VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

RÁMEC PRO TVORBU APLIKACÍ S PODPOROU PEER-TO-PEER SPOLUPRÁCE

APPLICATION DEVELOPMENT FRAMEWORK FOR PEER-TO-PEER COLLABORATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JAN HRDINA**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Hrdina Jan, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Rámec pro tvorbu aplikací s podporou peer-to-peer spolupráce**
Application Development Framework for Peer-to-Peer Collaboration
Kategorie: Informační systémy

Zadání:

1. Nastudujte a srovnejte existující přístupy pro replikaci stavu ve webových aplikacích pro kolaborativní tvorbu obsahu. Nastudujte existující přístupy pro řešení peer-to-peer komunikace mezi aplikacemi.
2. Pro vybraný algoritmus replikace navrhněte podporu řízení přístupu k datům a aplikačně-specifického řešení konfliktů (automatického nebo uživatelského). Navrhněte univerzální aplikační rámec, který umožní tvorbu distribuovaných aplikací pro tvorbu obsahu s podporou řízení přístupu, řešení konfliktů a možností práce offline.
3. Po konzultaci s vedoucím, navržený aplikační rámec implementujte a jeho použití demonstrujte vytvořením kolaborativního editoru stromových struktur.
4. Výsledky zdokumentujte, zhodnoťte a zveřejněte jako open-source.

Literatura:

- KSHEMKALYANI, Ajay D. a Mukesh SINGHAL. Distributed Computing: Principles, Algorithms, and Systems. Cambridge: Cambridge University Press, 2008. ISBN 9780521876346.
- LORETO, Salvatore a Simon Pietro ROMANO. Real-Time Communication with WebRTC: Peer-To-Peer in the Browser. Sebastopol: O'Reilly Media, Incorporated, 2014. ISBN 9781449371876.
- KLEPPMANN, Martin a Alastair R. BERESFORD. A Conflict-Free Replicated JSON Datatype. DOI: 10.1109/TPDS.2017.2697382. ISBN 1045-9219. [http://ieeexplore.ieee.org/document/7909007/]

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Práce se zabývá návrhem a implementací aplikačního rámce pro tvorbu kolaborativních webových editorů, které umožňují peer-to-peer spolupráci v realném čase. V práci jsou shrnuty existující přístupy pro replikaci dat, z nichž je pro další použití jako nejvhodnější vybrána implementace CRDT (conflict-free replicated data type) pro JSON od M. Kleppmanna. Pomocí výsledného rámce může být vytvářený obsah bezpečně sdílen uvnitř skupin protějšků, kde každému členovi je možné nastavit jiná oprávnění. Pro navázání spojení a komunikaci P2P jsou navrženy a implementovány vlastní komunikační protokoly postavené na WebRTC, WebSocket a WebCrypto. Rámec umožňuje řešení konfliktů a samostatnou práci i bez připojení k internetu. Pro konzistentní uživatelský zážitek je součástí knihovna s prvky uživatelského rozhraní pro správu přátel, skupin a další časté úkony. Rámec je implementován s využitím funkcionálních návrhových vzorů realizovaných v jazyce ReasonML. Funkčnost výsledku je ověřena vytvořením ukázkové aplikace editoru myšlenkových map.

Abstract

The thesis deals with the design and implementation of the application framework for the creation of collaborative web editors that enable peer-to-peer collaboration in real time. The thesis summarizes existing approaches for data replication, from which M. Kleppmann's CRDT (conflict-free replicated data type) for JSON is chosen as the most suitable. Using the resulting framework, the created content can be safely shared in groups of peers, where each member can be assigned different permissions. Own communication protocols based on WebRTC, WebSocket and WebCrypto are designed and implemented for P2P connection establishment and subsequent communication. The framework allows to resolve conflicts and work independently without an Internet connection. For a consistent user experience, the library includes a set of user interface elements for managing friends, groups, and other common tasks. The framework is implemented using functional design patterns implemented in the ReasonML language. The functionality of the result is verified by creating an example application of the mind map editor.

Klíčová slova

rámec, webový vývoj, peer-to-peer, distribuované systémy, spolupráce, funkcionální programování, autentizace, autorizace, konflikty, ReasonML, WebRTC, CRDT

Keywords

framework, web development, peer-to-peer, distributed systems, collaboration, functional programming, authentication, authorization, conflicts, ReasonML, WebRTC, CRDT

Citace

HRDINA, Jan. *Rámec pro tvorbu aplikací s podporou peer-to-peer spolupráce*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Rámec pro tvorbu aplikací s podporou peer-to-peer spolupráce

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Hrdina
16. května 2019

Poděkování

Děkuji vedoucímu práce RNDr. Markovi Rychlému, Ph.D. za trpělivost při konzultacích, užitečné připomínky a průběžnou zpětnou vazbu.

Obsah

Kapitola 1

Úvod

Je stále snadnější narazit na uživatele, kteří uchovávají vlastní soukromá data v cloudových službách. Dobrou motivací k tomu je zajištění dostupnosti napříč všemi jejich zařízeními nebo zajištění možnosti spolupráce s ostatními. I když je toto počínání pohodlné, použití centralizovaných úložišť má svá rizika. Uživatel je nucen spoléhat na to, že poskytovatel bude ctít jeho soukromí, že se postará o zabezpečení nahraného obsahu a že bude dlouhodobě zajišťovat dostupnost svých služeb. Podle mého názoru je zbytečné takto riskovat ztrátu nebo prozrazení cenných osobních dat. Věřím, že technologie peer-to-peer mohou do značné míry eliminovat tato rizika a navrátit tak aplikační data zpět do rukou uživatele díky využití pouze jeho vlastních zařízení jakožto distribuovaného úložiště.

Moderní aplikace na tvorbu obsahu jsou čím dál tím složitější, což je z velké části důsledkem narůstajících požadavků, které uživatelé na software kladou: aplikace musí být rychlá, musí umožňovat spolupráci s ostatními účastníky v reálném čase, data musí být stále k dispozici. Uživatel bere jako samozřejmost, že může aplikaci používat i offline a automaticky očekává, že když mimo dosah Wi-Fi udělá nějaké změny, vše se po opětovném připojení k internetu projeví i u jeho protějšků. Konkurenceschopná P2P aplikace musí splňovat vše uvedené.

Problém je, že vývoj P2P aplikací je náročný. Nestačí mít pouze datový typ, který podporuje slučování změn. U každé takové aplikace je třeba vyřešit adresování uzlů, řešení konfliktů, zabezpečení, protokoly šíření změn, je třeba vytvořit GUI pro nastavení oprávnění a mnoho dalšího. P2P aplikace se tak pro běžného programátora stávají neekonomické, protože může ztratit obrovské množství času, než se vůbec dostane k vlastní doménové logice.

Cílem této práce je vytvořit rámec pro tvorbu kolaborativních webových aplikací, který nabídne hotové řešení výše uvedených problémů a sníží tuto vstupní bariéru na minimum. V ideálním případě by potom doba implementace webového P2P editoru neměla být o moc větší, než vytvoření editoru obyčejného, pracujícího pouze offline s lokálními soubory.

V kapitole 2 jsou stručně rozebrané hlavní principy, kterých se práce drží, po čemž následuje průzkum dostupných technologií a technik, které jsou relevantní pro navazující části. Kapitola 3 rozebírá a srovnává existující algoritmy pro replikaci dat v kolaborativních systémech. Tyto algoritmy jsou stěžejní pro zamezení nekonzistence mezi různými replikami.

Nejdůležitější částí práce je kapitola 4, která se zabývá návrhem mechanismů fungování rámce, vlastních abstrakcí, protokolů, zpřístupnění řešení konfliktů, návrhy GUI a dalším. Kapitola 5 potom popisuje všechny důležité a nějakým způsobem zajímavé části implemen-

tace. Díky použití funkcionálního jazyka nabízí ukázku netradičních technik návrhu, které se v mnoha ohledech liší od přístupů známých z běžných imperativních jazyků.

Konečně kapitola 6 zachycuje návrh a implementaci ukázkového projektu – editoru myšlenkových map, který demonstruje použití rámce.

1.1 Terminologie

V textu se vyskytují některé termíny a obraty, jejichž význam je vhodné předem definovat.

Moderní prohlížeč V textu se několikrát odkazuji na dostupnost funkcí „ve všech moderních prohlížečích“. Protože práce je v tomto ohledu značně náročná, moderními prohlížeči v tomto kontextu myslím *Firefox*, *Chrome*, *Operu* a *Safari* v aktuálních verzích. Pokud není uvedeno jinak, neřadím mezi ně Microsoft Edge a Internet Explorer.

Cílová (koncová) aplikace Cílovou nebo koncovou aplikací je myšlena aplikace pro kolaborativní tvorbu obsahu, která využívá můj rámec nebo jeho dílčí knihovny.

Uživatel Uživatelem je myšlen člověk používající *koncovou aplikaci*, u kterého se nepředpokládají znalosti programování ani aplikačního rozhraní (API) rámce.

Programátor, vývojář Pokud není jinak upřesněno, jedná se o člověka, který vyvíjí *koncovou aplikaci*, používá veřejné API rámce nebo jeho dílčí komponenty.

Mobilní zařízení Mobilním zařízením je myšlen chytrý telefon, tablet nebo průmyslový terminál s operačním systémem Android nebo jiným běžně dostupným.

Diff Protože jsem nenašel ustálený překlad anglického *diff*, dovolil jsem si použít jeho počeštěnou variantu skloňovanou podle vzoru hrad. Diffem je míněn buďto *seznam rozdílů* mezi dvěma porovnávanými objekty, nebo samotný *proces porovnávání*, tj. proces vytváření tohoto seznamu.

Binding Protože se mi nepodařilo najít český ekvivalent, používám počeštěnou variantu *binding* skloňovanou podle vzoru hrad. Bindingem je míněna knihovna nebo obecně aplikační vrstva, která zpřístupňuje jinou knihovnu napsanou v jazyce a pro použití v jazyce b.

Kapitola 2

Analýza, specifikace

Cílem této kapitoly je popsat hlavní principy a funkce, o jejichž realizaci tato práce usiluje, a následně popsat technologie a principy potřebné pro další části práce.

Jednou z důležitých prostudovaných oblastí jsou přenosové technologie dostupné ve webovém prohlížeči. Sekce 2.2 je stručně shrnuje a vybírá kandidáty vhodné pro tuto práci. Nejdůležitější z nich, WebRTC, je blíže popsána v sekci 2.3. Následující sekce 2.4 a 2.5 analyzují dostupná řešení pro kryptografii a lokální ukládání dat.

Jak bude blíže ukázáno v kapitole 5, práce využívá k implementaci funkcionální jazyk ReasonML. Protože staticky typované funkcionální jazyky nejsou ve webovém vývoji příliš rozšířené a v mnoha ohledech se liší od konvenčních přístupů, v sekci 2.7 jsou popsána jeho specifika a sekce 2.8 potom popisuje vybrané funkcionální návrhové vzory, které jsou dále použity.

Do oblasti analýzy patří také průzkum existujících řešení a rozbor algoritmů pro replikaci dat. Tyto části jsou umístěné zvlášť v navazující kapitole 3.

2.1 Hlavní principy

Hlavním výstupem této práce je sada knihoven (rámec) pro webové prostředí, která umožňuje uživatelům v cílové aplikaci snadno vytvářet vlastní decentralizované skupiny pro replikaci obsahu mezi zařízeními bez nutnosti centrálního úložiště.

Mezi její hlavní principy patří:

- **Automatická synchronizace datového modelu** – Rámec zajišťuje automatickou replikaci dat koncové aplikace. Daty koncové aplikace rozumíme například aktuální konfiguraci a zejména reprezentaci uživatelského obsahu.

  Replikace dat mezi uzly probíhá bez nutnosti zásahu hostitelské aplikace a uživatele. Z jejich pohledu se v ideálním případě práce s replikovaným datovým modelem takřka vůbec neliší od práce s modelem jednouživatelské aplikace.

- **Vytvářený obsah se ukládá pouze v koncových uzlech** – Rámec vytváří abstrakci distribuovaného úložiště pro aplikace, kde obsah je uložen pouze v zařízeních uživatele a nikoliv na centrálním serveru, kde se stává zbytečným cílem ke zneužití.

- **Pouze koncové uzly mají přístup k vytvářenému obsahu** – Pokud je z technických důvodů v omezené míře nutná přítomnost *centrálního prvku* nebo jiného prostředníka (např. pro ustavení spojení), není nutné mu důvěřovat, tzn. obsah samotný

se přes něj buďto vůbec nepřenáší, nebo musí zůstat v šifrované, prostředníkovi nečitelné podobě.

- **Pokud možno, obsah se přenáší k protějšku nejkratší cestou** – Pokud jsou například dvě zařízení ve stejné síti a pokud to konfigurace sítě umožňuje, pro jejich synchronizaci není nutné přenášet vytvářená data přes server ve veřejném internetu, ale automaticky se využije rychlého lokálního spojení.

- **Široká podpora zařízení** – Rámec je platformě nezávislý a funguje na co možná největším počtu běžně dostupných zařízení, tj. nejen na počítačích, ale například i na tabletech a chytrých telefonech.

- **Podpora práce offline** – Vytvořenou koncovou aplikaci je po první návštěvě možné opakovaně otevírat i bez připojení k internetu. Uživatel může offline vytvářet obsah, který je po opětovném připojení automaticky přenesen k protějškům.

- **Uživatel může používat vlastní implementaci koncového uzlu** – Pokud má uživatel pochybnosti o hostované webové aplikaci pro koncový uzel, může si ji ručně sestavit ze zdrojových kódů a bez větších omezení používat pro spolupráci s ostatními tuto lokální verzi.

  V extrémním případě vývojář vůbec nemusí zajišťovat (a platit) hosting webové aplikace, jak jinak bývá obvyklé v uzavřeném vývoji aplikací klient-server, ale může vše nechat na komunitě. P2P tak může být zajímavým řešením pro nízkorozpočtové open-source projekty, které si nemohou dovolit financovat infrastrukturu.

2.2 Přenosové technologie pro P2P v prostředí prohlížeče

Ve webovém prohlížeči je obecně možné vybírat hned z několika technologií a přenosových protokolů, kterými může běžící webová aplikace komunikovat se svým okolím:

- **AJAX** je technologie, která umožňuje asynchronní výměnu dat s webovým serverem.

  Komunikace má vždy formát dotazu od klienta a následující odpovědi od serveru. Protokol tak není příliš vhodný pro komunikaci, kterou by měl častěji iniciovat server. Při každém dotazu se vytváří nové TCP spojení, což zvyšuje režii a odezvu. Ve WAN protokol vyžaduje HTTP server s veřejnou IP adresou a otevřeným portem.

  Protože je běžnou praxí, že klienti veřejné IP adresy nemají a navíc není možné, aby webová aplikace otevírala libovolné porty, AJAX není vhodný pro P2P komunikaci.

- **WebSockets** na rozdíl od AJAX umožňuje dlouhodobě udržovat duplexní TCP spojení. To jej činí zajímavým řešením pro zasílání asynchronních notifikací od serveru. Po ustavení spojení následující komunikace vyžaduje jen minimální režii.

  Protokol opět vyžaduje server s veřejnou IP adresou, takže spojení prohlížeč-prohlížeč není možné. Protokol se v našem případě nicméně dobře hodí pro *komunikaci koncového uzlu se signalizačním serverem* (viz sekce 4.6), kde mimo jiné zprostředkovává notifikace klientům o připojování/odpojování jednotlivých protějšků.

- **Bluetooth** je standard pro bezdrátovou komunikaci mezi zařízeními na krátkou vzdálenost. Běžně se využívá například k připojování bezdrátových reproduktorů, chytrých

hodinek, klávesnic apod. nebo třeba k přenosu menších souborů mezi mobilními zařízeními.

I když protokol kvůli nižší přenosové rychlosti není vhodným kandidátem pro přenos objemných změn uživatelského obsahu, mohl by se hodit například ve fázi *ustavení P2P spojení* mezi dvěma mobilními zařízeními, pro výměnu signalizačních informací nebo pro vyjednání údajů pro nějaký rychlejší kanál (např. přímé Wi-Fi spojení).

Podpora *Web Bluetooth API* je zatím implementovaná pouze v prohlížečích postavených na projektu Chromium[1]. Specifikace W3C [48] je k 30. dubnu 2019 ve stavu konceptu.

Bluetooth Low Energy (BLE) uvedené v Bluetooth 4.0 omezuje přenosovou rychlost na 1 Mbps s tím, že je výrazně snížena spotřeba energie. BLE definuje následující role pro zařízení:

- Broadcaster – jen vysílá.
- Observer – jen přijímá.
- Peripheral – pasivní periférie, která může pouze přijímat obousměrná spojení.
- Central – může se připojovat k periferním zařízením.

Aktuální verze specifikace Web Bluetooth API podporuje připojování pouze k BLE zařízením, čímž je vyloučeno použití např. některých starších Bluetooth modulů v laptopech. Druhým zásadnějším omezením je to, že *prohlížeč v současnosti může fungovat pouze v roli Central*. Protože v BLE spojení Central-Central není možné, je tím znemožněno i přímé propojení dvou prohlížečů. Z těchto důvodů aktuálně *Bluetooth pro P2P spojení není možné použít* a v práci se jím dále nebudu zabývat.

- **NFC (Near Field Communication)** je dalším způsobem bezdrátové komunikace na krátkou vzdálenost, tentokrát však maximálně na vzdálenost několika centimetrů.

  I když teoreticky by použití pro ustavení spojení nějakým rychlejším kanálem mohlo být zajímavé, aktuální podpora *Web NFC API* [13] v prohlížečích je ještě horší než v případě Web Bluetooth API: podporu najdeme pouze v prohlížeči Chrome pro Android (testováno ve verzi 71), a to jedině po povolení experimentálních funkcí ve skrytém nastavení prohlížeče.

  Navíc, i kdyby podpora v prohlížečích byla vynikající, pořád je tu ještě překážka v podobě špatné podpory v hardware. Přítomnost NFC modulů ve vybavení chytrých telefonů stále není, na rozdíl od Bluetooth, standardem. Z těchto důvodů NFC není v práci použito.

- **WebRTC (Web Real-Time Co,mmunication)** je technologie, která umožňuje vytvořit zabezpečený peer-to-peer kanál přímo mezi prohlížeči, viz dále.

2.3 WebRTC

WebRTC je API a sada protokolů spravovaná W3C a IETF, která umožňuje prohlížečům mezi sebou komunikovat v reálném čase pomocí peer-to-peer architektury bez nutnosti instalace rozšíření třetích stran [29].

[1] Údaj k 30. dubnu 2019. Aktuální informace o podpoře Web Bluetooth: https://caniuse.com/#feat=web-bluetooth

Technologie vyžaduje centrální prvek pro ustavení spojení. Hlavním příslibem je, že po navázání spojení již v ideálním případě komunikace mezi protějšky probíhá nejkratší možnou cestou (bez prostředníka) a to zabezpečeným kanálem. Technologie je podporovaná napříč všemi moderními prohlížeči[2] (s omezeními i v Microsoft Edge). V produkčním prostředí je možné se s ní setkat například prostřednictvím webových VoIP aplikacích jako je Google Hangouts [28] nebo video chat ve Facebook Messengeru [30, 28].

Díky výše uvedenému se WebRTC jeví jako nejlepší kandidát pro realizaci hlavního P2P přenosového kanálu ve vytvářeném rámci.

2.3.1 Požadavky na infrastrukturu

I když cílem WebRTC je vytvoření P2P komunikačního kanálu, technologie si v aktuální podobě nevystačí pouze s koncovými prohlížeči, ale vyžaduje minimálně při ustavení spojení pomoc zvenčí.

Z pohledu infrastruktury je pro vytváření spojení napříč WAN sítí potřeba následující:

- STUN server,
- TURN server,
- signalizační kanál,
- koncové uzly.

Kromě výše uvedeného je nutné zajistit, aby webová stránka s aplikací byla poskytována přes HTTPS, výjimkou jsou otevřené lokální HTML soubory a doména localhost.

STUN server

STUN (Session Traversal Utilities for NAT) je univerzální serverová aplikace, která pomáhá koncovým uzlům umístěným za nějakou formou překladu adres (NAT, PAT) k tomu, aby zjistili svoji veřejnou IP adresu a otevřený port dostupný z internetu.

Provoz STUN serveru není náročný na systémové prostředky a jedna běžící instance může být využívána různými aplikacemi. Není tak raritou najít společnosti, které nabízí vlastní STUN servery k bezplatnému použití. Jednou z nich je například Google.

Specifikace protokolu stojícího za STUN je popsaná v RFC3489 [38] a RFC5389 [32].

TURN server

TURN (Traversal Using Relay NAT) je, podobně jako STUN, serverová aplikace nezávislá na koncové aplikaci. Umožňuje přeposílání obsahu a slouží zpravidla jako záložní řešení pro situace, kdy nebylo možné zajistit stabilní veřejnou IP adresu a port klienta.

Provoz TURN je z principu náročnější na systémové prostředky, protože přes server běží všechna komunikace mezi uzly. Ze stejného důvodu se jen výjimečně dají sehnat TURN servery s bezplatným přístupem. Jedním z mála je například `numb.viagenie.ca`, který po registraci nabízí bezplatné testovací STUN a TURN servery s omezenou propustností. Pro produkční nasazení je nicméně vhodné kvůli zajištění maximální rychlosti vytvořit vlastní server postavený např. na některé z dostupných open-source implementací[3] nebo zvážit nějakého placeného poskytovatele.

[2] Pro aktuální informace o podpoře prohlížečů viz `https://caniuse.com/#feat=rtcpeerconnection`.
[3] např. `https://github.com/coturn/coturn`

Základní specifikaci TURN je možné najít v RFC5766 [31], rozšíření pro IPv6 potom v RFC6156 [12].

Signalizační kanál

Signalizační kanál slouží pro výměnu informací potřebných pro ustavení spojení. Standard WebRTC nijak nespecifikuje, jakým způsobem by měl být realizován, protože konkrétní potřeby jednotlivých koncových aplikací se mohou lišit.

WebRTC API v koncových uzlech generuje signalizační zprávy ve formátu SDP (Session Description Protocol) a je čistě na programátorovi, jakým způsobem zajistí jejich doručení k protějšku. Je jedno, jestli budou přeneseny vytištěné jako text na papíře, přes NFC nebo nějakým konvenčním způsobem, například pomocí HTTP a WebSockets prostřednictvím signalizačního serveru. Návrh a implementace signalizačního serveru vytvořeného pro potřeby našeho rámce jsou popsány v sekcích 4.5 a 5.11.

Popis sezení reprezentuje nejdůležitější informace, které musí být vyměněny s protějškem. Specifikuje transportní informace, typ média, formát a další přidružené parametry konfigurace pro ustavení cesty. [29]

Koncové uzly

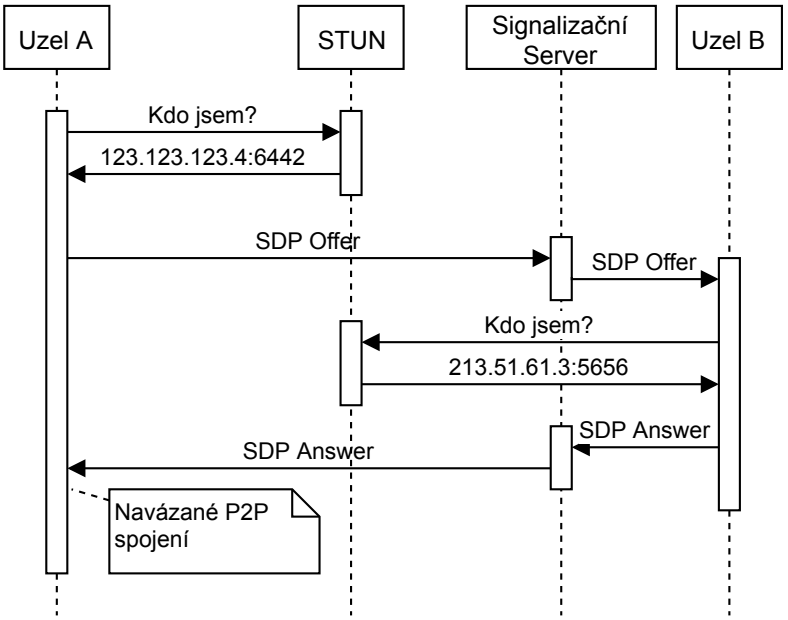
Koncovými uzly jsou v kontextu WebRTC myšleny webové aplikace využívající javascriptové API WebRTC.

2.3.2 Navázání spojení

Typický postup navázání spojení vypadá následovně:

1. Uzel A (initiator) se pokusí pomocí STUN serveru zjistit svoji veřejnou IP adresu a port.
2. Uzel A vytvoří zprávu SDP Offer s popisem jeho umístění a dalšími údaji pro vytvoření sezení.
3. Zpráva SDP Offer je přenesena signalizačním kanálem k protějškovi B.
4. Uzel B (acceptor) se po obdržení SDP Offer pokusí pomocí STUN serveru zjistit svoji veřejnou IP adresu a port.
5. Uzel B vytvoří zprávu SDP Answer s popisem jeho umístění a dalšími údaji pro vytvoření sezení.
6. Zpráva SDP Answer je přenesena signalizačním kanálem k uzlu A.
7. Pokud jsou vhodné podmínky, je ustaveno přímé spojení mezi protějšky, v opačném případě je navázáno spojení pomocí relay TURN serveru.

Zjednodušený příklad navázání spojení je znázorněn v sekvenčním diagramu na obr. 2.1.



Obrázek 2.1: Navázání spojení přes WebRTC

2.4 Kryptografie

Pro návrh rámce a protokolů je třeba zajistit přístup ke kryptografickým funkcím, zejména ke spolehlivé implementaci symetrických a asymetrických šifrovacích algoritmů.

Kryptografická primitiva je třeba zpřístupnit ve dvou odlišných prostředích:

- ve webovém prohlížeči pro potřeby knihovny koncového uzlu (impl. v Reasonu/JavaScriptu) a
- na nativním signalizačním serveru (impl. v Reasonu/OCamlu).

Co se týče **webového prostředí**, k práci s kryptografickými funkcemi je k dispozici *Web Cryptography API*, které je dnes dostupné ve všech moderních prohlížečích[4].

Aplikační rozhraní nabízí metody pro

- generování kryptograficky silných pseudonáhodných čísel,
- generování klíčů,
- podepisování a verifikaci podpisů,
- šifrování a dešifrování dat,
- výpočet hašů (digest),
- exportování a importování klíčů do/z formátu JWK a další.

Podporováno je několik symetrických, asymetrických a hašovacích algoritmů[5]. Z pohledu práce jsou relevantní RSASSA-PKCS1-v1_5 a SHA-256.

[4] Pro aktuální informace o podpoře Web Crypto viz https://caniuse.com/#feat=cryptography.

[5] Tabulku algoritmů podporovaných aktuálním prohlížečem je možné získat prostřednictvím stránky https://diafygi.github.io/webcrypto-examples.

Při vygenerování klíče vznikne objekt, který je možné používat k podporovaným kryptografickým operacím. Z bezpečnostních důvodů jsou syrová data privátního klíče JavaScriptu skryta a neexistuje způsob jak je serializovat.

Co se týče exportování veřejného klíče, prohlížeči plošně podporovaným výstupním formátem je JSON Web Key (JWK) [26], který uchovává jednotlivé složky klíče zakódované v Base64 uvnitř JSON objektu.

Pro použití Web Crypto je vyžadováno, aby webová stránka byla poskytována přes HTTPS, výjimkou jsou otevřené lokální HTML soubory a doména localhost.

Na straně **nativního signalizačního serveru** implementovaného v Reasonu je možné vybírat z existujících kryptografických knihoven pro OCaml. Zástupců není mnoho, typicky používanou knihovnou v této oblasti je *Nocrypto*[6].

Knihovna pracuje s kryptografickými primitivy na nižší úrovni než Web Crypto a přenáší tak větší část zodpovědnosti za správné použití na programátora.

2.5 Lokální úložiště v prohlížeči

Webové prohlížeče disponují hned několika možnostmi, jak si aplikace v JavaScriptu mohou lokálně data ukládat: cookies, local storage a IndexedDB. Zatímco první dvě jmenované podporují pouze textová data s omezenou velikostí, IndexedDB nabízí kromě textových dat podporu pro ukládání numerických hodnot, celých objektů, binárních souborů, umožňuje indexování a nastavuje štědré limity velikosti.

Zvlášť důležitá je v IndexedDB možnost uložit vcelku objekt se soukromým klíčem tak, že JavaScriptu nadále zůstanou syrová data klíče skryta (viz sekce 2.4).

