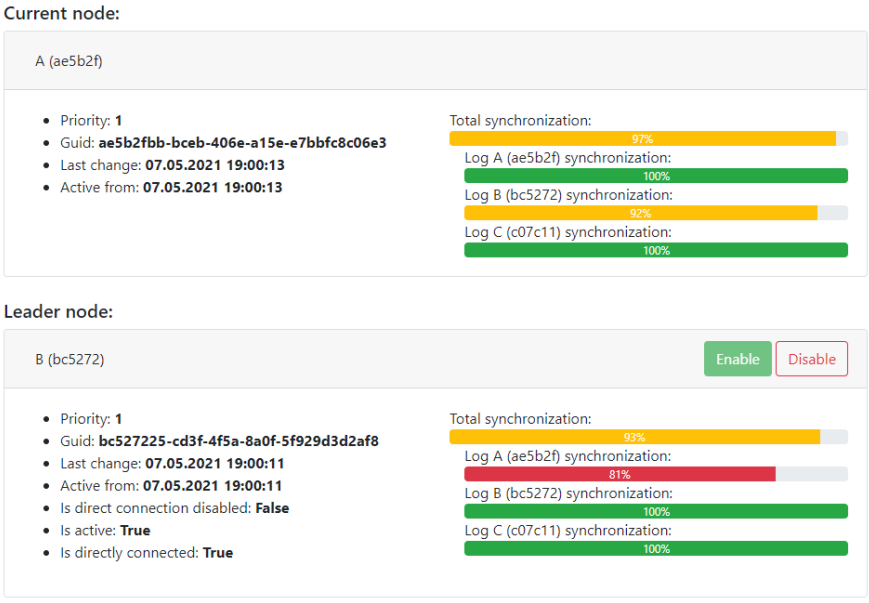
Obrázek 4.3: Obrazovka detailu uživatele

První část obrazovky uzlů 4.4 obsahuje vizualizaci aktuální topologie propojení jednotlivých uzlů z pohledu daného uzlu. Jednotlivé uzly jsou barevně odlišeny z pohledu konektivity a role.



Obrázek 4.4: Obrazovka vizualizace sítě a detailu uzlů – 1. část

V druhé části obrazovky uzlů 4.5 se nacházejí detailní informace o jednotlivých uzlech. Nejprve je zobrazen aktuální uzel, následně vůdce a nakonec všechny ostatní uzly. U každého z nich je zobrazen procentuální stav synchronizace multi-logu. Po najetí myší na jednotlivé logy se zobrazí přesný počet záznamů.



Obrázek 4.5: Obrazovka vizualizace sítě a detailu uzlů – 2. část

4.5 Komunikační API uzlů

Definice komunikační API uzlů implementované pomocí knihovny gRPC se spolu s dokumentací nachází na obrázku 4.6. Skládá se ze tří metod, které odpovídají návrhu z kapitoly 3.8.1. *SendAppendEntriesAsync* slouží k odesílání záznamů, *SendRequestVoteAsync* slouží pro získání hlasu při volbách, *SendPingAsync* slouží pro detekování, zda je uzel napřímo dostupný a k získání jeho směrovací tabulky.

```
namespace CSharpSync.Grpc.Contracts
{
    /// <summary>
    /// API for synchronization between nodes of cluster
    /// </summary>
    [ServiceContract]
    public interface ISyncService
    {
        /// <summary>
        /// Method for sending entries or heartbeats to another node.
        /// </summary>
        /// <param name="request">Request object holding logs to append and sender multi-log state</param>
        /// <returns>Multilog-state and leader GUID</returns>
        [OperationContract]
        Task<AppendEntriesReplyMessageContract> SendAppendEntriesAsync(AppendEntriesMessageContract request);

        /// <summary>
        /// Method for requesting votes from other nodes inside the elections process.
        /// </summary>
        /// <param name="request">Request object holding sender multi-log state and term</param>
        /// <returns>Election status</returns>
        [OperationContract]
        Task<RequestVoteReplyMessageContract> SendRequestVoteAsync(RequestVoteMessageContract request);

        /// <summary>
        /// Method for detecting if another node is accessible directly.
        /// </summary>
        /// <param name="request">Request object holding source node GUID.</param>
        /// <returns>Node's routing table</returns>
        [OperationContract]
        Task<PingReplyContract> SendPingAsync(PingRequestContract request);
    }
}
```

Obrázek 4.6: Definice komunikačního API uzlů

Kapitola 5

Testování

Jelikož synchronizační framework nelze použít sám o sobě, byly všechny testy prováděny nad demo aplikací, ve které docházelo k synchronizaci uživatelů. Nejdříve byly provedeny jednotkové testy klíčových metod pro manipulaci se záznamy, následně se provedly integrační testy z pohledu samostatného uzlu, a nakonec integrační testy z pohledu clusteru. K testování bylo využito testovacího frameworku NUnit [6] a balíčků pro integrační testování na platformě .NET [38]. Všechny testy se nacházejí v projektu *CSharpSync.Tests.UnitTests*.

5.1 Testovací prostředí

Testovacím prostředím, kde probíhalo vykonávání všech testů, byl notebook s operačním systémem Windows 10 Home (Build 19041) s 16 GB operační paměti typu LPDDR4X, procesorem AMD Ryzen 7 4700U a SSD úložištěm o kapacitě 512 GB.

