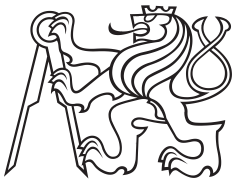
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Synchronizace stavu rozdílovými aktualizacemi

Bc. Matyáš Neuvirt

Vedoucí: RNDr. Ondřej Žára
Obor: Otevřená Informatika
Studijní program: Softwarové inženýrství
Květen 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Neuvirt** Jméno: **Matyáš** Osobní číslo: **474460**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Rozdílové aktualizace dat

Název diplomové práce anglicky:

Differential data updates

Pokyny pro vypracování:

Předmětem této práce je infrastruktura pro synchronizaci stavu pomocí rozdílových aktualizací.
Uvažte komunikační model, ve kterém vystupuje server jako zdroj pravdy (tj. majitel kompletního aplikačního stavu) a libovolný počet klientů, kteří potřebují přijímat aktualizace stavu tak, jak k nim v reálném čase dochází. Klienti nemusí mít zájem o celý stav a mohou se tak ucházet o aktualizace jen specifikované podmnožiny stavu.
Prozkoumejte existující řešení tohoto problému v kontextu klientů = webových prohlížečů.
Následně navrhněte a naimplementujte framework, který řeší popsanou úlohu. Klienti mohou jevit zájem o celý datový model, nebo jeho podmnožinu (např. pomocí JSONPath), server posílá klientům rozdílové aktualizace tak, aby
1) minimalizoval zpoždění mezi změnou stavu a informovaností klienta,
2) minimalizovat objem přenášených dat.
Implementujte protokol nezávisle na transportním médiu - uvažujte možnost použití HTTP, WebSocket, WebRTC.
Analyzujte a změřte, jak mají různé algoritmy na detekci a serializaci rozdílových dat vliv na rychlost celého systému, resp. objem přenášených informací.

Seznam doporučené literatury:

https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html
https://developer.mozilla.org/en-US/docs/Web
https://json-schema.org/

Jméno a pracoviště vedoucí(ho) diplomové práce:

RNDr. Ondřej Žára Katedra počítačové grafiky a interakce

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

Datum zadání diplomové práce: **02.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

RNDr. Ondřej Žára podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

. _____
Datum převzetí zadání

_____
Podpis studenta

Poděkování

Děkuji panu RNDr. Ondřeji Žárovi za veškeré odborné konzultace a za vedení této diplomové práce.

Prohlášení

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

V Praze, 20. května 2022

Abstrakt

Práce studuje možnosti synchronizace mezi několika klienty a autoritativním serverem v reálném čase. Velký důraz je kladen na minimalizaci množství přenášených informací a krátkou prodlevu pro obdržení aktuálního stavu. Výsledkem práce je framework NewSync využívající rozdílových aktualizací podpořených dalšími technikami ke snížení objemu posílaných informací s co nejvyšší mírou automatizace procesu detekování změn ve stavu aplikace.

U několika formátů byla zkoumána efektivita jejich komprese. Pro potřeby webové aplikace se nejvhodnějším formátem stal MessagePack. Pro automatizaci detekce změn bylo využito nativní Proxy API v Javascriptu. Dále je popsáno, jaká struktura zpráv zanechává nejmenší paměťovou stopu. Součástí řešení je i aplikace, která demonstruje integraci a používání frameworku. Na závěr je porovnán objem dat nutných pro synchronizaci kompletními aktualizacemi ve formátu JSON v kontrastu se synchronizačními zprávami frameworku NewSync a časová náročnost spojená s automatickou detekcí změn.

Klíčová slova: synchronizace, JavaScript, Node.js, rozdílové, aktualizace, server, klient, Proxy API

Vedoucí: RNDr. Ondřej Žára

Abstract

This thesis studies options of synchronization between multiple clients and an authoritative server in real time. Great care is taken to minimize the amount of data transferred and to keep the delay between receiving the current state to a minimum. The goal is to create a framework that will use differential updates and other techniques to minimize network bandwidth with an emphasis on automating the difference detection.

Many formats were tested for their ability to compress data effectively. In the context of web applications MessagePack ended up showing the best results. The process of automatic differences detection is driven by Proxy API which is natively present in the JavaScript language. Furthermore, we discuss efficient message structure that leaves smaller memory footprint. The solution also includes an application that demonstrates the integration and use of the framework. Lastly, the amount of data required for synchronization via complete updates is compared to synchronization data generated by the NewSync framework in respect to performance requirements for the automatic difference detection.

Keywords: synchronization, JavaScript, Node.js, differential, updates, server, client, Proxy API

Title translation: Differential data updates

Obsah

Obrázky

Tabulky

Seznam zkratek

ECMA *European Computer Manufacturers Association.*

ESM *ECMAScript Modules.*

HTTP *Hypertext Transfer Protocol.*

JS *JavaScript.*

NPM *Node Package Manager.*

TCP *Transmission Control Protocol.*

UDP *User Datagram Protocol.*

WebRTC *Web Real-Time Communication.*

Kapitola 1
Úvod

1.1 Předmluva

Neustále se zvyšuje digitalizace a roste počet IT systémů, které spolu musí komunikovat a sdílet informace. Spousta softwarových nástrojů se přesouvá z desktopových programů do formy volně přístupných webových aplikací v prohlížeči, které podporují spolupráci více uživatelů na jednom projektu. Z těchto důvodů se zvedá objem posílaných dat a nutnost synchronizovat stav systémů a uživatelů, přičemž při zvedající se komplexitě aplikací tyto nároky stále rostou. A to jak na výkon výpočetních prostředků a schopnosti vývojářů, tak na rychlost síťového připojení. Obzvláště pro pomalá připojení nebo omezená mobilní data je klíčové se snažit objem posílaných dat snížit.

Z těchto důvodů je v práci zkoumáno, jak synchronizovat data serveru mezi několik klientů, jak by bylo možné objem dat nutných ke komunikaci snížit, a jak toho docílit pro vývojáře pohodlným způsobem. Cílem je vytvořit framework, v němž se po základní inicializaci synchronizace děje "na pozadí". Vývojář, ať serveru nebo klientské aplikace, nemusí používat žádné speciální knihovní metody pro změny hodnot, pouze by napřímo upravoval stav dat v aplikaci.

1.2 Požadavky na řešení

Jedním z cílů práce je optimalizovat čas a výkon nutný pro komunikaci a minimalizovat velikost posílaných synchronizačních zpráv ve snaze zvýšit propustnost sítě a snížit nároky na rychlost připojení. Kratší zprávy ovšem neznamenají automaticky vyšší rychlost jejich zpracování. Na základě způsobu, kterým snížíme objem dat zprávy (např. komprese), může být práce s daty v konečném důsledku pomalejší. Sice snížíme množství dat putujících po síti, ale celkově se počet užitných dat sníží kvůli dlouhé transformaci dat před odesláním a po jejich přijetí. Je proto velice důležité správně balancovat mezi velikostí zpráv a režií nutnou na jejich kódování.

Dalším cílem je nalézt vhodné řešení pro architekturu server-klient, kdy server je autoritativní, jediným zdrojem pravdy a jeho úkolem je udržovat, spravovat a distribuovat stav aplikace mezi klienty. Klienti i server musí za

všech okolností vidět stav shodně a bez zbytečných prodlev. Předpokládáme, že simulace se v čase rychle mění a její stav je nutné co nejrychleji doručit ostatním.

Uživatelé mohou provádět úpravy stavu v aplikaci posíláním požadavků na server. Server požadavky zpracuje, změní data dle nutnosti a odešle informace o změně všem ostatním klientům.

Shrnutí požadavků:

1. snížení velikosti zpráv,
2. synchronizace stavu,
3. rychlé doručení nového stavu.

1.3 Hlavní cesty ke splnění požadavků

1.3.1 Formát zpráv

Velice snadný způsob, jak snížit velikost zpráv a nepřímo i rychlost doručení bez velikých nároků na implementaci, může být výběr vhodného formátu. Důraz je kladen na efektivitu komprese dat a náročnost na serializaci a deserializaci zpráv.

Při využití správných knihoven je přechod mezi formáty z pohledu vývojáře triviální. Rozhraní pro kódování/dekódování zpráv daného formátu se ve všech zkoumaných knihovnách během testování v kapitole 6 skládalo primárně ze dvou metod. Z tohoto důvodu by mohly z kompaktnějšího formátu těžit i již existující aplikace bez rozsáhlých změn v implementaci.

Různé formáty jsou podrobněji uvedeny v kapitole 2.

1.3.2 Synchronizace a obsah zpráv

Nejsnazším způsobem udržení synchronizace a konzistence je pravidelné posílání kompletního stavu aplikace a zajištění jeho doručení.

Pro menší projekty je toto dostačujícím řešením, avšak s přibývající komplexností aplikačního stavu může narůst velikost zpráv až do neúnosné míry.

Dalším aspektem je obsah samotných zpráv. Ještě před jejich zakódováním je možné uvažovat o jejich zmenšení a zároveň zachovat stejné množství užitných informací.

Posíláním pouze rozdílů ve stavu lze výrazně ušetřit objem dat, obzvláště, nemění-li se stav rychle, případně, pokud je část stavu statická. Nevýhodou je, že se sníží robustnost synchronizace. Ztráta nebo poškození nějaké zprávy nebo informace, kterou nese, může narušit aplikační stav a uvést jej do nekonzistence nebo úplně vyřadit aplikaci z provozu. V lepším případě se problém vyřeší při změně datové položky na serveru a příchodu nových hodnot, v horším zůstane nekonzistence až do okamžiku obdržení kompletní aktualizace celého stavu, typicky při odpojení a připojení klienta.

Jak s rozdílovými stavy efektivně pracovat je popsáno v kapitole 3.

1.3.3 Propojení serveru a klienta

V prostředí internetu existuje množství protokolů pro propojení účastníků komunikace různými způsoby a s rozdílnými vlastnostmi. Pro naše potřeby požadujeme protokol, který umožní rychlou a obousměrnou výměnu dat vhodný v kontextu webových aplikací. K dispozici existují již zaběhlé technologie, které vznikly přímo za tímto účelem.

Více do hloubky jsou jednotlivé možnosti spojení popsané v kapitole 4.

Kapitola 2
Formáty dat

2.1 JSON – JavaScript Object Notation

JSON je v dnešní době jeden z nejrozšířenějších formátů pro přenos zpráv a požadavků ve webovém prostředí [1]. Lze se s ním setkat i při ukládání záznamů v databázích, nastavení konfigurace programů atd.

JSON je lidsky čitelný a analyzovatelný formát, který je podmnožinou JavaScriptu [2]. Díky své jednoduchosti je hojně rozšířený a podporovaný napříč různými programovacími jazyky, ať již v základu nebo pomocí komunitních knihoven.

Struktura záznamu je velice podobná zápisu literálům objektů v JavaScriptu. Data jsou strukturovaná jako dvojice klíč-hodnota. Informace jsou zapsané jako obyčejný text ve formátu UTF-8. Záznamy tak lze velice snadno vytvářet i ručně, na rozdíl od binárních formátů. Struktura klíče se řídí stejnými pravidly jako klíče u objektů v JavaScriptu, tedy klíčem může být jakýkoliv textový řetězec. Hodnoty mohou být buďto také textové řetězce, čísla, pravdivostní hodnoty, pole hodnot, objekt nebo speciální označení pro prázdnou hodnotu (*null*). Objekty se mohou do sebe libovolně vnořovat a tvořit tak stromovou strukturu. Zmíněné vlastnosti tak mohou usnadnit vývoj aplikací ve fázi implementace nebo testování.

Kromě těchto předností se JSON stal populární především v souvislosti s menší velikostí zpráv v porovnání s dobovou alternativou, formátem XML, nativní[1] podpoře v JavaScriptu, jež je vedoucí programovací jazyk pro vývoj klientských webových aplikací, a vysoké rychlosti serializace a deserializace dat. Díky těmto skutečnostem není překvapující silné postavení formátu JSON v tomto prostředí.

Přestože si JSON získal vysokou popularitu, a leč je jistým vylepšením svého předchůdce, tj. formátu XML, co se poměru kompresního poměru užitných informací týče, stále existují efektivnější formáty. Je ale nutné zvážit jejich použitelnost a rychlost v kontextu internetových prohlížečů, protože v nich je nativně podporován pouze JSON. Funkcionalitu jiných formátů je nutno doplnit externě, např. knihovnami, které zpravidla nedosahují stejné rychlosti

[1] Specificky vyvinuto/připraveno pro využití v daném prostředí, zpravidla efektivnější než z vnějšku dodaná implementace.

zpracování dat. Více v kapitole Porovnání formátů v prostředí JavaScript.