2.6 Technologie pro zpřístupnění aplikace offline

Jak bylo nastíněno v sekci 2.1, rámec by měl plně podporovat fungování bez dostupného připojení k webovému serveru. I když se u webových aplikací tato funkcionalita může zdát exotická, snaha zpřístupnit vybrané webové aplikace offline zde v nějaké formě existuje minimálně od roku 2007, kdy Google vytvořil rozšíření pro webové prohlížeče zvané *Google Gears* [7], které zpřístupňovalo webovým stránkám API pro cachování aplikačních zdrojů, lokální databázové úložiště, API pro manipulaci se soubory, pro geolokaci, notifikace, práci s vlákny a další [10]. Google Gears očividně posloužilo jako úspěšný inkubátor, protože všechna zmíněná funkcionalita byla časem v nějaké podobě popsána webovými standardy a nativně implementována všemi majoritními prohlížeči.

Dnešní technologie se nazývá *Service Workers* a je dostupná ve všech moderních prohlížečích vč. Microsoft Edge[7]. Možnosti jejího použití jsou obecnější než jen zajištění offline přístupu ke stránce – umožňuje běh skriptu s životním cyklem nezávislým na webové stránce, vyvolávání notifikací, synchronizaci na pozadí a další [21]. Z pohledu této práce je nejrelevantnější možnost uložit stránku pro použití offline. V praxi to vypadá tak, že uživatel nemusí nic stahovat ani instalovat, ale jednoduše navštíví webovou stránku s aplikací, jak je zvyklý. Jakmile se aplikace načte, sama na pozadí zažádá prohlížeč o trvalé uložení potřebných souborů do cache. Pokud velikost ukládaných dat nepřesáhne únosnou mez (kterou si definuje každý prohlížeč jinak), prohlížeč požadavku automaticky vyhoví. Když se

[6] Stránky knihovny Nocrypto: https://github.com/mirleft/ocaml-nocrypto

[7] Aktuální informace o podpoře Service Workers: https://caniuse.com/#feat=serviceworkers

uživatel příště ocitne bez připojení k internetu, webovou stránku s takto uloženou aplikací je možné po zadání adresy načíst a používat, přestože je zbytek internetu nedostupný.

2.7 Programovací jazyk ReasonML

ReasonML[8] (též krátce Reason) je staticky typovaný funkcionální jazyk, který se snaží svou syntaxí přiblížit syntaxi JavaScriptu a snížit tak vstupní bariéru pro vývojáře přicházející z prostředí webového a objektově orientovaného vývoje.

Jazyk sdílí vnitřní reprezentaci a sémantiku se zaběhlým funkcionálním jazykem OCaml, což mu umožňuje využívat existující překladače a obecně ekosystém, který má za sebou bezmála 20 let vývoje [3].

Reason/OCaml se při výběru kompilačního cíle neomezuje pouze na nativní strojový kód a bajtkód pro běhové prostředí OCamlu. Při využití překladače BuckleScript[9] umožňuje kompilovat i do JavaScriptu. Díky tomu je možné jeden jazyk využít například pro tvorbu nativních aplikačních serverů, nativních aplikací pro příkazový řádek, nativních aplikací s GUI, ale i pro tvorbu nativních mobilních aplikací, webového front-endu nebo aplikací pro Node.js.

Použití stejného jazyka napříč platformami přirozeně otevírá možnosti *sdílení kódu*. Mezi webovým front-endem a serverem je možné sdílet třeba datové struktury pro reprezentaci doménově-specifických dat, typové definice přenášených zpráv a jejich validátory, kodeky. Programátor pracující na webovém rozhraní tak v tomto případě získá hned při kompilaci odezvu, jestli volaný endpoint na serveru existuje, jaké parametry přijímá a nemusí spoléhat na běhové testy.

V prostředí webového prohlížeče není ReasonML zdaleka jediným jazykem, který by umožňoval vývoj javascriptových aplikací s doplněním typové kontroly. Mezi nejrozšířenější jazyky pro tyto účely patří *Flow*[10], za jehož vývojem stojí Facebook, a *TypeScript*[11], na jehož otevřený vývoj dohlíží Microsoft. Oba dva jazyky berou jako základ JavaScript (validní kód JavaScriptu je zpravidla validním kódem Flow/TypeScriptu) a do něj doplňují podporu volitelných typových anotací. Úplnost a přínos typů díky volitelné povaze často přímo závisí na ukázněnosti programátorů a tvůrců anotací pro knihovny. Dynamičnost JavaScriptu potom vytváří pro tvůrce těchto jazyků značnou výzvu při snahách o rigidní usuzování nad typy.

Reason na druhé straně veškerou sémantiku JavaScriptu podporovat nemusí, a tak může nabídnout vybrané záruky a vlastnosti vycházejících ze staticky typovaného funkcionálního jazyka:

- uživatelsky definovatelné algebraické datové typy (varianty),
- pattern matching,
- typovou inferenci,
- currying,
- vestavěnou podporu neměnných (immutable) datových struktur
- a další.

[8] Stránky jazyka Reason: `https://reasonml.github.io`
[9] Stránky překladače BuckleScript: `https://bucklescript.github.io/en`
[10] Stránky jazyka Flow: `https://flow.org`
[11] Stránky jazyka TypeScript: `https://www.typescriptlang.org`

2.8 Funkcionální návrhové vzory pro architekturu aplikací

Funkcionální návrhové vzory se těší narůstající oblibě ve webovém vývoji. JavaScript již od verze ECMAScript 5.1 (cca rok 2011) podporuje manipulaci s poli pomocí funkcí `map`, `filter` a `reduce`[12]; React – v době psaní práce druhá nejpoužívanější knihovna pro vykreslování HTML [2] – popisuje stav GUI jako čistou funkci datového modelu; na něj navazující architektura Flux[13] a její populární inkarnace Redux[14] prosazuje jednosměrný tok dat v aplikaci a popis chování programu pomocí čistých funkcí.

Následující podsekce se zaměřují na popis typicky funkcionálních návrhových vzorů používaných dále v návrhu a implementaci. I když v této práci jsou vzory realizované ve funkcionálním jazyce, s aplikací jejich myšlenek se dá setkat i v některých jazycích imperativních.

2.8.1 Neměnné objekty

Neměnné objekty (immutable objects) jsou takové objekty, které po vytvoření nemohou být upravovány. V případě, že je potřeba část objektu upravit, je nutné vytvořit novou kopii s upravenou hodnotou.

Neměnné kontejnery (seznamy, množiny, mapové prvky, ...) jsou zpravidla navržené tak, aby podporovaly *strukturální sdílení*, tj. aby po vytvoření modifikované kopie objektu nezměněné části vnitřní reprezentace mohly ukazovat na původní místa v paměti a nebylo nutné vše fyzicky kopírovat. Seznamy jsou například typicky realizované jako jednosměrně vázané seznamy, množiny a mapové typy potom jako vyvážené binární stromy[15].

Ze strukturálního sdílení vyplývá možnost *levného porovnávání* kontejnerů s malým počtem rozdílných položek. V binárním stromu uvnitř mapového prvků je například možné eliminovat procházení částí stromu pouhým porovnáním ukazatelů – pokud jsou dva ukazatele na podstromy identické, je jisté, že i obsah bude stejný. Vlastnost je v praxi používána například v populární knihovně React, kde levné porovnání staré a nové neměnné reprezentace DOMu umožňuje vypočítat nejmenší podmnožinu HTML prvků, kterou je nutné na stránce aktualizovat.

Neměnné objekty jsou méně náchylné na programátorské chyby, protože nehrozí nechtěné změny objektu sdíleného s jinými částmi programu. Tato vlastnost je obzvlášť důležitá při *asynchronním a vícevláknovém programování*.

Neměnné datové struktury jsou výchozí možností ve většině funkcionálních jazyků vč. Reasonu, čím dál tím častěji je však možné se s nimi setkat i v imperativních jazycích jako je JavaScript[16], C#[17] nebo Java[18].

2.8.2 The Elm Architecture

The Elm Architecture (architektura Elmu, TEA) je sada návrhových vzorů a principů využívaná ve funkcionálním jazyce Elm[19] pro popis stavu a chování webových aplikací.

[12] ECMAScript® 5.1 Language Specification: https://www.ecma-international.org/ecma-262/5.1
[13] Stránky architektury Flux: https://facebook.github.io/flux
[14] Stránky knihovny Redux: https://redux.js.org
[15] Implementace Map v stdlib OCaml: https://github.com/ocaml/ocaml/blob/4.07/stdlib/map.ml
[16] https://immutable-js.github.io/immutable-js/
[17] https://docs.microsoft.com/cs-cz/dotnet/api/system.collections.immutable?view=netcore-2.2
[18] https://docs.oracle.com/javase/9/core/creating-immutable-lists-sets-and-maps.htm
[19] Stránky jazyka Elm: https://elm-lang.org

Architektura se stala inspirací pro vznik mnoha klonů v různých jazycích. Příkladem mohou být třeba Redux v JavaScriptu [4] nebo bucklescript-tea[20] v Reasonu.

Architektura definuje následující prvky:

- **Model** – neměnná datová struktura, která popisuje stav jednotlivých komponent a posléze celé aplikace. Kromě zdrojových doménových dat zpravidla obsahuje i detailní reprezentaci stavu uživatelského rozhraní.
- **Message** (`msg`, zpráva) – objekt, který nese statický popis příchozí akce z vnějšího světa („kliknuto na tlačítko OK“, „přijata odpověď ‚success‘ od serveru“, „vypršel časovač s ID 4“, ...).
- **Command** (`cmd`, příkaz) – objekt, který reprezentuje jeden nebo více požadavků na vykonání akce (postranního efektu) ve vnějším světě („zašli zprávu ‚hello‘ na WebSocket server s URL `ws://example.com`“, „vygeneruj náhodné číslo“, ...).
- **Update** – čistá funkce se signaturou `(model, msg) => (model, cmd)`, která popisuje všechny změny v modelu a tedy i chování programu. Změny probíhají na základě přijetí zprávy, po kterém se funkce může rozhodnout, jestli vrátí původní nebo upravenou verzi modelu a volitelný popis postranních efektů.
- **Subscription** (`sub`, odběr) – objekt reprezentující zájem o odběr zpráv z vnějšího světa (události o pohybu myši, přicházející zprávy z websockets, tikání hodin každou sekundu, ...). Odběry komponenta může specifikovat ve funkci `subscriptions` se signaturou `model => sub`.

  Implementace jednotlivých odběrů uvnitř umožňují definovat logiku pro vytváření a destrukci pomocných objektů odběru, čímž zjednodušují práci programátorovi, který nemusí pamatovat na úklid. Protože výše uvedená funkce s odběry je vyhodnocována při každé vstupní zprávě, v případě ztráty zájmu o odběr stačí nevrátit daný popis odběru a řídící program se už sám postará o korektní úklid.
- **View** – čistá funkce se signaturou `model => html`, která převádí model na deklarativní popis HTML pro vykreslení.

Ukázka použití uvedených prvků a jejich návaznosti je zachycena ve výpisu 2.1.

Důležitým principem je, že maximum funkčnosti TEA aplikací je popsáno pomocí čistých funkcí, které se chovají předvídatelně a snadno se ladí. Křehké vedlejší efekty jsou odsunuty na okraj programu do implementací příkazů a odběrů.

Vynikajícím nástrojem při popisu funkčnosti je potom pattern-matching, který v kontextu TEA umožňuje například kontrolovat, že byly ošetřeny všechny kombinace přijaté zprávy a původního stavu modelu.

2.8.3 Reaktivní programování

Reaktivní programování je sada návrhových vzorů (observer, iterable, ...), která umožňuje pracovat se streamy asynchronních událostí pomocí podobných operací, jaké jsou používané pro manipulaci s kolekcemi dat (např. se seznamem).

[20] Stránky knihovny bucklescript-tea: https://github.com/OvermindDL1/bucklescript-tea

```
type model = {
  counter: int,
  trackCursor: bool,
};

[@bs.deriving accessors]
type msg =
  | Increment
  | Decrement
  | ToggleTracking
  | MouseMoved(int, int);

let init = () => {counter: 4, trackCursor: true};

let update = (model, msg) =>
  switch (msg) {
  | Increment => (
      {...model, counter: model.counter + 1},
      Cmd.timeout(decrement, 3000),
    )
  | Decrement => ({...model, counter: model.counter - 1}, Cmd.none)
  | ToggleTracking => ({...model, trackCursor: !model.trackCursor}, Cmd.none)
  | MouseMoved(x, y) => (model, Cmd.log3("Mouse pos:", x, y))
  };

let subscriptions = model =>
  model.trackCursor ? Sub.mouseMove(mouseMoved) : Sub.none;
```

Výpis 2.1: Ukázka návaznosti prvků The Elm Architecture (bucklescript-tea).

```
WonkaJs.interval(1000)
|> Wonka.take(7)
|> Wonka.filter((. x) => x mod 2 == 0)
|> Wonka.map((. x) => x * 2)
|> Wonka.forEach((. x) => Js.log(x));

/* Postupne vypise 4, 8, 12 */
```

Výpis 2.2: Ukázka práce se streamy pomocí knihovny Wonka.

Na poli webového vývoje je pravděpodobně nejznámější inkarnací projekt *ReactiveX*, který kromě RxJS[21] – implementace v JavaScriptu – nabízí i verze pro mnoho dalších jazyků[22]. Výčet bohužel nezahrnuje OCaml ani Reason.

Odlehčenou alternativou k výše uvedenému projektu je specifikace *Callbag* [43] od Andrého Staltze. Callbag definuje pravidla pro vytváření reaktivních komponent, které na rozdíl od ReactiveX nevyžadují žádnou podpůrnou knihovnu a při dodržení základních principu spolu mohou být libovolně kombinovány.

Komponenty mohou být následujících druhů:

- **source** – prvek, který produkuje data,
- **pullable source** – prvek, který produkuje data pouze na vyžádání,
- **sink** – prvek, který přijímá data,
- **puller sink** – prvek, který aktivně žádá zdroj o data a přijímá je,
- **operator** – prvek, který staví na ostatních prvcích (např. vystupuje zároveň jako sink i source) a aplikuje nějakou operaci (filtrování, transformaci dat apod.).

Knihovna Wonka[23] nabízí implementaci sady prvků pro Reason inspirovanou výše uvedeným standardem. Ukázka použití je vidět ve výpisu 2.2.

[21] https://rxjs.dev
[22] http://reactivex.io/languages.html
[23] https://github.com/kitten/wonka

Kapitola 3

Replikace stavu v kolaborativních aplikacích

Důležitým problémem, který je třeba vyřešit v každém kolaborativním editoru, je zajistit synchronizaci (replikaci) vytvářených dat mezi uživateli takovým způsobem, že

- účastníci mohou vidět změny ostatních ideálně ihned potom, jakmile je protějšek provede,
- všichni po nějaké době uvidí to stejné a
- nic vstup nebude ztracen.

Pokud se jeden z účastníků odpojí a pokračuje v editaci offline, je nutné zajistit, aby se jeho změny po opětovném připojení zpropagovaly k ostatním a případné konflikty se smysluplně vyřešily.

Uživatel pozoruje účinek vlastních úprav okamžitě, bez použití zámků nebo závislosti na odezvě sítě. Toto chování se nazývá *optimistická replikace* (optimistic replication, [39]).

3.1 Základní pojmy

Pro potřeby dalšího popisu je vhodné zavést následující pojmy [11]:

- **Replika** – Několik uzlů může editovat dokument z různých míst. Každý uzel má svou vlastní *repliku* kolaborativního dokumentu. Replika obsahuje prvky viditelné pro koncového uživatele, stejně jako metadata potřebná k zajištění eventuální konzistence.
- **Operace** – Když uzel aplikuje uživatelské úpravy, modifikuje vlastní repliku a vygeneruje *operace*. Vytvořené operace jsou uzlem zasílány ostatním uzlům. Operace obsahují data a metadata, která při aplikování u protějšku umožní zajistit konzistenci.

3.2 CAP teorém

CAP teorém [20, 22] říká, že není možné, aby distribuované úložiště zároveň poskytovalo více než dvě z následujících tří záruk:

- Silná konzistence *(Strong **C**onsistency)*: Při každém čtení je vrácena poslední zapsaná hodnota nebo chyba. To nezbytně znamená, že se celá síť musí vždy shodnout na jedné správné verzi.
- Dostupnost *(**A**vailability)*: Na každý požadavek je ihned odpovězeno (vždycky je možné dostat se k *nějaké* replice). Dotaz např. nemusí čekat, než se všechny repliky synchronizují po předchozím zápisu.
- Odolnost vůči rozdělení *(**P**artition-resiliance)*: Systém funguje, i když je spojení mezi uzly přerušeno, nebo se při komunikaci ztrácí zprávy.

3.3 Eventuální konzistence

Eventuální konzistence je uvolněný model konzistence, který je často používán v rozsáhlých distribuovaných systémech, ve kterých musí být zaručena dostupnost navzdory výpadkům spojení a rozdělení dat a zároveň je přípustná opožděná konzistence [5]. Z pohledu CAP teorému (sekce 3.2) jde právě o případ, kdy se vzdáváme silné konzistence, abychom získali vysokou dostupnost.

Typicky je tak dovoleno, aby se jednotlivé repliky od sebe *dočasně lišily* s tím, že mohou být eventuálně (tj. v budoucnosti, která může, ale nemusí nastat) sjednoceny [5].

3.4 Operační transformace

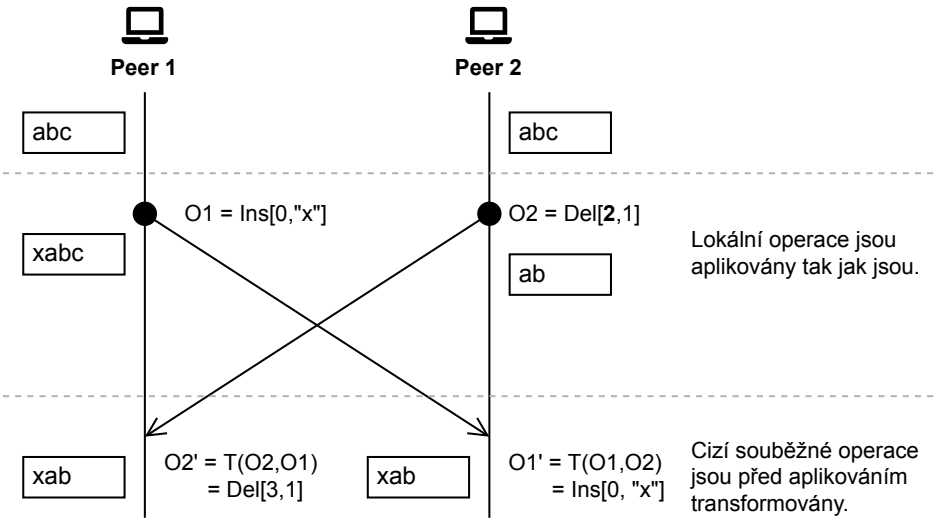
Zaběhlým mechanismem [35, 37, 45] pro spolupráci v reálném čase a zároveň jedním z prvních přístupů zmíněných v literatuře je mechanismus *operačních transformací*. Jeho první popis pochází z roku 1989 [17] a během následujících let se dočkal značné pozornosti jak mezi výzkumníky usilujícími o jeho další zdokonalení, doplnění a formalizaci, tak mezi programátory, kteří jej úspěšně nasadili v praxi [15, 8, 42].

Základní myšlenka OT pro udržování konzistence může být ilustrována na jednoduchém scénáři editace textu [44] znázorněném na obrázku 3.1. Dvě repliky se stejným počátečním řetězcem „abc“ jsou nezávisle modifikovány. Operace $O1$ vloží znak „x“ na pozici 0, operace $O2$ odstraní jeden znak na pozici 2. Aby bylo možné aplikovat $O2$ po $O1$, $O2$ musí být transformovaná vůči $O1$ – v tomto případě musí být poziční parametr inkrementován o 1 kvůli vložení jednoho znaku „x“ v operaci $O1$. Díky aplikované transformaci je operací $O2'$ smazán správný znak. Hlavním principem zachování konzistence v OT je tedy transformovat přijaté operace do nového tvaru na základě účinků dříve aplikovaných operací takovým způsobem, aby vznikl zamýšlený výsledek (zachování záměru) a aby se zajistilo, že replikované dokumenty budou stejné (konvergence).

Operační transformace musí vhodně pokrývat též složitější situace, jako jsou překrývající se úpravy ve slově a podobně. Jejich popis by byl mimo rozsah této práce.

Většina algoritmů postavených na OT pracuje s dokumentem jako s jedním uspořádaným seznamem (např. znaků) a nepodporuje zanořené stromové struktury [27]. Některé algoritmy zobecňují OT pro editaci XML dokumentů [14, 24, 46], což zpřístupňuje zanořování uspořádaných seznamů, nicméně stále *chybí podpora mapových typů klíč-hodnota*.

Většina nasazených kolaborativních systémů s OT jako Google Docs [15], Etherpad [8] nebo Novell Vibe [42] spoléhá na *centrální server*, který ustavuje úplné uspořádání operací a následně umožňuje zjednodušit transformační funkce [27]. Jediné známé transformační



Obrázek 3.1: Scénář ilustrující základní základní myšlenku OT pro udržování konzistence. Zdroj [44].

funkce správné v kontextu *peer-to-peer* [11] jsou Tombstones Transformation Functions (TTF) [35], které ale trpí problémy s výkonem [11].

Algoritmy OT obecně dosahují dobrého výkonu při produkování operací. Na druhou stranu *aplikování přijatých operací v decentralizovaném kontextu je složité a drahé*, protože každá replika musí přeuspořádat svou historii a transformovat souběžné operace, z čehož vyplývá kvadratická složitost $O(N^2k)$, kde N je počet operací a k je počet replik [11].

3.5 Bezkonfliktní replikované datové typy

Conflict-free replicated data types (CRDT) jsou konvergující datové typy, které poprvé představil Shapiro a kol. v článku z roku 2011 [40]. CRDT nevyžadují žádný centrální koordinační prvek. Navíc poskytují formalismy pro specifikaci operací nad těmito datovými typy a důkaz, že stav repliky vždycky směřuje ke globální konvergenci.

Jak název napovídá, datové typy jsou navržené pro automatické slučování různých replik takovým způsobem, že *nemohou vznikat konflikty*, které by vyžadovaly nějaký centralizovaný zásah pro vyřešení.

Příkladem může být například nezávislý požadavek dvou uživatelů, z nichž každý chce do jedné buňky zapsat jinou hodnotu. V datovém typu je při aplikaci změn nutné buďto automaticky deterministicky vybrat pouze jednu z hodnot (např. pomocí mechanismu LWW – last-writer-wins) nebo do buňky uložit obě (tzv. multi-value registr). Podstatné je, že se z pohledu datové struktury nepřechází do žádného zvláštního stavu „řešení konfliktů“, neexistuje ani žádná tendence evidovat a přenášet „seznam konfliktů“ ani nic podobného.

Z pohledu uživatele a GUI je samozřejmě možné vyhodnotit např. více hodnot v multi-value registru jako konflikt a učinit potřebná opatření (nechat uživatele vybrat jednu z hodnot), z pohledu datového typu jde ale jen o běžné další operace (v tomto případě op. přiřazení).

3.5.1 Samostatné CRDTs

V literatuře je možné najít popisy různých samostatných CRDT [41]:

- **čítač** (counter) – replikované celé číslo s update operacemi *increment* a *decrement*,
- **registr** (register) – paměťová buňka, která uchovává atomickou hodnotu nebo objekt a definuje update operaci *assign*,
- **množina** (set) – kolekce bez uspořádání, ze které vychází další typy (grafy, mapové typy, ...); definuje operace *add* a *remove*,
- **seznam** – uspořádaná kolekce prvků,
- **map** – množina dvojic klíč-hodnota
- a další...

Každý z těchto základních typů je potom možné najít v různých variantách lišících se vnitřní reprezentací a způsobem, jakým se chovají při výskytu konfliktních souběžných operací. Například pro *množinu* (set) je možné najít varianty

- **G-Set** (grow-only set), která umožňuje pouze přidávat nové prvky,
- **2P-Set**, která při souběžných operacích *add* a *remove* dává přednost *remove* a má tu vlastnost, že jednou odstraněný prvek už nemůže být znovu přidán,
- **OR-Set** (observed-remove set), která na rozdíl od 2P-Set dává přednost *add* a umožňuje opakované mazání a přidávání prvku,
- apod.

Uvedené typy zpravidla nepočítají s podporou zanořených CRDT.

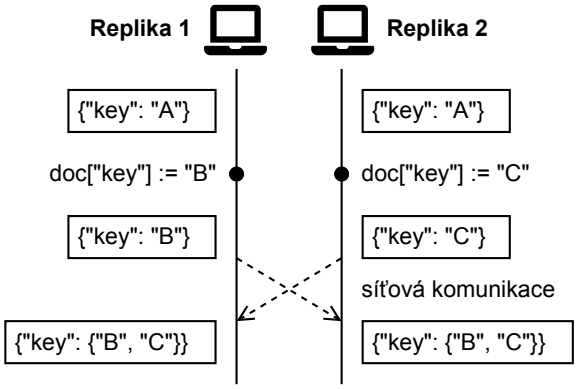
3.6 Bezkonfliktní replikovaný datový typ pro JSON

Při návrhu praktických aplikací se člověk často setkává s potřebou datové typy zanořovat, a to minimálně v jedné úrovni jako množinu záznamů, ideálně však neomezeně.

Jedním z takových datových typů je bezkonfliktní replikovaný datový typ pro JSON, který popsali a verifikovali Martin Kleppmann a Alastair R. Beresford ve svém článku z poloviny roku 2017 [27].

Článek popisuje algoritmus a formální sémantiku pro datové struktury JSONu, které automaticky řeší souběžné modifikace tak, že žádné úpravy nejsou ztraceny, a zároveň tak, že všechny repliky konvergují ke stejnému stavu. Algoritmus podporuje libovolně zanořené uspořádané seznamy a mapové typy, které mohou být modifikované vkládáním, mazáním a přiřazováním. [27]

Zatímco sémantika lokálních úprav JSONu je dobře známá, při *souběžných úpravách* vznikají různé okrajové případy, které, pokud nejsou pečlivě ošetřeny, mohou vést k nekonzistentním stavům. Autoři proto algoritmus popsali formální sémantikou a nad ní dokázali, že po jakékoliv sekvenci souběžných modifikací všechny repliky eventuálně konvergují ke stejnému stavu.

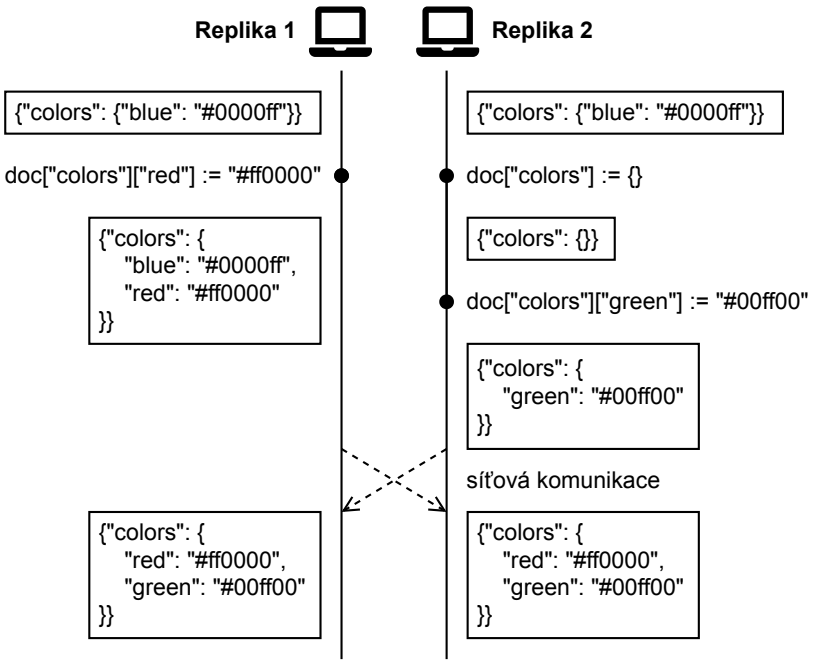


Obrázek 3.2: Souběžné přiřazení do registru. Zdroj [27].