Všechny testy nejprve spouští instanci demo aplikace. V případě jednotkových a integračních testů uzlů se jedná o testovací server vytvořený skrze knihovnu *Microsoft.AspNetCore.TestHost* a v případě integračních testů clusteru se jedná o více klasických instancí demo aplikace [38]. Každá instance demo aplikace využívá vlastní databázi, která se nachází v databázovém systému PostgreSQL [23].

5.2 Jednotkové testy

Jmenný prostor *CSharpSync.Tests.UnitTests* obsahuje jednotkové testy. Po spuštění instance demo aplikace se z IoC kontejneru získá třída, skrze kterou bude prováděno testování. Po vykonání operací se kontroluje stav požadovaných objektů. Jednotkové testy se zaměřují na otestování manipulace se synchronizovanými entitami a multi-logem, dále pak testují správnou detekci konfliktů popsaných v kapitole 3.3 a jejich korektní řešení popsané v 3.4.

5.3 Integrační testy

Integrační testy byly zaměřeny na celkový průchod dotazu aplikací. V obou sadách integračních testů bylo použito provolávání komunikačního API jednotlivých uzlů vystaveného pomocí gRPC.

5.3.1 Integrační testy uzlu

V *CSharpSync.Tests.NodeIntegrationTests* jsou obsaženy testy, které ověřují korektnost chování stavů vůdce a následovníka. V obou případech se vytvoří pro testovací server gRPC klient, skrze kterého se zprávy posílají [24]. Využíváno je volání základních API metod *AppendEntry* a *RequestVote*. Po zavolání metody *AppendEntry* se ověřuje odpověď, zda její parametry odpovídají očekávaným hodnotám, dle chování, které bylo navrženo v kapitole 3. Typicky se jedná o stav multi-logu či jaký vůdce je zvolen. V případě metody *RequestVote* se kontroluje, zda byl uzlem udělen hlas či byly volby zakázány.

Při testování stavu následovník je skrze konfiguraci prodloužen čas na vypršení limitu, do kterého se musí ozvat vůdce. Tím je zajištěno, že v průběhu testu nedojde ke změně do stavu kandidát. Testuje se základní přidávání nových verzí od různých uzlů, duplicitní přidání záznamů a případně chybové stavy.

Při testování stavu vůdce je limit pro jeho zvolení naopak skrze konfiguraci zkrácen. Před každým testem se tedy čeká, než je uzel zvolen do této role. Následně se provádí testy typu korektního přidání nových záznamů, reakce na heartbeat zprávy od jiných vůdců apod.

V obou případech se také testují reakce na žádost o hlas od jiných uzlů v různých situacích, např. kdy žadatel má více či méně záznamů v logu oproti testovanému uzlu, rozdílné číslo termu či prioritu.

5.3.2 Integrační testy clusteru

Při testování cluster uzlů bylo použito 5 samostatných instancí demo aplikace, které všechny běžely na stejném fyzickém stroji. Po spuštění testu se vyčkalo na zvolení jednoho vůdce a propagaci této informace všem uzlům.

První sada integračních testů clusteru se zabývá propagací jednotlivých záznamů mezi uzly. Zda při vytvoření záznamu na uzlu s rolí vůdce je propagován všem ostatním uzlům, či zda je při vytvoření na uzlu s rolí následovník záznam propagován nejprve vůdci a následně ostatním uzlům. Pro vytváření záznamů bylo využito toho, že v aplikaci je zaveden návrhový vzor Unit of Work. Byla tedy použita implementace UoW pro manipulaci s entitou uživatele, kdy na jednom uzlu se skrze ni uživatelé vytvářeli a na ostatních uzlech se naopak kontrolovalo, že obsahují stejné nově vytvořené uživatele.

Další sady testů se zabývají testováním synchronizace při výpadku spojení mezi dvěma uzly. Zda budou zprávy přeposílány přes jiný uzel a zda po rozpojení sítě a následnému opětovnému propojení dojde k synchronizaci chybějících záznamů na všech uzlech. Pro účely simulace výpadků byl do aplikace zaveden parametr *IsDirectConnectionDisabled*, který dokáže simulovat výpadek spojení mezi dvěma uzly. Při výpadku přímé cesty, uzel nemohl odesílat cílovému uzlu data napřímo, ale přes jiný uzel. Rozdělení sítě do více částí bylo opět realizováno pomocí tohoto parametru. Pro každý uzel z jedné sítě byla vyřazena přímá cesta do všech uzlů z ostatních sítí.