Ukázka JSON záznamu:

```
{"breakfast_menu": [
    {"name": "Belgian Waffles",
     "price": "$5.95" },
    {"name": "French Toast",
     "price": "$4.50"}
  ]
}
```

2.2 BSON – Binary JSON

BSON byl vyvinut specificky pro použití v databázovém systému MongoDB k efektivnějšímu ukládání záznamů. Cílem formátu je umožnit jejich rychlejší zpracování, procházení, vyhledávání a snížit potřebnou velikost na disku při ukládání.

BSON se velice se podobá JSONu. Také pracuje s dvojicí klíč-hodnota, navíc obsahuje další datové typy, jako například datum, časovou značku, celá čísla, čísla velikého rozsahu a další, které jsou typicky využívané v databázových systémech. Hlavním rozdílem je ukládání informací v binární podobě. Data jsou takto rychleji strojově čitelná, jelikož není třeba prvně přečíst textové symboly a ty poté parsovat. V záznamu jsou pro ještě rychlejší procházení přibalená metadata, která umožňují vyhledat konkrétní části bez nutnosti prvně načíst a zpracovat celý záznam. BSON tak splňuje všechny důležité vlastnosti pro prostředí, kterému byl určen.

Přestože jedním z cílů BSONu je menší velikost zpráv, v porovnání s JSONem není jeho velikost výrazně menší. Naopak v některých případech může být větší, protože: [3]

- Přídavná metadata snižují efektivitu binární komprese.
- BSON pro ukládání čísel v binárním zápisu vždy používá 4 bajty. JSON reprezentuje čísla pomocí znaků číslovek, kdy jeden znak zabírá 1 bajt. Čísla složená ze tří a méně číslovek tedy zabírají v JSONu méně místa. Úspory se začínají projevovat až pro čísla složených z 5 a více číslic.
- Klíče a hodnoty v podobě textových řetězců mohou tvořit podstatnou část dat. Textové řetězce v binární podobě nelze rozumně zkracovat.

Vlastnosti BSONu se určitě vyplatí v databázových systémech, které obsluhují velké množství klientů a uchovávají velké množství dat, ve kterých je třeba efektivně vyhledávat. Ovšem pro případ synchronizace v reálném čase, kdy pravidelně dostáváme zprávy o aktualizaci stavu, potřebujeme načíst a zpracovat celou zprávu, nikoliv jen vybrané části[2]. Funkce přidávání metadat navíc pro rychlejší vyhledávání pouze konkrétních částí zpráv pozbývá smysl a stává se spíše přítěží, protože snižuje kompresní poměr.

[2]Předpokládáme, že v rámci optimalizace a snahy snížit objem posílaných dat server posílá pouze ta data, která jsou nutná pro zajištění synchronizace.

2.3 MessagePack

MessagePack je obdobně jako BSON binární formát dvojic klíčů-hodnota, jehož cílem je minimalizovat velikost zpráv a zajistit efektivní zapisování a čtení. Efektivní zápis spočívá například v proměnném počtu bajtů nutných k reprezentaci menších čísel.

Nevýhodou formátu MessagePack jsou některá omezení související s reprezentací dat. Protože čísla se zapisují binárně, jejich maximální hodnota je omezena počtem bajtů vyčleněných pro jejich zápis. Limitovaný je například i počet prvků v poli nebo počet klíčů ve vnořeném objektu. Všechny limity jsou dostatečně vysoké, aby za běžného používání nemohly nastat problémy (zpravidla $(2^{32} - 1)$ [4]). Navíc většina programovacích jazyků má podobné limity pro velikosti čísel, zpravidla 32 nebo 64 bitů. Ovšem v databázích je možné se setkat se speciálními datovými typy překračující tato omezení. Dále například skriptovací jazyk Python ve verzi 3 umožňuje teoreticky zapsat čísla až nekonečné velikosti. Přenos těchto čísel pomocí MessagePacku by provázely komplikace.

MessagePack umožňuje definovat až 127 vlastních datových typů. Vlastní datový typ obsahuje unikátní číselný kód od 1 do 127 a informaci, jak jej binárně zapsat a zrekonstruovat. Všichni účastníci komunikace pomocí takových zpráv musí lokálně definovat datové typy shodně. Ty lze definovat dynamicky až za běhu programu, aplikace je tak teoreticky nemusí znát během spuštění, ale může si je vyžádat až později od jiných účastníků komunikace.

Za další specifikum MessagePacku lze považovat absenci oficiálního standardu ve formě RFC[3] dokumentu nebo zaštítění nějakou organizací. Původně byl MessagePack komunitní projekt, který nabyl veliké popularity.

2.4 CBOR – Concise Binary Object Representation

Podle slov autorů je CBOR založený na JSONu[5] a inspirovaný MessagePackem, leč si neklade za cíl být jeho následníkem[6]. Navzdory tomuto tvrzení vznikla myšlenka na vytvoření CBORu z nedostatku v tehdy již existujícím MessagePacku. Podstatou problému bylo nejednoznačné rozlišení datového typu pro textový řetězec a sekvenci binárních dat. Sepsání specifikace nového formátu se ujala organizace IETF (Internet Engineering Task Force). Přestože před zhotovením specifikace CBORu byl již daný nedostatek v MessagePacku dlouho opraven, IETF bylo stále odhodlané CBOR dokončit[7].

Obdobně jako u předchozích formátů se jedná hlavně o zápis dvojic klíčů a hodnot v binární podobě. V porovnání s JSONem obsahuje navíc datový typ sekvence bajtů. Datový typ číslo byl rozdělen na kladné celé číslo, záporné celé číslo a číslo s pohyblivou desetinnou čárkou. Podobně jako MessagePack umožňuje definovat vlastní datové typy pomocí štítků (tagů). Již samotný standard jich několik definuje [6]. Celkově CBOR zvládne pracovat až s $(2^{64})-1$

[3]Request for Comments – veřejně dostupné dokumenty, které se často využívají k publikaci a popisu standardů a norem především v oblasti výpočetních technologií.

štítky (MessagePack umožňuje definovat pouze 127 vlastních datových typů). Organizace IANA[4] dokonce spravuje veřejně dostupný registr[9] štítků, do kterého mohou své vlastní štítky přidávat na vyžádání i běžní uživatelé.

2.5 Ostatní

NPM (Node Package Manager) – jeden z hlavních repositářů uživatelských balíčků a knihoven pro JavaScript a zároveň aplikace pro jejich snadnou správu, stahování a instalaci.

Momentálně je široký výběr různých formátů. Mezi další formáty typu JSON lze zařadit například UBJSON – další binární zápis JSONu a následník BSONu. Kvůli jeho nízké popularitě mu v práci není věnována další pozornost (implementace je dostupná v repositáři NPM s pouhými 758 týdenními staženími [10]).

XML záznamy v porovnání s předchozími formáty zabírají příliš mnoho místa. Na vině je veliká redundance ve stylu jeho zápisu. Například jednotlivé záznamy jsou vždy uvozené tagy, které se následně musí uzavřít. Během uzavírání se vždy musí zopakovat jejich jméno, které se zbytečně musí zapsat dvakrát. Výhodou je naopak lepší expresivnost a strukturalizace dat, která usnadňuje vyhledávání v záznamech.

Ukázka zápisu stejného objektu, který byl demonstrován v sekci 2.1 JSON – JavaScript Object Notation (110 znaků), tentokrát v XML (152 znaků), o stejné výpovědní hodnotě.

```
<breakfast_menu>
    <food>
        <name>Belgian Waffles</name>
        <price>$5.95</price>
    </food>
    <food>
        <name>French Toast</name>
        <price>$4.50</price>
    </food>
</breakfast_menu>
```

Pro přehlednost je zde znovu uvedena ukázka zápisu ve formátu JSON ze sekce 2.1.

```
{"breakfast_menu":
  [
    {"name": "Belgian Waffles",
     "price": "$5.95" },
    {"name": "French Toast",
```

[4] Internet Assigned Numbers Authority je organizace, která spravuje především přidělování IP adres a DNS kořenových záznamů, avšak nabízí i správu číselných indexů pro různé protokoly[8].

```
    "price": "$4.50"}
  ]
}
```

2.6 Formáty se schématem

Všechny doposud vyjmenované formáty byly tzv. bez schématu, tedy čitelné bez předchozí znalosti struktury záznamu, protože ta je obsažena ve zprávě samotné. Díky této vlastnosti je formát snáze použitelný a více flexibilní. Nevýhodou je, že v sobě musí obsahovat informace, jak data číst a co reprezentují (např. jména klíčů a datový typ položek), což jsou z našeho pohledu zbytečná data, je-li našim cílem minimalizace.

Protikladem jsou formáty se schématem. Těm se úmyslně nevěnuji do hloubky, neboť je s nimi složitější pracovat, a protože vyžadují připravit a definovat schéma v aplikační vrstvě. Protože tato práce se zabývá především prostředím JavaScriptu, který není typovaný jazyk, používání těchto formátů by vyžadovalo zmapovat používané objekty v aplikaci a zajistit dodržování datových typů určených ve schématu.

Dalším faktorem, proč se formátům bez schématu vyhnout, je použít v rámci zadání práce metodu rozdílových aktualizací. Jednotlivé změny (obzvláště u vnořených objektů) lze obtížně předvídat a špatně strukturovat. Rigidní schéma není vhodné, protože položky různých datových typů v záznamu mohou různě přibývat a ubývat podle prováděných změn. Pro tyto případy by šlo u formátů se schématem využít datový typ “mapa”, který funguje na principu klíč-hodnota. A to je stejný přístup k zápisu jako u formátů bez schématu.

Mezi formáty se schématem patří například Apache Avro, Apache Thrift nebo Protobuf.

Kapitola 3

Rozdílové aktualizace a snížení objemu dat

Smyslem rozdílových aktualizací je neposílat celý stav aplikace, ale pouze nastalé změny od posledního synchronizačního cyklu. Už jen samotným přechodem na rozdílové aktualizace se výrazně sníží objem posílaných dat, ale stále existuje prostor pro další zlepšení. Hlavním problémem je, jak účinně zjišťovat rozdíly a efektivně je rozesílat.

Změny se detekují pomocí tzv. výchozího snímku, vůči kterému porovnáváme všechny nastalé rozdíly. Pokud je potřeba ukládat celkovou historii stavu, jednotlivé změny si postupně zaznamenáváme. Jakmile se změn nahromadí příliš mnoho, vytvoří se nový výchozí snímek aplikováním všech dosavadních změn. Rekonstrukce stavu do libovolného bodu v minulosti totiž spočívá v nalezení nejbližšího výchozího snímku od daného momentu, nalezení všech změn v mezičase výchozího snímku a času, který nás zajímá, a aplikování nalezených změn. Častější kompresí změn do nových výchozích snímků se zvyšuje redundance dat, ale rekonstrukce do jakéhokoliv momentu se pak stává rychlejší, protože není třeba aplikovat tolik rozdílů.

V rámci práce uvažujeme o demo aplikaci, která se zaměřuje především na synchronizaci v reálném čase bez nutnosti ukládat historii. V tomto případě si stačí pamatovat pouze jeden výchozí stav. Provedené změny se rozešlou klientům a poté ihned aplikují.

V rámci rešeršních prací nebylo nalezeno mnoho relevantních zdrojů nebo literatury, která by se tématu věnovala do hloubky. V diskuzních fórech jsou obsaženy informace obecného charakteru nebo návrhy či postupy pro jednoduché implementace, ne však složitější konkrétní algoritmy či strategie.

3.1 Strategie zjišťování rozdílů

3.1.1 Celkové porovnání

Pokud máme k dispozici dva kompletní snímky stavu v různých časech, nezbývá nám, než zevrubně projít celou hierarchii vlastností objektů a zaznamenávat si postupně nalezené rozdíly. Změny můžeme reprezentovat jako pole párů změněných klíčů a nových hodnot, nebo, v netypovaných jazycích, jako nový objekt, který obsahuje pouze změněné atributy a jejich hodnoty.

Celkové porovnání je nutné použít převážně v případech, kdy nemáme

možnost objekt měnit napřímo, ale postupně se nám odjinud dostává nových verzí (např. výsledky dotazů do databáze, externího API, nebo při práci s immutable[1] objekty).

3.1.2 Průběžné zapisování změn

Pokud změny provádí samotná aplikace, nikoliv externí zdroje, můžeme její stav různě upravovat a okamžitě si zapisovat dané rozdíly do seznamu. Efektivita této techniky je závislá na počtu změn v mezičase dvou synchronizačních cyklů – okamžitý zápis rozdílů do seznamu s sebou nese jistou režii navíc. Čím větší je monitorovaný stav, a čím méně změn v něm nastane, o to vyšší efektivitu bude tato technika mít oproti celkovému porovnání. Protože předpokládáme distribuci stavu v reálném čase, pravděpodobnost extrémního množství změn mezi cykly není vysoká a dále klesá s vyšší distribuční frekvencí.