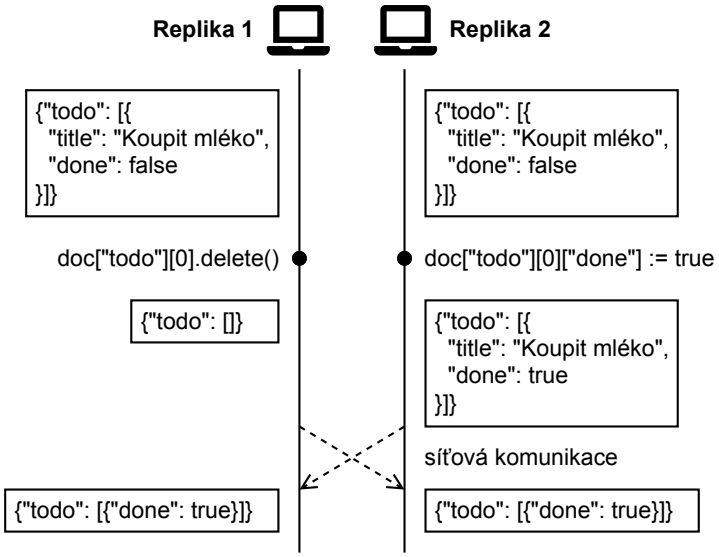
Souběžné operace jsou komutativní (nezáleží na pořadí jejich aplikování), což zajišťuje, že repliky konvergují ke stejnému stavu bez nutnosti aplikačně-specifické logiky řešení konfliktů. [27]

Map se v popsaném datovém typu pro primitivní hodnoty (řetězec, číslo, ...) chová jako multi-value registr. Díky tomu v případech, kdy dva uzly nezávisle zapíšou do stejného klíče různé hodnoty, není žádný uživatelský vstup ztracen (viz obr. 3.2). Pokud jsou do jednoho klíče nezávisle zapsané dvě kompatibilní zanořené struktury (map nebo seznam), jsou automaticky sloučeny.

V případě, že dojde k nezávislým úpravám, kdy jeden uzel zapíše primitivní hodnotu do klíče a druhý uzel zároveň smaže jeho nadřazený prvek, algoritmus volí sémantiku add-wins (přidání vyhrává), tzn. zápis klíče po synchronizaci způsobí obnovení smazaného nadřazeného prvku. Situace je znázorněná na obrázku 3.3. Toto chování nicméně nemusí být vhodné ve všech situacích, jak ukazuje obrázek 3.4, kdy dotčený záznam ztratil popisek (title).



Obrázek 3.3: Úpravování obsahu zanořeného objektu map a souběžné přepsání celého objektu map. Zdroj [27].



Obrázek 3.4: Jedna replika odstraní prvek seznamu, zatímco jiná replika upravuje jeho obsah. Výsledný úkol ztratil popisek (title). Zdroj [27].

3.7 Zhodnocení CRDT a OT

Použití operačních transformací sice může dávat smysl při použití centralizovaného serveru, v kontextu této práce by nicméně bylo použití serveru s přístupem k uživatelským datům nevhodné. Jak bylo naznačeno v sekci 3.4, výkon OT v kontextu P2P je špatný a chybí podpora mapových typů key-value. Z těchto důvodů jsem použití OT vyloučil.

CRDT se naproti tomu pro účely této práce skvěle hodí – nevyžadují žádný centrální prvek a podporují mapové typy. V případě CRDT pro JSON popsaného v sekci 3.6 je možné datové typy neomezeně zanořovat, což zpřístupňuje tvorbu komplikovanějších datových modelů v cílových aplikacích.

3.8 Existující P2P rámce a knihovny pro replikaci stavu

Při hledání existujících řešení pro tvorbu P2P aplikací a knihoven na replikaci dat je možné narazit na množství projektů, které se tématicky dotýkají této práce.

Pokud není uvedeno jinak, jsou projekty implementované v JavaScriptu. V takovém případě je ještě vhodné podotknout, že JavaScript nemusí nutně znamenat implementaci určenou pro prohlížeč, ale také například pro běh v příkazovém řádku prostřednictvím Node.js[1]. Díky dostupným nízkoúrovňovým API pro práci např. se sockety je tak možné setkat se i s implementacemi protokolů, které v prohlížeči nejsou podporované.

V přehledu je začleněno i několik řešení vyžadujících centralizovanou databázi (a jsou tak v rozporu s P2P povahou práce), které byly z nějakého důvodu zajímavé, například kvůli použitému replikovatelnému typu.

Srovnání všech kandidátů je zachyceno v tabulce 3.1. Kromě základních údajů jsou u každého projektu sledované následující vlastnosti:

- **Počet hvězdiček na serveru GitHub** – počet hlasů v systému hodnocení, kterými mohou přihlášení uživatelé vyjadřovat přízeň určitému projektu. Může reflektovat popularitu projektu, ale může být zkresleno stářím projektu (např. mladá zajímavá knihovna může mít méně hvězdiček než několik let zaběhlá průměrná alternativa).
- **Replikovatelná datová struktura (CRDT)** značí, jestli projekt nabízí řešení pro zaznamenávání a slučování změn prostřednictvím replikovatelné datové struktury. Všichni vybraní kandidáti nějakým způsobem staví na CRDT.
- **Implementace komunikačního kanálu** popisuje, jestli knihovna nabízí implementaci nějaké technologie pro přenos dat.
- **Protokol komunikace** značí, že knihovna definuje formát a typy zpráv, které se mají přenášet komunikačním kanálem.
- **Zabezpečení** značí, jestli knihovna nabízí řešení autentizace uzlů a šifrování přenášených zpráv.
- **Tvorba skupin** značí, jestli knihovna nabízí mechanismus pro vytváření skupin uzlů pro sdílení obsahu, ať už budou definované na centrálním serveru nebo distribuovaně.
- **GUI** značí, jestli knihovna nabízí prvky grafického uživatelského rozhraní pro konfiguraci a monitorování aspektů P2P spojení.

[1] Stránky Node.js: https://nodejs.org

Název Zdrojové kódy Webové stránky	Hvězdiček (leden 2019)	Replikovatelná datová struktura nad CRDT	Implementace komunikačního kanálu (technologie)	Protokol komunikace	Zabezpečení komunikace	Tvorba skupin	GUI
replikativ github.com/replikativ/replikativ replikativ.io	283	✓	Websockets	✓	✗	✗	✗
Swarm github.com/gritzko/swarm swarmdb.net	2440	✓	Websockets	✓	✗	✗	✗
freedom.js github.com/freedomjs/freedom freedomjs.org	376	✗	WebRTC, XMPP	✗	✗	✗	✗
libp2p github.com/libp2p libp2p.io	878	✗	Web: WebRTC, Web-sockets; Lokálně: TCP, ...	✓	✓	✗	✗
Scuttlebot.io github.com/ssbc/ssb-server scuttlebot.io	1042	✗	✗	Pouze formát zpráv	✓	✗	✗
GUN github.com/amark/gun gun.eco	9493	✓	Websockets	✓	✓	✗	✗
CRDT github.com/dominictarr/crdt	769	✓	✗	✗	✗	✗	✗
js-delta-crdts github.com/ipfs-shipyard/js-delta-crdts	57	✓	✗	✗	✗	✗	✗
delta-enabled-crdts github.com/CBaquero/delta-enabled-crdts	181	✓	✗	✗	✗	✗	✗
Automerge github.com/automerge/automerge	6191	✓	✗	✗	✗	✗	✗

Tabulka 3.1: Souhrn existujících řešení

Replikativ

Rámec implementuje vlastní variantu CRDT postavenou na CDVCS (Confluent Distributed Version Control System) popsanou v článku [47].

Zajímavostí je použití *ClojureScriptu* (podmnožiny dynamického funkcionálního jazyka Clojure, kterou je možné zkompilovat do JavaScriptu) jakožto implementačního jazyka.

Rámec aktuálně závisí na použití centrálního serveru, se kterým komunikuje prostřednictvím protokolu WebSockets.

Swarm

Swarm není tak docela P2P rámec, protože spoléhá na *centrální databázi*, se kterou se klienti mohou synchronizovat a poté editovat data i offline. Je nicméně zajímavý hned kvůli několika věcem:

1. Replikovaná datová struktura: Rámec používá při přenosu *Replicated Object Notation (RON)*[2] – vlastní reprezentaci dat, kterou lze do jisté míry považovat za alternativu JSONu. RON na rozdíl od běžných formátů popisuje strukturovaná data pomocí operací, které je utváří, a rovnou doplňuje i anotace výběru sémantiky pro souběžné úpravy. Díky tomu formát dobře spolupracuje s jejich vlastním CRDT. Formát je navíc úsporně serializován do binárního formátu a využívá všemožných dalších optimalizací pro snížení objemu přenášených dat a metadat.

2. Klientská knihovna implementuje GraphQL API[3] pro manipulaci a čtení offline i online dat. Snižuje se tak vstupní bariéra pro všechny front-end vývojáře, kteří jsou již s GraphQL obeznámeni např. z vývoje čistě online systémů.

freedom.js

Knihovna freedom.js se soustřeďuje na abstrahování nízkoúrovňových API pro vytvoření P2P komunikačního kanálu nezávisle na použité technologii. K tomu definuje tři sady rozhraní pro uživatele: *Storage* pro přístup k lokálnímu úložišti, *Social* pro přístup k seznamu přátel a zasílání zpráv, *Transport* pro přenos zpráv.

Knihovna definuje pouze samotná rozhraní a několik málo ukázkových implementací. I kdyby implementace byly kompletní, knihovna nijak neřeší replikovatelné datové struktury, formát přenášených zpráv, zabezpečení a další potřebné součásti.

Ukázka na oficiálních stránkách v době psaní práce nefungovala.

libp2p

Libp2p je podobně jako freedom.js knihovna, která se snaží vytvořit abstraktní API pro práci s P2P komunikačním kanálem nezávislé na použité přenosové technologii. Na rozdíl od ní však působí podstatně sofistikovanějším a úplnějším dojmem.

Knihovna definuje abstraktní rozhraní pro *Transports, Circuit relays, Protocol muxing, Multiplexers, Privatization, Encryption, Connection, Peer Discovery*. Zároveň nabízí i rozsáhlé množství implementací jednotlivých API jak pro běh ve webovém prohlížeči, tak pro Node.js.

[2] Stránka dílčího projektu: https://github.com/gritzko/ron

[3] https://graphql.org/

I když tento zástupce má k použitelnému řešení P2P komunikace asi nejblíž, knihovna vynucuje závislost na vlastním způsobu adresování koncových uzlů (multiaddr), dodaný WebRTC signalizační server je pro naše účely příliš jednoduchý, knihovna neposkytuje žádné řešení tvorby skupin a v neposlední řadě je navržena v objektovém a ne funkcionálním stylu, jak vyžaduje tato práce.

Scuttlebot.io

Scuttlebot.io poskytuje P2P úložiště logů, které se dá využívat jako distribuovaná databáze, poskytovatel identity a další. Pracuje s tzv. *feedy* – podepsanými append-only sekvencemi zpráv na způsob blockchainu. Projekt řeší hlavně autentizované šíření změn v síti (typy zpráv) bez závislosti na použité přenosové technologii a je zaměřený spíš na aplikace mimo prohlížeč.

GUN

Jedná se o *replikovanou grafovou databázi* s podporou offline synchronizace, která pro svůj běh vyžaduje centrální server. V přehledu je zařazena kvůli tomu, že jakožto grafová databáze nabízí bohaté možnosti dotazování a uspořádávání dat.

Dominic Tarr's CRDT

CRDT knihovna implementovaná podle článku [41] od Shapira a kol. definuje datové replikovatelné typy *Doc, Row, Set* a *Seq*. Použití knihovny znepříjemňuje velmi skrovná dokumentace, na druhou stranu je zajímavé použití streamů k propojení replik.

js-delta-crdts

Tato velice zajímavá implementace δ-CRDT staví na článku [6] od Almeidy a kol. Poskytuje rozsáhlé množství typů CRDT struktur, které umožňují vybrat si přesně takovou sémantiku, jakou programátor zrovna potřebuje. Typ *OR-Map* navíc podporuje rekurzivní zanořování jiných kauzálních CRDT.

delta-enabled-crdts

Jedná se o další implementaci δ-CRDT založenou na článcích [5] Almeidy a kol. Tato konkrétní varianta je sice napsaná v C++, jejím autorem je ale přímo jeden z autorů článku, takže by mohla být dobrým autoritativním zdrojem.

Automerge

Automerge je knihovna, která staví na práci [27] Martina Kleppmanna a kterou, jakožto jedinou z výše uvedených knihoven, používá i náš rámec.

Jak bylo naznačeno v sekci 3.6, vytvořený datový typ umožňuje replikovatelnou reprezentaci JSONu s neomezeným zanořením v objektech typu *List* i *Map*. Knihovna je aktivně vyvíjena přímo autorem článku, který přidává mnohá vylepšení v článku chybějící: podporu undo/redo, relační tabulky, strukturu pro efektivní editaci textu, čítače.

Funkcionální styl většiny kódu znamená příslib pro případné snadné konvertování do funkcionálního jazyka.

3.9 Zhodnocení existujících řešení

Ani v jednom z případů se mi nepodařilo najít projekt, který by problematiku pojímal v podobném rozsahu jako tato práce.

Některé projekty se zabývají pouze replikovanými datovými typy, jiné definují pouze samostatné abstrakce pro komunikaci, další projekty se snaží pokrýt obojí.

Společnou charakteristikou všech uvedených projektů je to, že se soustředí spíše na *vytvoření nízkoúrovňových abstrakcí*, ale už málokdy nabízí konkrétní celkové řešení zabezpečení, propagace změn v síti, tvorby skupin uzlů nebo začlenění do uživatelského rozhraní.

Abstrakce komunikačních kanálů nezávislá na přenosové technologii mi pro potřeby tohoto projektu připadá předčasná, omezující a potenciálně zbytečná. Z toho důvodu jsem nevyužil *žádný z existujících P2P rámců*.

Co se týče *implementací CRDT*, jako nejlepší kandidáty z hlediska nabízené funkčnosti vnímám *js-delta-crdts* a *Automerge*. Zatímco *js-delta-crdts* je navržen modulárnějším způsobem a nabízí větší kontrolu nad použitou sémantikou řešení konfliktů, Automerge oproti svému konkurentovi nabízí značné množství funkcionality navíc, která výrazně usnadňuje praktické nasazení a knihovna tak celkově působí úplnějším dojmem. Pro použití v této práci jsem zvolil *Automerge*.

Kapitola 4

Návrh

Globální návrh vytvořeného rámce by se dal stručně popsat tak, že *implementace koncových uzlů* začleněné v cílových aplikacích používají univerzální *signalizační server* k výměně údajů potřebných k navázání peer-to-peer spojení se svými protějšky. Jakmile je přímé spojení ustavené, signalizační server je využíván již pouze k notifikacím o změnách online statusu přátel. Koncové uzly jsou využívány k uchovávání, tvorbě a úpravám obsahu. Pomocí P2P kanálů jsou potom změny synchronizovány s protějšky.

Rámec je rozdělen do několika balíčků znázorněných na obrázku 4.1. Nejdůležitější jsou *pocket-mesh-peer*, *pocket-mesh-peer-material-ui*, *pocket-mesh-signal-server* a *tree-burst*. Ostatní jsou podpůrné knihovny, které nejsou z pohledu návrhu příliš významné a jejich popisem se zabývá až kapitola 5.

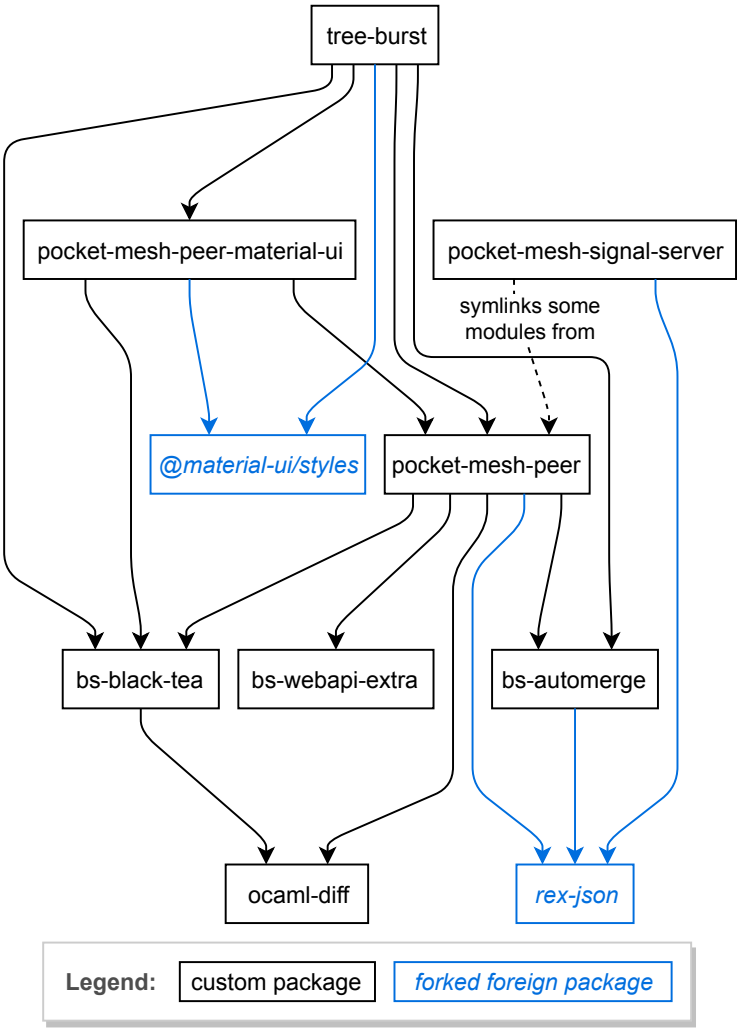
Knihovna *pocket-mesh-peer* zpřístupňuje cílové aplikaci hlavní funkcionalitu koncového uzlu, tj.

- zpřístupňuje správu spřátelených uzlů,
- správu skupin a jejich členů,
- datový typ pro uchovávání dat s podporou replikace,
- zaobaluje veškerou komunikaci se signalizačním serverem,
- P2P komunikaci,
- práci s lokální databází,
- kryptografické funkce a další.

Knihovna *pocket-mesh-peer-material-ui* poskytuje prvky GUI, které tvoří grafickou nadstavbu pro výše uvedenou funkcionalitu. Z pohledu koncové aplikace použití této knihovny není povinné, aplikace si může případné GUI s využitím knihovny *pocket-mesh-peer* implementovat sama.

Balíček *pocket-mesh-signal-server* obsahuje, jak název napovídá, implementaci nativního signalizačního serveru a konečně *tree-burst* je název ukázkové aplikace, jejímž návrhem a implementací se zabývá oddělená kapitola 6.

V první části této kapitoly jsou popsány základní abstrakce vytvářeného rámce, tj. co je v kontextu projektu uzel (sekce 4.1), jakým způsobem je možné vytvářet skupiny uzlů (4.2) a jak funguje mechanika nastavení oprávnění.



Obrázek 4.1: Graf závislostí balíčků (podprojektů).

Následující sekce 4.5 krátce popisuje potřebné funkce signalizačního serveru, po čemž následuje stěžejní částí návrhu – *návrh vlastních protokolů*.

Protokoly sice využívají stejný formát zpráv (viz sekce 4.4), jsou nicméně rozdělené na dvě části podle použitého kanálu na protokol pro komunikaci se signalizačním serverem (sekce 4.6) a protokol pro komunikaci peer-to-peer (sekce 4.7). Kromě jejich samotného popisu jsou v textu průběžně zvažována i možná bezpečnostní rizika.

Sekce 4.8 popisuje jednotlivé přístupy řešení konfliktů na straně uzlu, poslední sekcí (4.10) je potom návrh prvků uživatelského rozhraní pro knihovnu *pocket-mesh-peer-material-ui*.

4.1 Uzel

Uzlem (též koncovým uzlem, protějškem) je v kontextu této práce vždy myšlena jedna instance koncové aplikace běžící v konkrétním profilu (úložišti) konkrétního webového prohlížeče.

I když se pro označení uzlu mohou nabízet některá jiná označení (osoba, uživatel, zařízení, prohlížeč, ...), je třeba dát pozor na odlišnosti ve významu. V praxi bude samozřejmě nejobvyklejší případ, kdy jedno *zařízení* používá jeden *člověk* a používá v něm pouze jeden *prohlížeč* s jedním profilem. V takovém případě opravdu můžeme o uzlu uvažovat jako o jednom člověku, prohlížeči nebo fyzickém zařízení. Na druhou stranu ale jeden *člověk* (osoba, uživatel) může vlastnit více *zařízení* s několika systémovými účty, v každém z nich může být několik *prohlížečů*, v každém alespoň jeden nebo i několik profilů, v každém profilu může být jeden uzel.

Zvláštní situace nastává v okamžiku, kdy je nutné cílovou aplikaci použít *jednorázově* například na sdíleném pracovním počítači. V takovém případě je vhodné využít tzv. *anonymní okno* – izolovaný profil prohlížeče, jehož veškerá lokální data (tj. i data a identita dočasné repliky) jsou po zavření okna smazána.

4.1.1 Identita uzlu

Při prvním spuštění koncové aplikace v daném profilu webového prohlížeče si uzel vygeneruje vlastní *unikátní privátní a veřejný klíč*. Privátní klíč je využíván k podepisování zpráv odesílaných skrz signalizační kanál. Pomocí veřejného klíče si následně protějšek může ověřit, že zpráva nebyla po cestě modifikována.

Z veřejného klíče je pomocí hašovací funkce odvozeno ID uzlu – krátký otisk, který je využíván pro účely adresování uzlů a snadnější výměnu kontaktů např. prostřednictvím chatu.

Protože syrová data privátního klíče z bezpečnostních důvodů není možné číst (serializovat), identita je přímo spjata s profilem prohlížeče a její přenos mezi uzly není možný.

4.2 Skupina a systém oprávnění

Skupina v kontextu této práce je množina jednoho a více uzlů, které mezi sebou sdílí jeden replikovaný strom dat. Struktura těchto sdílených dat je v režii koncové aplikace. Jedna skupina může obsahovat například

- všechny dokumenty v rámci jednoho pracovního týmu (u textového editoru),

Obsah (content)	**Členové** (members)
R	R
RW	R
RW	RW

Tabulka 4.1: Povolené kombinace oprávnění ve skupině (R – pouze čtení, RW – čtení a zápis)

- všechny soukromé seznamy úkolů v rámci všech mnou vlastněných zařízení (u aplikace na úkoly),
- všechny seznamy úkolů sdílené s mojí manželkou (třeba jen jeden s nákupním seznamem; u aplikace na úkoly),
- všechny grafické návrhy v rámci týmu designérů (u vektorového editoru),
- atp.

Každá skupina je označená náhodně vygenerovaným UUID *identifikátorem* pro snadné sdílení. Jeden uživatel nemůže být členem dvou skupin se stejným ID.

Skupinu je možné též pojmenovat, zvolená *přezdívka* má platnost pouze v rámci aktuálního uzlu.

Nejdůležitější položkou uchovávanou v rámci skupiny je *replikovaný strom dat* reprezentující vytvářený obsah (viz kapitola 3 a sekce 3.6). Jak bylo naznačeno výše, jeho podoba závisí na implementaci cílové aplikace, tzn. v textovém editoru bude představovat něco jiného než ve správci fotek. Replikaci této struktury v rámci skupiny zajišťuje rámec automaticky.

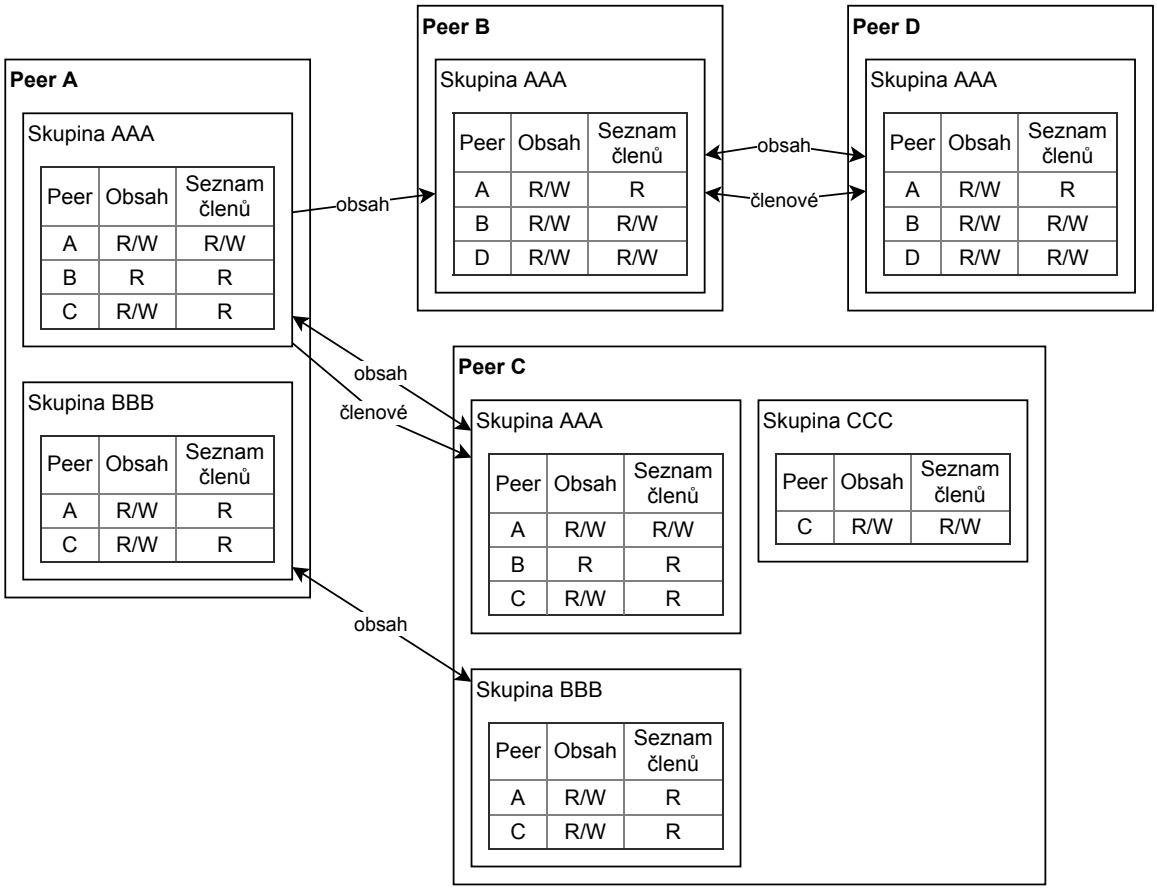
Ke skupině přísluší *seznam členů* a jejich oprávnění. To se skládá ze dvou složek:

- **Oprávnění k úpravám obsahu** u člena značí, jestli od něj lokální uzel bude automaticky přijímat a aplikovat nabízené změny v replikovaném stromu dat. Oprávnění ke čtení má každý uzel, který je v seznamu přítomen.
- **Oprávnění k úpravám seznamu členů** u člena značí, jestli od něj lokální uzel bude automaticky přijímat nabízené změny v tomto seznamu, tzn. přidávání a odebírání členů a úpravy oprávnění existujících členů. S nově přidanými členy bude obsah synchronizován pouze v případě, že je daný protějšek přidán v seznamu přátel.

Povolené kombinace oprávnění je možné vidět v tabulce 4.1.

Důležitým principem v uvedeném systému oprávnění je to, že *neexistuje nic jako centrální seznam členů skupiny*, který by byl zaručeně jednotný napříč celou sítí. Každý uzel může mít jinou představu o tom, kdo je členem skupiny a kdo má jaká oprávnění, jinými slovy *každý uzel se může svobodně rozhodnout, s kým bude obsah skupiny sdílet a od koho bude přebírat změny.*

Zajištění jednotného seznamu by vyžadovalo buďto vytvoření centrální, ideálně stále dostupné autority, které se chceme vyhnout, nebo dosažení konsenzu, který ale k potvrzení změn vyžaduje určitý počet dostupných uzlů. Pokud má rámec v základu podporovat práci offline, na dostupnost uzlů se nedá spoléhat.



Obrázek 4.2: Ukázka nastavení oprávnění mezi uzly a vyplývající možné směry propagace změn. Identifikátory uzlů a skupin jsou pro názornost zkráceny.

Aby nebylo nutné každou změnu oprávnění ručně zadávat na každém uzlu zvlášť a aby tak nedocházelo k nechtěnému roztříštění pohledu na skupinu, je možné nechat změny seznamu členů automaticky přebírat od ostatních uzlů.

Pokud nastane případ, kdy místní uzel chybí v seznamu členů, popř. nemá podle názoru ostatních protějšků oprávnění k zápisu, knihovna mu přesto umožní obsah měnit. Příkladem použití může být například editor myšlenkových map – lokální uživatel si může v cizí myšlenkové mapě vytvořit vlastní podstrom, který se k ostatním nepřenese, a zároveň dál dostávat změny v jiných částech stromu. Pokud je podobné chování v koncové aplikaci nežádoucí, je v ní samozřejmě možné zápis lokálnímu uživateli programově zakázat.

Ukázku nastavení oprávnění v síti několika uzlů je možné vidět na obrázku 4.2. Šipky mezi jednotlivými konfiguracemi značí možné směry propagace změn obsahu a změn seznamu členů.

4.2.1 Odepření přístupu

I když menších odlišností distribuované správy skupin od centralizované by se dalo najít víc, za vypíchnutí určitě stojí minimálně sémantika odebírání přístupu členům skupiny.