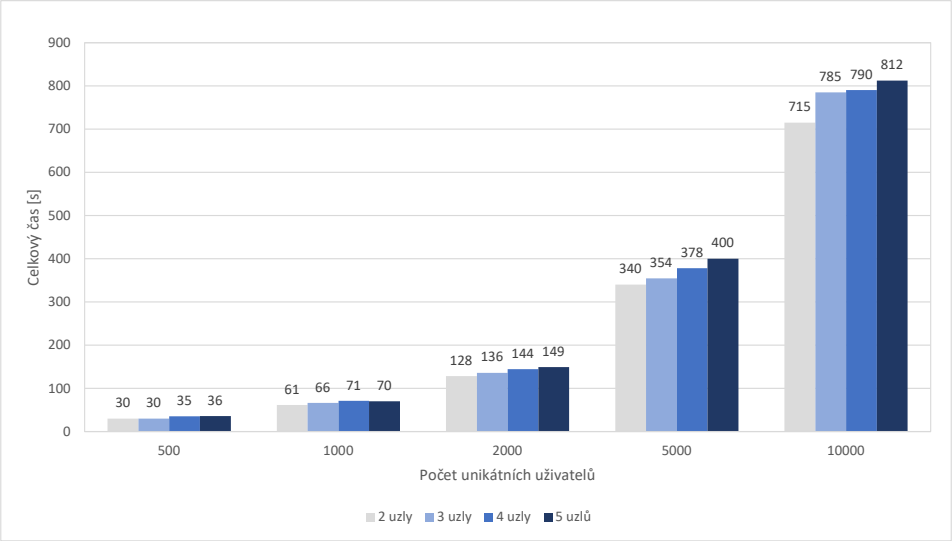
Poslední testovací sada se zabývá synchronizací uživatelů mezi uzly v případě vytváření více konfliktních verzí. Testuje se tady, zda dojde k automatickému vyřešení konfliktů a změny se vypropagují na všechny uzly.

Pro ověření synchronizace byl vytvořen ještě samostatný parametrizovaný test o velikosti clusteru 4-5 uzlů, kde se postupně zvyšoval počet unikátních uživatelů od 500 až po 10 000. Po vygenerování uživatelů se vyčkalo na jejich synchronizace mezi všemi uzly. Následně každý uzel začal generovat dvojnásobný počet verzí nových uživatelů oproti počtu unikátních uživatelů. Mezitím docházelo k simulaci rozpadům sítě na různé části a jejich následnému opravení. Na konci testu se ověřilo, že všechny uzly obsahují stejné záznamy.

5.4 Benchmark

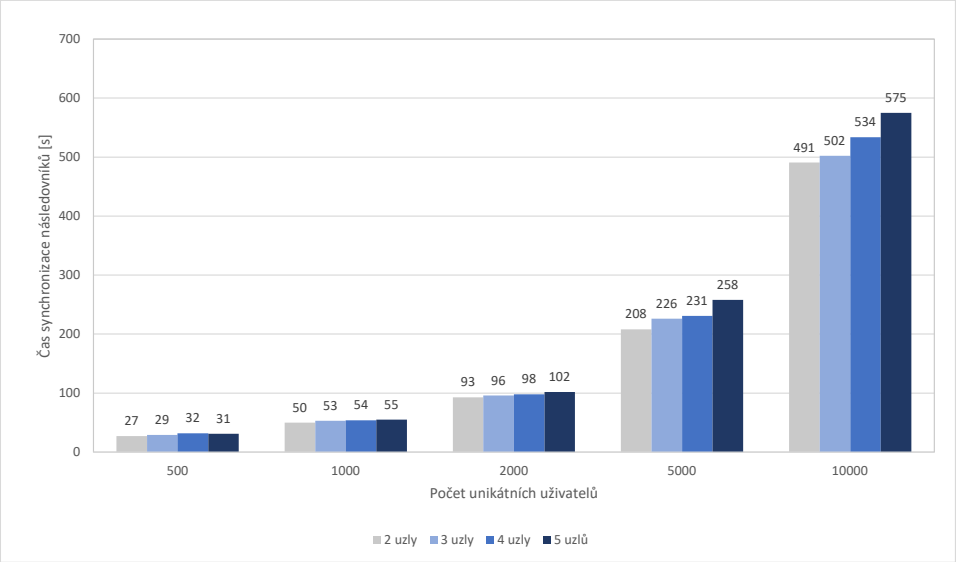
Kromě samotných testů bylo provedeno měření výkonu clusteru z pohledu, za jakou dobu se v určitých případech dokáže dostat do synchronizovaného stavu. Každé měření bylo opakováno 5x a poté byly hodnoty měření zprůměrovány.

Prvním případem je, kdy je síť po celou dobu měření propojena a po zvolení vůdce, se v síti začnou generovat uživatelé. Uživatele generuje pouze vůdce a jelikož je síť propojena, rovnou je odesílá svým následovníkům. Měření bylo prováděno pro sítě o velikosti 2, 3, 4 a 5 uzlů. Pro každou velikost sítě se měřilo s postupně se zvyšujícím počtem unikátních uživatelů od 500 po 10 000. Po vygenerování unikátních uživatelů došlo ke generování jejich nových verzí. Vygeneroval se dvojnásobný počet verzí, než bylo uživatelů, tedy každý uživatel měl v průměru 3 verze celkem. Výsledek tohoto měření je zaznamenán v grafu 5.1. Z grafu lze vidět, že větší počet uzlů v síti má zanedbatelný vliv na dobu potřebnou k synchronizace celé sítě. Dále lze vidět, že při zvětšení počtu uživatelů 2-2,5x se čas nutný pro synchronizaci zvýšil 2-3x.