Problémem průběžného zapisování změn je nutnost po provedení jakékoliv změny tuto změnu navíc ještě explicitně zapsat. Vystavujeme se tak riziku, že vývojář by mohl na toto zapomenout. Lepší řešení by bylo změny provádět pouze přes pomocné funkce. Bohužel tento způsob je těžkopádný a u přechodu na takovouto detekci změn u již existujících aplikací by muselo dojít k výraznému přepisování zdrojového kódu. Konkrétně u přepisování hodnot bychom mohli v JavaScriptu využít tzv. setterů. Nejedná se o metodu v pravém slova smyslu – nevolá se pomocí kulatých závorek, ale automaticky při přiřazení hodnoty pomocí znaménka "=". Opět, u existujícího kódu by bylo nutno všechny tyto settery dodatečně manuálně definovat. Bohužel něco obdobného nelze využít pro klíčové slovo "delete" na mazání vlastností objektů.

3.1.3 Proxy API

Proxy API v JavaScriptu umožňuje vytvořit proxy z libovolného objektu a automaticky na něm monitorovat prováděné operace, jako např. přístup k jeho datům, přidání, odstranění atributu, změnu jeho hodnoty a další. Proxy tak můžeme využít k automatickému zaznamenávání všech provedených změn v aplikačním stavu. Omezíme tak chyby lidského faktoru, kdy vývojář může zapomenout změnu explicitně zapsat.

Nevýhodou je, že všechny operace prochází skrze tuto proxy za vzniku režie navíc, i když na tyto změny nedefinujeme jakékoliv vlastní chování. Přístup k vlastnostem objektu může být až 200krát pomalejší [11]. Přestože se jedná o vysoké relativní zpomalení, při převedení do absolutního času se stále jedná o velmi krátké časové intervaly při ojedinělých přístupech, které by neměly znatelně zpomalit běh aplikace. Existují krajní případy, kdy by toto zpomalení mohlo být citelné, naštěstí můžeme do proxy přidat různé optimalizace, které tento dopad limitují. Více o těchto situacích a jejich řešení je uvedeno v sekci 7.5.1 Optimalizace proxy.

[1] Immutable objekty nelze měnit. Můžeme pouze získat jejich kopii s požadovanými změnami jako nový objekt.

3.2 Slovník klíčů

Velikost synchronizačních zpráv můžeme kromě rozdílových aktualizací ještě více zmenšit zkrácením klíčů, které se ve formátech bez schématu musí posílat společně s hodnotami. Můžeme se inspirovat například Huffmanovým kódováním, kdy nejfrekventovanější hodnoty jsou reprezentované nejkratším řetězcem symbolů. V našem případě bychom metriku museli vhodně upravit tak, aby v závislosti na délce klíče a jeho frekvenci výskytu v synchronizačních zprávách docházelo k optimálnímu mapování na krátké klíče.

Je nezbytné zajistit, aby ve slovníku nedocházelo ke kolizi v překladech, kde by zkrácený zápis klíče odpovídal normálnímu zápisu jiného klíče a naopak. Snadným řešením může být pro zkrácený klíč vyčlenit speciální prefixní znak. JavaScript umožňuje, aby se skládaly z jakýchkoliv znaků textového řetězce. Vybrat takový znak, aby nedocházelo ke konfliktům v běžných situacích, je velice snadné – zkrácené klíče můžeme prefixovat jakýmkoliv symbolem, klidně i pomocí uvozovky, apostrofu, diakritiky, matematických znamének atd. K interakci s danou položkou pak musíme použít méně příjemný přístup k vlastnostem pomocí textového řetězce uvnitř hranatých závorek, protože standardní systém přístupu pomocí tečky neumožňuje uvozovat klíče pomocí čísel a špatně reaguje na přítomnost speciálních znaků kdekoli v klíči.

Ve zprávách s rozdílovými aktualizacemi budeme posílat krátké klíče. Při obdržení se klíče pomocí slovníku automaticky transformují zpět na výchozí. Programátor tak bude odstíněn od celého procesu a s daty bude nadále pracovat tak, jak je zvyklý.

Slovník můžeme tvořit dynamicky společně s detekcemi změn. Musíme pak ale zajistit, aby každý účastník komunikace měl aktuální slovník. Pro jednoduchost a zamezení kolizí mezi slovníky serveru a klientů by jej tvořil pouze server, který by do něj přidával zápisy a synchronizoval jej s klienty stejným způsobem jako stav simulace.

3.3 Využití schématu ve zprávách

Přestože pro výsledný framework není plánované využít formát se schématem, můžeme se jimi inspirovat a využít některé jejich aspekty.

Díky znalosti schématu můžeme reprezentovat jednoduché objekty jenom pomocí seznamu hodnot, v binárnímu zápisu pak jen pouze bajty bez jakýchkoliv informací navíc. Obejdeme se úplně bez klíčů a specifikace datových typů. Schéma musí být ovšem známé v momentu čtení zprávy, a hlavně shodné u obou komunikujících stran. Bez něj jsou data pouze náhodnou směsicí bitů, o kterých nevíme, jak je interpretovat nebo co znamenají.

Schématem můžeme reprezentovat i složitější objekty s proměnnými délkami (v některých případech mohou klíče chybět, počet hodnot v poli není dopředu znám atd.). Efektivita pak mírně klesne, protože například u pole musíme zavést symboly pro jeho uvození a ukončení, pro položky zprávy, které nemusí být vždy přítomné, je nutné zavést uvozující znak označující jejich přítomnost.

U typovaných jazyků je práce se schématem mnohem snazší, protože existují nástroje, které jejich tvorbu usnadní. Přečtením zdrojového kódu nástroje vytvoří schéma převážně automaticky a vygenerují všechny potřebné doprovodné soubory. Pro čistý JavaScript by se muselo schéma pro všechny objekty tvořit z velké části ručně. Nástroje by sice mohly detekovat alespoň základní strukturu objektů, ale měly by problém detekovat konkrétní datové typy jednotlivých položek. Navíc, protože datový typ proměnných se za běhu programu může libovolně měnit, není jistota, že schéma zůstane korektní po celou dobu.

V rámci frameworku se posílá několik konkrétních typů protokolových zpráv, které ve struktuře na nejvyšší úrovni vypadají vždy stejně a uživatel ani aplikační stav zde nemá na jejich podobu vliv. Schéma se může simulovat pomocí pole, u kterého je dopředu znám datový typ a význam jednotlivých položek na konkrétních indexech.

3.4 Frekvence synchronizace a balíčkování zpráv

Při posílání jakýchkoliv zpráv přes internet, ať už se používá spojení postavené na UDP nebo TCP (viz kapitola 4 Druhy síťového připojení), se musí data zabalit do tzv. paketu a pak poslat. Pakety v sobě vždy musí kromě užitných dat obsahovat i další informace, tzv. hlavičku, kde se specifikuje např. cílová a výchozí IP adresa. Tento proces se děje automaticky na pozadí. Důsledkem je, že poslání stejného množství informací napříč několika zprávami je méně efektivní než poslat jednu větší. Příliš velké zprávy se sice automaticky rozdělí na menší, jakmile přesáhnou jistou velikost, kvůli tzv. fragmentaci, ale obecně se stále vyplatí data seskupit do jedné větší zprávy.

Prakticky to znamená, že místo posílání jednotlivých změn stavu okamžitě, jak nastanou, se může chvíli počkat, než se jich nastřádá více, a poslat takto celý balíček změn. Podle potřeb aplikace je možné mít tuto umělou prodlevu libovolně dlouhou. Tento interval bude v práci dále označován jako synchronizační cyklus.

Pokud by se hodnota nějaké položky v mezičase synchronizačního cyklu měla změnit vícekrát, ve výsledné zprávě se může poslat pouze finální podoba atributu (za předpokladu, že klient nepotřebuje znát úplnou historii změn, ale zajímá ho pouze aktuální hodnota), čímž se ještě více omezí posílaná data.

3.5 Efektivní reprezentace změn

Pomocí rozdílů můžeme zapsat kromě struktury objektů i samotné hodnoty. Změnu hodnoty z 1 000 na 1 001 můžeme v JSONu reprezentovat jako +1 (2 znaky = 2 bajty) místo výsledného čísla 1 001 (4 znaky = 4 bajty) a ušetřit tak 2 bajty. Tento způsob je ovšem závislý na použitém formátu. Posílaná data by v sobě obsahovala kromě konkrétních hodnot také operátory a příkazy, jak s daty nakládat. Pro BSON by byl uvedený příklad kontraproduktivní,

protože je v něm číslo vždy zapsané pomocí 4 bajtů. Přidáním znaménka plus bychom zvedli celkovou paměťovou náročnost o jeden bajt.

Praktičtější se tak stává pomocí rozdílů zapisovat změny např. v textových řetězcích nebo polích. Místo poslání nového pole můžeme pouze poslat změny v konkrétních indexech nebo nové položky pole, a na která místa je vložit, nebo indexy položek, které z pole smazat. Obecně se nám celé pole vyplatí poslat pouze, pokud by počet změn ve struktuře pole převýšila jeho délku po aplikaci všech změn – místo informací o smazaní 6 položek pole s původní velikostí 10 pošleme raději nové pole o 4 položkách.

Dalším vylepšením by mohlo být označení smazání položky, např. na posledním indexu pole, a pozdějšího přidání jiného prvku na konec pole pouze jako změnu hodnoty na posledním indexu. Pro tento typ optimalizace se musí obě změny odehrát v mezičase posílání synchronizačních zpráv.

Kapitola 4

Druhy síťového připojení

Spolehlivé připojení je takové, u kterého je zajištěné, že všechny zprávy protistraně dorazí a navíc i v takovém pořadí, v jakém byly odeslány. Pokud se nějaká zpráva během přenosu ztratí, připojení za nás automaticky zajistí poslání náhrady a pozdržení zpracování zpráv dorazivše mimo pořadí.

TCP (Transmission Control Protocol) je spolehlivé připojení. Toho je zajištěno posíláním navíc kontrolních informací o doručení zprávy, které mírně zvyšují režii a objem zpráv. Výhodou je, že programátor nemusí věnovat úsilí dohlížení na korektnost komunikace a doručování zpráv. Vhodným využitím jsou např. aplikace, které vyžadují kompletní a neporušené informace, jejichž ztráta dat by způsobila problémy v konzistenci aplikačního stavu. Významným uživatelem TCP je např. HTTP.

UDP (User Datagram Protocol) spojení není spolehlivé a v základu neposkytuje žádné garance doručení zpráv, ani zpracování jejich pořadí, zato má v porovnání s TCP menší režii pro přenos zpráv. S výpadky, případně se špatným pořadím, aplikace využívající UDP musí počítat a správně na ně reagovat podle svých možností. Ať už se jedná o vlastní systém v aplikační vrstvě nahrazující nespolehlivost nebo schopnost vypořádat se s chybějícími informacemi. Tento způsob je vhodný například pro aplikace posílající neustálý proud dat, u kterých mají informace krátkou životnost a jejich ztráta se výrazně neprojeví, nebo je lze spolehlivě dopočítávat. Ukázkovým příkladem je streamování videí, audia, hovorů atd. Běžná ztráta informací je natolik malá, že se projeví např. pouze chybějící částí jednoho snímku videa. Takový výpadek buďto nemusíme řešit vůbec, protože za okamžik stejně dostaneme nový snímek, nebo se chybějící část obrazu můžeme pokusit nějak zaplnit.

4.1 HTTP – HyperText Transfer Protocol

HTTP, založené na TCP, je jedním z hlavních způsobů komunikace na internetu, který běžně používají webové prohlížeče. Hlavním principem jsou jednorázové požadavky a odpovědi na ně. Pro každý dotaz se musí vytvořit nové spojení, které se po obdržení odpovědi opět uzavře. Opakované vytváření

tohoto spojení pro každý dotaz zbytečně zvedá množství posílaných dat a tím i časovou prodlevu. Z těchto příčin není HTTP a na něm postavené metody vhodné pro pravidelnou synchronizaci v reálném čase.

Další problém v dnešním internetu je přítomnost síťových prvků jako jsou firewally a NAT. Zjednodušeně, toto vytváří situaci, kdy server zpravidla nemůže sám od sebe poslat nějakou informaci klientům, kteří předtím sami nevyslali na server předchozí dotaz. Z tohoto důvodu ani není možné, aby server, jakmile bude mít k dispozici nová data k synchronizaci, automaticky odeslal informace klientům.

Tento problém lze obejít pomocí techniky zvané "long polling": klient pošle požadavek o nový aplikační stav na server, ten udržuje spojení aktivní, ale pozdrží odpověď, dokud nenastane nový synchronizační cyklus. Jakmile klient obdrží nový stav simulace, okamžitě pošle nový požadavek a celý proces se opakuje [12].