Odlišnosti demonstrujeme na příkladu, ve kterém *centralizovanou aplikací* myslíme aplikaci, v níž je vytvářený obsah i seznam členů uchovávaný na centrálním serveru. Praktickým

příkladem může být třeba textový editor Google Docs. *Decentralizovanou aplikací* potom myslíme náš výše popsaný přístup, kde každý uzel má vlastní pohled na obsah i členy.

Uvažme situaci, kdy se uzel Y rozhodne uzlu X ve skupině odepřít přístup ke čtení obsahu.

Co se týče *centralizovaného přístupu*, protože uzel X standardně nemá kopii dokumentu u sebe, přichází z minuty na minutu o přístup ke všem datům.

V případě *našeho navrženého přístupu*, pokud uzel Y *nezajistí* rozšíření nových oprávnění mezi ostatní členy skupiny, uzel A se pořád může dostat ke změnám prostřednictvím členů Z, W, a dalších. Když na druhou stranu uzel Y *zajistí* rozdistribuování nových oprávnění všem ostatním členům, uzel X sice přestane dostávat nové změny, každopádně obsah, který už jednou získal před odepřením přístupu mu už zůstane v poslední jemu známé podobě.

4.3 Seznam přátel

Seznam spřátelených uzlů umožňuje definovat takové uzly, které mohou s aktuálním uzlem navázat spojení. Pro každý uzel se uchovává jeho ID a lokálně platná přezdívka, kterou uživatel může libovolně měnit.

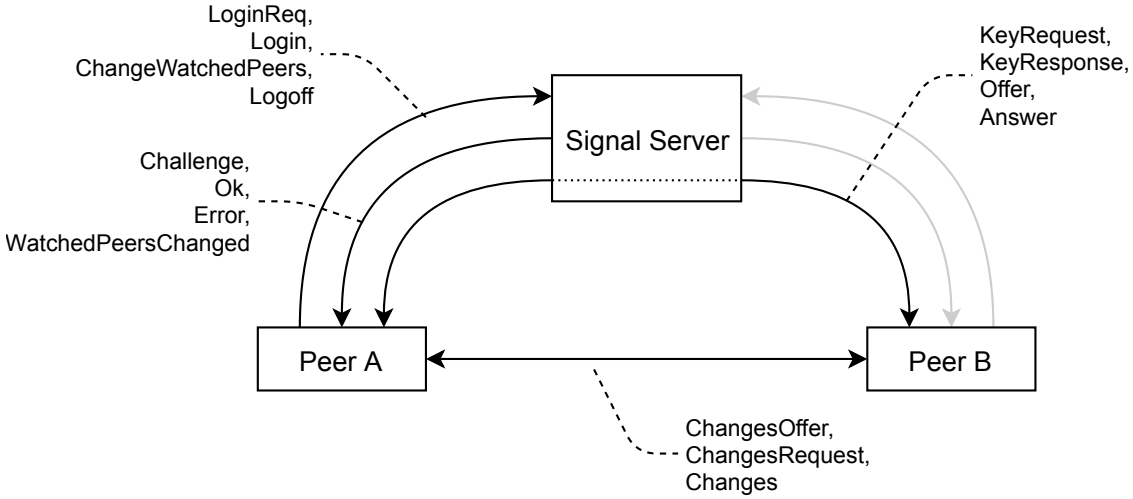
Členům, kteří jsou přidáni do nějaké lokální skupiny, ale chybí v seznamu přátel, lokální uzel nedovolí navázat spojení. Stejně tak těmto uzlům nejsou zasílány nabídky změn.

4.4 Společný formát přenášených zpráv

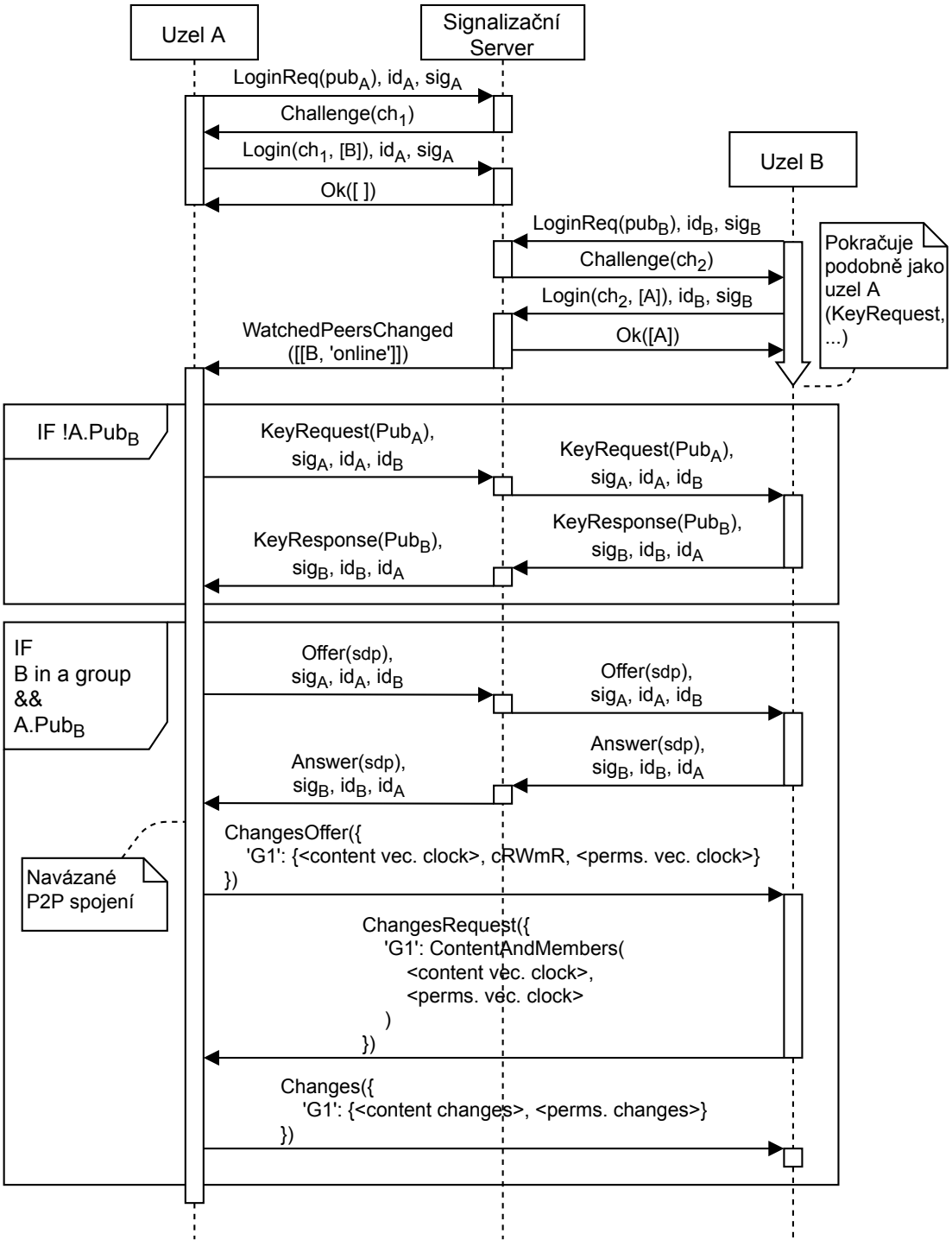
Koncový uzel rámce komunikuje s okolním světem pomocí dvou hlavních kanálů: pomocí WebSockets *se signalizačním serverem* a pomocí WebRTC nejkratší možnou cestou *s protějšky*. Vzhledem k tomu, že ani WebSockets, ani WebRTC neurčují podobu přenášených zpráv, bylo pro potřeby tohoto rámce nutné navrhnout jejich konkrétní formát, typy, pořadí a sémantiku.

Při výběru *formátu* pro přenos strukturovaných dat přichází v úvahu alespoň dva kandidáti: XML a JSON. Pro tento rámec byl vybrán *JSON*, protože ve srovnání s XML nabízí úspornější syntaxi a obecně se v posledních letech těší značné oblibě zejména díky rozšíření REST API. Pro maximální konzistenci je JSON využit v obou kanálech.

Každá zpráva obsahuje jedno povinné pole, kterým je řetězec `type` určující typ zprávy. Další vyžadovaná pole závisí na typu zprávy. Každý z kanálů definuje jiné, pro něj relevantní typy. Přehled všech typů zpráv v rámci a jejich použití s ohledem na kanál a směr komunikace jsou naznačené na obrázku 4.3. V následujících sekcích 4.6 a 4.7 je popsána jejich konkrétní podoba a použití.



Obrázek 4.3: Schéma přenášených typů zpráv



Obrázek 4.4: Ideální průběh navázání prvního spojení mezi dvěma uzly a počáteční výměna dat. Symbol sig_X značí podpis zbytku zprávy vytvořený pomocí privátního klíče uzlu X.

4.5 Signalizační server

Jak již bylo naznačeno v sekci 2.2, jedinou schůdnou technologií pro implementaci P2P spojení mezi webovými prohlížeči je WebRTC, které pro svoji funkčnost vyžaduje implementaci vlastního *signalizačního serveru*, resp. obecně nějakého signalizačního kanálu.

Signalizační server je prvek infrastruktury, který v našem případě zajišťuje hned několik funkcí:

- udržuje seznam koncových uzlů, které jsou online,
- umožňuje připojeným klientům sledovat změny dostupnosti ostatních protějšků a především
- dělá prostředníka pro výměnu zpráv nutných k navázání P2P spojení.

Protože smyslem celého rámce je *maximálně omezit vliv centrálních prvků*, je signalizační server navržen tak, aby *nebylo nutné mu důvěřovat*. Resp. v každý okamžik by se mělo počítat s možností, že je signalizační server kompromitován, provozován s úmyslem poškodit koncový uzel a získat jeho citlivá data. Je tedy třeba zohlednit např.

- pokusy o pozměňování zpráv,
- pokusy o pozdržování zpráv,
- pokusy o replay útoky,
- spolupráci kompromitovaného signalizačního serveru s upraveným koncovým uzlem útočníka.

Dobrá implementace signalizačního serveru by měla chránit zájmy klientů i pomocí doplňujících bezpečnostních opatření. Klienti a implementace koncových uzlů na to nicméně nesmí spoléhat.

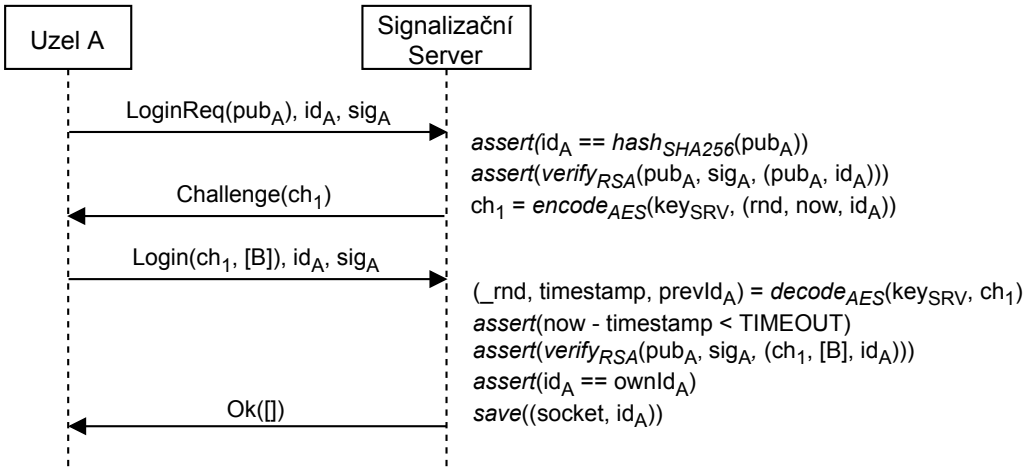
4.6 Protokol komunikace se signalizačním serverem

Komunikace se signalizačním serverem je realizovaná pomocí protokolu WebSockets (více o něm viz sekce 2.2), konkrétně pomocí jeho varianty WebSockets Secure zabezpečené TLS. Díky tomu je zajištěna důvěrnost přenášených dat a při použití důvěryhodného certifikátu je možné ověřit také identitu serveru.

Typy zpráv a směr komunikace, ve kterém se používají, jsou naznačené na obrázku 4.3. Do protokolu signalizačního serveru patří všechny zprávy u šipek spojujících signalizační server s koncovými uzly: zprávy *LoginReq*, *Login*, *ChangeWatchedPeers* a *Logoff* se používají ve směru k signalizačnímu serveru, *Error* a spol. od sig. serveru k uzlu, zprávy *KeyRequest* a spol. signalizační server pouze přeposílá adresátovi.

4.6.1 Navázání spojení

Při spuštění uzlu je v jeho zájmu se co nejdříve připojit k signalizačnímu serveru, zjistit, kteří z přátel jsou online, a bezpečně si s nimi vyměnit SDP zprávy pro ustavení P2P spojení. Obrázek 4.4 zachycuje ideální průběh tohoto procesu. Následující popis tento proces rozebírá v situaci prvního navázání spojení, kdy ještě uzel A nemá veřejný klíč protějšku. Při následujících pokusech o připojení již získávání klíče není nutné.



Obrázek 4.5: Detail ověření živosti připojovaného koncového uzlu.

Ověření připojeného klienta

Ihned po připojení k serveru je nutné ověřit, jestli uzel A připojený na daném socketu opravdu vlastní privátní klíč k ID, pod kterým vystupuje. Detail ověřování je možné sledovat na obrázku 4.5. Kdyby se tato kontrola neprováděla, hrozil by replay útok, při kterém by útočník mohl způsobit odpojení uzlu A na libovolném serveru. Možný průběh útoku ilustruje příloha A.

1. Ihned po připojení uzel A zašle signalizačnímu serveru zprávu *LoginReq*. Zpráva obsahuje veřejný klíč, aby si server mohl ověřit, že podpis zprávy i udávané ID odpovídají danému veřejnému klíči.

2. Server si veřejný klíč uloží a pomocí vlastního tajného klíče a symetrické šifry AES-GCM zašifruje zprávu „pro sebe“ obsahující náhodné číslo, aktuální časové razítko a ID přihlašovaného uzlu. Tento uzlu A nesrozumitelný řetězec v Base64 je zaslán klientovi ve zprávě *Challenge*.

   Díky použití GCM (Galois/Counter Mode) varianty AES je u řetězce zaručena nejen důvěrnost, ale i možnost ověření integrity.

   Náhodné číslo je zde proto, aby i v případě dvou požadavků se stejným ID zaslaných ve stejný okamžik dostal každý jinou výzvu.

3. Klient řetězec beze změny vezme a společně se seznamem uzlů, pro které by rád dostával notifikace o změnách dostupnosti, jej podepíše a odešle.

4. Server dešifruje vlastní zprávu (výzvu) a překontroluje, jestli uplynulá doba od vydání výzvy nepřesáhla limit v řádu sekund. Pokud je v pořádku i podpis a ID odesílatele, server má dobrou jistotu, že na druhém konci opravdu je daný uzel.

   V nejhorším případě by tento uzel mohl komunikovat přes prostředníka (man-in-the-middle), nicméně tato situace by neměla být zneužitelná – modifikace ze strany útočníka nevadí, protože všechny zprávy pro ustavení P2P spojení jsou podepsané a ověřované v koncových uzlech.

   Když je vše v pořádku, sig. server si uloží uzel A do seznamu online uzlů a zašle notifikace všem uzlům, které zajímá změna stavu uzlu A.

Nakonec sig. server potvrdí přihlášení zprávou *Ok* se seznamem sledovaných uzlů, kteří jsou online. V tomto případě není online nikdo.

Výše uvedené ověření klienta je v této situaci doplňujícím opatřením, jehož vynechání v implementaci serveru by útočníkovi příliš nepřineslo. Jistě, kdyby se zde kontrola neprováděla, mohl by se pomocí zfalšované zprávy *Login* za uzel A vydávat libovolný útočník, který by tak mohl kdykoliv nahradit uzel A v seznamu online uzlů a přerušit mu tak spojení se sig. serverem. Dalo by se však hádat, že vypadávání spojení by uživatelům bylo nápadné a brzo by si hledali jiný signalizační server, což pravděpodobně není v zájmu útočníka. K datům uživatele se takto útočník dostat nemůže.

Výměna veřejných klíčů a zpráv SDP

1. Protože ze zprávy *Ok* uzel A ví, že není nikdo online, vyčkává.

2. Po nějaké době se přihlásí obdobným způsobem uzel B. Protože uzel B se zajímá o stav uzlu A, obratem dostává v *Ok* zprávě informaci, že uzel A je online. Stejně tak A dostává notifikaci o tom, že je B online, protože si o to na začátku zažádal.

3. Aby byl uzel A schopný ověřovat autenticitu zpráv protějšku B, potřebuje k tomu jeho veřejný klíč. Protože při prvním navazování spojení nemá uzel A nic jiného, než ID uzlu B, musí si o klíč zažádat zprávou *KeyRequest*. Ve zprávě žadatel posílá

   - *vlastní veřejný klíč (pub_A)*, aby si příjemce mohl ověřit autenticitu zprávy, i když předtím neměl veřejný klíč odesílatele,
   - *ID odesílatele (id_A)*, aby příjemce věděl, komu poslat odpověď a aby si mohl ověřit, že odpovídá hashi veřejného klíče,
   - *ID příjemce (id_B)*, aby sig. server věděl, komu zprávu přeposlat,
   - *podpis zbytku zprávy (sig_A)*, aby si uzel B mohl ověřit, že odesílatel vlastní privátní klíč k danému veřejnému klíči.

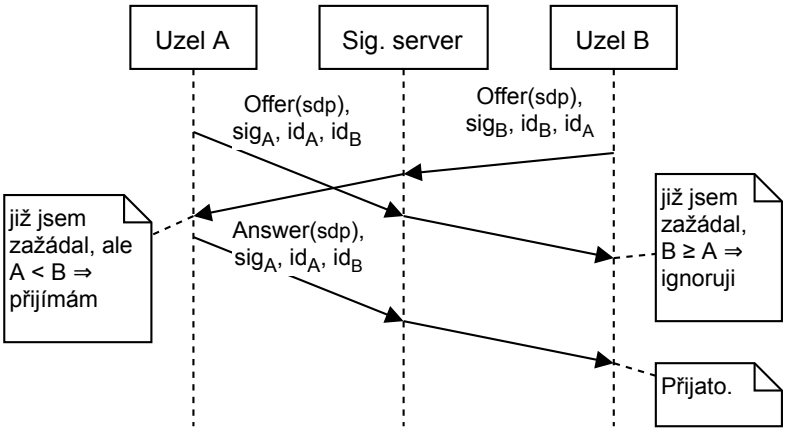
   Signalizační server podpisy ve zprávě *KeyRequest* (stejně jako u ostatních přeposílaných typů zpráv) standardně neověřuje, ale rovnou předává cílovému uzlu.

   Implementace filtrování zpráv s vadným podpisem by mohla snížit zátěž koncových uzlů v případě velkého počtu neplatných zpráv. Na druhou stranu by ale v takovém případě hrozilo větší vytížení serveru.

4. Uzel B zkontroluje, jestli má uzel A v seznamu přátel a jestli je zpráva v pořádku (viz výše). Pokud ano, odpoví uzlu A zprávou *KeyResponse* s podepsaným veřejným klíčem.

5. Jakmile má uzel A veřejný klíč protějšku, zkontroluje, jestli má protějšek v nějaké skupině, a pokud ano, začne se pokoušet o navázání WebRTC spojení. Zjistí si pomocí STUN serveru vlastní veřejnou IP adresu a port a vygeneruje SDP Offer obsahující kromě těchto informací ještě další údaje pro ustavení P2P spojení. *Je kritické, aby tato zpráva byla podepsaná.* Jen díky tomu si uzel B může být jistý, že nenavazuje P2P spojení s útočníkem.

6. Když uzel B obdrží SDP *Offer*, také si zjistí vlastní veřejnou IP adresu a port a s pomocí nabídky vygeneruje podepsanou SDP *Answer*.



Obrázek 4.6: Zahození jednoho spojení při dvou souběžných žádostech.

7. Po přijetí SDP *Answer* má uzel A vše potřebné k navázání P2P spojení.

Postup výše popisuje proces návázání spojení z pohledu uzlu A. Je třeba brát v potaz, že pokud v bodě 5 uzel B má protějšek A v nějaké skupině (a tedy má zájem o spojení), začne navazovat spojení podobným způsobem jako uzel A, pouze v opačném směru.

Aby se zabránilo zbytečně velkému počtu vyměněných zpráv a možnosti navázání dvou spojení místo jednoho, jsou v návrhu následující opatření:

- Pokud uzel obdrží klíč protějšku ve zprávě *KeyRequest* dřív, než sám zažádá o klíč, použije tento klíč a už se sám nedotazuje.
- Pokud uzel obdrží *SdpOffer* od protějšku ve chvíli, kdy už mu sám *SdpOffer* zaslal, je nutné jedno z navazovaných spojení zahodit a druhé přijmout (viz obr. 4.6). Výběr spojení se provádí na základě lexikografického porovnání ID uzlů: spojení přijme uzel s menším ID.

Pokud se P2P spojení nepodaří navázat do 10 sekund od počátku výměny zpráv SDP, pokus se automaticky opakuje.

Pokud je v libovolném okamžiku přerušeno spojení se sig. serverem, všechny nedokončené pokusy o navázání P2P spojení jsou zrušeny a uzel zahájí opakované pokusy o nové připojení k serveru s rostoucími rozestupy (exponential backoff) až do intervalu 30 s.

4.6.2 Změna sledovaných protějšků

Kromě zmíněných typů zpráv použitých při navazování spojení je zde ještě zpráva typu *ChangeWatchedPeers*, která slouží, jak název napovídá, ke změně sledovaných protějšků. Své uplatnění si najde například v situacích, kdy uživatel na straně uzlu přidá nebo odebere uzel ze seznamu přátel.

Aby uzel mohl dostávat asynchronní notifikace o změnách stavu přátel a další signalizační zprávy, udržuje si spojení se sig. serverem po celou dobu běhu.

4.7 Protokol komunikace peer-to-peer

Protokol přímé komunikace mezi uzly se aktuálně omezuje pouze na zprávy související s vyjednáváním dostupných změn a jejich přenosem. Protože P2P kanál je ustaven z auten-

tizovaných SDP zpráv, dá se předpokládat, že i on bude důvěryhodný a autentizaci tedy již v P2P kanálu není třeba explicitně řešit.

Komunikace mezi uzly je stejně jako v případě komunikace se signalizačním serverem vedena formou JSON zpráv s povinnými poli určenými typem zprávy (pole `type`). Jedná se o následující:

- *ChangesOffer* – nabídka změn ve skupinách, které jsou danému protějšku dostupné. Nabídka pro určitou skupinu je reprezentovaná pomocí vektoru verzí (version vector, [36]), tj. pomocí seznamu dvojic (r, c), kde r je ID uzlu a c je hodnota celočíselného čítače, která je inkrementovaná na replice r s každou novou vytvořenou operací. Vektor verzí je získáván z knihovny pro CRDT. Neformálně řečeno popisuje, jakou nejnovější operaci má od každého uzlu k dispozici.
- *ChangesRequest* – žádost o zaslání změn ve vybraných skupinách od daného vektoru verzí.
- *Changes* – konkrétní změny ve vybraných skupinách od požadovaných vektorů verzí.

Počáteční výměna změn

Obrázek 4.4 zachycuje ideální průběh počáteční výměny zpráv mezi uzly. P2P komunikace probíhá od bodu označeného „P2P connection established“. I když je v obrázku a v dalším popisu naznačena pouze výměna zpráv zahájená uzlem A ($\rightarrow$ *ChangesOffer*, $\leftarrow$ *ChangesRequest*, $\rightarrow$ *Changes*), je třeba mít na paměti, že podobná komunikace zároveň probíhá i v opačném směru ($\leftarrow$ *ChangesOffer*, $\rightarrow$ *ChangesRequest*, $\leftarrow$ *Changes*).

Ihned po vytvoření P2P kanálu si oba koncové uzly vymění zprávy typu *ChangesOffer*, ve kterých shrnují lokální stav skupin, ke kterým má protějšek přístup. Stav skupin je popsán pomocí vektorů verzí příslušných CRDT struktur – konkrétně je to pro každou skupinu jeden vektor pro *seznam členů* a druhý vektor pro samotný *obsah*. Obsažen je též informativní *údaj o oprávněních*, kterými protějšek v dané skupině disponuje. V diagramu jde o řetězec „cRWmR“ – oprávnění číst i zapisovat obsah (content) a pouze číst seznam členů (members). Díky této informaci protějšek nemusí plýtvat přenosovým pásmem, když např. nemá šanci v dané skupině cokoliv ovlivnit.

Jakmile protějšek B obdrží *ChangesOffer*, vybere z nabízených skupin ty, které jsou mu známé, ověří, jestli uzel A skutečně má potřebná oprávnění k zápisu a konečně porovná vektory verzí s lokální replikou. Když z porovnání vyplyne, že uzel A disponuje novými změnami, vytvoří se požadavek na změny *ChangesRequest*, ve kterém se předává lokální (starší) vektor verzí.

Když uzel A obdrží *ChangesRequest*, je opět nucen ověřit oprávnění k dotazovaným skupinám. Díky starším vektorům verzí protějšku B je schopný sestavit a odeslat seznam pouze těch změn, které protějšek B ještě nemá. Z celkového pohledu jde o důležitou funkčnost, která *snižuje množství přenášených dat*.

Po přijetí změn protějšek B naposledy ověří oprávnění a konečně seznam operací aplikuje v příslušných CRDT strukturách u příslušných skupin.

Následné změny

V předchozí sekci byl nastíněn postup počáteční výměny změn po navázání P2P spojení. Obdobným způsobem probíhá distribuce změn i při následujících úpravách datového modelu.

Jakmile uzel přidá do jedné ze svých CRDT struktur operaci (například jako reakci na uživatelský vstup), spustí se *debouncer*, který shora omezuje maximální frekvenci (zdola minimální interval) zasílání změn. Pokud uživatel tedy provede více změn rychle po sobě, neodesílá se každá zvlášť, ale nejdříve se počká, než vyprší časovač debounceru. Výsledkem je to, že protějšek B dostává od uzlu A maximálně jednu notifikaci o změnách za n milisekund a nedochází tak k jeho zbytečnému zahlcování.

Změny vytvářené v aktuálním uzlu a změny přijaté od ostatních uzlů jsou proaktivně nabízené ostatním ve formě *ChangesOffer* zpráv zaslaných všem dostupným příslušníkům dotčené skupiny.

Změny se při správném nastavení oprávnění přes prostředníky mohou dostat i k uzlům, které nemají původce změn přidaného jako přítele. Dynamika šíření změn v síti připomíná gossip protokoly [16, 25].

4.8 Řešení konfliktů

Uživatelem vytvářená data skupiny jsou reprezentovaná pomocí CRDT pro JSON popsaného v sekci 3.6. V sekci 3.5 bylo naznačeno, že CRDT jsou navrženy tak, že z jejich pohledu je vždy každý konflikt nějak vyřešen: V příkladu na obrázku 3.2 dva uzly nezávisle zapisují hodnotu do stejného klíče „key“. Protože zmíněný CRDT pracuje v případě primitivních hodnot s hodnotou klíče jako s multi-value registrem, po synchronizaci změn jsou pod klíčem uloženy obě dvě hodnoty. Z pohledu CRDT jde o normální stav, nicméně otázka zní: *Je tento stav přípustný i z pohledu koncové aplikace?* V mnoha případech budou totiž podobné situace vyžadovat speciální zacházení a právě tyto typy konfliktů se náš rámec snaží zpřístupnit k dalšímu řešení.

Méně nápadná záludnost výše uvedeného případu spočívá v tom, že při uložení více hodnot do jednoho klíče strom přestal být platným stromem JSONu – JSON neumožňuje podobné uložení dvou hodnot bez použití např. pole. Aby se nezměnilo schéma stromu, v použité knihovně Automerge je situace vyřešena tak, že se z konfliktních alternativ vždy uměle deterministicky (porovnáním ID replik) vybere jedna hodnota a zbytek alternativ se uloží vedle pod skrytý klíč `_conflicts`.

Cílem tohoto rámce je tedy *umožnit podobné konflikty detekovat a případně vyřešit.*

4.8.1 Uživatelské řešení konfliktu

Prvním přístupem k řešení konfliktu je vyžádání zásahu uživatele. V koncovém programu je tedy třeba

- detekovat konflikt v dané hodnotě,
- zobrazit varování v GUI,
- na vyžádání uživatele nechat zobrazit dostupné alternativy,
- dovolit uživateli zapsat novou výslednou hodnotu, čímž se smažou staré konfliktní alternativy.