Obrázek 5.1: Graf času synchronizace uživatelů při propojené síti

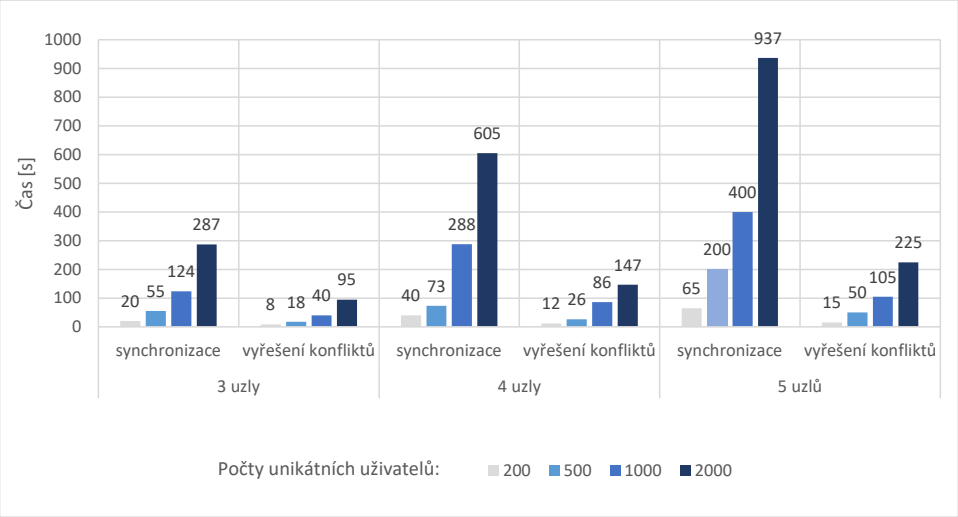
Druhým případem měření výkonu byla situace, kdy vůdce vygeneroval unikátní uživatele a následně jejich verze. Ale až poté je začal posílat svým následovníkům – síť byla tedy po dobu generování uživatelů rozpojena. V tomto případě se neměřila celková doba synchronizace i s generováním uživatelů, nýbrž jenom čas potřebný pro propagaci záznamů od vůdce k následovníkům. Velikosti sítí a počty uživatelů byly použity stejné jako u předcházejícího případu. Výsledky měření (obrázek 5.2) jsou obdobné jako u předchozího případu – přidáním uzlů se čas výrazně nezvýší a se zvětšením počtu uživatelů na dvojnásobek se i čas minimálně zdvojnásobí.



Obrázek 5.2: Graf času synchronizace uživatelů po připojení uzlů do sítě

V posledním případě se nejprve vygeneruje daný počet uživatelů a následně všechny uzly náhodně generují dvojnásobný počet verzí. Prvním měřeným údajem je čas potřebný pro vygenerování verzí a jejich synchronizování skrze celou síť. Druhým údajem je doba nutná po synchronizaci k dořešení všech konfliktů a distribuci těchto řešení na všechny uzly v síti. Z výsledků vyobrazených v grafu 5.3 lze pozorovat veliký nárůst času potřebného k synchronizace se zvyšujícím se počtem uzlů v síti. To je zapříčiněno také zvedajícím se celkovým počtem záznamů, které je nutné synchronizovat, jelikož každý uzel generuje záznamy. Roste i čas potřebný k vyřešení konfliktů, protože se jich vyskytne více, opět kvůli dalšímu uzlu generujícímu záznamy.

Všechny instance běžely na stejném fyzickém stroji, tudíž docházelo ke sdílení hardwarových prostředků. To se mohlo nejvíce projevit v posledním měření, kdy všechny uzly generovaly data a zároveň je přijímaly od ostatních.



Obrázek 5.3: Graf času synchronizace a řešení konfliktů při generování uživatelů všemi uzly

5.5 Shrnutí výsledků testování

Testování ověřilo, že implementovaný synchronizační framework je funkční a chová se dle specifikace z kapitoly 3 Návrh. Nejprve se ověřily základní operace jako jsou ukládání a získávání záznamů. Následně se testovala schopnost synchronizace jednotlivých uzlů mezi sebou a to i při situacích, kdy docházelo k výpadkům spojení mezi uzly, či rozdělení sítě na více částí.

Při měření výkonu bylo zjištěno, že při generování záznamů jedním uzlem dochází k lineárnímu zvětšování potřebného času k synchronizaci s lineárně zvětšujícím se počtem záznamů, které je nutné synchronizovat. Na tento případ nemá vliv ani zvětšování počtu uzlů v síti. Potřebný čas se neparně zvyšuje, ale lineární trend zůstává stejný. Každý uzel je za této situace schopen synchronizovat 15-20 záznamů za sekundu na stroji s parametry uvedenými v kapitole 5.1 Testovací prostředí. Lineární nárůst času je z důvodu sériového zpracování jednotlivých záznamů. Případnou paralelizací zpracování by se dal zvětšit počet záznamů, které je uzel schopen zpracovat během jedné sekundy.

V případě, kdy každý uzel generuje nové záznamy, dochází ke zvyšování času potřebného k synchronizaci uzlů. Je tak z důvodu většího množství záznamů, které jsou vytvářeny po řešení konfliktů. Zároveň zpracovávání konfliktů je časově náročnější než přidávání nekonfliktních verzí, jelikož je nutné procházet stromovou strukturu a nastavovat verzím nové synchronní rodiče. Tím dochází i ke snížení počtu záznamů, které je schopen uzel synchronizovat za sekundu.