4.2 WebSocket

Výše popsané komplikace řeší dnes velice populární technologie WebSocket, také postavená na TCP. Oblíbenosti dosáhla především díky své jednoduchosti používání a dobré podpory v JavaScriptu.

Navázání spojení začíná odesláním specifického HTTP požadavku klienta na server. Po této jednorázové režii je vyjednáno stálé spojení, ve kterém mohou data libovolně proudit oběma směry bez značné prodlevy.

WebSocket splňuje požadavky pro vhodné propojení dvou stran, které spolu potřebují frekventovaně komunikovat v reálném čase. Stává se tak vhodným kandidátem pro účely této práce.

4.3 WebRTC

WebRTC je novější technologie zaměřená především na streamování dat, jako hovorů nebo videa. Konkrétně pro práci s médii je k dispozici přímočaré API, které je jednoduché používat, přičemž komplementuje funkce pro práci s médii v prohlížeči.

Pro účely synchronizace se více hodí poskytnutá funkcionalita provozování vlastního způsobu posílání dat v podobném stylu jako Websocket. Nejedná se ovšem o propojení, kdy jedna strana musí být server a druhá klient, ale o tzv. peer to peer připojení – propojení dvou a více rovnocenných účastníků komunikace. V praxi to znamená, že můžeme propojit napřímo i dva webové prohlížeče. WebSocket tohoto není schopný jinak než napojením obou prohlížečů na jeden server, který by mezi nimi komunikaci zprostředkovával přeposíláním dat.

Podle našich potřeb WebRTC může fungovat v režimu TCP i UDP. Pokud využijeme zabudované funkcionality pro streamy, automaticky se použije UDP. Lze ale vytvořit specializovaný kanál pro aplikační data a libovolně jej nakonfigurovat.

4.3.1 Navázání spojení

Na rozdíl od Websocketu je z vývojářského hlediska složitější připravit nutné zázemí pro používání WebRTC, protože je prvně nutné zprovoznit signální a ICE (Interactive Connectivity Establishment) servery, které se dále dělí na STUN (Session Traversal Utilities for NAT) a TURN (Traversal Using Relays around NAT).

Než je možné navázat spojení mezi dvěma peery, musí se kontaktovat prostředník – signální server – který zprostředkuje prvotní výměnu nezbytných informací. Signální servery musíme využít kvůli podobným důvodům, jako to, že HTTP server nemůže svévolně kontaktovat libovolné klienty. Způsob výměny zpráv v rámci signálního serveru není specifikovaný ve standardu. Je na vývojáři, aby přes něj odeslal a doručil všechny požadované informace. Programátor má úplnou svobodu, jak tyto prvotní informace doručit, využít může např. předtím zmíněný WebSocket.

STUN servery se používají ze stejných důvodů jako signální servery, ale plní odlišnou funkci. Jejich úkolem je najít vhodnou přímou cestu mezi dvěma peery. Na rozdíl od signálních serverů je možné využít volně přístupné STUN servery provozované např. Googlem.

TURN servery se využívají, pokud není kvůli síťovým podmínkám možné navázat přímé peer to peer spojení. V takové situaci je jejích úkolem síťový provoz směrovat přes sebe a vytvořit alternativní cestu mezi peery.

Kapitola 5

Existující řešení

Většina existujících řešení, která by poskytovala automatickou synchronizaci, je součástí větších frameworků nebo služeb, ve kterých je synchronizace pouze část jinak více rozsáhlé funkcionality, kterou lze stěží, jestli vůbec, oddělit a využít samostatně. Často také pracují pouze v režimu replikace stavu databáze na klientskou stranu. Funkcionalitu synchronizovat přímo aplikační stav v paměti serveru poskytují zejména menší knihovny.

Informace, jakým způsobem synchronizace funguje v daných implementacích jsou velice střídmé, jakákoliv dokumentace se obvykle zabývá pouze vysvětlením používání poskytovaného API, nikoliv způsobem implementace. Výjimkou jsou předtím zmíněné knihovny, jejichž způsob synchronizace není explicitně popsán, ale je k dispozici jejich zdrojový kód.

5.1 Firebase

Firebase je rozsáhlá cloudová služba od Google, která, mimo jiné, řeší synchronizaci v reálném čase pomocí rozdílových aktualizací a cachování, kdy při dočasném výpadku spojení na server umožňuje omezený provoz klientské aplikace s lokálními změnami. Po opětovném připojení se automaticky sesynchronizuje klient se serverem a vyřeší se případné konflikty. Kromě těchto vlastností umožňuje sledovat chování uživatelů, poskytovat analytické a diagnostické informace, provozovat aplikační servery v cloudu a další služby.

Jakákoliv data, která mají spadat pod automatickou synchronizaci, se musí ukládat do jedné ze dvou nabízených databází:

1. **Firebase Realtime Database** je starší, původně jediná nabízená možnost.

2. **Firestore** založený na Firebase Realtime Database implementuje efektivnější a expresivnější dotazy pro přístup k datům a lepší možnosti škálování.

Obě databáze jsou tzv. noSQL a záznamy ukládají ve formátu velmi podobnému JSONu. Implementace synchronizace není veřejně známá, protože se jedná o proprietární technologii [13].

K datům lze přistupovat pouze skrz dodané API poskytované v mnoha programovacích jazycích. Kromě nutnosti použít jednu z proprietárních Firebase databází je důležité přizpůsobit strukturu dat vybrané databázi, avšak novější Firestore nevyžaduje tak striktní formu. Pro maximální efektivitu zodpovídání dotazů se při strukturaci dat doporučuje dodržovat konvence uvedené v oficiální dokumentaci, např. nezanořovat data příliš do hloubky a neshlukovat je do velkých celků – data bychom se měli snažit rozprostřít tzv. horizontálně [14].

Kromě aplikačních dat, která se ukládají do databáze, Firebase umožňuje hostovat také další libovolný obsah, jako obrázky, videa a jiné soubory.

Protože klíčové prvky Firebase není možné provozovat na vlastní infrastruktuře, nelze jinak než využít cloudu. Tato služba je zadarmo pouze do jistého množství požadavků. Požadavky se dělí na zápis, čtení a mazání. Pro každý typ je nastavena denní hranice. Za každé její překročení o konkrétní množství se automaticky účtuje odpovídající částka. Ceník není jednotný, záleží na geografické poloze serverů, které jsou využívány [15].

Při automatické synchronizaci se každé propagování změn právě připojenému klientovi považuje za jeden požadavek na čtení. Provádění mnoha změn v simulaci a jejich okamžitá distribuce klientům rychle zvedne celkový počet požadavků na čtení. Provoz aplikace, která potřebuje pravidelně distribuovat stav v reálném čase se může výrazně prodražit, podle počtu právě připojených klientů. Jednou z mitigací by bylo snížit frekvenci aktualizací.

Další nevýhodou a možným problémem je, že kvůli využití cloudu není plně pod kontrolou, kde a jak jsou data fyzicky uložena. Pokud se pracuje s osobními daty uživatelů EU, může vzniknout problém s GDPR, které vyžaduje, aby osobní data těchto uživatelů byla uložena na serverech na území EU [16].

5.2 Meteor

Framework Meteor obsahuje klientskou i serverovou část zároveň. Pro klientskou část je již v základu podpora pro populární frameworky jako React, Vue, Angular a další. Jako server Meteor využívá Node.js a jako databázi MongoDB. Všechny tyto částí využívají JavaScript. Filozofií Meteoru je mít všechny části aplikace vedle sebe v rámci jednoho projektu pod jedním programovacím jazykem.

Velkou výhodou je, že po první instalaci frameworku a jeho spuštění automaticky fungují všechny komponenty, tj. klientská část, server i databáze. Vývojář se okamžitě může soustředit na implementaci podstatných částí aplikace.

Dalšími poskytovanými funkcemi, kromě automatické synchronizace a fungování aplikace v offline režimu, je možnost z jednoho zdrojového kódu vytvořit aplikaci fungující ve webových prohlížečích i mobilních zařízeních. [17]

Mezi nevýhody lze zařadit nutnost využívat konkrétní technologie, na kterých byl Meteor postaven. Nahrazení jedné z předdefinovaných částí za vlastní s sebou provází různá úskalí. Protože z našeho pohledu po instalaci

"vše nějak funguje" a jednotlivé části jsou na sebe automaticky napojeny, není zřejmé, co všechno se na pozadí děje a jak zde dosadit vlastní řešení.

5.3 Couchbase

Couchbase označuje skupinu produktů, která je dohromady schopná zajistit synchronizaci dat a částečnou offline funkcionalitu. [18]

- Couchbase Server je noSQL dokumentová databáze, ve které se záznamy ukládají způsobem podobnému JSON.
- Couchbase Lite je samostatná integrovaná aplikační verze Couchbase Serveru a API pro práci s daty. Jinými slovy se jedná o lokální offline databázi v rámci klienta.
- Couchbase Sync Gateway je přemostění mezi lokální Couchbase Lite a Couchbase Server databázemi. Na této vrstvě mezi nimi probíhá synchronizace. Gateway také automaticky reaguje na ztrátu spojení během provozu aplikace s dodatečnou synchronizací a vyřešením konfliktů po obnovení spojení.

Opět platí podobná omezení jako u předchozích řešení, kdy synchronizovaná data musí být uložena v databázi.

Výhodou je, že u Couchbase, na rozdíl od Firebase, lze provozovat všechny nutné komponenty na vlastní infrastruktuře.

5.4 SyncIt

SyncIt je JavaScriptová knihovna složená z klientské a serverové části, která se stará o synchronizaci dat mezi klienty a serverem a zároveň umožňuje aplikaci fungovat i v omezeném rozsahu při dočasné ztrátě spojení. Po jeho obnovení se lokálně provedené změny přenesou na server a aktualizuje se lokální stav. Automaticky řeší případné konflikty v datech, ale umožňuje také vývojáři implementovat vlastní strategii řešení konfliktů [19].

Na rozdíl od výše zmíněných řešení zde synchronizace probíhá na vyžádání a data nemusí být předtím uložena do databáze.

Bohužel knihovna se jeví jako neudržovaná. Poslední commit byl proveden v roce 2018 (jednalo se pouze o změnu verze používané knihovny, poslední skutečná změna ve zdrojovém kódu byla provedena v roce 2015) a repositář obsahuje pouze 3 issues, poslední z roku 2017. V repositáři NPM SyncIt dosahuje průměrně 22 týdenních stáhnutí [20].

5.5 Jsynchronous

Jsynchronous je minimalistická knihovna, jež je schopná obalit jakýkoliv existující objekt v JavaScipt, sledovat na něm prováděné změny a ty automaticky synchronizovat. Data není třeba ukládat do databáze, a hlavně

knihovnu můžeme využít pro synchronizaci stavu v paměti serveru, ne pouze dat zapsaných v databázi. Obsažena je klientská i serverová část.

Knihovna neřeší, jak konkrétně danou změnu dopravit ostatním, místo toho se očekává, že uživatel knihovny dodá definici automaticky volané metody, která určuje, co se má s vygenerovanou zprávou udělat. Tím je zpravidla poslání zprávy ostatním jakýmkoliv způsobem, jež vývojář uzná za vhodný.

Zároveň i přijetí a rozpoznání zprávy se u klienta musí dodefinovat samostatně. Data ze zprávy pak už jen stačí předat kódu klientské implementace knihovny.

Díky tomuto řešení je knihovna velice flexibilní a dopřává volnost výběru vlastních technologií.

5.6 Shrnutí

Všechna nalezená existující řešení, kromě SyncIt a Jsynchronous, toho nabízejí mnohem více, než je potřeba, a často si vynucují použití technologií a práci s daty, která nemusí být vhodná pro některé druhy aplikací, a navíc všechny s sebou přináší do jisté míry komplexitu navíc. Prezentované produkty působí dojmem, že se jejich použití vyplatí především pro střední a velké projekty. Pro malé projekty by mělo být možné využít SyncIt a Jsynchronous.

Za další problém se dá považovat nutnost výrazně měnit zdrojový kód, pokud bychom již měli nějakou existující aplikaci a chtěli využít některé z prezentovaných řešení. Ať už se jedná o přechod na konkrétní vyžadované databáze nebo úpravu práce s daty ve zdrojovém kódu. Knihovny nebo frameworky zpravidla stojí na používání jimi poskytovaných metod pro manipulaci stavu. Za předpokladu, že chceme převést aplikaci, která spravuje svůj stav v paměti, museli bychom, kromě instalace databáze a napojení se na ni, při jakékoliv změně stavu explicitně přidat volání na metody, které rozdíly propíšou z lokální paměti aplikace do databáze.

Jedině Jsynchronous umožňuje pohodlný přechod pro aplikace se stavem v lokální paměti bez potřeby velkých zásahů do způsobu práce s daty.