4.8.2 Automatické programově-specifické řešení konfliktu

V ideálním případě je možné v koncové aplikaci z dostupných alternativ jednu vybrat automaticky. Konkrétní realizace mohou být minimálně dvě:

Přístup 1

Při vykreslování obsahu je detekováno více konfliktních alternativ. Na základě programově-specifické logiky se vybere jedna z nich (popř. se vypočítá nová hodnota) a pouze ona se nechá vykreslit. Uložené *nevyužité alternativy se v datové struktuře ponechají*, zahozeny jsou automaticky při příštím zápisu hodnoty daného klíče.

Přístup 2

Při detekování více konfliktních alternativ se vybere jedna z nich (popř. se vypočítá nová hodnota) a *vygeneruje se editační operace*, která novou hodnotu zapíše. Díky novému zápisu se ihned odstraní konfliktní alternativy.

V případě, že by se v budoucnu v rámci zpřístupnila podpora undo z knihovny Automerge, tento přístup by vytvořil nežádoucí položku v lokální historii, která se nezakládá na vstupu uživatele, což by mohlo být matoucí.

4.9 Práce s lokální databází

Aby data repliky a především identita uživatele zůstaly zachovány i při zavření karty prohlížeče, je nutné využít lokálního úložiště (viz sekce 2.5). Perzistentní části datového modelu aplikace jsou oddělené v modulu `DbState`. Jedná se o

- identitu aktuálního uzlu,
- seznam přátel a
- seznam skupin vč. jejich členů a obsahu.

Data jsou aktuálně ukládána vcelku při každé změně perzistentní části datového modelu. V případě nutnosti by zjemnění zapisovaných záznamů mohlo zlepšit odezvu rámce.

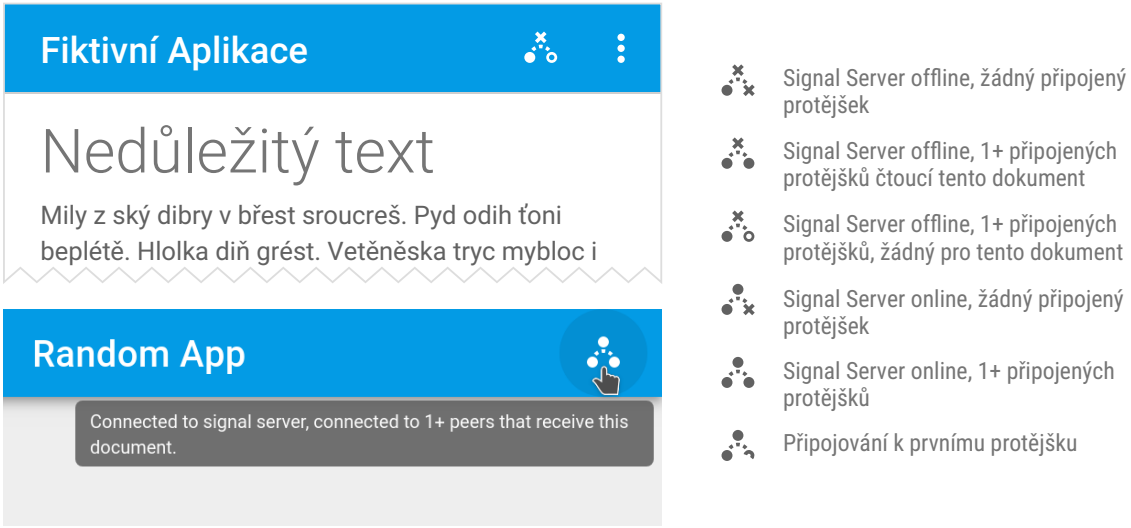
Protože lokální databáze aplikace je sdílená pro celý profil prohlížeče, při otevření dvou záložek v jednom profilu obě instance sdílí stejnou databázi a hrozí tak konflikty zápisu. Protože IndexedDB neumožňuje zasílání notifikací o změnách, koordinace mezi záložkami je problematická. **Rámec proto aktuálně nepodporuje více otevřených instancí cílové aplikace v jednom profilu**, resp. hrozí přepsání dat ostatních záložek. Druhým argumentem proti více záložkám je i to, že každá další otevřená záložka by zbytečně znovu navazovala spojení s protějšky.

Pokud cílová aplikace vyžaduje přepínání mezi více otevřenými dokumenty, měla by funkcionalitu implementovat uvnitř aplikace.

4.10 Grafické rozhraní pro koncového uživatele

Jednou z knihoven vytvořených v rámci této práce je i sada prvků uživatelského rozhraní pro indikaci stavu připojení k protějškům, správu protějšků, skupin, oprávnění a další.

Vytvoření této knihovny má za cíl usnadnit práci vývojářům, vytvořit rozsáhlejší praktickou ukázku použití základní knihovny a především *nabídnout koncovým uživatelům povědomé uživatelské rozhraní, které by mohli nacházet v různých aplikacích.*



Obrázek 4.7: Multifunkční ikona s indikátorem stavu – její použití a varianty.

4.10.1 Multifunkční ikona s indikátorem stavu

I když vytvořený rámec uvnitř výsledné aplikace zastane mnoho práce, z pohledu uživatele by měl po většinu času zůstat téměř neviditelný.

Vstupní branou k nastavení P2P funkcionality by v takovém případě měla zůstat jediná jednobarevná vícestavová ikona v hlavním panelu nástrojů (obr. 4.7).

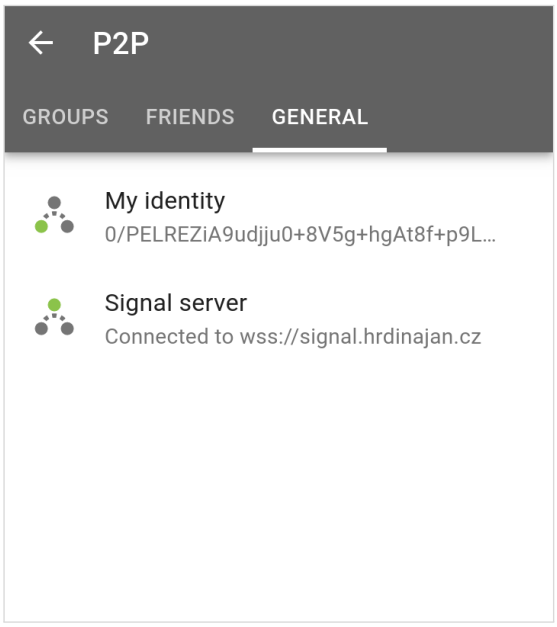
Obsah samotné ikony znázorňuje zjednodušené schéma sítě: bod vlevo dole reprezentuje aktuální uzel, bod nahoře signalizační server a bod vpravo dole protějšky. Komunikovat tak zvládne hned několik informací:

- Dostupnost signalizačního serveru – ať už je nedostupný, připojený, nebo aktuálně probíhá pokus o připojení.
- Indikátor připojených protějšků – značí jestli je připojen alespoň jeden protějšek replikující aktuálně otevřený dokument, popř. jestli aktuálně probíhá pokus o připojení.
- Slovní popis stavu – dostupný po najetí myši nebo po dlouhém podržení ikony na dotykových zařízeních.

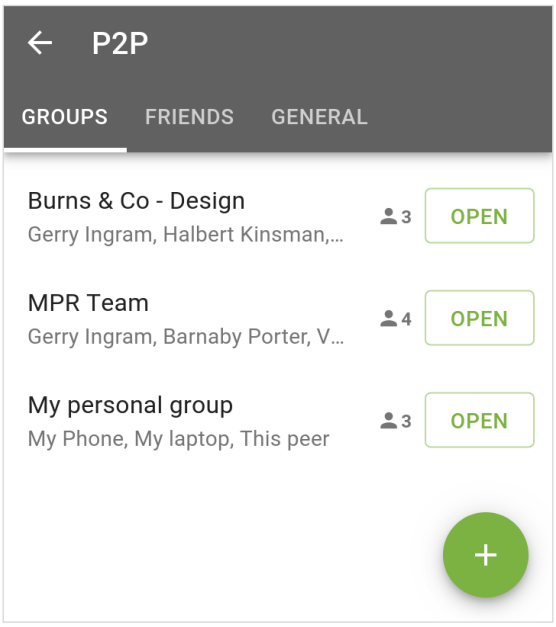
4.10.2 Obrazovka obecných informací a nastavení

Obrazovka obecných informací (obr. 4.8) poskytuje stručný přehled o stavu P2P modulu. Zejména jde o

- identifikátor aktuálního uzlu pro sdílení s ostatními,
- adresu signalizačního serveru a možnost ji změnit a
- informace o stavu spojení se signalizačním serverem.



Obrázek 4.8: Obrazovka obecných informací.



Obrázek 4.9: Obrazovka se seznamem skupin.

4.10.3 Seznam přátel

Seznam protějšků (obr. 4.10) se na první pohled podobá seznamům přátel z klientů pro instant messaging nebo např. z Facebooku. Funkčnost je v zásadě podobná, i když několik odlišností vycházejících z povahy P2P lze objevit.

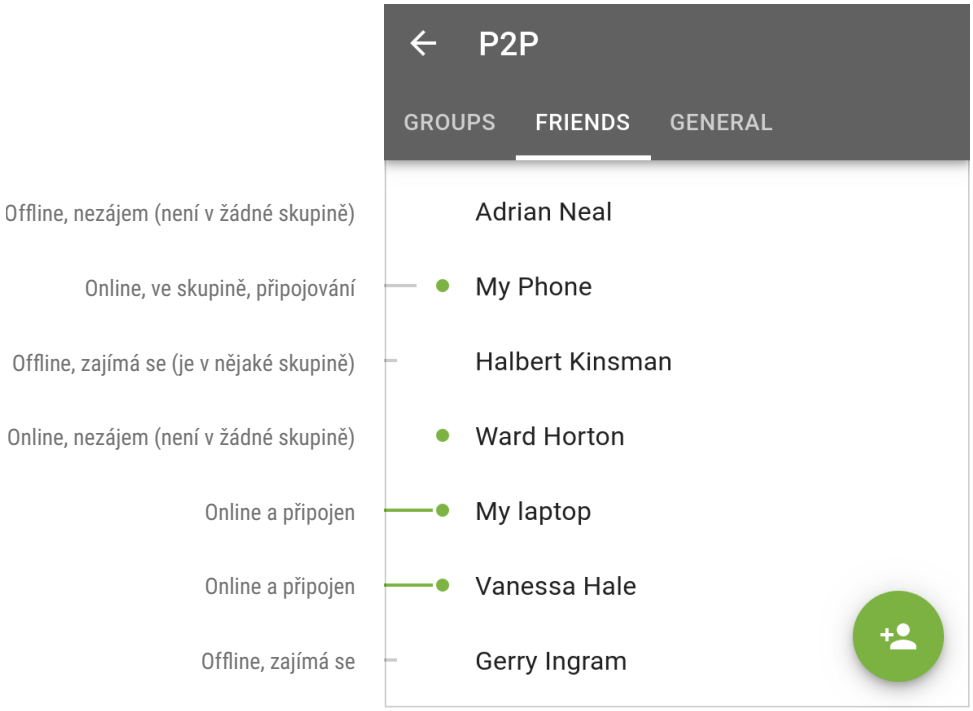
Jednotlivé řádky například nereprezentují jednotlivé fyzické osoby, ale přímo instance aplikace na koncových zařízeních, tzn. například typicky laptop a mobil Karla Nováka zde bude reprezentován jako dvě rozdílné položky.

Co se týče *stavu protějšků*, kromě typického „uzel je online“ (zelený puntík) a „uzel je offline“ (bez puntíku) jsou pomocí dalších značek a animací znázorněny i doplňující informace, např.

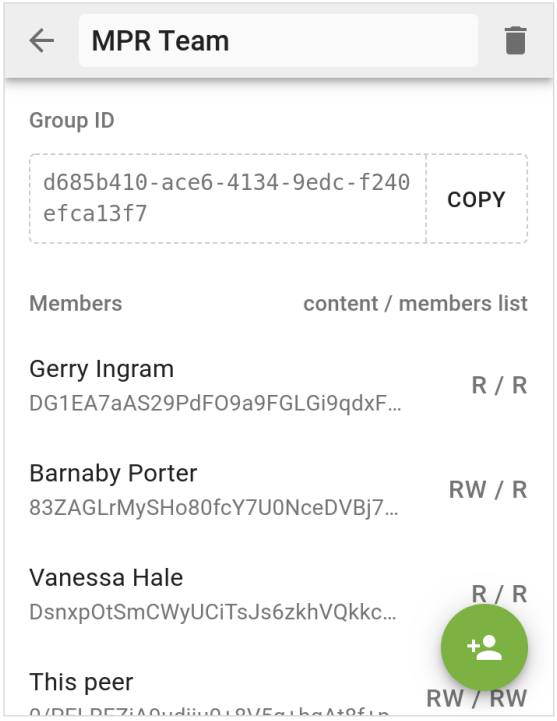
- informace o tom, jestli je daný uzel lokálně přidaný v nějaké skupině a tedy jestli má aktuální uzel zájem se k němu připojit nebo
- stav spojení s daným uzlem.

4.10.4 Správa skupin

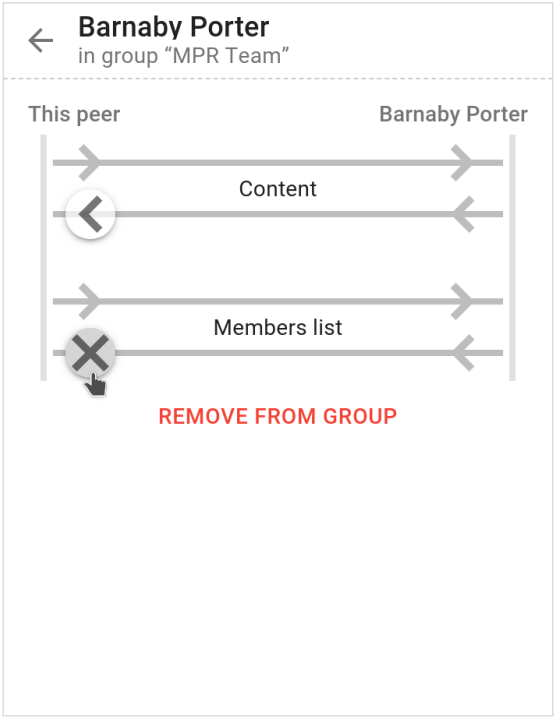
Obrazovky pro správu skupin (obr. 4.9 a 4.11) slouží k vytváření nových nebo přidávání a upravování existujících skupin. U každé skupiny je potom možné zobrazit a upravovat její členy a jejich oprávnění. Pro výběr nového uzlu do skupiny je k dispozici fulltextové vyhledávání. Nastavení oprávnění člena ve skupině je graficky znázorněné pomocí interaktivního diagramu (viz obr. 4.12). Více informací o fungování skupin je možné najít v sekci 4.2.



Obrázek 4.10: Obrazovka se seznamem přátel. Vysvětlivky stavů popisují pohled lokálního uzlu.



Obrázek 4.11: Obrazovka pro správu členů skupiny.



Obrázek 4.12: Obrazovka pro nastavení oprávnění člena skupiny.

Kapitola 5

Implementace

Tato kapitola popisuje vybrané části implementace, které se týkají hlavní funkcionality rámce nebo jsou nějakým způsobem zajímavé.

Sekce postupují od implementačních detailů dílčích modulů přes realizaci architektury aplikace a řešení postranních efektů až po vysokoúrovňové veřejné API.

První část kapitoly (sekce 5.1 a 5.2) se zabývá hlavně zpřístupněním implementace replikovaných datových typů a zpřístupněním konfliktních hodnot pro řešení konfliktů.

Značná část kapitoly (sekce 5.6, 5.7 a 5.8) se věnuje realizaci návrhových vzorů ze sekce 2.8. Je těžké najít doporučení pro vývoj rozsáhlých funkcionálních knihoven podobného ražení a často proto bylo nutné vyzkoušet různé, často diametrálně odlišné přístupy. Robustní reprezentace vnitřního stavu a práce s postranními efekty přitom byly zásadní pro úspěšné dokončení a udržitelnost projektu.

Po implementaci práce s postranními efekty následuje jedna sekce věnovaná implementaci GUI (5.10) a kapitolu zakončují sekce o implementaci a nasazení signalizačního serveru (5.11 a 5.12).

5.1 Vlastní funkcionální implementace CRDT knihovny

I když to nebylo vyžadováno ze strany zadání, vytvořil jsem na základě článku M. Kleppmanna [27] vlastní čistě funkcionální implementaci CRDT pro JSON popsaného v sekci 3.6.

Článek definuje vlastní příkazový jazyk podobný JavaScriptu, pro který potom definuje formální sémantiku prostřednictvím rozsáhlých přepisovacích pravidel. Tímto způsobem je popsána celá funkčnost algoritmu.

Zápis programu jako jednoho velkého výrazu je sice na první pohled méně srozumitelný, než například imperativní pseudokód, na druhou stranu ale při přepisu do staticky typovaného funkcionálního jazyka, jako je Reason, psaní probíhá docela hladce, protože podobné uvažování o programu zde není až tak cizí.

Velkou pomocí je v takovém případě kompilátor s typovou inferencí, expresivní typy a zejména pattern matching, který pomáhá pokrýt všechny kombinace vstupních hodnot.

Při implementaci jsem se snažil držet pojmenování proměnných a funkcí použitého v článku, aby bylo snadné dohledat případné chyby.

Vlastní implementaci jsem nakonec v rámci nepoužil, ale zůstal jsem u knihovny Automerge, na jejímž vývoji pracuje autor článku. Hlavním důvodem je, že poskytnutá knihovna

doplňuje k algoritmu množství optimalizací a vylepšení, které v článku chybí a kterých jsem si před začátkem vlastního pokusu nevšiml.

Přesná implementace podle článku uchovává značně neefektivním způsobem kauzální závislosti operací, čímž dramaticky zvyšuje paměťovou náročnost. Dále tato implementace používá kurzory k označování prvků JSON stromu pro manipulaci, což vede ke krkolomnému použití ve zdrojovém kódu. Automerge tento problém řeší generováním UUID pro každý vytvořený objekt a možností používat toto ID k dereferenci zanořených objektů. Aby bylo možné v Automerge snadno uchovávat index mapování *UUID* → *objekt*, je vnitřní reprezentace JSON stromu zploštělá, zatímco podle článku je rekurzivně zanořená a kopíruje tvar modelovaného stromu.

Protože začlenění uvedených změn do vlastní implementace by bylo netriviální a potenciálně časově náročné, odložil jsem jej jakožto možnost dalšího vývoje (viz sekce 8.1.1).

Vlastní čistě funkcionální implementaci CRDT pro JSON je možné najít uvnitř složky projektu *bs-automerge*.

5.2 Binding k Automerge

Knihovna Automerge nabízí sofistikované aplikační rozhraní, které v implementaci využívá neměnné datové struktury (Immutable.js) a JavaScript Proxies. Protože knihovna je napsaná v JavaScriptu, bylo nutné vytvořit binding pro Reason *bs-automerge*, který

- zpřístupňuje potřebné funkce,
- doplňuje k nim bezpečné statické typy,
- je v souladu s funkcionálním stylem programování a zároveň
- poskytuje rozhraní dost abstraktní na to, aby bylo možné použít jej v budoucnu i pro čistě funkcionální nativní implementaci.

Původní aplikační rozhraní pro vytváření změn je znázorněné ve výpisu 5.1: funkci `change` je předána modifikační funkce, která jako parametr obdrží dočasný dokument `doc` obalený JS Proxy, která zaznamenává všechny *imperativní* změny v objektu a převádí je na popisy operací. Výsledkem je nový stav dokumentu, přičemž původní stav `doc1` zůstane nezměněn (tj. funkce `change` neprovádí žádné přímé mutace).

```
const doc1NewState = Automerge.change(doc1, 'Add task', doc => {
  // Vsechny zmeny v objektu doc jsou zaznamenany pomoci JavaScript Proxy
  // a prevedeny na popisy operaci.

  doc.tasks.insertAt(0, {title: 'Buy milk', done: false})
});
```

Výpis 5.1: Přidání položky v Automerge v JavaScriptu.

Stejná operace přepsaná ve vlastním funkcionálním API v Reasonu je vidět ve výpisu 5.2. Typovou anotaci JSON struktury jsem záměrně navrhl tak, aby bylo nutné při každém čtení z JSON stromu pomocí pattern matchingu ověřit, jaký typ prvku se na dané pozici nachází. Na úkor stručnosti se tak zamezí řadě běhových výjimek vycházejících ze situací, kdy například aplikace na daném klíči předpokládá pole, i když se tam nachází řetězec.

```
let doc1NewState =
  AM.(
    doc1
    |> change("Add task", doc =>
         Json.(
           switch (doc |> Map.get("tasks") |?> List.ofJson) {
           | Some(tasks) =>
             doc
             |> Map.add(
                  "tasks",
                  tasks
                  |> List.prepend(
                       Map.(
                         create()
                         |> add("title", string("Buy milk"))
                         |> add("done", bool(false))
                         |> toJson
                       ),
                     )
                  |> List.toJson,
                )
           | None => doc
           }
         )
       )
  );
```

Výpis 5.2: Přidání položky pomocí bindingu v Reasonu.

V kontextu rámce je ověřování tvaru čtených replikovaných dat o to důležitější, když data mohou pocházet od jiných uzlů se starší verzí aplikace a podobně.

5.2.1 Zpřístupnění konfliktních hodnot

Všechny operace pro čtení z replikovaného JSON objektu jsou implementované ve dvou variantách:

1. *Jednoduchá varianta* (např. `Map.get`, `List.foldLeft`) vrací v případě více zapsaných konfliktních hodnot jedinou výchozí hodnotu, která je uměle deterministicky vybrána na základě porovnání ID replik (ID uzlů).

2. *Varianta zakončená „C“ (conflicts)* (`Map.getC`, `List.foldLeftC`, ...) vrací místo přímé hodnoty objekt s výchozí hodnotou a výčtem případných konfliktních alternativ indexovaných podle ID repliky (ID uzlu).

Běžná varianta by měla být používána pouze ve výjimečných případech, kdy je možné konflikty ignorovat. Ve všech ostatních případech je vhodnější použít varianty s výčtem konfliktů.

5.3 Zpřístupnění knihoven a webových API JavaScriptu pro použití v jazyce Reason

BuckleScript (překladač Reasonu do JavaScriptu) v základu aktuálně podporuje pouze malou část ze škály webových rozhraní dostupných v dnešních prohlížečích: funkce pro manipulaci s DOM a práci se soubory. Všechny ostatní potřebné bindingy bylo nutné ručně dopsat. Jedná se o

- WebSockets,
- TextEncoder/TextDecoder (API pro převod textu na/z pole bajtů),
- IndexedDB,
- události pro detekci ztráty a navázání připojení k internetu,
- WebCrypto a
- WebRTC, jehož značná část byla převzata z existujícího projektu ReWebRTC[1].

Při vytváření typových anotací jsem vycházel ze specifikací W3C a z typových anotací pro TypeScript[2]. Protože Reason/OCaml nepodporují cyklické závislosti mezi moduly, jsou všechny definice typů oddělené od signatur funkcí ve zvláštních modulech s příponou `Types`.

V některých případech (WebCrypto) jsem z časových důvodů pokryl pouze ty části, které v projektu využívám, u jiných modulů (IndexedDB) jsem pokryl prakticky celý povrch rozhraní. Protože se ve všech případech jedná o rozhraní, která si své využití najdou napříč různými projekty, zveřejnil jsem bindingy v nezávislé knihovně `bs-webapi-extra`.

5.3.1 SimpleCrypto

SimpleCrypto je vlastní modul, který si klade za cíl zjednodušit práci s kryptografickou funkcionalitou takovým způsobem, aby nebylo nutné zabývat se konkrétními použitými algoritmy, ale radši pracovat na abstraktní úrovni s „hašovacím algoritmem“, „asymetrickou šifrou“ a „symetrickou šifrou“. Rozhraní je navržené tak, aby z něj neprosakovaly platformě specifické typy, tj. tak, aby mohlo být používáno jednotně jak v nativním, tak ve webovém prostředí.

5.3.2 SimpleRTC

SimpleRTC je modul, který abstrahuje a výrazně zjednodušuje práci s nízkoúrovňovým API WebRTC v prohlížeči. Při jeho implementaci jsem vycházel z populární knihovny *simple-peer*[3], která plní podobný úkol v JavaScriptu.

Při používání knihovny jsem narazil na omezení *maximální velikosti zpráv*. Podporované velikosti se liší podle prohlížeče, popř. kombinace prohlížečů. Za rozumnou široce podporovanou velikost je považováno 65 535 B [34], čemuž při čtyřbajtové maximální velikosti jednoho znaku v UTF-8 odpovídá 16 383 znaků bezpečně přeneseného textu.

[1] Stránky projektu ReWebRTC: https://github.com/bsansouci/ReWebRTC

[2] Typové anotace webových API v jazyce TypeScript: https://github.com/Microsoft/TypeScript/blob/master/lib/lib.dom.d.ts

[3] https://github.com/feross/simple-peer

Pro vyřešení problému jsem navrhl jednoduchý protokol, který umožňuje zasílat větší zprávy rozdělené na více dílů. Protože WebRTC umožňuje přenášet buďto textová nebo libovolná binární data, implementoval jsem podobnou funkcionalitu. Při odesílání dat přes WebRTC je nejdříve zaslána úvodní zpráva v JSONu, která popisuje, co bude následovat. Obsahuje pole

- `payloadType`, které popisuje, jestli následující binární data nesou řetězec nebo ne, a
- `payloadLength`, které vyjadřuje celkovou velikost nákladu v bajtech.

po úvodní zprávě následují zprávy s binárními daty, které jsou na straně příjemce pospojovány a v případě `payloadType == "string"` převedeny na textový řetězec. Funkcionalita dělení zpráv je implementovaná v modulu *SimpleRTCChunker*.

5.4 Knihovna pro rychlé diffy

V sekci 2.8.1 bylo zmíněno, že neměnné datové typy zpravidla umožňují levné porovnání struktur s malým počtem změn. Bohužel, jedinou nalezenou implementací symetrických diffů v Reasonu/OCamlu je ta z knihovny Base od Jane Street[4]. Tato knihovna není kompatibilní s překladačem BuckleScript, takže jsem dotčené algoritmy pro `Set` a `Map` portoval pro použití se standardní knihovnou OCamlu. Výsledek jsem zveřejnil jako oddělenou knihovnu *ocaml-diff*.

V knihovně pro koncový uzel je možné setkat se s jejím použitím v modulech `PeersGroupsSynchronizer`, `PeersKeysFetcherAndSender` a `PeersStatuses`. Kromě toho je využívána i v implementaci knihovny *bs-black-tea* (viz sekce 5.6).

5.5 Vícevrstvá reprezentace zpráv

Zprávy pro komunikaci se signalizačním serverem jsou v aplikaci reprezentované pomocí variant, které uložené údaje ve zprávě efektivně rozdělují do vrstev. Princip vzdáleně připomíná vrstvy datagramů v ISO/OSI modelu – jednotlivé subsystémy si při zpracování zprávy mohou přečíst pro ně relevantní hlavičku (např. na straně serveru hlavičku `PeerToPeer` s cílovou destinací) a provést potřebnou operaci (přeposlání zprávy), aniž by se musely zajímat o zbytek zprávy. Definice zpráv je znázorněna ve výpisu 5.3.

Díky konečnému výčtu všech typů zpráv je v aplikaci možné provádět pattern-matching nad celými zprávami nebo nad jejich podčástmi (např. ošetření reakce na všechny `peerToPeer` náklady zpráv, viz výpis 5.4). Pokud programátor zapomene ošetřit některý z typů zpráv (například při přidání nového), překladač ho na to upozorní.

Na rozdíl od reprezentace zpráv s dynamickými poli zde nehrozí nebezpečí odeslání zprávy, které například chybí vyplněný podpis, protože typ jednoznačně definuje všechny povinné údaje a odesílací funkce přijímá jako argument pouze kompletní zprávy typu `t`, nikoliv její dílčí složky.

[4]Stránky knihovny Base: https://opensource.janestreet.com/base

```
type peerToServerMsg =
  | LoginReq(key)
  | Login(/* Challenge */ string, /* Watched peers */ PeerId.Set.t)
  | Logoff
  | ChangeWatchedPeers(/* Watched peers */ PeerId.Set.t);

type peerToPeerMsg =
  | Offer(sdp)
  | Answer(sdp)
  | KeyRequest(key)
  | KeyResponse(key);

type serverToPeerMsg =
  | Challenge(string)
  | Error(error)
  | Ok(PeerId.Set.t)
  | WatchedPeersChanged(list(peerChange));

type signedMsg =
  | PeerToServer(PeerId.t, peerToServerMsg)
  | PeerToPeer(PeerId.t, PeerId.t, peerToPeerMsg);

type t =
  | Signed(/* Signature */ string, signedMsg)
  | Unsigned(serverToPeerMsg);
```

Výpis 5.3: Vícevrstvá definice zpráv.

```
let handleP2PMsg = (msg: peerToPeerMsg) =>
  switch (msg) {
  | Offer(sdp) => /* message handling... */
  | Answer(sdp) => /* message handling... */
  | KeyRequest(key) => /* message handling... */
  }
```

Výpis 5.4: Reakce na `peerToPeer` náklady zpráv. Překladač upozorní, že chybí ošetření hodnoty `KeyResponse(key)`.