Kapitola 6

Závěr

Cílem této diplomové práce bylo navrhnout a implementovat synchronizační framework pro distribuovaný systém, který bude schopen trvale ukládat data i když nebudou dostupné ostatní uzly systému. Právě z důvodu dynamické změny clusteru, bylo v systému využito principu optimistické replikace dat. Z definice CAP teorému vyplývá, že takový systém je náchylný na vzniku konfliktů mezi upravovanými daty.

V rámci analýzy byly porovnány dva konsenzus algoritmy – Paxos [30] a RAFT [39]. Následně v rámci kapitoly 3 Návrh byla navržena modifikace algoritmu RAFT, aby mohl být využit pro optimistickou replikaci dat mezi uzly. Pro adaptování algoritmu do dynamického prostředí, kde může docházet k rozpadům sítě, bylo nutné modifikovat původní log uzlů na multi-log, který zajistí trvalého uložení záznamů na všech uzlech. Jednotlivé záznamy v multi-logu jsou verzovány aby bylo možné detekovat a řešit případné konflikty, vzniklé v dynamickém prostředí. V rámci modifikací byly navrženy změny volebního mechanismu tak, aby mohl být v případě rozpadu sítě zvolen vůdce pro každou její část. Vzniklé konflikty jsou detekovány díky verzování synchronizovaných entit a následně je umožněno, je řešit programově a to jak na syntaktické, tak na sémantické úrovni, případně přenechat řešení uživateli aplikace.

Navrhovaná architektura synchronizačního frameworku od sebe odděluje logiku synchronizace od přímého přístupu do databáze a také od komunikačního API mezi uzly.

Dle navržených úprav byl implementován synchronizační framework, který se dokáže vyrovnat s odpojením jednotlivých instancí a jejich opětovným připojením. V případě detekce výpadku přímého spojení s jiným uzlem, dochází k přeposílání zpráv skrze ostatní uzly. Dále umožňuje dynamicky přidávat i odebírat uzly, aniž by muselo docházet k restartování ostatních

uzlů v síti. K frameworku byla implementována demo aplikace, která slouží k vyzkoušení frameworku a demonstraci jeho použití. Zároveň byla tato demo aplikace využita pro účely testování.

Testování probíhalo zejména z pohledu případů, kdy při synchronizaci entit docházelo k rozpadům sítě na různé části. Při měření výkonu bylo zjištěno, že při synchronizaci vůdce s nově připojenými uzly čas lineární narůstá s lineárně zvětšujícím se množství záznamů, které je nutné synchronizovat. Přidání více uzlů do sítě má na tento případ minimální vliv. Při vytváření konfliktních verzí dochází k nárůstu času potřebného k synchronizaci, jelikož detekce konfliktů a jejich následné řešení je časově náročnější.

V rámci implementovaného frameworku se podařilo splnit všechny funkční a nefunkční požadavky vycházející ze zadání diplomové práce.

6.1 Návrhy na rozšíření

V průběhu návrhu a implementace se ukázalo, že je možné dále rozšiřovat funkcionalitu synchronizačního frameworku nad rámec stanovených požadavků. Rozšířit lze framework o detekci stejných entit – případ kdy dva uzly založí stejnou novou entitu, která je ale v databázi vedena jako dvě odlišné. Konkrétní mechanismus pro detekci stejných entit by se následně musel implementovat pro každý typ synchronizované entity zvlášť, stejně jako proces sloučení těchto verzí. S tímto rozšířením se pojí i optimalizace zamykacího mechanismu tak, aby bylo umožněno zpracovávat paralelně více synchronizovaných entit stejného typu, které na sobě ale nejsou závislé.

Dalším rozšířením by mohlo být přidání pesimistického synchronizačního módu, který by byl volitelný pro synchronizované entity.

Jedním z možných vylepšení je efektivita ukládání a transportu dat, kdy místo celých verzí by se přenášely a ukládaly pouze rozdíly oproti předchozí verzi.

Vylepšena by mohla být i volba cest v síti v případě, kdy síť není zcela propojena a je nutné zprávy přeposílat přes jiné uzly. K tomu by mohly posloužit již existující směrovací protokoly jako jsou např. RIP nebo OSPF [17].

Příloha A

Literatura

[1] *APACHE LICENSE, VERSION 2.0* [online]. Apache Software Foundation, 2004. [cit. 6.05.2021]. Dostupné z: https://www.apache.org/licenses/LICENSE-2.0.

[2] *Apache Cassandra* [online]. [cit. 14.12.2020]. Dostupné z: https://cassandra.apache.org/.

[3] *CVS—Concurrent Versions System v1.11.23* [online]. 2019. [cit. 28.04.2021]. Dostupné z: https://www.gnu.org/software/trans-coord/manual/cvs/cvs.htm.

[4] *About - Git* [online]. 2021. [cit. 28.04.2021]. Dostupné z: https://git-scm.com/about/distributed.

[5] *NHibernate - Relational Persistence for Idiomatic .NET* [online]. 2021. [cit. 6.05.2021]. Dostupné z: https://nhibernate.info/doc/nhibernate-reference/index.html.

[6] *NUnit.org* [online]. 2021. [cit. 10.01.2021]. Dostupné z: https://nunit.org/.

[7] *Protocol Buffers* [online]. Google, 2021. [cit. 6.05.2021]. Dostupné z: https://developers.google.com/protocol-buffers.

[8] A TECHNICAL OVERVIEW OF RIAK KV ENTERPRISE. Dostupné z: https://riak.com/content/uploads/2016/05/RiakKV-Enterprise-Technical-Overview-6page.pdf.