Dále žádné z uvedených řešení nevyzdvihuje, ba vůbec nezmiňuje efektivitu přenosu synchronizačních dat z hlediska množství posílaných dat.

Kapitola 6

Porovnání formátů v prostředí JavaScript

6.1 Specifika prostředí JavaScriptu

6.1.1 Kompilované a interpretované jazyky

Kompilované jazyky bývají výkonnostně rychlejší než interpretované, protože zdrojový kód se musí před spuštěním zkompilovat neboli převést do strojových instrukcí. Tyto zkompilované instrukce lze potom velice rychle provádět. Kvůli jedinečnosti procesorů je ale nutné kompilovat instrukce různými způsoby podle cílové architektury nebo operačního systému. Kompilovanými jazyky jsou např. C, C++, BASIC, Pascal a další.

Pomocí interpretovaných jazyků se zbavíme nutnosti kód nejdříve zkompilovat do několika různých podob. Docílíme toho zavedením abstrakční vrstvy tak, že na naši platformu nainstalujeme na míru vytvořený interpret daného jazyka. Ten pak postupně čte pro všechny platformy jednotný zdrojový kód v "surové" podobě. Úkolem interpretu je za běhu kód překládat do strojových instrukcí srozumitelných pro konkrétní hardware.

Některé interprety jsou schopné zvednout svou výkonnost pomocí techniky Just in Time (JIT) kompilace. Často volané pasáže zdrojového kódu se za běhu programu zkompilují. Místo pomalé interpretace se pak rovnou pouští kompilovaný kód dané části. Důsledkem této optimalizace mohou být zkreslené výsledky výkonnostního testování, když se během opakování měření náhle spustí JIT kompilace. K předejití problému s nepřesným měřením se u testování doporučuje prvně spustit tzv. zahřívací fázi, kdy test nejdříve opakovaně probíhá, aby se interpretu naskytla příležitost aplikovat JIT kompilaci ještě před začátkem měření.

Přes dodatečné kompilování pomocí JIT se lze přiblížit podobné rychlosti jako cíleně kompilovaného kódu, alespoň pro konkrétní části, kde lze JIT aplikovat.

6.1.2 JavaScript a interpret

JavaScript je interpretovaný jazyk, pro který na trhu momentálně existuje několik různorodých interpretů. V internetových prohlížečích se můžeme setkat s integrovanými interprety jako V8 (Google Chrome, Microsoft Edge),

Spidermonkey (Mozilla Firefox) a dalšími. Kromě těchto jsou k dispozici i samostatně použitelní interpreti jako Node.js nebo Deno.

Hlavním problémem existence několika interpretů je jejich rozdílná implementace, která se projevuje různou mírou výkonnosti obecně, ale také konkrétně v některých případech pro specifické instrukce. Dále se může lišit poskytovaná funkcionalita nebo vystavené rozhraní pro jednotlivé části jazyka a funkcionalit.

Prvotním smyslem JavaScriptu bylo umožnit interakci a provádění dynamických změn na internetových stránkách bez nutnosti jejich opětovného načítání. Protože JavaScriptový kód do prohlížeče dostáváme od třetích stran (provozovatelů stránek), hrozí reálné nebezpečí stažení a automatického spuštění závadného kódu. Proto mají interprety v prohlížečích různě omezené pravomoci, díky čemuž je nemožné napáchat škody v systému. Některé standardní funkce programovacích jazyků tak nemusí být v prohlížečích k dispozici.

Když později vznikli samostatní interpreti, u kterých uživatel cíleně a uvědoměle pouští zdrojový kód, bylo nutno doplnit chybějící funkcionalitu, především práci se systémem. Navíc přibyla možnost rozšířit nativní implementaci interpretu o vlastní předkompilovaný zdrojový kód. Uživatelské knihovny a balíčky tak mohou zvednout svou výkonnost za předpokladu, že pro danou verzi interpretu pro konkrétní platformu existuje tato předkompilovaná verze. Tuto možnost interpreti v internetových prohlížečích nenabízí, samostatně fungující Node.js ovšem ano. Přestože pomocí JavaScirptu a Node.js lze pouštět kód s takřka libovolnou funkcí, v průmyslu se velmi často používá k provozování serverových částí webových aplikací, protože pak máme jak klientskou, tak serverovou část v jednom programovacím jazyce.

Protože JavaScript nativně podporuje pouze JSON, implementaci pro ostatní formáty je nutno dodat přes různé knihovny. Ne všechny ovšem fungují jak v Node.js, tak v prohlížečích. Navíc kvůli různé implementaci interpretů nemusí knihovna dosahovat stejné výkonnosti napříč různými prostředími.

6.2 Testování

6.2.1 Metodika testování

Protože pro každý formát existuje více dostupných implementací, byly otestovány ty nejsnáze vyhledatelné a nejvíce používané v rámci NPM.

Testy byly prováděny na malém, středním a velkém objektu. Objekty byly složeny z vnořených objektů, číselných hodnot a polí obsahující další čísla. V rámci každého testu se daný objekt serializoval. Mimo časově měřený úsek se ještě nakonec zpětně deserializoval a zkontrolovalo se, zda deserializovaný objekt je shodný s prvotním objektem. Každé dílčí testy byly celkově zopakovány desetkrát pro zmírnění dopadů odchylek při měření.

- Malý objekt: 560 skalárních veličin - 10 vnořených objektů, v každém 16 klíčů s číselnou hodnotou a další 4 klíče s poli, v každém obsaženo 10

čísel

- Střední objekt: 10 640 skalárních veličin - 190 vnořených objektů, v každém 16 klíčů s číselnou hodnotou a další 4 klíče s poli, v každém obsaženo 10 čísel
- Velký objekt: 104 000 skalárních veličin - 800 vnořených objektů, v každém 30 klíčů s číselnou hodnotou a další 4 klíče s poli, v každém obsaženo 25 čísel

6.2.2 Paměťové testy

Paměťové testy byly provedeny celkově čtyři, kdy čísla nabývala náhodných hodnot, vždy od 0 do 5 tisíc, od 0 do 250 tisíc, od 0 do 5 milionů a od 0 do 500 milionů. Smyslem samotných testů, a jejich opakování pro různé maximální velikosti čísel, je pozorovat efektivitu komprese obecně a v závislosti na odlišných způsobech reprezentace různě velikých čísel mezi formáty. Nehledě na prostředí, ve kterém testy probíhají, by měly být výsledky komprese vždy stejné, proto byly prováděny pouze v Node.js.

6.2.3 Výkonnostní testy

Výkonnostní testy byly opakovány celkem třikrát, a to v interpretu Node.js a v internetových prohlížečích Google Chrome a Mozilla Firefox, aby bylo možné pozorovat rozdíly ve výkonu v těchto prostředích pro konkrétní knihovny a pokusit se vybrat tu obecně nejrychlejší. Rozmezí hodnot čísel v tomto případě nemá na testování citelný vliv. Testy probíhaly shodně jako paměťové, jen se místo velikosti výsledného objektu měřil čas nutný pro serializaci a deserializaci daného objektu.

6.3 Výsledky testování

Kompletní naměřená data jsou k dispozici v příloze v souboru *formatTests.xlsx*.

Tabulky v této kapitole vychází z průměrných hodnot ze všech 10 měření. Další statistické údaje, jako minima, maxima a intervaly spolehlivosti, jsou uvedeny v příloze.

6.3.1 Node.js

Některé z knihoven využívají schopnosti Node.js využít své předkompilované verze, které by měly být rychlejší než jejich interpretovaní konkurenti.

Ze všech možných balíčků a knihoven byly pro testování vybrány následující:

- JSON
  - nativní implementace v JavaScriptu
- BSON

  - bson
  - bson-ext – akcelerovaný nativní knihovnou
- CBOR
  - borc
  - cbor
  - cbor-js
  - cbor-x – akcelerovaný nativní knihovnou
- MessagePack
  - msgpack – akcelerovaný nativní knihovnou
  - messagepack
  - msgpackr – akcelerovaný nativní knihovnou
  - msgpack-lite
  - msgpack5
  - msgpack-js

Výkonnost – naměřené hodnoty

V tabulce 6.1 jsou v prvním řádku zanesené délky trvání jednotlivých operací v milisekundách pro nativní JSON implementaci. V dalších pak relativní hodnoty vyjadřující, kolikrát déle trvalo knihovně provést konkrétní operace v porovnání s JSONem, takže čísla menší než jedna znamenají zrychlení oproti JSONu, čísla větší než jedna zpomalení. První písmeno sloupce označuje velikost objektu (s = malý, m = střední, l = velký), druhé serializaci (s) nebo deserializaci (d). Knihovny používající nativní akceleraci jsou v tabulce 6.1 označené hvězdičkou.

Knihovna	s-s	s-d	m-s	m-d	l-s	l-d
json	0,082 ms	0,065 ms	1,280 ms	0,809 ms	12,166 ms	8,429 ms
bson-ext*	5,66	3,94	6,42	5,75	7,35	7,24
bson	18,06	15,27	5,52	4,01	4,45	3,14
cbor	60,79	70,98	18,33	52,83	19,01	35,09
cbor-js	21,56	10,71	3,21	3,00	1,70	1,12
cbor-x*	9,16	8,57	1,22	1,70	0,73	0,80
borc	44,42	36,92	9,33	5,77	7,06	2,85
msgpack*	3,73	3,28	3,36	4,15	3,61	4,76
messagepack	36,07	74,76	53,16	51,57	296,86	32,43
msgpackr*	8,87	6,91	1,49	1,66	0,75	1,02
msgpack-lite	17,73	15,37	3,46	6,26	1,84	3,47
msgpack5	54,11	50,03	17,45	26,53	18,89	27,15
msgpack-js	297,25	10,85	4,35	4,13	3,49	2,25

Tabulka 6.1: Výkonnostní tabulka pro Node.js

Výkonnost – shrnutí

Pouze dvě knihovny byly schopné předčit výchozí JSON implementaci – cbor-x a msgpackr. Obě jsou od stejného autora a využívají akcelerace přes nativní kompilované knihovny. Lepší výsledky než pro JSON se ale dostavily až u největší instance. Pravděpodobným důvodem je drobná režie navíc při volání metod nativních knihoven, která se stává zanedbatelnou až u větších instancí. Stále lze ale pozorovat vyšší výkon u akcelerovaných knihoven v porovnání s interpretovanými. Zvláštní nicméně zůstává, že akcelerovaná knihovna msgpack jako jediná toto zpomalení u malé instance nezaznamenává.

Z interpretovaných knihoven nejlepších výsledků dosáhly msgpack-lite, bson a cbor-js, který se obzvláště osvědčil pro veliké instance, avšak u menších instancí ztrácí na serializaci, ale získává na deserializaci.

U knihovny msgpack-js bylo zjištěno zvláštní chování – pro malé objekty serializace trvá až nezvykle dlouho. Serializační test pro malé instance byl specificky pro msgpack-js zopakován několikrát se stále stejnými výsledky. V deserializaci v kategorii malých objektů je přitom pravidelně rychlejší než celkově výkonný msgpack-lite.

Akcelerovaná knihovna bson-ext se ukázala jako dobrá pro menší instance, avšak z neznámých důvodů pro větší instance zaostává za interpretovanou verzí knihovny bson.

Celkově nejlepších výkonnostních výsledků, až na výjimky v kategorii velkých objektů, dosahuje výchozí implementace v JSONu. V kategorii výsledné velikosti záznamů již tak dobrých výsledků nedosahuje, viz sekce 6.3.1.

Vybrat nejrychlejší knihovnu mimo výchozí JSON nelze jednoznačně, protože záleží převážně na tom, s jak velkými daty se v aplikaci pracuje.

Uvážíme-li ovšem, že budeme využívat rozdílových aktualizací, u kterých lze očekávat menší rozsah, můžeme zvažovat využití knihoven msgpack, msgpackr a bson-ext, z neakcelerovaných pak msgpack-lite, bson nebo cbor-js.

Přestože pro malé instance jsou všechny neakcelerované knihovny relativně až 20krát pomalejší, absolutně se stále jedná o krátký časový úsek, viz *s-s* a *s-d* pro JSON, tj. 0.082 ms, respektive 0.065 ms.

Paměťová náročnost - naměřené hodnoty

V tabulkách 6.2, 6.3, 6.4 a 6.5 je v prvním řádku velikost serializovaných objektů ve formátů JSON v bajtech. Na dalších řádcích je pak vyznačen podíl výsledné velikosti serializovaného objektu danou knihovnou a JSONu. První písmeno sloupce označuje velikost objektu (s = malý, m = střední, l = velký).