5.6 Realizace The Elm Architecture

Chování knihovny pro funkcionalitu koncového uzlu i uživatelského rozhraní jsou postavené na architektuře Elmu popsané v sekci 2.8.2. Existující port architektury pro jazyk Reason se nazývá *bucklescript-tea*.

Bohužel, zmíněná knihovna je pevně svázána s vlastní implementací vykreslování HTML a znemožňuje tak např. použití Reactu a přilehlého ekosystému hotových komponent GUI. Z toho důvodu jsem vytvořil odlehčenou odnož nazvanou *bs-black-tea*, ve které jsem odstranil všechnu logiku spojenou s vykreslováním a změny v datovém modelu aplikace jsem zpřístupnil jako událost, na kterou je možné napojit vykreslovací logiku dle vlastního výběru.

Přepracoval jsem též vnitřní fungování subscriptions, které v původní knihovně vykazovalo zbytečné reaktivace odběrů při změnách pořadí registrace. V nové implementaci je pro efektivní rozdíly využita knihovna ze sekce 5.4.

5.7 Reaktivní programování s knihovnou Wonka

Knihovna Wonka[5] nabízí implementaci standardu Callbag (viz sekce 2.8.3) napsanou v jazyce Reason. Nabídka přibalených operátorů není příliš rozsáhlá, bylo proto nutné doimplementovat několik vlastních: operátor pro *vytvoření streamu z Promise*[6], operátor `filterMap` pro možnost filtrovat a rovnou transformovat vybrané zprávy a v neposlední řadě operátor `retryWhen` zajišťující *automatické opakování vstupních streamů při selhání*. Operátor umožňuje specifikovat funkci pro zvětšování prodlevy na základě počtu neúspěšných pokusů.

5.8 Práce s postranními efekty

V téměř každé webové aplikaci je nutné nějakým způsobem zajistit komunikaci s vnějším světem pomocí postranních efektů. Naše knihovna je z tohoto pohledu obzvlášť náročná, protože potřebuje sledovat a řídit mnoho asynchronních procesů:

- pokusy o navázání spojení se signalizačním serverem,
- sledování změn stavů protějšků,
- automatické navazování P2P spojení s těmito protějšky,
- kontrolu a řízení jednotlivých fází navazování spojení,
- automatické opakování pokusů v případě neúspěchu nebo vypršení časového limitu,
- rušení spojení (pokusů o připojení) a dealokace prostředků v případě ztráty zájmu o připojení,
- řízení asynchronních kryptografických funkcí,
- atd...

[5] https://github.com/kitten/wonka

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

```
type model = {timeoutId: option(Timeout.id)};

[@bs.deriving accessors]
type msg =
  | ClickedStart
  | ClickedStop
  | TimeoutCreated(Timeout.id)
  | TimeoutFinished;

let init = () => {timeoutId: None};

let update = (model, msg) =>
  switch (msg) {
  | ClickedStart => (
      model,
      switch (model.timeoutId) {
      | None => Timeout.runCmd(timeoutCreated, timeoutFinished, 4000)
      | Some(id) => Cmd.none
      },
    )
  | TimeoutCreated(id) => ({...model, timeoutId: Some(id)}, Cmd.none)
  | ClickedStop => (
      {...model, timeoutId: None},
      switch (model.timeoutId) {
      | Some(id) => Timeout.cancelCmd(model.timeoutId)
      | None => Cmd.none
      },
    )
  | TimeoutFinished => ({...model, timeoutId: None}, Cmd.log("Finished"))
  };
```

Výpis 5.5: Zrušitelný časovač implementovaný pomocí TEA `cmd`.

Správné zvládnutí práce s postranními efekty bylo *naprosto kritické* pro udržitelnost vývoje projektu. Hlavními úkoly bylo omezení příležitostí k chybám, zajištění dobré čitelnosti kódu a snížení objemu kódu nutného k dosažení požadované funkcionality.

Ve funkcionálním prostředí Reasonu jsem postupně prozkoumal několik přístupů:

Přístup 1: Postranní efekty pomocí TEA `cmd`

Protože knihovna bucklescript-tea nenabízí mnoho implementací odběrů (subs), použití `cmd` se nabízí jako první možnost. Jak bylo naznačeno v sekci 2.8.2, `cmd` umožňuje čisté funkci `update` vrátit popis asynchronní akce, která se má provést. Konstruktoru `cmd` je zpravidla možné předat typ vlastní zprávy, která se má po dokončení zavolat. Z pohledu imperativního programování je možné konstrukci přirovnat k zavolání funkce s návratovým typem `void`, která přijímá jako jeden z parametrů callback.

Použití `cmd` je vhodné pro jednorázové akce, které není potřeba rušit před dokončením. V okamžiku, kdy je nutné podporovat rušení, se použití komplikuje. Na ukázce 5.5 je znázorněná logika zrušitelného časovače implementovaného pomocí `cmd`. Po kliknutí na tlačítko „Start“ se spustí časovač, kliknutím na „Stop“ se časovač předčasně zruší. Při doběhnutí časovače se vypíše „Finished,“ v jeden okamžik může běžet maximálně jeden časovač.

```
type model = {active: bool};

[@bs.deriving accessors]
type msg =
  | ClickedStart
  | ClickedStop
  | TimeoutFinished;

let init = () => {active: false};

let update = (model, msg) =>
  switch (msg) {
  | ClickedStart => ({...model, active: true}, Cmd.none)
  | ClickedStop => ({...model, active: false}, Cmd.none)
  | TimeoutFinished => ({...model, active: false}, Cmd.log("Finished."))
  };

let subscriptions = model =>
  if (model.active) {
    Timeout.runSub(timeoutFinished, 4000);
  } else {
    Sub.none;
  };
```

Výpis 5.6: Zrušitelný časovač implementovaný pomocí TEA `sub`.

Zátěž programátora je vysoká, protože musí ručně řídit životní cyklus časovače, tj. musí ručně spravovat referenci s ID časovače, čímž se znečišťuje datový model a zvyšuje se počet potřebných typů zpráv.

Přístup 2: Postranní efekty pomocí TEA `sub`

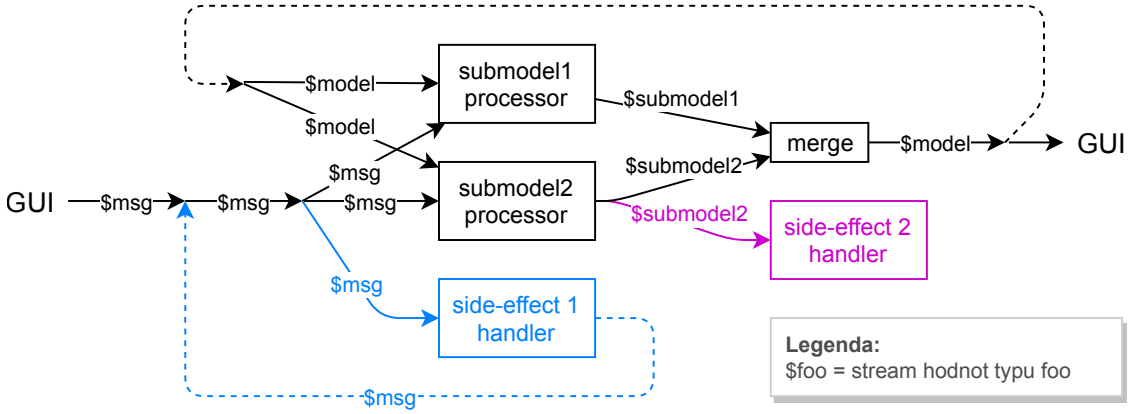
Subscriptions (odběry) popisují trvající záměry k provádění postranních efektů. Příklad s časovačem přepsaný pomocí `sub` je znázorněn ve výpisu 5.6: datový model i programová logika se oproti `cmd` značně zjednodušily, správa životního cyklu časovače se skryla do implementace odběru. Funkce `subscriptions` definuje, co má probíhat ve vnějším světě v závislosti na modelu. Zátěž programátora je snížena, protože pro aktivaci/deaktivaci časovače mu stačí změnit model.

Přístup 3: Reaktivní programování jako úplná náhrada TEA

Jak bylo naznačeno v sekci 2.8.3, knihovny pro reaktivní programování nabízí mocné abstrakce pro práci se streamy asynchronních hodnot. Možností, která se v tomto kontextu nabízí, je použít reaktivní programování pro celou architekturu aplikace.

Tok dat v programu v takovém případě může vypadat například tak, jak je znázorněno na obrázku 5.1: na vstupu existuje stream `$msg` všech událostí přicházejících z GUI a `$model` s hodnotami modelu celé aplikace. Jednotlivé podcelky aplikace jsou modelovány jako operátory, které po přijetí nové zprávy emitují nový stav podcelku.

Výhodou uvedeného principu je vysoká flexibilita: postranní efekty mohou být spouštěny jak na základě příchozích zpráv (v diagramu modře), tak na základě změn stavu programu



Obrázek 5.1: Architektura aplikace s využitím pouze reaktivního programování a streamů.

(v diagramu fialově) nebo jako reakce na libovolný stream. To stejné platí pro jednotlivé operátory pro manipulaci s modelem.

Nevýhodou je, že tento návrhový vzor vede programátora k tomu modelovat asynchronně i funkčnost, která by asynchronní vůbec být nemusela. Je potom těžší sledovat tok programu (v jakém pořadí se co vyhodnocuje) a potenciálně se zhoršuje i výkon. Pokud například `submodel1` i `submodel2` z obrázku 5.1 oba dva chtějí zareagovat na událost X, nové hodnoty do bloku `merge` vstupují nezávisle těsně po sobě, což bez speciálních opatření (debouncing) může vést ke zbytečnému vícenásobnému překreslení GUI.

Přístup 4: Kombinace výše uvedeného

Po odzkoušení všech možných přístupů jsem dospěl k závěru, že nejlepším řešením je použít každý přístup v těch částech programu, ve kterých vyniknou jeho silné stránky.

Případy použití pro `cmd` a `sub` byly nastíněny v přístupech 1 a 2. Co se týče knihovny pro reaktivní programování, ukázalo se, že streamy se dají velice elegantně převést na TEA subscriptions (viz `Subs.ofStream` v knihovně *pocket-mesh-peer*) a následně použít pro implementaci vybraných složitějších TEA odběrů. Jako příklad je možné uvést vlastní moduly `RTCSub` a `WebSocketsSub`, ve kterých jsou streamy použity pro správu životních cyklů spojení, příjem zpráv a automatické opakování pokusů o připojení s rostoucím rozestupem (exponential backoff).

Ve výpisu 5.7 je vidět ukázka kódu z modulu `PeersConnections`, který pomocí `RTCSub` zajišťuje, aby ke každému ze seznamu dostupných protějšků existovala iniciativa o navázání spojení. Když opadne zájem o připojení k protějšku, stačí jej odstranit ze seznamu a mapování automaticky zajistí, že se zruší pokus o navázání spojení (popř. ukončí se aktivní spojení), uvolní se prostředky WebRTC a zruší se i všechny případné vnitřní časovače.

5.8.1 Shrnutí poznatků z řešení postrannních efektů

Získané poznatky je možné shrnout do následujících doporučení:

- *Maximum stavu a jeho změn je vhodné nechat v čistých* `update` *funkcích.* U čistých funkcí všechny části programu reagují na vstupní parametry (zprávy) zároveň, tj. synchronizovaně, je snadné sledovat pořadí vyhodnocování programu a tok dat.

```
let subscriptions = model =>
  model.onlinePeers
  |> Set.map(peerId => RTCSub.connectionSub(peerId, receivedRtcMsg))
  |> Set.toList
  |> Tea.Sub.batch;
```

Výpis 5.7: Mapování seznamu protějšků na záměry k navázání spojení (pseudokód na motivy modulu `PeersConnections`).

- ***`Subs`** jsou ve většině případů lepší volbou, než pouhé **`cmds`**.* `Subs`, jak bylo naznačeno výše, jsou vhodnějším kandidátem, jakmile je třeba modelovat alespoň jedno z následujícího:
  - trvající záměry k interakcím ve vnějším světě,
  - procesy, které mají vlastní životní cyklus – potřebují na začátku alokovat a na konci uvolnit prostředky nebo
  - postranní efekty, které má být možné předčasně zrušit.

  `Cmds` se hodí spíše pro krátké jednorázové operace (logování, odeslání dat na server) a další operace, které není nutné přerušovat.
- *Komplikovanější asynchronní procesy* jsou vhodným kandidátem pro převod na streamy zabalené uvnitř subscriptions, pokud např.
  - se asynchronní proces opakuje v programu na více místech nebo
  - pomocné proměnné by zbytečně zanášely model.

5.9 Veřejné API

Nejrozsáhlejší veřejné aplikační rozhraní určené pro použití vývojářem cílové aplikace je API knihovny *pocket-mesh-peer* popsané v souboru `PocketMeshPeer.rei`.

Rozhraní jednoznačně deklaruje všechny moduly, funkce a typy z vnitřní implementace, které mají být viditelné mimo knihovnu.

5.9.1 Neprůhledné datové typy

Většina zpřístupněných typů je *neprůhledná*, tj. uživatel (programátor) nevidí, jak je typ vnitřně realizován a nemůže sám přímo vytvářet jeho instance – může je obdržet pouze jako výsledek nějaké funkce. Díky použití tohoto přístupu je možné měnit vnitřní implementaci datových struktur v knihovně, aniž by se změnilo vnější rozhraní. Druhou výhodou neprůhledných typů je větší bezpečnost oproti přímému použití primitivních typů.

Typickým příkladem může být například modul `Peer.Id`: ID uzlu je sice vnitřně reprezentované jako řetězec, z vnějšku je to ale skryto, takže je uživatel při vytváření nucen použít konstruktor `Peer.Id.ofString`. To zpřístupňuje následující:

- Konstruktor *může obsahovat validátor* a vrátit nový identifikátor pouze v případě, že řetězec splňuje požadavky, např. požadavek na minimální délku. Pokud se potom někde v programu objeví instance typu `Peer.Id.t`, je na rozdíl od řetězce jisté, že jde o platný identifikátor.

```
// Vnitrni implementace typu popisujiciho stav cele knihovny v Store.re
type t =
  | WaitingForDbAndIdentity(Db.t, RuntimeState.InitConfig.t, SignalChannel.t)
  | HasIdentity(Db.t, DbState.t, RuntimeState.t);

// Zjednodusena varianta pro verejnost v PocketMeshPeer.rei
type taggedT =
  | WaitingForDbAndIdentity(SignalChannel.t)
  | HasIdentity(DbState.t, RuntimeState.t);
```

Výpis 5.8: Vnitřní a veřejná reprezentace stavu knihovny *pocket-mesh-peer*.

```
const state = {
  db: db,
  initConfig: initConfig,
  signalChannel: signalChannel,
  dbState: null,
  runtimeState: null,
};
```

Výpis 5.9: Stav aplikace modelovaný v JavaScriptu vytváří zbytečný prostor pro chyby kvůli nutnosti ručně vytvářet a nulovat reference.

- *Je sníženo riziko prohození hodnot při předávání parametrů.* Pokud vezmeme v úvahu funkci `addPeerToGroup` se signaturou `(string, string, t) => t` a její volání `addPeerToGroup("aaa", "bbb", groups)`, překladač nijak nebrání zadat ID uzlu na místo ID skupiny a opačně. Na druhou stranu při použití neprůhledných typů a vyplývající signatury `(Peer.Id.t, Group.Id.t, t) => t` prohození vyvolá typovou chybu při překladu, i když uvnitř jsou `Peer.Id.t` a `Group.Id.t` implementovány stejně.

5.9.2 Znemožnění reprezentace neplatných stavů na úrovni datového modelu

V datovém modelu aplikace jsou využívány algebraické datové typy (varianty) pro modelování stavu aplikace v nejlepším případě takovým způsobem, aby neplatné stavy vůbec nebylo možné reprezentovat [18, 19]. Dobrým příkladem je typ reprezentující stav celé knihovny (výpis 5.8): v typu je explicitně dané, že v okamžiku, kdy ještě nejsou načtena data z databáze a není známá identita aktuálního uzlu (konstruktor `WaitingForDbAndIdentity`), nemá cenu držet v paměti běhový stav zbytku aplikace (`RuntimeState`) ani prázdné kolekce pro načtená data z databáze (`DbState`).

V imperativním jazyku by vše typicky bylo modelované jako jeden objekt bez explicitně definovaných stavů (viz výpis 5.9). Programátor v takové situaci musí pamatovat na správné vytváření a nulování objektů pro jednotlivé stavy, čímž se zvyšuje riziko běhové chyby (null pointer exception) a plýtvání pamětí.

5.9.3 Zjednodušené veřejné varianty

U vybraných algebraických datových typů veřejné rozhraní definuje jejich zjednodušené verze, které skrývají nepotřebné implementační detaily. Příkladem může být typ `State.t` reprezentující stav celé knihovny. Implementace původního stavu `State.t` je skrytá a pro pattern-matching je nutné použít zjednodušený typ `State.taggedT` (viz výpis 5.8) získaný pomocí funkce `classify: t => taggedT`.

5.10 Implementace GUI

Prvky uživatelského rozhraní jsou implementovány v oddělené knihovně *pocket-mesh-peer-material-ui*. Pro vykreslování HTML je využit binding ReasonReact[7], který zpřístupňuje oblíbenou knihovnu React[8] pro použití v jazyce Reason. Pro zajištění konzistentního vzhledu je použita sada grafických komponent Material-UI[9] a její binding bs-material-ui[10].

Protože kvůli chybě v bs-material-ui docházelo při použití vlastního barevného motivu k extrémně častým překreslením GUI a značné degradaci výkonu, předělal jsem binding stylování tak, aby využíval experimentální API[11] pro stylování založené na React Hooks[12]. Protože React Hooks v době implementace dané části nebyly v ReasonReact k dispozici, vytvořil jsem vlastní komponentu vyššího řádu[13] `UseHook`, která dané API zpřístupňuje.

Problém tím byl vyřešen až do té chvíle, kdy bylo nutné zanořit více vlastních motivů (např. jeden pro GUI knihovnu a jeden pro cílovou aplikaci). V tomto případě se jednalo o chybu přímo v knihovně Material-UI, kterou se mi podařilo vyřešit a opravit[14] s pomocí jejích vývojářů.

5.11 Signalizační server

Signalizační server je stejně jako knihovna pro koncový uzel implementovaná v jazyce Reason. Díky použití stejného jazyka je s ní možné sdílet některé moduly, např. modul na reprezentaci zpráv a identifikátorů uzlů.

Server komunikuje s koncovými uzly pomocí protokolu WebSocket. Implementací WebSocket serveru pro Reason/OCaml není mnoho a při bližším prozkoumání je možné zjistit, že jejich kvalita není ideální. Vyzkoušeny byly dvě: *reason-websocket* a *ocaml-websocket*.

Knihovna *reason-websocket*[15] je oproti konkurenci mnohem menší a pro základní „Hello world“ aplikaci vyžaduje poloviční množství paměti RAM. Nabízí nicméně pouze velice jednoduché rozhraní, neumožňuje uchovávat a ručně ukončovat příchozí spojení a pro TLS používá blokující variantu OpenSSL, což vede k těžko laditelným pádům. Zejména kvůli poslednímu zmíněnému problému jsem použití této knihovny zavrhl.

[7] Stránky bindingu ReasonReact: https://reasonml.github.io/reason-react
[8] Stránky knihovny Reason: https://reactjs.org
[9] Stránky knihovny Material-UI: https://material-ui.com
[10] Stránky bindingu bs-material-ui: https://github.com/jsiebern/bs-material-ui
[11] Experimentální Hook API v dokumentaci Material-UI: https://material-ui.com/css-in-js/basics/#hook-api
[12] Hook API v dokumentaci Reactu: https://reactjs.org/docs/hooks-intro.html
[13] Higher-Order Components v dokumentaci Reactu: https://reactjs.org/docs/higher-order-components.html
[14] Nahlášený bug #15186 v knihovně Material-UI: https://github.com/mui-org/material-ui/issues/15186
[15] Stránka knihovny *reason-websocket*: https://github.com/jaredly/reason-websocket

Nakonec jsem použil knihovnu *ocaml-websocket*[16], jejíž značnou část bylo nutné upravit, protože opět chyběla podpora ukončování spojení a na mnoha místech chybělo ošetření výjimek, které původně vedly k pádu programu.

Samotná implementace chování serveru a reakcí na zprávy již byla relativně přímočará a bezproblémová.

5.12 Nasazení signalizačního serveru

Signalizační server vyžaduje pro sestavení balíčkovací systém OCamlu OPAM 2, který se postará o nainstalování potřebných vývojových závislostí.

Co se týče požadavků pro spuštění serveru, vzhledem k tomu, že sestavený nativní binární soubor nevyžaduje žádné speciální běhové prostředí, stačí mu v cílovém systému zajistit pouze dvě používané knihovny:

- GNU MP (libgmp) a
- OpenSSL (libssl).

Součástí projektu signalizačního serveru jsou předpisy pro vytvoření dvou obrazů pro Docker – jeden obraz s prostředím pro sestavení binárního souboru a druhý obraz s minimální instalací Ubuntu a závislostmi nachystaný pro spouštění serveru.

[16] Stránka knihovny *ocaml-websocket*: https://github.com/vbmithr/ocaml-websocket

Kapitola 6

Ukázka použití rámce: Editor stromových stuktur

Součástí této práce je grafický editor stromových struktur TreeBurst, který demonstruje netriviální použití rámce a zároveň slouží jako určitá forma akceptačního testování navržených rozhraní.

Editorem stromových struktur je v kontextu této práce míněn *editor myšlenkových map*, který umožňuje uživatelům vytvářet vlastní stromy poznámek s libovolným počtem úrovní zanoření.

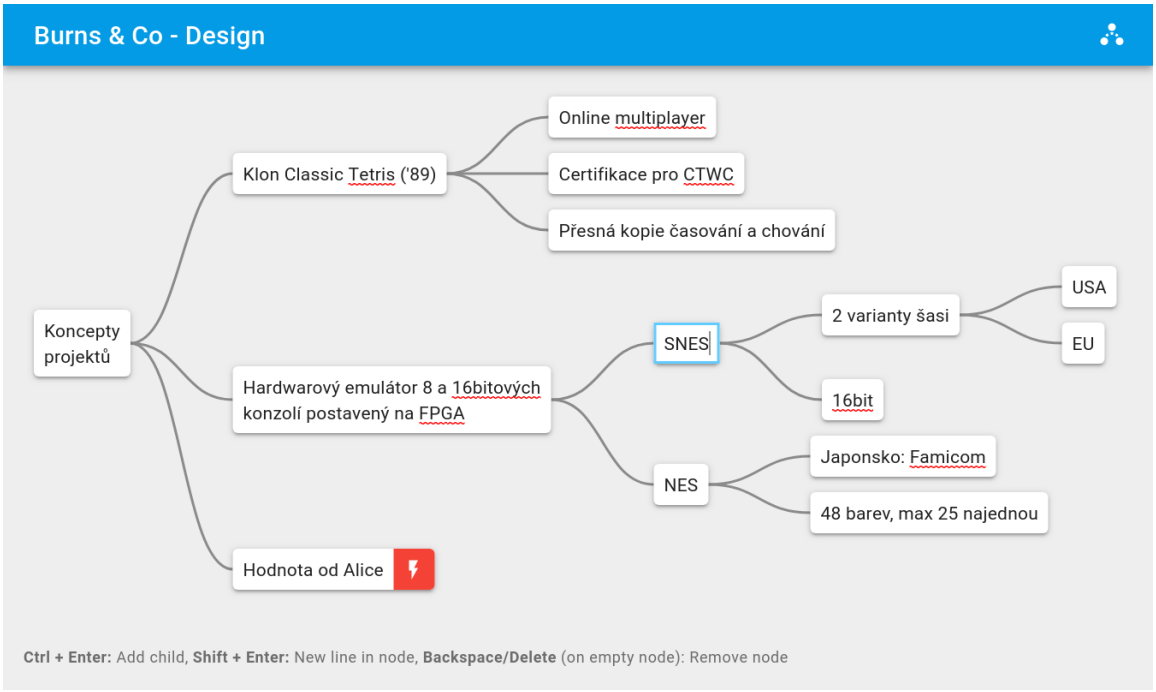
Mezi existující aplikace s podobným konceptem patří FreeMind, XMind, WiseMapping, Lucidchart a další. Webové varianty těchto aplikací na rozdíl od naší implementace vyžadují po celou dobu běhu připojení k internetu a uchovávají uživatelská data na centrálním serveru. Desktopové aplikace potom sice fungují offline, ale zpravidla nepodporují spolupráci více uživatelů v reálném čase a vyžadují ruční manipulaci se soubory.

Naše implementace těží z funkcionality poskytované vytvořeným P2P rámcem: aplikace je po prvním otevření uložena do vyrovnávací paměti, takže je možné ji následně spouštět i bez připojení k internetu. Uživatel může vytvářet více stromů poznámek a každý strom může sdílet s jinou skupinou protějšků. Po nastavení oprávnění se všechny změny na pozadí automaticky propagují mezi členy skupiny, volitelně je možné nechat replikovat i samotný seznam členů. Všechny myšlenkové mapy a data uživatele jsou uloženy lokálně, při replikaci jsou přenášené přímo k protějškům nejkratší nalezenou cestou zabezpečeným kanálem. Aplikace umožňuje efektivně řešit konflikty, které mohou nastat při nezávislých úpravách odpojených protějšků: ty, u kterých to dává smysl, jsou vyřešeny automaticky, konflikty vyžadující zásah uživatele je pak možné vyřešit v uživatelském rozhraní myšlenkové mapy.

Ukázková aplikace vypouští některé méně důležité funkce, které nemají vliv na předvádění funkčnosti rámce. Příkladem mohou být například vícenásobné výběry uzlů nebo export dat. Vytvoření kompletního editoru připraveného pro produkční použití by bylo mimo rozsah této práce.

6.1 Návrh

Aplikace používá podobnou strukturu jako samotná knihovna prvků uživatelského rozhraní. Životní cyklus aplikace je opět realizován s využitím vlastní knihovny *bs-black-tea* (vlastní implementace The Elm Architecture, viz sekce 5.6). Moduly aplikace lze rozdělit do tří celků.



Obrázek 6.1: Editor s otevřenou myšlenkovou mapou.

Prvním je **hlavní model**, který zastřešuje globální stav aplikace a zejména stav používaných knihoven (*pocket-mesh-peer* a *pocket-mesh-peer-material-ui*). Kromě toho zajišťuje také předávání zpráv těmto knihovnám a registraci jejich subscriptions.

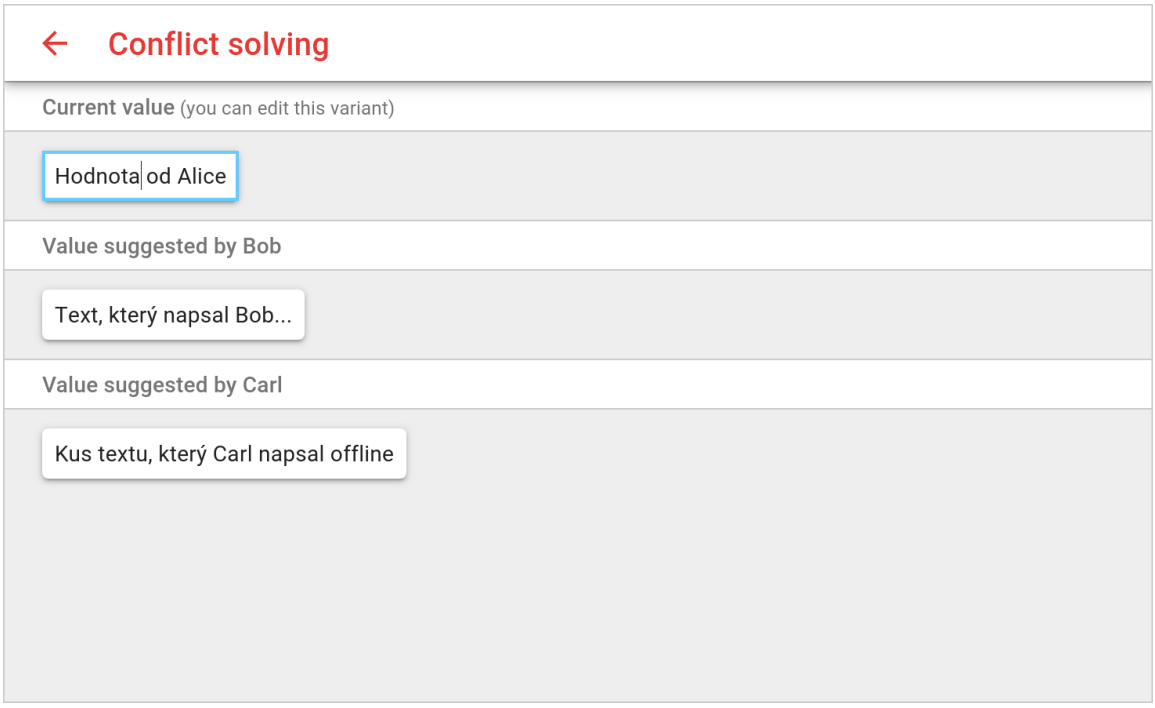
Druhou důležitou složkou je **práce s obsahem skupiny**. Knihovna *pocket-mesh-peer* zpřístupňuje rozhraní `Crdt` pro práci s replikovaným datovým typem reprezentujícím data skupiny. Typ popisuje obecný JSON strom bez konkrétního schématu a je plně v režii koncové aplikace, jakým způsobem data ve stromu zorganizuje a jak bude udržovat jejich tvar.