[9] AHMED, R. What Is Git ? – Explore A Distributed Version Control Tool, 2020. Dostupné z: https://www.edureka.co/blog/what-is-git/.

[10] BELSHE, M. – PEON, R. – THOMSON, M. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015. Dostupné z: https://rfc-editor.org/rfc/rfc7540.txt.

[11] Benz, S. – Pedone, F. Elastic Paxos: A Dynamic Atomic Multicast Protocol. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, s. 2157–2164, 2017. doi: 10.1109/ICDCS.2017.84.

[12] BERNSTEIN, P. A. – GOODMAN, N. The Failure and Recovery Problem for Replicated Databases. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, s. 114–122, New York, NY, USA, 1983. Association for Computing Machinery. doi: 10.1145/800221.806714. Dostupné z: `https://doi.org/10.1145/800221.806714`. ISBN 0897911105.

[13] BREWER, E. A. Towards robust distributed systems. In *PODC*, 7, s. 343477–343502. Portland, OR, 2000.

[14] CASTRO, M. – LISKOV, B. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, s. 173–186, USA, 1999. USENIX Association. ISBN 1880446391.

[15] CHACON, S. – STRAUB, B. *Pro Git*. Springer Nature, second edition, 2014. ISBN 1484200772.

[16] CLOW, J. – JIANG, Z. A byzantine fault tolerant raft, 2017. Dostupné z: `http://www.scs.stanford.edu/17au-cs244b/labs/projects/clow_jiang.pdf`.

[17] CONER, D. E. *Internetworking with TCP/IP: principles, protocols, and architecture*. Prentice Hall, 1995. ISBN 0132278367.

[18] KLERK, J. *gRPC on HTTP/2 Engineering a Robust, High-performance Protocol* [online]. 2018. [cit. 6.05.2021]. Dostupné z: `https://grpc.io/blog/grpc-on-http2/`.

[19] FISCHER, M. J. – LYNCH, N. A. – PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*. April 1985, 32, 2, s. 374–382. ISSN 0004-5411. doi: 10.1145/3149.214121. Dostupné z: `https://doi.org/10.1145/3149.214121`.

[20] GAMMA, E. et al. Elements of reusable object-oriented software. *Reading: Addison-Wesley*. 1995.

[21] GILBERT, S. – LYNCH, N. Perspectives on the CAP Theorem. *Computer*. 2012, 45, 2, s. 30–36. doi: 10.1109/MC.2011.389.

[22] GILBERT, S. – LYNCH, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*. June 2002, 33, 2, s. 51–59. ISSN 0163-5700. doi: 10.1145/564585.564601. Dostupné z: `https://doi.org/10.1145/564585.564601`.

[23] GROUP, T. P. G. D. *PostgreSQL 13.2 Documentation* [online]. 2021. [cit. 7.05.2021]. Dostupné z: `https://www.postgresql.org/files/documentation/pdf/13/postgresql-13-A4.pdf`.

[24] AUTHORS. *gRPC – A high-performance, open source universal RPC framework* [online]. 2021. [cit. 6.05.2021]. Dostupné z: https://grpc.io/.

[25] HAERDER, T. – REUTER, A. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* December 1983, 15, 4, s. 287–317. ISSN 0360-0300. doi: 10.1145/289.291. Dostupné z: https://doi.org/10.1145/289.291.

[26] HOWARD, H. ARC: analysis of Raft consensus. Technical report, University of Cambridge, Computer Laboratory, 2014. Dostupné z: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf.

[27] JAKOB, M. Konsensus a Algoritmu Raft. 2017. Dostupné z: https://cw.fel.cvut.cz/b172/_media/courses/b4b36pdv/lectures/l11_konsensus_a_raft.pdf.

[28] KUMAR, V. et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. USA : Benjamin-Cummings Publishing Co., Inc., 2nd edition, 2003. ISBN 0201648652.

[29] LAMPORT, L. Fast paxos. *Distributed Computing*. 2006, 19, 2, s. 79–103.

[30] LAMPORT, L. The Part-Time Parliament. *ACM Trans. Comput. Syst.* May 1998, 16, 2, s. 133–169. ISSN 0734-2071. doi: 10.1145/279227.279229. Dostupné z: https://doi.org/10.1145/279227.279229.

[31] LAMPORT, L. Generalized consensus and Paxos. 2005.

[32] LAMPORT, L. – SHOSTAK, R. – PEASE, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* July 1982, 4, 3, s. 382–401. ISSN 0164-0925. doi: 10.1145/357172.357176. Dostupné z: https://doi.org/10.1145/357172.357176.

[33] LAMPORT, L. et al. Paxos made simple. *ACM Sigact News*. 2001, 32, 4, s. 18–25. Dostupné z: http://lamport.azurewebsites.net/pubs/paxossimple.pdf.

[34] LOUKAS, D. *C# 8 and .NET Core 3.1 Recipes*. PACKT Publishing Limited, 2nd edition, 2020. Dostupné z: https://www.packtpub.com/product/c-8-and-net-core-3-1-recipes-2nd-editionsecond-edition-video/9781838986728. ISBN 9781838986728.

[35] LYNCH, N. A. *Distributed Algorithms*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1996. ISBN 9780080504704.