Knihovna	s-size	m-size	l-size
json	4,059 KiB	77,503 KiB	956,766 KiB
bson-ext	123,56%	123,16%	144,36%
bson	123,56%	123,16%	144,36%
cbor	59,87%	60,02%	61,69%
cbor-js	59,87%	60,02%	61,69%
cbor-x	60,39%	60,50%	61,77%
borc	59,87%	60,02%	61,69%
msgpack	52,57%	52,51%	57,85%
messagepack	60,01%	60,19%	62,04%
msgpackr	60,25%	60,38%	62,28%
msgpack-lite	60,01%	60,19%	62,04%
msgpack5	60,01%	60,19%	62,04%
msgpack-js	60,01%	60,19%	62,04%

Tabulka 6.2: Paměťová náročnost – náhodná čísla od 0 do 5 tisíc

Knihovna	s-size	m-size	l-size
json	5,204 KiB	99,035 KiB	1,235 MiB
bson-ext	96,36%	96,38%	109,18%
bson	96,36%	96,38%	109,18%
cbor	65,70%	65,73%	67,62%
cbor-js	65,70%	65,73%	67,62%
cbor-x	66,11%	66,11%	67,68%
borc	65,70%	65,73%	67,62%
msgpack	60,26%	60,09%	64,99%
messagepack	66,05%	66,10%	68,17%
msgpackr	66,11%	66,10%	68,17%
msgpack-lite	66,05%	66,10%	68,17%
msgpack5	66,05%	66,10%	68,17%
msgpack-js	66,05%	66,10%	68,17%

Tabulka 6.3: Paměťová náročnost – náhodná čísla od 0 do 250 tisíc

Knihovna	s-size	m-size	l-size
json	5,966 KiB	113,758 KiB	1,442 MiB
bson-ext	84,06%	83,91%	93,52%
bson	84,06%	83,91%	93,52%
cbor	62,38%	62,42%	63,79%
cbor-js	62,38%	62,42%	63,79%
cbor-x	62,74%	62,75%	63,84%
borc	62,38%	62,42%	63,79%
msgpack	57,65%	57,52%	61,55%
messagepack	62,71%	62,75%	64,26%
msgpackr	62,74%	62,75%	64,26%
msgpack-lite	62,71%	62,75%	64,26%
msgpack5	62,71%	62,75%	64,26%
msgpack-js	62,71%	62,75%	64,26%

Tabulka 6.4: Paměťová náročnost – náhodná čísla od 0 do 5 milionů

Knihovna	s-size	m-size	l-size
json	7,231 KiB	137,846 KiB	1,781 MiB
bson-ext	69,35%	69,25%	75,72%
bson	69,35%	69,25%	75,72%
cbor	51,74%	51,75%	51,91%
cbor-js	51,74%	51,75%	51,91%
cbor-x	52,03%	52,02%	51,95%
borc	51,74%	51,75%	51,91%
msgpack	47,83%	47,70%	50,09%
messagepack	52,01%	52,02%	52,29%
msgpackr	52,03%	52,02%	52,29%
msgpack-lite	52,01%	52,02%	52,29%
msgpack5	52,01%	52,02%	52,29%
msgpack-js	52,01%	52,02%	52,29%

Tabulka 6.5: Paměťová náročnost – náhodná čísla od 0 do 500 milionů

Paměťová náročnost – shrnutí

Formáty MessagePack a CBOR dosáhly podobných kompresních hodnot.

BSON nepřinesl až tak výrazné úspory v porovnání s ostatními formáty. Naopak pro měření s čísly maximálních hodnot do 5 tisíc zabíral více místa pro všechny objekty, u maximálních hodnot do 250 tisíc pouze u velkého objektu. Na vině jsou přidaná metadata, která urychlují navigování výsledných záznamů bez nutnosti jej celý deserializovat, a dále kvůli ukládání čísel s malou hodnotou za použití 4 bajtů, i když by stačilo méně, což se velice negativně projevilo u prvních dvou testů. V porovnání s formáty CBOR a MessagePack vykazuje nejhorší výsledné hodnoty testu.

Je s podivem, že přestože by všechny CBOR a MessagePack knihovny měly odpovídat standardům, velikosti výsledných záznamů v rámci stejné kategorie

velikosti objektu se drobně lišily napříč všemi čtyřmi testy. Nejznatelnější jsou rozdíly pro kategorii malých objektů (s-size), konkrétně u knihoven cbor-x, msgpack a msgpackr lze pozorovat rozdíly od velikosti výsledných záznamů ostatních implementací daného formátu. U knihovny msgpackr jsou rozdíly nejméně významné. Kromě testu s náhodnými čísly od 0 do 5 tisíc (tabulka 6.2) lze odchylky pozorovat pouze v kategorii malých objektů (s-size).

Rozdíly ve výsledné velikosti ovšem nejsou moc výrazné, až na msgpack, který pro malou a střední instanci v porovnání s ostatními implementacemi MessagePacku ušetřil nezanedbatelné množství dat.

I přes rozdíly byla stále zachována vzájemná kompatibilita mezi knihovnami daných formátů, kdy byly schopny správně rekonstruovat původní objekt serializovaný jinou knihovnou, nehledě na rozdílné velikosti.

6.3.2 Mozilla Firefox 95.0.2

V prostředí prohlížeče byla testována pouze výkonnost.

Z testu byly vyřazeny balíčky bson-ext a msgpack, protože v prohlížeči není možné využít nativní kompilované knihovny, na kterých závisí.

Akcelerované balíčky cbor-x a msgpackr jsou schopné přepnout v prohlížečích na svou interpretovanou verzi.

Naměřené hodnoty

Jako u tabulky 6.1, v tabulce 6.6 čísla menší než jedna znamenají zrychlení oproti JSONu, větší než jedna zpomalení.

Knihovna	s-s	s-d	m-s	m-d	l-s	l-d
json	0,103 ms	0,040 ms	0,957 ms	0,657 ms	11,251 ms	7,094 ms
bson-ext	-	-	-	-	-	-
bson	8,53	13,06	3,44	3,05	3,31	2,60
cbor	96,83	185,66	164,24	171,36	207,17	195,86
cbor-js	4,188	8,553	17,387	13,910	4,544	4,229
cbor-x	12,88	5,72	1,99	1,26	1,53	1,54
borc	49,05	46,07	85,44	11,48	41,99	6,13
msgpack	-	-	-	-	-	-
messagepack	55,39	29,08	511,99	14,10	5729,40	10,89
msgpackr	2,68	5,14	4,69	1,46	2,86	1,24
msgpack-lite	6,46	7,99	2,92	2,34	1,71	2,53
msgpack5	67,28	106,37	130,41	121,52	130,79	376,05
msgpack-js	8,62	12,90	153,48	7,45	30,79	6,98

Tabulka 6.6: Tabulka výkonnosti pro Mozilla Firefox 95.0.2

Shrnutí

Ve Firefoxu pro formát MessagePack dopadla nejlépe knihovna msgpackr ve své interpretované variantě, pokud počítáme s prací převážně na malých

objektech. Jestliže bude v aplikaci třeba převážně serializovat větší objekty, stojí za zvážení msgpack-lite, jež v kategorii serializace středních a velkých objektů dosahuje lepších výsledků než msgpackr.

Jediný zástupce formátu BSON, knihovna bson, byla sice rychlejší než většina ostatních testovaných knihoven, ale v obou formátech CBOR a MessagePack jsou zástupci, kteří vykazují lepší výsledky než bson ve všech kategoriích, až na serializaci malých objektů (u knihovny cbor-x) a serializaci středního objektu (u knihovny messagepackr).

U CBORu nejvyšší rychlosti dosahuje cbor-x v interpretované variantě, ale vykazuje až 13násobné hodnoty v porovnání s nativní implementaci JSON při serializaci malých objektů, což je jediná kategorie, kde je cbor-js rychlejší než cbor-x.

Balíček messagepack se vyznačuje výrazně pomalejší serializací objektů. Pro největší objekt dosáhl dobou serializace až na závratných 60 sekund.

6.3.3 Chrome 96.0.4664.110

Obdobně jako u Mozilla Firefox, byla testována pouze výkonnost a z testu byly vyřazeny balíčky bson-ext a msgpack.

Internetové prohlížeče omezují přesnost měření času, aby pomohly předcházet speciálnímu druhu útoku zvanému Spectre [21]. V prohlížeči Mozilla Firefox lze toto úmyslné zkreslení měření vypnout, bohužel Google Chrome podobnou možnost nenabízí. Nejvyšší přesnost měření je tak pouze na setiny milisekund.

Naměřené hodnoty

Výsledky měření jsou zaneseny do tabulky 6.7. V tabulce jsou hodnoty pro sloupec s-d jsou až neobvykle nepříznivé pro všechny knihovny. S přihlédnutím k rychlosti deserializace pro menší objekt je pravděpodobné, že se tento úkaz objevil v důsledku snížené přesnosti měření. Zaokrouhlení na setiny milisekund mohlo pro JSON zakrýt, vzhledem k celkové délce této operace (0.02 ms), poměrově dlouhý časový úsek v řádu tisícin sekundy.

Shrnutí

Na rozdíl od Firefoxu, v Google Chrome byly některé knihovny rychlejší než nativní JSON implementace, a to v kategorii největších objektů, jak pro serializaci, tak deserializaci (sloupce l-s a l-d).

Podobně jako u Firefoxu, i zde nejvyšší rychlosti dosahují knihovny cbor-x pro CBOR a msgpackr pro MessagePack. Rozdílem je, že u Firefoxu se byl msgpack-lite schopný vyrovnat výkonu balíčku msgpackr v serializaci, ve Chrome ale není rychlejší než msgpackr v žádné kategorii. Dále knihovna cbor-x není zatížena dlouhým "startem" pro serializaci malých objektů v porovnání s výsledky testování ve Firefoxu.

Knihovna	s-s	s-d	m-s	m-d	l-s	l-d
json	0,090 ms	0,020 ms	1,330 ms	0,690 ms	10,680 ms	6,670 ms
bson-ext	-	-	-	-	-	-
bson	12,00	43,50	3,44	4,57	4,36	5,33
cbor	108,11	297,00	71,59	100,57	119,94	105,71
cbor-js	9,444	19,500	2,444	3,116	1,843	1,637
cbor-x	4,56	19,50	1,29	1,48	0,75	0,79
borc	43,78	84,50	17,97	4,42	16,09	3,22
msgpack	-	-	-	-	-	-
messagepack	24,00	98,50	63,34	27,07	518,95	20,58
msgpackr	4,11	17,00	1,77	1,72	0,77	0,96
msgpack-lite	9,22	29,00	2,17	4,96	2,00	3,85
msgpack5	67,00	211,50	63,76	92,49	74,20	101,33
msgpack-js	17,22	33,00	7,27	7,70	7,16	6,65

Tabulka 6.7: Tabulka výkonnosti pro Chrome 96.0.4664.110

6.3.4 Celkové shrnutí

Vezmeme-li v potaz výkonnost knihoven napříč testovanými prostředími, msgpackr dosáhl nejlepších výsledků. V závěsu je poté cbor-x, jehož nevýhodou je pomalejší serializace malých objektů ve Firefoxu.

BSON bohužel kvůli nízké, někdy až neexistující úspoře objemu dat, není vhodné používat pro účely rozdílových aktualizací. Přestože je CBOR v porovnání s MessagePackem mírně efektivnější v kompresi dat obecně (okolo 1 %), použitím konkrétního balíčku pro MessagePack – msgpack – se naopak CBOR stává méně efektivní. Nevýhodou je, že msgpack funguje pouze v prostředí Node.js, kde je navíc pro práci s malými objekty nejrychlejší. Postupně začíná ztrácet až s přibývající velikostí zpracovaných dat. Díky vzájemné kompatibilitě knihoven pro formát MessagePack, přestože produkují výsledky jiné velikosti, je možné msgpack použít na serverové straně a na straně klientů použít k deserializaci jinou MessagePack knihovnu.

Pro maximalizaci rychlosti a úspory dat nejlépe vychází na serveru použít msgpack a na klientech msgpackr. Pokud bude upřednostněna vyšší rychlost pro práci s velkými objekty před vyšší úsporou balíčku msgpack v zájmu vyšší rychlosti práce s velkými objekty, vyplatí se msgpackr použít i na serverové straně.

Kapitola 7
Implementace

Zdojový kód je dostupný v příloze v archivu "newsync.zip", nebo v GitHub repositáři přístupném pod url adresou `https://github.com/neuvimat/newsync`. Všechny soubory frameworku jsou uvnitř složky /lib.

Součástí repositáře je kromě zdrojového kódu frameworku také zdrojový kód výkonnostních testů, demo aplikace popsané v kapitole 8 Demo aplikace a příklady používaní frameworku.