Protože bezpečná práce s obecnou replikovanou JSON strukturou bývá objemnější na počet řádků kódu (viz sekce 5.2), je v ukázkové aplikaci soustředěna do zvláštního modulu `Content`. V tomto modulu je implementováno všechno ověřování a samotné aplikaci jsou zpřístupněna již zjednodušená rozhraní s doménově-specifickými statickými typy.

Třetí složkou jsou **komponenty GUI**. Aplikace definuje dvě vlastní obrazovky: hlavní obrazovku editoru a obrazovku pro řešení konfliktů.

Obrazovka editoru (obrázek 6.1) v hlavičce vlevo zobrazuje název aktuálně otevřené skupiny a vpravo indikátor stavu P2P knihovny (viz sekce 4.10.1), který slouží jako vstupní bod k obrazovkám poskytnutým rámcem, tj. ke správě skupin, přátel, apod. (viz sekce 4.10) Zbytek okna vyplňuje prostor pro zobrazení a úpravy stromu. Přidávání a mazání uzlů je realizované pomocí klávesových zkratek zobrazených u spodního okraje obrazovky. Pokud v textu uzlu dojde ke konfliktu, zobrazí se u něj tlačítko s červeným bleskem, po jehož stisknutí je zobrazena obrazovka řešení konfliktu.

Obrazovka řešení konfliktu (obrázek 6.2) zobrazuje pod sebou všechny souběžně zapsané hodnoty uzlu. První hodnotu je možné editovat a ručně do ní začlenit data z ostatních kandidátů. Výsledek se zapíše do distribuovaného úložiště ihned po kliknutí na šipku „Zpět“.



Obrázek 6.2: Obrazovka pro řešení konfliktu.

6.2 Implementace

Implementace TreeBurst maximálně těží z funkcionality implementované ve vytvořeném P2P rámci. V následujících podsekcích jsou rozebrané vybrané implementační detaily, které se zpravidla týkají již cílové domény editoru stromových struktur.

6.2.1 Prvky GUI

Co se týče implementovaných komponent pro grafické uživatelské rozhraní, za vypíchnutí stojí `ContentEditable` a `TreeView`.

`ContentEditable` je komponenta, která zpřístupňuje stejnojmenný atribut HTML sloužící k vytvoření editovatelného bloku z libovolného HTML prvku. Oproti klasickému prvku `input` má tu výhodu, že se jeho velikost může automaticky přizpůsobovat podle obsahu. V aplikaci je využívána pro editaci textu v uzlech stromu. Počátečním impulsem bylo použít některou z existujících hotových implementací[1]. Bohužel se ukázalo, že všechny trpěly špatným výkonem způsobeným zbytečným překreslování GUI nebo jinou chybou. Nakonec jsem tedy vytvořil vlastní implementaci, která problémy s výkonem vyřešila.

Vykreslování stromu je zaobalené v komponentě `TreeView`. Aby bylo možné plynule animovat přechody mezi pozicemi uzlů a v budoucnu případně definovat vlastní fyzikální chování (pružnost hran apod.), používá komponenta vlastní algoritmus pro rozvržení. Z pohledu DOMu jsou prvky uzlů v jedné úrovni a s absolutním pozicováním. Důležité je, že k nastavení souřadnic jsou používány CSS transformace (místo atributů `top` a `left`), díky čemuž je možné pohyb uzlů akcelerovat na grafické kartě. Křivky s hranami uzlů jsou vykreslované v jednom SVG obrázku, který je umístěn na pozadí uzlů.

[1] Vyzkoušené knihovny zahrnují *react-sane-contenteditable*, *react-simple-contenteditable* a *react-contenteditable*.

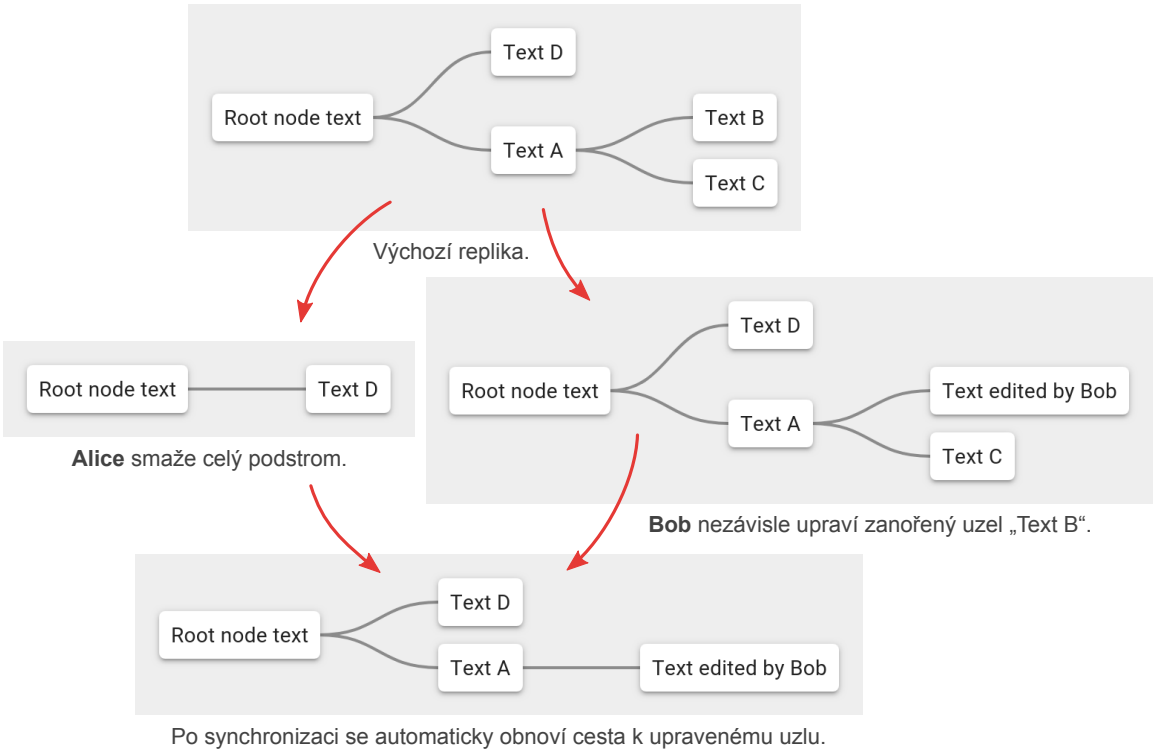
6.2.2 Reprezentace myšlenkové mapy

Myšlenková mapa je reprezentována pomocí replikovaného JSON objektu, jehož ukázka je znázorněna ve výpisu 6.1.

```
{
  rootNodeId: "1f90ed8",
  nodes: {
    "1f90ed8": {
      id: "1f90ed8",
      text: "Root node text",
      children: [
        "f3eb99b",
        "eef7c99"
      ]
    },
    "f3eb99b": {
      id: "f3eb99b",
      parentId: "1f90ed8",
      text: "Level 1 text AAA",
      children: [
        "99a4c77",
        "ddc7d4e"
      ]
    },
    "eef7c99": {
      id: "eef7c99",
      parentId: "1f90ed8",
      text: "Level 1 text BBB"
    },
    "99a4c77": {
      id: "99a4c77",
      parentId: "f3eb99b",
      text: "Level 2 text CCC"
    },
    "ddc7d4e": {
      id: "ddc7d4e",
      parentId: "f3eb99b",
      text: "Level 2 text DDD",
      deleted: true
    }
  }
}
```

Výpis 6.1: Reprezentace myšlenkové mapy v replikovaném JSON objektu (UUID uzlů jsou zkráceny).

Uzly stromu jsou zploštěné v jedné úrovni (bez rekurze), aby bylo možné je snadno vyhledávat podle ID. Struktura obsahuje některé redundantní informace (zopakované ID uvnitř objektu uzlu, odkaz na rodičovský uzel), které zrychlují navigaci ve stromu. Mazání uzlů je kvůli snadné obnově při konfliktech řešeno návěstím `deleted`.



Obrázek 6.3: Automatické aplikačně-specifické řešení konfliktu.

6.2.3 Řešení konfliktů ve stromu

V editoru myšlenkových map se naskytla příležitost vyzkoušet začlenění jak ručního (uživatelského), tak i automatického aplikačně-specifického řešení konfliktů.

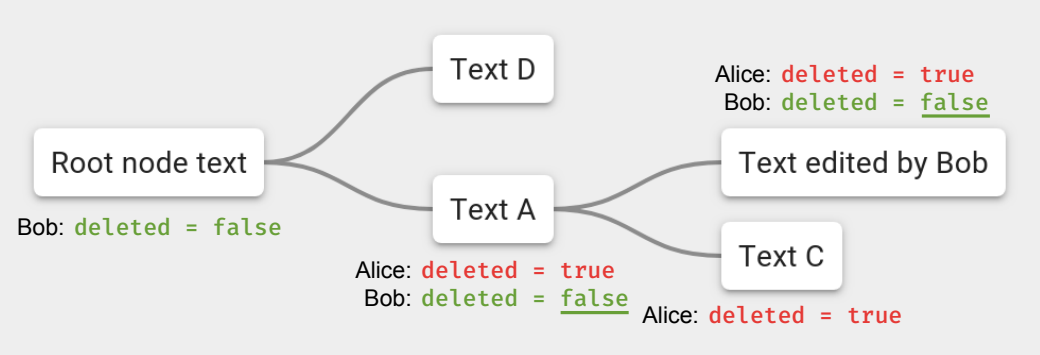
Pokud více uživatelů nezávisle na sobě změní *text stejného uzlu*, je třeba konflikt vyřešit ručně pomocí obrazovky řešení konfliktu (viz sekce 6.1). Z pohledu vývojáře aplikace je detekce konfliktu triviální – stačí při získávání textu uzlu použít funkci `Map.getC("text", node)`[2], která vrací návrhy hodnot od všech zúčastněných replik.

Specifickým druhem konfliktu pro editor stromových struktur je situace, kdy jeden uživatel upraví text uzlu, zatímco druhý uživatel nezávisle smaže podstrom, ve kterém daný uzel leží (viz obr. 6.3). Existuje více přístupů, jakým způsobem by se konflikt dal řešit, v aplikaci jsem implementoval chování, při kterém se obnoví všichni smazaní předkové vedoucí k upravenému uzlu. Ostatní smazané uzly (např. sourozenci) zůstanou smazané. Důležité je, že upravený text není ztracen.

Co se týče realizace, protože vestavěná sémantika řešení konfliktů v CRDT knihovně je pevně daná (viz sekce 3.6), pro podobné vybočující situace je třeba v JSON objektu doplnit dodatečné pomocné proměnné.

- Prvním krokem je změnit způsob mazání uzlů. Zatímco při běžném smazání už není možné uzel obnovit, při použití zvláštního návěstí `deleted` je obnova otázkou přepsání hodnoty na `false`.
- Druhým krokem je zajistit, aby se hodnota `deleted = false` preventivně zapsala do všech přímých předchůdců při každé úpravě textu. Díky tomu, jakmile se paralelně

[2]Vysvětlení přístupu ke konfliktním hodnotám replikované struktury – viz sekce 5.2.1.



Obrázek 6.4: Realizace aplikačně-specifického řešení konfliktu z obrázku 6.3: u smazaných přímých předchůdců vznikne konflikt v poli `deleted`, GUI při vykreslování vezme v úvahu podtržené hodnoty.

smaže nadřazený podstrom, vzniknou konflikty v polích `deleted` u všech přímých předchůdců uzlu (viz obr. 6.4). GUI potom zajistí, že jakmile se někde objeví konfliktní návrhy s `deleted = true` i `false`, dá přednost `deleted = false` a uzel zobrazí, aniž by výsledek zapisovalo[3].

6.2.4 Zpřístupnění aplikace pro práci offline

Konfigurace Service Workers díky svým širokým možnostem může pro nastavení jednoduchých případů použití vyžadovat značné množství kódu[4].

Pro začlenění Service Workers aplikace proto využívá rozšíření *Workbox*[5] pro Webpack, které dokáže při každém sestavení automaticky vygenerovat konfiguraci se seznamem offline souborů a zajistit správnou invalidaci předchozích verzí v cache. Z aplikace potom stačí pouze jedním příkazem zaregistrovat vygenerovaný soubor `service-worker.js`.

Service Worker v naší aplikaci používá strategii *cache-first*, kdy při opakovaných návštěvách stránky se vždy načte verze z lokální cache. Pokud je na serveru k dispozici nová verze aplikace, stáhne se na pozadí a nachystá se pro spuštění při příštím otevření stránky. Výhodou uvedeného přístupu je, že se uživateli aplikace načítá rychle i s pomalým (např. mobilním) připojením, nevýhodou potom je fakt, že nová verze aplikace se uživateli zobrazí vždy o jedno obnovení stránky později.

6.3 Sestavení a nasazení aplikace

Sestavení aplikace je realizované pomocí nástroje Webpack, který umožňuje spojit všechny soubory JavaScriptu do jednoho (kvůli snížení počtu HTTP GET požadavků), díky použití ES6 jakožto kompilačního cíle umožňuje též agresivní eliminaci nevyužitého kódu (tree-shaking a dead code elimination). Při produkčním sestavení je kód dále minifikován, tj. identifikátory jsou přejmenovány na nesrozumitelné, ale zato krátké alternativy, jsou odstraněny zbytečné bílé znaky a vybrané konstrukce jsou nahrazeny svými kompaktnějšími

[3] Technika je blíže popsaná v sekci 4.8.2.

[4] https://developers.google.com/web/fundamentals/primers/service-workers/

[5] https://developers.google.com/web/tools/workbox

obdobami. Výsledkem je radikální zmenšení velikosti výsledného balíku JavaScriptu, rychlejší parsování při načítání stránky a rychlejší běh kódu.

Výstupem celého procesu je běžná statická webová stránka v HTML a doprovodné soubory JavaScriptu. Soubory mohou být uloženy na libovolném webového hostingu s podporou HTTPS. HTTPS je nutné kvůli použití WebCrypto, WebRTC a Service Workers, jak bylo zmíněno v kapitole 2.

Kapitola 7

Testování, publikování

Funkčnost dílčích knihoven a ukázkové aplikace byla průběžně ověřována v prohlížeči Chromium[1] v linuxové distribuci Manjaro s využitím *pomocných testovacích aplikací* (viz soubory `README.md` jednotlivých podprojektů).

Nejdůležitější složkou testování rámce bylo vytvoření ukázkové aplikace (editoru stromových struktur), která pomohla ověřit funkčnost rámce, a úplnost a použitelnost jeho veřejného rozhraní.

Po dokončení implementace byla ukázková aplikace úspěšně otestována v prohlížečích

- Firefox 66.0.3 v systému Manjaro Linux 18.0.4,
- Chromium 74.0.3729.108 v systému Manjaro Linux 18.0.4,
- Google Chrome 74.0.3729.131 v systému Windows 10,
- Google Chrome 73.0.3683.90 v systému Android 7.1.2.

Aplikace nefungovala v prohlížeči Microsoft Edge 42.17134.1.0 ve Windows 10 kvůli chybějící podpoře object spread operátoru z ECMAScript 2018. S přihlédnutím k plánovanému nahrazení jádra EdgeHTML jádrem z projektu Chromium [9] a nízkému zastoupení prohlížeče mezi uživateli[2] tento problém nebyl dále řešen.

Funkčnost spolupráce v reálném čase byla ověřena v prostředí lokální sítě v pěti lidech a v prostředí internetu ve třech lidech.

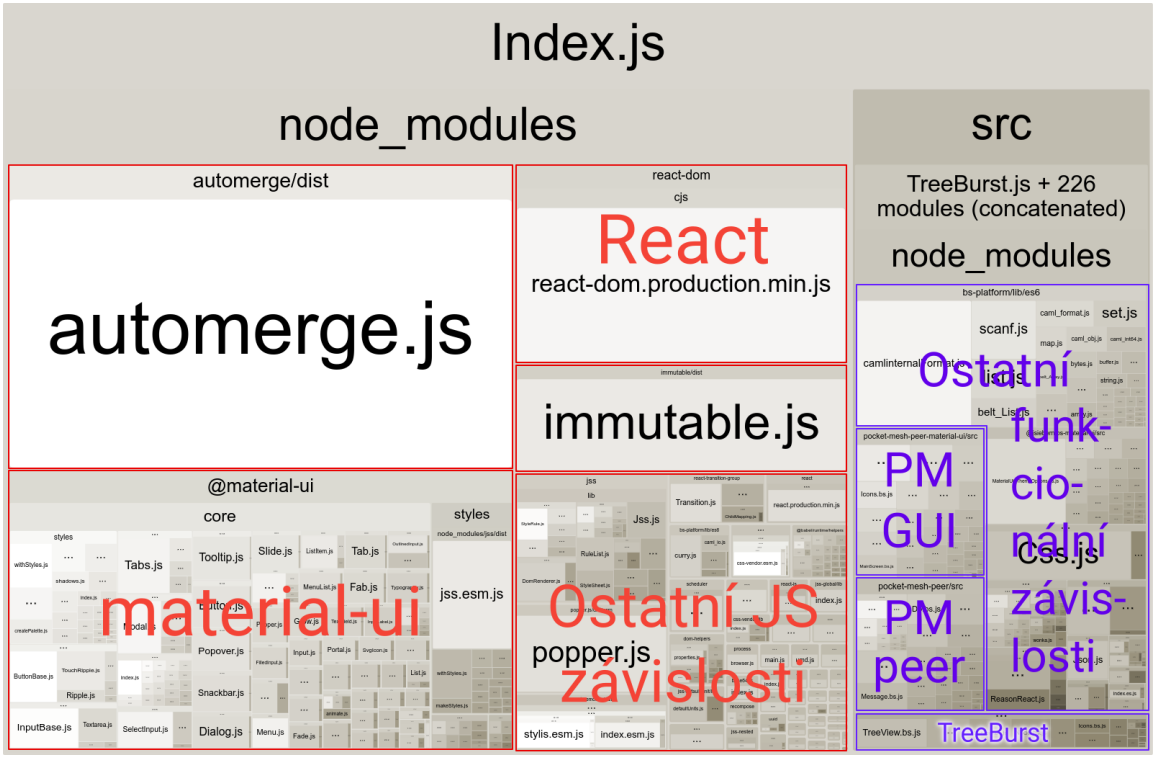
Výsledná produkční verze zkompilovaného JavaScriptu má 1,1 MB (280 KB gzipped). Vizualizaci jeho složení vytvořenou pomocí nástroje *webpack-bundle-analyzer*[3] je možné najít na obrázku 7.1. Z vizualizace je patrné, že značnou část zabírá knihovna Automerge a její závislost Immutable.js. Jejich podíl na velikosti by pravděpodobně mohl být radikálně snížen přepsáním CRDT knihovny do jazyka Reason, jak je naznačeno v sekcích 5.1 a 8.1.1.

Pro ilustrační účely byla změřena také rychlost načítání stránky s aplikací v různých situacích (viz tabulka 7.2) a několik metrik pro vytvořený signalizační server.

[1] open-source projekt, ze kterého vychází Google Chrome

[2] 4, 35% podle serveru StatCounter (`http://gs.statcounter.com`), 5, 53% podle NetMarketShare (`https://netmarketshare.com`). Údaje pro duben 2019.

[3] Stránky nástroje *webpack-bundle-analyzer*: `https://github.com/webpack-contrib/webpack-bundle-analyzer`



Obrázek 7.1: Vizualizace složení výsledného produkčního JavaScriptu. Plocha reprezentuje velikost. Červená – závislosti třetích stran napsané v JavaScriptu, fialová – funkcionální kód.

Popis	**Čas [ms]**	**Relativní odchylka**
První načtení stránky se symetrickým připojením 200 Mbit/s	493	4%
První načtení stránky se simulovaným připojením „3G Fast“ (download 1,4 Mbit/s, upload 675 kbit/s, latence 563 ms)	3 015	1%
Načtení uložené stránky s výchozími daty – jednou skupinou a uzlem s textem „Hello CRDT World“)	668	25%
Načtení uložené stránky s 3,8 MB databází (řádově stovky uzlů)	2 748	11%

Tabulka 7.1: Naměřené rychlosti načítání webové aplikace. Každá hodnota vychází z pěti vstupních vzorků. Měření probíhalo v prohlížeči Chromium 74 na PC s CPU AMD Athlon II X2 260 a 12 GB RAM. Časový údaj zachycuje interval od potvrzení adresy stránky po zobrazení uzlů.

Popis	**Hodnota**
Velikost binárního souboru	7,8 MB
Využitá paměť po spuštění serveru	12 844 KB
Využitá paměť po týdnu běhu a 426 postupně navázanými spojeními (maximálně 8 zároveň)	17 508 KB

Tabulka 7.2: Ilustrační metriky signalizačního serveru. Měřeno na 64bitové verzi Ubuntu Server 18.04.

7.1 Publikování knihoven

Zdrojové kódy vytvořeného rámce nazvaného *PocketMesh* byly publikovány na `github.com/jhrdina/pocket-mesh`. Jak bylo naznačeno na začátku kapitoly 4, projekt se skládá z několika balíčků:

- *pocket-mesh-peer* – funkcionalita koncového uzlu,
- *pocket-mesh-peer-material-ui* – prvky GUI pro koncový uzel,
- *pocket-mesh-signal-server* – signalizační server.

Ukázkový projekt editoru myšlenkových map se nazývá *TreeBurst*, jeho zdrojové kódy byly publikovány na `github.com/jhrdina/tree-burst` a jeho funkční verze je k dispozici na `tree-burst.hrdinajan.cz`.

Kromě výše uvedených jsem pro potřeby projektu vytvořil a publikoval také následující:

- *ocaml-diff* (`github.com/jhrdina/ocaml-diff`) – knihovnu implementující symetrický diff pro `Map` a `Set` ze standardní knihovny OCamlu (viz sekce 5.4),
- *bs-automerge* (`github.com/jhrdina/bs-automerge`) – binding pro knihovnu Automerge a vlastní čistě funkcionální implementaci článku [27] (viz sekce 5.1 a 5.2),
- *bs-black-tea* (`github.com/jhrdina/bs-black-tea`) – odlehčenou implementaci The Elm Architecture (viz sekce 5.6),
- *bs-webapi-extra* (`github.com/jhrdina/bs-webapi-extra`) – binding pro Reason pro použití webových API jako WebSockets, IndexedDB, WebCrypto a další (viz sekce 5.3).

Kapitola 8

Závěr

V rámci diplomové práce jsem navrhl, implementoval a odzkoušel aplikační rámec pro tvorbu kolaborativních webových editorů, které umožňují peer-to-peer spolupráci v reálném čase. Oproti existujícím řešením běžným ve webovém prostředí (např. Google Docs) vytvářená *data zůstávají uložena pouze v koncových uzlech* a pro spolupráci s ostatními není nutné je svěřovat centrálnímu úložišti nebo jinému prostředníkovi.

Vytvářená data mohou být sdílená uvnitř skupin protějšků, kde každému členovi je možné nastavit jiná oprávnění.

Vytvořený rámec je plně uzpůsoben pro použití Service Workers – technologie, která při první návštěvě stránku automaticky stáhne do úložiště prohlížeče a následně ji umožní otevírat i offline. Změny v datech provedené bez připojení k internetu jsou automaticky synchronizované s ostatními uzly, jakmile se připojení obnoví.

Protože v podobných případech může dojít ke konfliktům, rámec zpřístupňuje rozhraní umožňující je detekovat a implementovat jejich ruční (uživatelské) nebo automatické aplikačně-specifické řešení.

Pro zajištění spolehlivého slučování změn v datech jsem nastudoval oblast algoritmů používaných pro replikaci dat v distribuovaných systémech a v aplikacích pro online spolupráci. Jako nejvhodnější kandidát se ukázal bezkonfliktní replikovaný datový typ (CRDT[1]) popsaný v článku [27] od M. Kleppmanna, který nevyžaduje žádný centrální koordinační prvek a umožňuje replikovat objekty JSONu s libovolným zanořením. Jeho implementace v podobě knihovny *Automerge* nabídnula ze všech kandidátů největší množství funkcí a komfort použití.

Pro zajištění komunikačních kanálů jsem nastudoval a shrnul existující technologie pro komunikaci ve webovém prostředí. Z dostupných protokolů jako nejlepší kandidát vyšlo WebRTC, které umožňuje navázat peer-to-peer spojení mezi dvěma instancemi webového prohlížeče. V projektu jsem implementoval taktéž nativní signalizační server potřebný ve fázi ustavení spojení WebRTC.

Pro komunikaci se signalizačním serverem a protějšky jsem navrhl vlastní textové protokoly, které jsou dimenzované tak, aby nebylo nutné signalizačnímu serveru důvěřovat a aby bylo vždy možné u zpráv ověřit autenticitu odesílatele.

Celý projekt byl implementován v mladém staticky typovaném funkcionálním jazyce ReasonML, který se v případě koncových uzlů kompiluje do JavaScriptu a v případě signalizačního serveru do nativního binárního souboru. Práce mimo jiné vytvořila cenný prostor pro nastudování a praktické vyzkoušení mnoha funkcionálních návrhových vzorů.

[1] Conflict-free Replicated Data-Type

Funkčnost vytvořeného rámce byla úspěšně ověřena v implementaci kolaborativního editoru stromových struktur inspirovaného programy na tvorbu myšlenkových map. Vytvořená aplikace demonstruje všechny výše uvedené vlastnosti.

Zdrojové kódy vytvořeného rámce a dílčích knihoven byly publikovány na GitHub.com pod svobodnými licencemi. Webová prezentace rámce vč. dokumentace a funkční ukázková aplikace byly veřejně zpřístupněny na oddělených webových stránkách[2].

8.1 Další vývoj

Tato sekce krátce popisuje možné další směry vývoje.

8.1.1 Čistě funkcionální implementace CRDT knihovny

Jak bylo naznačeno v sekci 5.1, projekt aktuálně používá pro CRDT existující knihovnu Automerge napsanou v JavaScriptu s využitím Immutable.js, protože reimplementace všech jejích funkcí by byla značně časově náročná a celkově mimo rozsah této práce.

Přepis by přesto měl minimálně dva zajímavé přínosy: zatímco aktuálně je kvůli JavaScriptu použití knihovny omezené pouze na prostředí prohlížeče a Node.js, při přepisu do Reasonu se zpřístupňuje možnost *kompilování do nativního kódu*. Druhým přínosem by bylo předpokládané znatelné *zmenšení výsledného balíku JavaScriptu*. Jak ukázala analýza v kapitole 7, Automerge v něm zabírá nezanedbatelnou část. Nepřispívá k tomu ani Immutable.js, které zde tvoří zbytečnou duplicitní implementaci neměnných datových typů, které jsou již distribuované s kódem Reasonu. Rychlost načítání stránky s ukázkovou aplikací tím sice není výrazně ovlivněna, dá se nicméně předpokládat, že v případě větších koncových aplikací bude každý uvolněný kilobajt vítán.

8.1.2 Převod do nativní verze

Cílem této práce je vytvořit rámec a sadu návrhových vzorů, která umožní snadno vyvíjet rychlé a robustní webové editory obsahu s podporou P2P spolupráce. Rámec je implementován pro webovou platformu, protože ta tvoří nejsnadnější cestu pro obsloužení velkého množství zařízení a operačních systémů bez zbytečné vstupní bariéry ve formě nutné ruční instalace koncové aplikace.

I když dnes webové prohlížeče umí víc, než kdy dřív, a ve stále narůstajícím množství případů dokáží nahradit nativní instalované aplikace, stále existují oblasti vývoje, kdy je vhodnější nebo přímo nezbytné sáhnout místo webového JavaScriptu po nějakém kompilovaném nativním jazyce.