[36] MICROSOFT. *What is a DLL* [online]. 2020. [cit. 5.05.2021]. Dostupné z: https://docs.microsoft.com/en-US/troubleshoot/windows-client/deployment/dynamic-link-library.

[37] MONGODB. *The most popular database for modern apps* [online]. [cit. 14.12.2020]. Dostupné z: https://www.mongodb.com/.

[38] NELSON, J. C. – SMITH, S. – TIL, J. *Integration tests in ASP.NET Core* [online]. 2020. [cit. 10.01.2021]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/test/integration-tests?view=aspnetcore-5.0.

[39] ONGARO, D. *Consensus: Bridging theory and practice*. PhD thesis, Stanford University, 2014.

[40] ONGARO, D. – OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, s. 305–320, USA, 2014. USENIX Association. Dostupné z: https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14. ISBN 9781931971102.

[41] ÖZSU, T. – VALDURIEZ, P. Principles of Distributed Database Systems, 2019.

[42] PRITCHETT, D. BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability. *Queue*. May 2008, 6, 3, s. 48–55. ISSN 1542-7730. doi: 10.1145/1394127.1394128. Dostupné z: https://doi.org/10.1145/1394127.1394128.

[43] SAITO, Y. – SHAPIRO, M. Optimistic Replication. 2003. Dostupné z: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2003-60.pdf.

[44] SAITO, Y. – SHAPIRO, M. Replication: optimistic approaches. Technical report, Technical Report HPL-2002-33, HP Laboratories Palo Alto, 2002.

[45] SCHNEIDER, F. B. The state machine approach: A tutorial. In *Fault-tolerant distributed computing*. Springer, 1990. s. 18–41.

[46] SCHNEIDER, F. B. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* December 1990, 22, 4, s. 299–319. ISSN 0360-0300. doi: 10.1145/98163.98167. Dostupné z: https://doi.org/10.1145/98163.98167.

[47] VAN STEEN, M. – TANENBAUM, A. S. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 3.01 edition, 2017. ISBN 9781543057386.

[48] ZYCHLA, W. *How to implement XPO's Soft Delete pattern for NHibernate* [online]. 2012. [cit. 1.05.2021]. Dostupné z: https://www.wiktorzychla.com/2012/06/how-to-implement-xpos-soft-delete.html.

Příloha B

Seznam použitých zkratek

ACID	Atomicity, consistency, isolation, durability
AP	Availability, partition tolerance
API	Application Programming Interface
BASE	Basic availability, soft state, eventual consistency
CAP	Consistency, availability, partition tolerance
CA	Consistency, availability
CP	Consistency, partition tolerance
DLL	Dynamic-link library
ER	Entity-relationship
FLP	Fisher, Lynch, and Paterson Impossibility
gRPC	gRPC Remote Procedure Calls
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
IoC	Inversion of Control
IP	Internet protocol
ORM	Object relational mapping
RDBMS	Relational Database Management System
SQL	Structured Query Language

TTL	Time To Live
UoW	Unit of Work
URL	Uniform Resource Locator

Příloha C

Příručka použití synchronizačního frameworku

Tento návod popisuje jak použít synchronizační frameworku v aplikaci. Návod je demonstrován za použití implementované demo aplikace, která je typu ASP.NET Core Web App.

Ukázka C.1 zobrazuje, jakým způsobem zaregistrovat do IoC kontejneru konfigurace pro jádro frameworku a pro komunikační gRPC API. Konfigurace se získávají z *appsettings.json* souboru a následně pomocí extension metody *Configure* registrují. Dále je vyobrazena registrace obou komponent. Metoda *RegisterCSharpSyncService* slouží k zaregistrování jádra synchronizačního frameworku, zatímco metoda *RegisterCSharpSyncGrpcApi* slouží k registraci gRPC API. V případě použití gRPC API je také nutné zaregistrovat službu *GrpcSyncService* jako endpoint aplikace – ukázka C.2.

```
public void ConfigureServices(IServiceCollection
    services)
{
    ...
    var clusterConfig = m_config.GetSection("Cluster");
    services.Configure<ClusterConfiguration>(clusterConfig);

    var syncApiConfig = m_config.GetSection("SyncApi");
    services.Configure<SyncApiConfiguration>(syncApiConfig);

    services.RegisterCSharpSyncService();
    services.RegisterCSharpSyncGrpcApi();

}
```