7.1 Obecný popis frameworku NewSync

Na základě zadání a výsledků z testování konceptu vznikl framework NewSync, jehož cílem je usnadnit synchronizaci mezi serverem a klientem v prostředí webových aplikací. Součástí frameworku je klientská a serverová verze, obě napsané v jazyce JavaScript. Klientská část je přizpůsobena pro fungování v internetových prohlížečích, serverová pro interpret Node.js.

Hlavním úkolem je minimalizovat velikost posílaných zpráv rozdílovými aktualizacemi sestavovanými automatickou detekcí změn v simulaci. Programátor nemusí frameworku změny nijak tlumočit, aby byly rozpoznány. Proces detekce ovšem může urychlit poskytováním nápověd a správným zacházením s daty, případně využitím poskytnutých frameworkových metod.

Jednou z předností je možnost integrace do již existujících aplikací. Framework není závislý na konkrétních technologiích a díky automatické detekci změn je výrazně omezen rozsah nutného zásahu do již existujícího kódu pro napojení stavu simulace do správy frameworku. Vhodné aplikace pro migraci na NewSync jsou ty, které pracují se stavem simulace ve formě objektů, které se serializují do formátu JSON, protože NewSync svazují podobná omezení, viz 7.2.1 Kruhové reference.

Kromě posílání rozdílových aktualizací framework navíc data serializuje do formátu MessagePack, který v předešlém testování dosahoval nejlepších výsledků. Dále dochází k redukci zpráv s využitím slovníku klíčů a kompresí změn v datových strukturách objektů a polí.

Obsluhované zprávy nejsou omezené pouze na ty generované frameworkem. K dispozici je i mechanismus uživatelských zpráv a událostí, jež je úmyslně navržený tak, aby emuloval již existující API dvou populárních technologií pro navázání spojení v prostředí prohlížečů, tj. WebSocketu a knihovny Socket.io.

Programátor tak může přes framework posílat vlastní aplikační data, která jsou automaticky převáděna do formátu MessagePack s využitím slovníku dlouhých klíčů.

7.2 Omezení

7.2.1 Kruhové reference

Kruhová reference je výstižným názvem pro stav, kdy pomyslný objekt A má v sobě referenci na objekt B, B na objekt C a C na objekt A. Vzniklá struktura referencí tvoří kruh. Například formát JSON není schopný serializovat stav aplikace, ve kterém se kruhová reference vyskytuje, protože během postupného procházení objektů a jejich serializování se vytvoří nekonečný cyklus.

Ze stejného důvodu NewSync nepodporuje kruhové reference, protože se při jejich výskytu během sepisování synchronizační zprávy a mapování struktury změn zacyklí podobným způsobem jako JSON.

Kruhových referencí se lze zbavit nahrazením reference na konkrétní objekt jeho unikátním identifikátorem. Při deserializaci zprávy se pak místo identifikátoru dosadí reference na specifický objekt.

7.2.2 Různé typy spojení současně

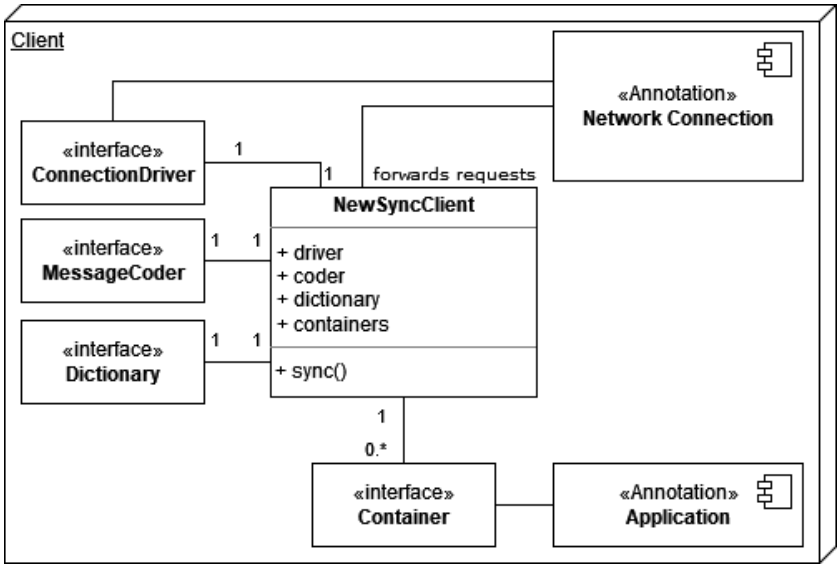
Ve stávající verzi frameworku není možnost sdílet data simulace přes různé druhy spojení současně. Využít lze jeden ze dvou již existujících ovladačů pro WebRTC nebo WebSocket, ale ne oba v jednu chvíli. Vývojář si v současném stavu musí vybrat jeden z dostupných ovladačů, nebo si vytvořit vlastní, ať pro úplně jiný druh spojení, nebo který podporuje více typů spojení zároveň.

7.2.3 Jednotná synchronizace

Synchronizace stavu pomocí odeslání rozdílů se musí aplikovat pro všechny klienty. Vývojář může nastavit frekvenci synchronizačního cyklu nebo jej spouštět manuálně, ale změny se vždy odešlou všem klientům. Momentálně není možné z technických důvodů pro různé klienty nastavit různé prodlevy v synchronizaci nebo nechat klienty, ať si sami žádají o konkrétní čas pro aktualizace.

7.3 Síťový model

Framework předpokládá síťový model, v němž figuruje autoritativní server (který je jediným zdrojem pravdy), a na něj napojení klienti. Ti mohou na server posílat požadavky, jimiž interagují se simulací. Server v pravidelných intervalech posílá aktuální stav simulace, nebo jeho části, všem klientům, kteří o něj aktuálně jeví zájem. Klienti mohou být napojeni na serverovou část frameworku, přesto v daný moment nevyžadovat aktualizace stavu. Stále ale mohou naslouchat uživatelským zprávám. Technicky tak lze NewSync



Obrázek 7.1: Zjednodušený diagram zobrazující použití frameworku NewSync v klientské části aplikace.

využít pouze pro kompresi aplikační komunikace bez využití automatických synchronizačních mechanismů.

Framework není závislý na konkrétním způsobu propojení účastníků. Je úkolem programátora, aby zajistil navázání spojení a zprostředkoval doručení vygenerovaných synchronizačních zpráv. Pro usnadnění posílání zpráv jsou součástí frameworku dva síťové ovladače, které tuto část výrazně zjednodušují.

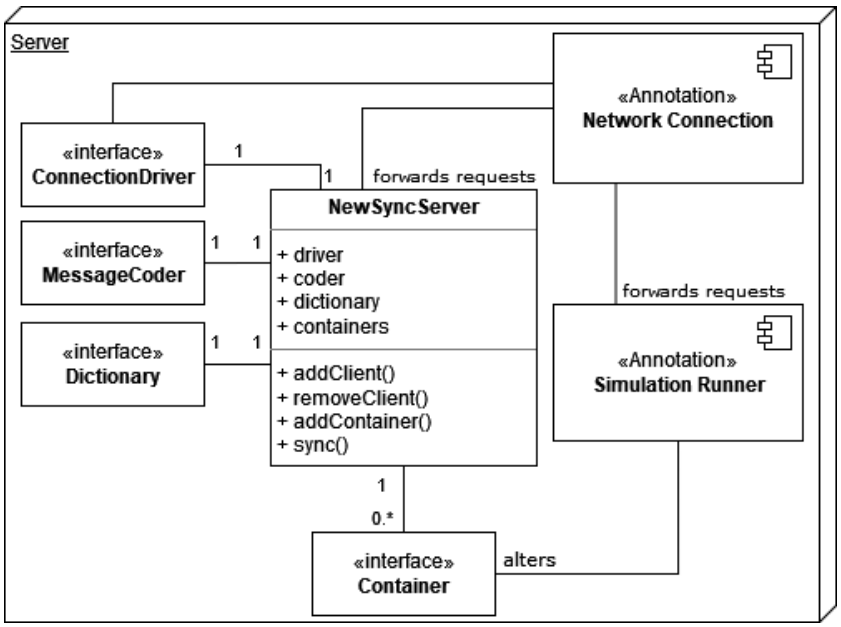
7.4 Komponenty frameworku

Základními stavebními kameny frameworku jsou: instance třídy NewSync, ovladač síťového spojení, kodér zpráv, kontejnery a slovník dlouhých klíčů. Jednotlivé části jsou dále rozvedené v následujících sekcích. Dále framework pracuje s komponentami pro automatické detekování změn, s těmi ale uživatel frameworku nepřijde do přímého kontaktu. Blíže jsou popsané v sekci 7.5 Detekce změn.

Framework je koncipován tak, aby si vývojář mohl implementovat všechny podpůrné komponenty svým vlastním způsobem, dokud dodrží předepsané rozhraní. Zpravidla bude třeba, aby si vývojář napsal vlastní verzi síťového ovladače, rozhodne-li se využít jiného typu připojení než WebSocket nebo WebRTC.

7.4.1 NewSync instance

Třída NewSync existuje ve dvou variantách, jedna pro roli klienta (NewSyncClient) a druhá pro server (NewSyncServer). Pro uživatele frameworku se jedná o hlavní přístupový bod k jeho funkcím. Třída je ztělesněním návrhového vzoru fasáda, jejíž cílem je odstínit vývojáře od komplexity a provázanosti



Obrázek 7.2: Zjednodušený diagram zobrazující použití frameworku NewSync v serverové části aplikace.

několika vnořených komponent vystavením jednoduchého rozhraní, jež řeší vzájemné interakce celého systému na pozadí.

Konfigurace NewSyncu probíhá pomocí injektování konkrétních implementací různých komponent do konstruktoru. Jednotlivé komponenty se konfigurují předáním parametrů při jejich inicializaci.

Na straně serveru i klienta vývojář používá NewSync instanci pro volání synchronizačního cyklu, specifikaci uživatelských zpráv a událostí (viz sekce 7.7 Uživatelská data), zpracování příchozích synchronizačních zpráv a vytváření a odebírání kontejnerů.

NewSync ve své serverové variantě navíc uchovává a spravuje seznam všech připojených klientů. Zprávy obsahující změnu stavu musí obdržet vždy všichni klienti, viz 7.2.3 Jednotná synchronizace. U uživatelských zpráv lze vyčlenit konkrétní příjemce.

7.4.2 Kontejner

Veškerý aplikační stav, jež se má automaticky sdílet a synchronizovat klientům, je reprezentovaný kontejnerem. Kontejner je objekt, jehož vnitřní stav je automaticky monitorován, přičemž ho lze libovolně upravovat. Jakákoliv změna uvnitř kontejneru se u něj zaznamená do seznamu změn. Jakmile nastane synchronizační cyklus, NewSync si vyžádá od kontejneru jednotlivé změny a sestaví na jejich základě synchronizační zprávu, jež se skládá pouze z posledních relevantních změn. Pokud se jeden atribut během synchronizačního cyklu několikrát změnil, aby byl nakonec z kontejneru odstraněn, ve zprávě bude obsažena informace pouze o konečném stavu atributu, tedy o jeho smazání.

Kontejnerů může existovat v jednu chvíli libovolné množství. Klienti si mohou vybrat konkrétní kontejnery, jejichž stav budou přijímat a synchronizovat. Proto by kontejnery měly reprezentovat samostatnou disjunktní podmnožinu celkového stavu aplikace. Pokud se data uvnitř kontejnerů prolínají, nebo jeden kontejner odkazuje na data jiného kontejneru, je na vývojáři, aby ošetřil situaci, kdy klient neodebírá odkazovaný kontejner a předešel chybě u klienta vzniklou absencí některých dat.

Synchronizace kontejnerů probíhá v automatizované formě pouze směrem od serveru ke klientům. Klienti z důvodu zabezpečení a zachování konzistence nemohou provádět změny v simulaci přímou manipulací dat v kontejnerech. Vývojář musí vyvinout svoje vlastní rozhraní pro manipulaci stavu, přičemž pro přenos požadavků může využít systém uživatelských zpráv v NewSyncu.

Kontejnery kromě vlastnosti, jež reprezentuje stav určený k synchronizaci, vystavují další pomocné atributy, jejichž manipulací lze ovlivnit proces synchronizace a zápisu změn, viz sekce 7.5 Detekce změn.

7.4.3 Síťový ovladač (driver)

Ovladač je realizací návrhového vzoru strategie. Vždy, když dochází k interakci se síťovým připojením, NewSync instance se odkazuje na funkcionalitu ovladače. Aby bylo možné začlenit komunikaci do již existujícího WebSocket připojení i v případě, kdy přes něj proudí i jiná aplikační data než ta generovaná NewSyncem, připravený WebSocketDriver umožňuje automaticky přidat před NewSync zprávy konfigurovatelnou předponu pro identifikaci frameworkových zpráv. Ovladač dále vystavuje metodu pro detekci zvolené předpony a identifikaci synchronizačních zpráv.