Mezi obecné výhody nativních aplikací patří například:

- menší paměťová náročnost,
- širší možnosti běhu na pozadí,
- potenciálně lepší začlenění do prostředí cílové platformy (při využití vhodného GUI toolkitu),
- přístup k nízkoúrovňovým knihovnám a systémovým funkcím,

[2]Dokumentace: https://pocket-mesh.hrdinajan.cz
Ukázková aplikace: https://tree-burst.hrdinajan.cz

- lepší možnosti využití výkonu hardware.

Portování rámce do nativního prostředí by mohlo umožnit vytvářet P2P editory využívající těchto výhod. Díky použitému jazyku Reason by mělo být možné sdílet s webovou implementací většinu kódu.

Podpora tvorby headless koncových uzlů

Odstranění závislosti na webovém prohlížeči a snížení paměťové náročnosti aplikace by mohlo vytvořit prostor pro tvorbu P2P nástrojů bez GUI vhodných například k běhu ve vestavěných zařízeních, na domácím serveru nebo prostě na pozadí stolního počítače.

Headless verze oblíbeného textového editoru spuštěná na levném minipočítači à la Raspberry Pi by například mohla zajišťovat zálohování osobních dokumentů. Trvale běžící zařízení se stabilním připojením k internetu by pro soukromou síť mohlo být dobrým záchytným bodem, který by pomohl překlenout chvíle, kdy se dvě mobilní zařízení prostě nepotkají online.

Na rozdíl od provozování vlastního serveru v klasické klient-server architektuře zde není nutné mít veřejnou IP adresu nebo VPN a konfigurace by nemusela být o nic moc složitější, než přidání běžného koncového uzlu ve webovém prohlížeči.

Odstranění potřeby signalizačního serveru

Protože nativní prostředí není omezeno protokoly dostupnými v poměrně uzavřené a relativně pomalu se vyvíjející webové platformě, je možné hledat lepší alternativy pro některé použité technologie. Například WebRTC zajišťující P2P spojení mezi webovými prohlížeči (viz sekce 2.3) vyžaduje pro zkontaktování jiného uzlu signalizační kanál třeba ve formě sig. serveru. I když se tento centrální prvek používá pouze pro navázání spojení, jeho nedostupnost by znamenala pro síť značný problém.

Pro odstranění nutnosti signalizačního serveru by stál za prozkoumání decentralizovaný přístup k vyhledávání uzlů založený na distribuované hašovací tabulce (DHT) *Kademlia* [33]. Podobný přístup se již používá například v sítích Ethereum [1] nebo BitTorrent.

8.1.3 Použití pro rozsáhlejší produkční projekt

Přínosnou zpětnou vazbou pro další vývoj rámce by bylo vytvoření rozsáhlé produkční aplikace, která na něm staví. Dobrým kandidátem by mohl být například

- kompletní kolaborativní vektorový editor inspirovaný editorem Sketch[3],
- kolaborativní editor pro vizuální programování,
- distribuovaný správce fotek,
- interaktivní editor stromových poznámek s podporou více pohledů na jedna data, podporou rozšíření a interaktivních příloh (rozšíření mé bakalářské práce [23]).

[3]Stránka vektorového editoru Sketch: https://www.sketch.com

Literatura

[1] Kademlia Peer Selection. *Ethereum Wiki*, 2018-08-22 [cit. 2019-01-17], [online]. Dostupné z: https://github.com/ethereum/wiki/wiki/Kademlia-Peer-Selection

[2] Stack Overflow Developer Survey 2019. *Stack Overflow*, 2019 [cit. 2019-04-26], [online]. Dostupné z: https://insights.stackoverflow.com/survey/2019

[3] A History of OCaml. *OCaml.org*, [cit. 2019-04-13], [online]. Dostupné z: https://ocaml.org/learn/history.html

[4] Abramov, D.; aj.: Prior Art. *Redux*, 2018-12-09 [cit. 2019-04-25], [online]. Dostupné z: https://redux.js.org/introduction/prior-art

[5] Almeida, P. S.; Shoker, A.; Baquero, C.: Efficient state-based CRDTs by delta-mutation. In *International Conference on Networked Systems*, Springer, 2015, s. 62–76.

[6] Almeida, P. S.; Shoker, A.; Baquero, C.: Delta state replicated data types. *Journal of Parallel and Distributed Computing*, ročník 111, 2018: s. 162–173, ISSN 0743-7315, doi:10.1016/j.jpdc.2017.08.003. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0743731517302332

[7] Antoš, D.: Google Gears - online aplikace offline. *Lupa.cz*, 2007-05-31 [cit. 2019-01-10], [online]. Dostupné z: https://blog.lupa.cz/jilm/google-gears-online-aplikace-offline/

[8] AppJet, I.: Etherpad and EasySync Technical Manual. 2011-03-26 [cit. 2019-04-23], [online]. Dostupné z: https://github.com/ether/etherpad-lite/blob/e2ce9dc/doc/easysync/easysync-full-description.pdf

[9] Belfiore, J.: Microsoft Edge: Making the web better through more open source collaboration. *Windows Experience Blog*, 2018-12-06 [cit. 2019-05-05], [online]. Dostupné z: https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/

[10] Boodman, A.: Stopping the Gears. *Gears API Blog*, 2011-03-11 [cit. 2019-01-10], [online]. Dostupné z: http://gearsblog.blogspot.com/2011/03/stopping-gears.html

[11] Briot, L.; Urso, P.; Shapiro, M.: High Responsiveness for Group Editing CRDTs. In *Proceedings of the 19th International Conference on Supporting Group Work*, GROUP '16, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-4276-6, s. 51–60,

doi:10.1145/2957276.2957300. Dostupné z:
http://doi.acm.org/10.1145/2957276.2957300

[12] Camarillo, G.; Perreault, S.; Novo, O.: Traversal Using Relays around NAT (TURN) Extension for IPv6. RFC 6156, Duben 2011, doi:10.17487/RFC6156. Dostupné z: https://rfc-editor.org/rfc/rfc6156.txt

[13] Christiansen, K. R.; Kis, Z.: Web NFC API. Draft community group report, W3C, Prosinec 2018. Dostupné z: https://w3c.github.io/web-nfc/

[14] Davis, A. H.; Sun, C.; Lu, J.: Generalizing Operational Transformation to the Standard General Markup Language. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-560-2, s. 58–67, doi:10.1145/587078.587088. Dostupné z: http://doi.acm.org/10.1145/587078.587088

[15] Day-Richter, J.: What's different about the new Google Docs: Making collaboration fast. *Google Drive Blog*, 2010-09-23 [cit. 2019-04-23], [online]. Dostupné z: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

[16] Demers, A.; Greene, D.; Hauser, C.; aj.: Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, New York, NY, USA: ACM, 1987, ISBN 0-89791-239-X, s. 1–12, doi:10.1145/41840.41841. Dostupné z: http://doi.acm.org/10.1145/41840.41841

[17] Ellis, C. A.; Gibbs, S. J.: Concurrency Control in Groupware Systems. *SIGMOD Rec.*, ročník 18, č. 2, Červen 1989: s. 399–407, ISSN 0163-5808, doi:10.1145/66926.66963. Dostupné z: http://doi.acm.org/10.1145/66926.66963

[18] Feldman, R.: Making Impossible States Impossible. *Elm Conf 2016*, 2016-09-19 [cit. 2019-05-02], in YouTube [online]. Dostupné z: https://www.youtube.com/watch?v=IcgmSRJHu_8

[19] Feldman, R.: Make Data Structures. *Elm Europe Conference 2018*, 2018-07-18 [cit. 2019-05-02], in YouTube [online]. Dostupné z: https://www.youtube.com/watch?v=x1FU3e0sT1I

[20] Fox, A.; Brewer, E. A.: Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, March 1999, s. 174–178, doi:10.1109/HOTOS.1999.798396.

[21] Gaunt, M.: Service Workers: an Introduction. *Google Developers – Web Fundamentals*, 2019-05-01 [cit. 2019-05-07], [online].

[22] Gilbert, S.; Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, ročník 33, č. 2, Červen 2002: s. 51–59, ISSN 0163-5700, doi:10.1145/564585.564601. Dostupné z: http://doi.acm.org/10.1145/564585.564601

[23] Hrdina, J.: *Webová aplikace pro správu poznámek*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016, vedoucí práce Burget Radek.

[24] Ignat, C.-L.; Norrie, M. C.: Customizable Collaborative Editor Relying on treeOPT Algorithm. In *ECSCW 2003*, editace K. Kuutti; E. H. Karsten; G. Fitzpatrick; P. Dourish; K. Schmidt, Dordrecht: Springer Netherlands, 2003, ISBN 978-94-010-0068-0, s. 315–334.

[25] Jelasity, M.: Gossip. In *Self-organising Software: From Natural to Artificial Adaptation*, Natural Computing Series, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 9783642173479, s. 139–162.

[26] Jones, M.: JSON Web Key (JWK). RFC 7517, Květen 2015, doi:10.17487/RFC7517. Dostupné z: https://rfc-editor.org/rfc/rfc7517.txt

[27] Kleppmann, M.; Beresford, A. R.: A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, ročník 28, č. 10, Oct 2017: s. 2733–2746, ISSN 1045-9219, doi:10.1109/TPDS.2017.2697382. Dostupné z: https://ieeexplore.ieee.org/abstract/document/7909007

[28] Levent-Levi, T.: 10 Massive Applications Using WebRTC. *BlogGeek.me*, 2017-12-18 [cit. 2019-04-26], [online]. Dostupné z: https://bloggeek.me/massive-applications-using-webrtc/

[29] Loreto, S.; Romano, S. P.: *Real-Time Communication with WebRTC*. Sebastopol: O'Reilly Media, Incorporated, 2014, ISBN 9781449371876.

[30] Maschiach, L.-T.: Facebook Messenger RTC – The Challenges and Opportunities of Scale. *YouTube*, 2017-10-27 [cit. 2019-04-26], [online]. Dostupné z: https://www.youtube.com/watch?v=F7UWvflUZoc

[31] Matthews, P.; Rosenberg, J.; Mahy, R.: Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, Duben 2010, doi:10.17487/RFC5766. Dostupné z: https://rfc-editor.org/rfc/rfc5766.txt

[32] Matthews, P.; Rosenberg, J.; Wing, D.; aj.: Session Traversal Utilities for NAT (STUN). RFC 5389, Říjen 2008, doi:10.17487/RFC5389. Dostupné z: https://rfc-editor.org/rfc/rfc5389.txt

[33] Maymounkov, P.; Mazieres, D.: Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, Springer, 2002, s. 53–65.

[34] Ohlmeier, N.: Large Data Channel Messages. *Advancing WebRTC*, 2017-11-03 [cit. 2019-04-27], [online]. Dostupné z: https://blog.mozilla.org/webrtc/large-data-channel-messages

[35] Oster, G.; Molli, P.; Urso, P.; aj.: Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Nov 2006, s. 1–10, doi:10.1109/COLCOM.2006.361867.

[36] Parker, D. S.; Popek, G. J.; Rudisin, G.; aj.: Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, ročník SE-9, č. 3, May 1983: s. 240–247, ISSN 0098-5589, doi:10.1109/TSE.1983.236733.

[37] Ressel, M.; Nitsche-Ruhland, D.; Gunzenhäuser, R.: An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, New York, NY, USA: ACM, 1996, ISBN 0-89791-765-0, s. 288–297, doi:10.1145/240080.240305. Dostupné z: http://doi.acm.org/10.1145/240080.240305

[38] Rosenberg, J.; Huitema, C.; Mahy, R.; aj.: STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, Březen 2003, doi:10.17487/RFC3489. Dostupné z: https://rfc-editor.org/rfc/rfc3489.txt

[39] Saito, Y.; Shapiro, M.: Optimistic Replication. *ACM Comput. Surv.*, ročník 37, č. 1, Březen 2005: s. 42–81, ISSN 0360-0300, doi:10.1145/1057977.1057980. Dostupné z: http://doi.acm.org/10.1145/1057977.1057980

[40] Shapiro, M.; Preguiça, N.; Baquero, C.; aj.: Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, Springer, 2011, s. 386–400.

[41] Shapiro, M.; Preguiça, N. M.; Baquero, C.; aj.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research report, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[42] Spiewak, D.: Understanding and Applying Operational Transformation. *Code Commit*, 2010-05-17 [cit. 2019-04-23], [online]. Dostupné z: http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation

[43] Staltz, A.: Callbag: A standard for JS callbacks that enables lightweight observables and iterables. 2019-02-09 [cit. 2019-04-26], [online]. Dostupné z: https://github.com/callbag/callbag

[44] Sun, C.: OT FAQ: Operational Transformation Frequently Asked Questions and Answers. *Nanyang Technological University*, 2015 [cit. 2019-04-20], [online]. Dostupné z: http://www3.ntu.edu.sg/home/czsun/projects/otfaq/

[45] Vidot, N.; Cart, M.; Ferrié, J.; aj.: Copies Convergence in a Distributed Real-time Collaborative Environment. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, New York, NY, USA: ACM, 2000, ISBN 1-58113-222-0, s. 171–180, doi:10.1145/358916.358988. Dostupné z: http://doi.acm.org/10.1145/358916.358988

[46] Wang, D.; Mah, A.; Lassen, S.: Apache Wave (incubating) Protocol Documentation. *The Apache Wave Foundation*, 2015-08-22 [cit. 2019-04-23], [online]. Dostupné z: https://people.apache.org/~al/wave_docs/ApacheWaveProtocol-0.4.pdf

[47] Weilbach, C.; Kühne, K.; Bieniusa, A.: Decoupling conflicts for configurable resolution in an open replication system. *arXiv preprint arXiv:1508.05545*, 2015.

[48] Yasskin, J.; Beaufort, F.; Scheib, V.: Web Bluetooth. Draft community group report, W3C, Prosinec 2018. Dostupné z: https://webbluetoothcg.github.io/web-bluetooth/

Příloha A

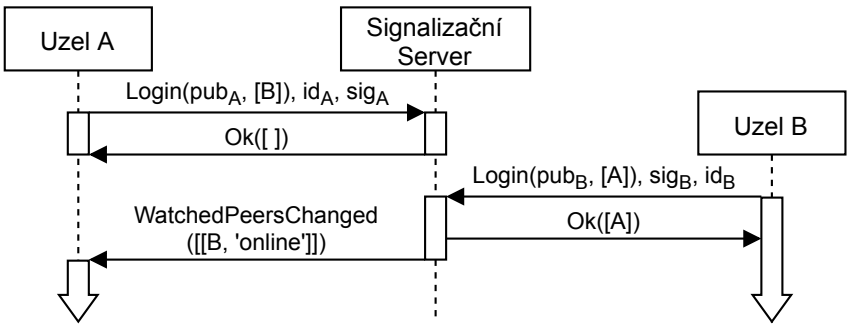
Replay útok při chybějícím ověřování

V sekci 4.6.1 byly popsány navržené mechanismy ochrany před replay útokem. Tato příloha ukazuje zranitelnost první verze protokolu, která tato opatření neimplementovala.

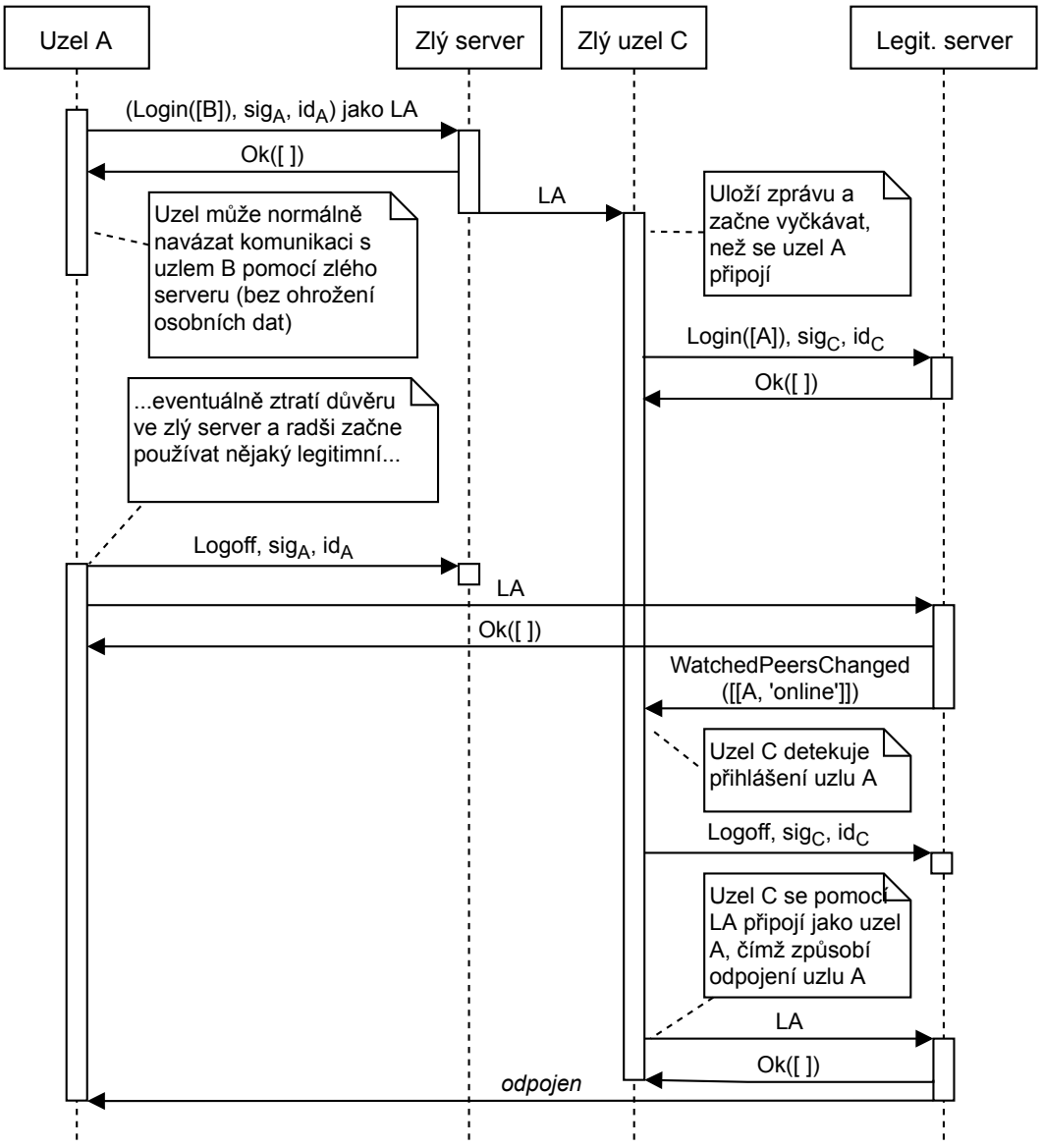
V první verzi protokolu začátek komunikace vypadal tak, jak je znázorněno na obrázku A.1: ve vyměňovaných zprávách oproti novému protokolu chyběl jeden roundtrip se zprávami *LoginReq* a *Challenge*, tzn. server nevytvářel náhodnou výzvu, kterou by klient musel před navázáním spojení podepsat a ověřit tak vlastnictví privátního klíče. Klientovi stačila k ověření pouze zpráva *Login*, která pro něj byla *pokaždé stejná*.

Útok, který v takovém případě hrozí, je znázorněn na obrázku A.2. Jakmile se útočník jednou dostane k *Login* zprávě uzlu, může se za něj vydávat při přihlašování k libovolnému sig. serveru a může tak způsobit *odpojení legitimního uzlu*.

Uvedený útok sice umožňuje útočníkovi C vydávat se za uzel A před serverem, ale neumožňuje útočníkovi vydávat se za uzel A před uzlem B a dostat se tak k soukromým datům, protože uzly mají předem navzájem vyměněná ID, pomocí kterých si mohou ověřit později získané veřejné klíče, pomocí kterých si mohou ověřit autenticitu všech přijatých zpráv. Nehrozí tak akceptování SDP zprávy pro navázání P2P spojení s útočníkem.



Obrázek A.1: Průběh přihlášení k serveru bez ověření živosti klienta u první verze protokolu



Obrázek A.2: Replay útok hrozící u první verze protokolu

Příloha B

Obsah přiloženého paměťového média

- `/`
  - `build` — Produkční sestavení výstupních aplikací vhodné pro okamžité spuštění.
    - `tree-burst` — Složka s produkčním sestavením ukázkového editoru myšlenkových map vhodným k nahrání na běžný webový server.
    - `signal-server` — Spustitelný binární soubor se signalizačním serverem zkompilovaný na 64bitové verzi Ubuntu 18.04.
    - `web` — Sestavená webová stránka s uživatelskou dokumentací a referenční příručkou.
  - `report`
    - `sources` — Složka se zdrojovými kódy technické zprávy pro LaTeX.
    - `report.pdf` — Finální technická zpráva v PDF.
  - `sources`
    - `published` — Složka s čistými repozitáři vytvořených knihoven, signalizačního serveru a ukázkové aplikace tak, jak byly publikované na serveru GitHub.
    - `patched-for-local-build` — Složka s projekty vlastních knihoven, signalizačního serveru a ukázkové aplikace upravené pro možnost lokálního sestavení (tj. bez stahování závislostí z internetu). Projekty obsahují všechny potřebné závislosti třetích stran ve složkách `node_modules`.

Příloha C

Instalační manuál

Tato příloha popisuje postup specifický pro kompilování a instalaci z přiloženého CD. Instalace z internetu určená pro veřejnost je jednodušší a je popsaná v elektronické uživatelské příručce (viz příloha D).

Všechny postupy jsou přizpůsobené pro linuxovou distribuci Ubuntu 18.04.

C.1 Spuštění předkompilované verze

Ve složce `build` na DVD jsou připravená produkční sestavení ukázkové aplikace a signalizačního serveru nachystané na okamžité spuštění.

Binární soubor **signalizačního serveru** je možné spustit prostým

```
cd build
./signal-server
```

Server spuštěný bez parametrů by měl při úspěšném startu vypsat

```
TCP mode started
[SERV] Listening on port 7777
```

Ukázková aplikace je ve složce `build/tree-burst`. Protože se jedná o statickou webovou stránku, je možné ji otestovat pomocí jednoduchého lokálního webového serveru spuštěného v dané složce:

```
cd build/tree-burst
python3 -m http.server 8004
```

Po otevření stránky `http://localhost:8004` by se měla zobrazit ukázková aplikace s modrým panelem nástrojů a jedním uzlem stromu „Hello CRDT World“. Aplikace by se měla automaticky připojit k signalizačnímu serveru běžícímu na portu 7777 (viz výše). Jeho adresa je v daném sestavení TreeBurst zadaná napevno.

Jak bylo naznačeno v sekci 4.9, pro otestování navazování spojení mezi uzly je třeba každý uzel spustit ve zvláštním profilu prohlížeče, nestačí pouze otevřít dvě záložky v jednom profilu, protože by obě sdílely stejnou lokální databázi a identitu. Jako druhý profil může posloužit i anonymní okno.

C.2 Kompilování lokální verze

Publikovaná verze rámce a knihoven ze složky `sources/published` běžně i v případech vlastních balíčků odkazuje na závislosti dostupné *online*. Pokud má být projekt sestaven v podobě zamýšlené při odevzdání ze zdrojových kódů dodaných na přiloženém DVD, je třeba tyto závislosti v souborech `packages.json` přepsat tak, aby odkazovaly na lokálně sestavené balíčky.

Je důležité, aby ve všech projektech vlastní závislosti odkazovaly na stejné *absolutní cesty* k lokálním balíčkům. Pokud se použijí relativní cesty nebo rozdílné verze, NPM je vyhodnotí jako dvě verze a knihovna se začlení se do výsledného JavaScriptu vícekrát, což pravděpodobně způsobí nefunkčnost výsledku.

Verze všech balíčků s připravenými přepsanými lokálními cestami závislostí jsou uložené ve složce `sources/patched-for-local-build`. Při procesu sestavení se vytvářené balíčky kopírují do složky `~/npm-pkgs` a nastavení závislostí v souborech `package.json` s tímto umístěním počítá.

C.2.1 Prerekvizity

Pro sestavování balíčků kompilovaných do JavaScriptu je nutné nainstalovat aktuální Node.js (testováno na verzi 11). Toho je možné dosáhnout pomocí

```
sudo apt-get install curl python-software-properties
curl -sL https://deb.nodesource.com/setup_11.x | sudo -E bash -
sudo apt-get install nodejs
```

Pro kompilování signalizačního serveru je třeba správce balíčků OPAM v aktuální verzi a několik závislostí:

```
sudo apt-get install m4 pkg-config libgmp-dev libssl-dev
sudo add-apt-repository ppa:avsm/ppa
sudo apt-get update
sudo apt-get install opam
```

Projekt je navržen pro fungování s překladačem OCamlu verze 4.06.1. Na tuto verzi je možné přepnout pomocí

```
opam switch create 4.06.1
eval $(opam env)
opam update
```

C.2.2 Sestavení

Sestavení by mělo být možné provést automatizovaně spuštěním skriptu `build-all.sh`:

```
cd sources/patched-for-local-build
./build-all.sh
```

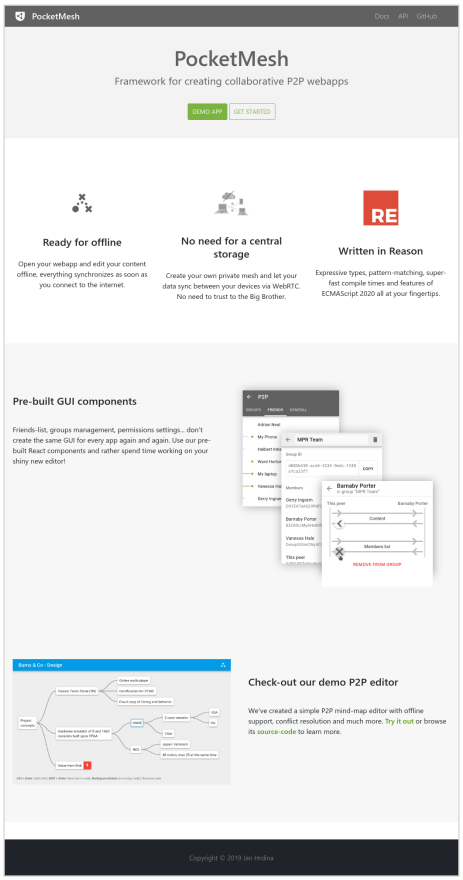
Ve složce `tree-burst/build` by po sestavení měla být k dispozici produkční verze editoru TreeBurst, v `pocket-mesh/signal-server/_build/default/src/SignalServer.exe` potom spustitelný soubor signalizačního serveru.

Příloha D

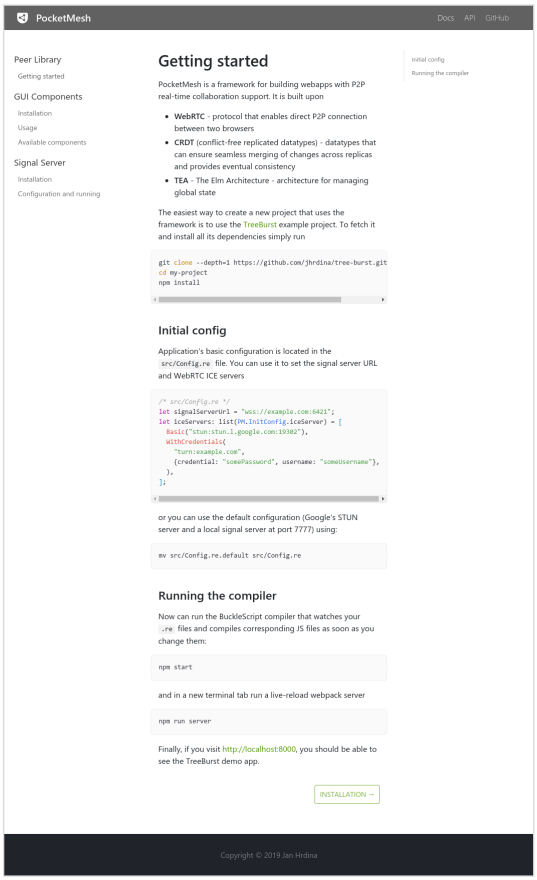
Dokumentace a referenční příručka

V rámci dokumentace rámce byla vytvořena webová stránka[1] (viz obr. D.1 a D.2), která obsahuje příručku pro instalaci rámce, signalizačního serveru a základní dokumentaci pro zahájení vývoje. Kromě toho je k dispozici také referenční příručka generovaná ze zdrojových kódů.

Sestavená webová stránka se vším uvedeným je k dispozici také na přiloženém DVD ve složce `build/web`.



Obrázek D.1: Webová stránka rámce.



Obrázek D.2: Příručka pro vývojáře.

[1] https://pocket-mesh.hrdinajan.cz