Listing C.1: Registrace gRPC API a synchronizačního frameworku do IoC kontejneru

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env, IContainer container)
{
    ...
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGrpcService<GrpcSyncService>();
    });
}
```

Listing C.2: Registrace GrpcSyncService jako endpoint aplikace

Framework je vázán na knihovnu NHibernate, proto je nutné provést registraci třídy splňující rozhraní *ISessionFactory* do IoC kontejneru [5]. Příklad této registrace za použití nadstavby FluentNHiberante se nachází ve třídě *NHibernateInstaller* obsažené v demo aplikaci. V každém případě je nutné zaregistrovat pro NHibernate mapování databázových entit, které se nachází ve jmenném prostoru *CSharpSync.Core.DataEntities.Entities*.

Dalším krokem je vytvoření synchronizovaných entit. Každá entita určená k synchronizaci musí dědit od třídy *SyncEntityBase* a musí mít přiřazen atribut *[Serializable]*, příkladem této entity je *UserEntity*. Pro každou entitu pak musí být vytvořen Unit of Work, který bude zajišťovat její zpracování dat v transakci – viz *UserUoW*, tento UoW musí dědit od abstraktní třídy *SyncEntityBaseUoW*. Nutný je také repozitář zajišťující její uložení do databáze – viz UserRepository. Posledním vyžadovanou částí je třída implementující rozhraní *IMerger*, jenž bude zajišťovat automatické řešení konfliktů mezi verzemi. Pokud není automatické požadováno, musí být tato třída stejně zaregistrováno, ale místo, aby metoda pro řešení konfliktů vracela novou sloučenou verzi řešící konflikt, bude vracet hodnotu *null*.

Všechny tyto komponenty je nutné opět zaregistrovat do IoC kontejneru a to jako typ *singleton*. Registrace je vyobrazena na ukázce C.3. Konkrétní Unit of work je nutné zaregistrovat pod rozhraním *ISyncEntityUoW*. Třída s automatickým řešením konfliktů musí být registrována pod rozhraním *IMerger*.

```
private static void
    RegisterUserSyncEntityComponents(this
    IServiceCollection services)
{
  services.AddSingleton<UserUoW>();
  services.AddSingleton<ISyncEntityUoW>(provider =>
     provider.GetRequiredService<UserUoW>());

  services.AddSingleton<IMerger, UserMerger>();

  services.AddSingleton<UserRepository>();
}
```

Listing C.3: Registrace tříd sloužících k manipaluci s danou synchronizovanou entitou

Konfigurace pro gRPC API (*SyncApiConfiguration*) vyžaduje pouze parametr *PingTimeout* – číslo v milisekundách vyjadřující, jak často se má posílat ping dotaz na ostatní uzly v síti. Parametry konfigurace pro samotné jádro frameworku (*ClusterConfiguration*) jsou následující:

- *SynchronizationEnabled* – vynutí/zapnutí synchronizace.
- *CurrentNode* – informace o aktuálním uzlu.
  - *Guid* – unikátní identifikátor uzlu ve formě GUID.
  - *Priority* – priorita uzlu.
  - *Name* – název uzlu pro snadnější identifikaci.
  - *Address* – komunikační adresa uzlu (IP nebo URL).
- *ElectionsTimeout* – čas v milisekundách udávající minimální dobu, než budou vyhlášeny nové volby v případě že nepřijde heartbeat zpráva od vůdce. Skutečný čas bude náhodně vygenerován z rozsahu toho minima po dvojnásobek této hodnoty aby se minimalizovala situace, kdy dva uzly vyhlásí volby ve stejnou chvíli.
- *Nodes* – seznam sousedních uzlů. Vždy je vyžadován alespoň jeden soused, aby se uzel mohl připojit do sítě. U uzlů musí být uvedena jejich komunikační adresa (IP nebo URL) a také jejich unikátní identifikátor (GUID).

Příloha D

Použité knihovny v implementaci synchronizačního frameworku

V následující tabulce D.1 se nacházejí .NET knihovny použité při implementaci. Tyto knihovny mohou mít vlastní závislosti na další knihovny. Tyto závislosti jsou u každé verze dohledatelné na webu `nuget.org`.

Název	Verze	Licence
AutoMapper	10.1.1	MIT
AutoMapper.Extensions.Microsoft.DependencyInjection	8.1.1	MIT
DryIoc.Microsoft.DependencyInjection	5.1.0	MIT
FluentNHibernate	3.1.0	BSD
Grpc.Net.Client	2.34.0	Apache 2.0
Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation	5.0.5	Apache 2.0
Microsoft.AspNetCore.TestHost	5.0.5	Apache 2.0
Microsoft.Extensions.DependencyInjection.Abstractions	5.0.0	MIT
Microsoft.Extensions.Hosting.Abstractions	5.0.0	MIT
Microsoft.Extensions.Logging.Abstractions	5.0.0	MIT
Microsoft.Extensions.Options	5.0.0	MIT
Microsoft.NET.Test.Sdk	16.9.4	MSLT
NHibernate	5.3.8	LGPL v2.1
NLog	4.7.9	BSD
NLog.Web.AspNetCore	4.12.0	BSD
Npgsql	5.0.3	PostgreSQL
NUnit	3.12.0	MIT
NUnit3TestAdapter	3.16.1	MIT
protobuf-net.Grpc	1.0.140	Apache 2.0
protobuf-net.Grpc.AspNetCore	1.0.140	Apache 2.0

Tabulka D.1: Použité .NET knihovny v implementaci synchronizačního frameworku

Tabulka D.2 obsahuje knihovny využité při tvorbě GUI. Tyto knihovny mohou mít závislosti na další knihovny. Tyto závislosti lze dohledat na webu `npmjs.com`, odkud se dají knihovny stáhnout.

Název	Verze	Licence
Amcharts	4.10.18	Free amCharts license
Bootstrap	4.3.1	MIT
Font Awesome	5.15.3	CC BY 4.0
jQuery	3.5.1	MIT
jQuery Validation Plugin	1.17.0	MIT
jQuery Unobtrusive Validation	3.2.11	Apache 2.0
vis-network	9.0.4	Apache 2.0

Tabulka D.2: Použité knihovny při implementaci GUI demo aplikace