Ovladač pro WebRTC možnost předpon nepodporuje, protože v rámci jednoho WebRTC připojení lze vytvořit libovolné množství samostatných datových kanálů. Místo zavedení předpony stačí ovladači předat jemu vyhrazený kanál.

7.4.4 Kodér zpráv

NewSync je sice postaven na formátu MessagePack, ale kdyby měl vývojář potřebu využít jiný formát, může tak snadno učinit vytvořením si vlastního kodéru zpráv. Kodér musí implementovat pouze dvě metody rozhraní, jednu pro zakódování a druhou pro dekódování zprávy. Ve frameworku je v základu připravený MessagePackCoder. Kodér zpráv je zpravidla pouze realizací návrhového vzoru adaptér, který umožní instanci třídy NewSync pracovat s různými formáty pod jednotným rozhraním.

7.4.5 Slovník dlouhých klíčů

Princip

Protože framework používá formát bez schématu (MessagePack), jednotlivé vlastnosti přenášených objektů musí být označené klíči. Vývojář zpravidla

volí takovou podobu klíče, která výstižně popisuje danou vlastnost a dělá kód čitelným. Ale každý znak v klíči zvyšuje délku výsledné přenášené zprávy. Proto, než se zpráva odešle, projdou se v ní všechny obsažené klíče a nahradí se kratšími variantami. Při přijmutí zprávy klient automaticky vrátí všechny krátké klíče do jejich původní podoby. Celý proces se děje transparentně na pozadí. Vývojář při práci s daty vždy pracuje s původním zněním klíčů.

Aby klient věděl, jak klíče rekonstruovat, součástí synchronizačních zpráv je i slovník. Klient při prvním připojení k serveru obdrží plné znění slovníku, poté přijímá pouze nově přidané překlady během běhu aplikace. Na začátku komunikace klient sice přijme více dat kvůli plnému znění slovníku, ale s postupně zvedajícím se počtem obdržených zpráv se začne projevovat úspora dat. Efektivita slovníku je silně závislá na počtu unikátních klíčů, se kterými se v aplikaci pracuje, a na jejich průměrné délce a frekvenci výskytu. Lze očekávat, že struktura objektů je statického charakteru, a že se ve zprávách budou z tohoto důvodu klíče často opakovat.

Aby slovník správně fungoval, musí se zajistit jednoznačné mapování oběma směry z krátkého klíče na dlouhý a naopak. Dále pro nejvyšší možnou efektivitu musí být kódová slova co nejkratší.

Kódová slova

Protože JavaScript umožňuje, aby byl klíč jakýkoliv textový řetězec, krátké klíče využívají pouze speciální znaky, u kterých je minimální pravděpodobnost výskytu v klíčích při běžné práci s daty.

Jednotlivé znaky jsou generovány pomocí metody `String.fromCharCode()`. WebSocket i WebRTC přenáší text v kódování UTF-8 [22][23]. V tomto kódování mají různé znaky různou délku zápisu v bajtech. Znaky reprezentované kódem 0 až 127 zabírají jeden bajt, znaky od 128 do 2047 dva bajty, znaky od 2048 do 65535 tři bajty a znaky od 2048 do 1 114 111 čtyři bajty [24].

Všechny alfanumerické znaky jsou v kódovém rozmezí 30 až 122, zabírají tedy pouze jeden bajt. Slovník sleduje, jaké znaky má k dispozici pro následující krátký klíč, a proto překládá pouze ty klíče, u nichž se překladem ušetří nějaké bajty.

Slovník pracuje pouze se znaky v kódovém rozpětí 200 až 55295. Jakmile se vyčerpají všechna jednoznaková kódová slova, začnou se používat slova o dvou znacích, poté o třech atd. Znaky od kódu 200 je obtížné napsat na běžné klávesnici, proto lze očekávat, že se nebudou objevovat ve jménu nějakého klíče. Navíc, jakékoliv jiné než alfanumerické znaky nelze použít v běžném zápisu klíčů v tečkové notaci. Když klíč obsahuje nestandardní znaky, musí se použít notace využívající hranaté závorky obsahující uvozovkami, nebo apostrofy, uvozený textový řetězec.

Horní hranice 55295 byla vybrána, protože po ní následuje sekvence netisknutelných znaků, které v běžném prostředí nejsou reprezentované symbolem. Konkrétně například znak 55296 není ani v prohlížeči Firefox, ani v Node.js zobrazitelný, místo toho je nahrazen zástupným symbolem v Node.js, v prohlížeči poté zapsán jako “\ud800”. Pokud by se takový znak přenášel ve formátu

JSON, bude reprezentovaný zmíněnou sekvencí, tedy dohromady 6 alfanumerickými znaky (tedy 6 bajty), i když v binárním zápise stále zabírá pouze 3 bajty. Pro binární formát MessagePack toto nepředstavuje problém, ale protože NewSync umožňuje použít libovolný formát implementací vlastního kodéru zpráv, byla zvolena tato horní hranice.

7.5 Detekce změn

Detekce změn může probíhat automaticky, ale za cenu jisté režie navíc. Framework poskytuje i možnost provádět změny stavu bez automatické detekce, přičemž změny pak musí vývojář zaznamenat manuálně. Dále jsou poskytnuté doprovodné metody pro asistované změny stavu, jejichž použitím je framework schopný různé změny popsat a zpracovat efektivněji.

Kontejner vystavuje následující důležité vlastnosti pro práci se svým stavem:

pristine: reference na stav kontejneru, přičemž prováděné změny nebudou automaticky zapsány.

proxy: proxy reference na stav kontejneru. Změny prováděné přes tuto referenci se automaticky detekují.

merges: seznam klíčů a hodnot, které je nutné sloučit do minulého výchozího snímku k dosažení synchronizace.

deletes: seznam klíčů, které je nutno odstranit z výchozího snímku k dosažení synchronizace.

meta: seznam klíčů, hodnot a konkrétního typu operace, který je nutný provést ve výchozím snímku k dosažení synchronizace.

Protože operace pro sloučení a mazání klíčů jsou ty nejčastější, je pro ně vytvořena vlastní kategorie. Při zpracování synchronizační zprávy tak lze bez rozmyslu aplikovat danou operaci pro jednotlivé položky ve specifickém seznamu. Pokud zrovna např. není třeba mazat žádnou položku, seznam "deletes" bude v synchronizační zprávě chybět.

Specifičtější operace, převážně ty pro práci s poli, jsou umístěné ve vlastnosti kontejneru "meta". Zde zapsané změny obsahují identifikátor operace a další různé parametry (např. pro kopírování položky v poli jsou parametry zdrojový a cílový index).

7.5.1 Automatická detekce změn

Automatická detekce je postavená na Proxy API, které umožňuje vytvořit proxy referenci na daný objekt (obalit jej) a definovat pro referenci tzv. handler. Původní objekt a reference na něj zůstává nedotknuta, vytvořením proxy vznikne nová reference na původní objekt. Kdykoli se provádí operace na objektu přes proxy referenci, proxy danou operaci zachytí a spustí jednu z 13 tzv. *traps* (pastí) [25], na kterou poskytnutý handler libovolně reaguje.

Může si například pouze zaznamenat, že daná operace proběhla, upravit její efekt, nebo ji zamítnout.

V kontejnerech je původní objekt reprezentující jeho stav dostupný pod klíčem “pristine”, proxy reference na tento stav pod klíčem “proxy”.

Implementace handleru

Handler může kromě definice metod volaných při zachycení jednotlivých pastí obsahovat i jakékoliv další vlastnosti, na jejichž základě lze měnit jeho chování i později za běhu aplikace. Každý handler v sobě obsahuje referenci na původní objekt a sekvenci klíčů, jež určuje cestu k původnímu objektu ve stavu kontejneru.

Pokud jsou ve stavu kontejneru vnořené objekty, pro každý se musí rekurzivně vytvořit nová proxy a handler, protože proxy vždy snímá operace prováděné pouze na své úrovni. Při přístupu k objektu pod cestou “state.nestedObject.value”, přičemž “state” je objekt obalený proxy a “nestedObject” není obalený, zachytí se pomocí pasti pouze přístup k objektu pod klíčem “nestedObject”, ale jakékoliv změny prováděné na něm by se nezaznamenaly.

Rekurzivní instancování nových proxy a jejich handlerů se děje automaticky pomocí techniky *lazy loading*. To znamená, že objekty se obalí do proxy až při prvním přístupu k nim. Nemůže se stát, že by nějaká prováděná změna byla opomenuta, protože při nastavení hodnoty v objektu, který ještě není obalený do proxy, se k němu nejdříve musí přistoupit. V tu chvíli se vytvoří proxy reference, pokud dosud neexistovala. Pokud se do proxy reference přiřadí objekt, jež má v sobě vnořené další objekty, na úrovni proxy se jako změna zaznamená přiřazení celého objektu včetně vnořených objektů, i když ty zatím zůstávají ve své původní, neobalené podobě.

Protože kontejner vždy poskytuje možnost procházet stavem i bez automatické detekce, tedy pomocí původních, do proxy neobalených referencí, nemůže nahradit původní reference za jejich proxy variantu, jinak by se znemožnil přístup k objektu bez využití automatické detekce změn. Aby se nemusela vytvářet druhá variantu stavu, jež by zrcadlila původní strukturu, kde by původní reference byly nahrazené za proxy reference, vždy, když se objekt poprvé obalí do proxy, se k němu pomocí symbolu přidá nová vlastnost, jež odkazuje na jeho proxy variantu.

Symbol je v JavaScriptu speciální druh primitivní hodnoty, kterou lze použít jako klíč. Smyslem symbolu je vytvoření unikátního klíče, k jehož hodnotě nejde přistoupit jinak než přes referenci v kódu na daný symbol. Symboly navíc nejsou iterovatelné ve standardní “for in” a “for of” smyčce a nepodléhají serializaci. K objektům lze takto přidat vlastnosti, které nemohou kolidovat s vlastnostmi reprezentujícími stav objektu a ani nejsou součástí synchronizačních zpráv.

Všechny handlery během přístupu k vlastnostem objektů vždy využívají proxy variantu referencí, které jsou uložené v symbolu u původního objektu.

Díky této skutečnosti lze jednoduše přepínat mezi procházením proxy verzí stavu s automatickou detekcí pouze tím, zdali se pro přístup ke stavu kontejneru použije původní reference “pristine”, nebo proxy verzi reference pod vlastností “proxy”.

Náročnost proxy

Provádění operací na proxy verzi objektu s sebou vždy přináší režii navíc, i když je handler prázdný (nedefinuje žádné vlastní chování na jednotlivé pasti a v podstatě “nic nedělá”). Protože největší zpomalení způsobuje už jenom to, že je použita proxy, samotná implementace metod volaných při spuštění pasti se z relativního úhlu pohledu na zpomalení výrazně nepodílí.

Během výkonnostního testování pouhé použití proxy verze objektu s prázdným handlerem způsobilo v krajních, speciálně připravených případech, až 250násobné zpomalení, viz sekce 9.2.

Optimalizace proxy

Největší zpomalení u proxy způsobuje opakovaný přístup k vlastnostem objektů nebo nastavování jejich hodnot. Vývojář může zrychlit běh aplikace tak, že si bude ukládat reference na objekty, s kterými je třeba opakovaně pracovat, do pomocných proměnných. Pokud je třeba nahradit všechny vlastnosti nějakého objektu, může se provést pouze jedna změna tak, že se vytvoří nový objekt, a ten jako celek nahradí minulý objekt, místo aby se musely postupně projít všechny jeho vlastnosti v rámci jednotlivých změn.

Volání metod na proxy objektech způsobuje, že i volaná metoda, pokud manipuluje se stavem daného objektu, přistupuje k jeho vlastnostem pomocí proxy reference. Pokud metoda pouze čte vlastnosti objektu, je zbytečné, aby k nim přistupovala pomocí proxy. Pokud metoda mění stav objektu, pravděpodobně je požadované změny zaznamenat, pak se hodí, že metoda automaticky využije proxy.

Jelikož NewSync si v handlerech uchovává referenci na neobalenou verzi objektu, může metodu přesměrovat tak, aby k přístupu nebo ke změnám v objektu proxy nepoužívala. Této možnosti se využívá hlavně při volání nativních metod na polích, kdy proxy ve skutečnosti zavolá upravenou, tzv. obalenou, variantu konkrétní metody. Protože je dopředu známo, jaký účel daná metoda má a jaké změny její volání provede, obalená metoda zavolá nativní metodu na původním objektu bez automatické (a kvůli proxy pomalé) detekce změn a změny zaznamená přímo obalená metoda, nikoliv handler v proxy, jehož volání způsobuje implicitně jistou míru zpomalení.

Pokud je v aplikaci při spuštění nutné nastavit výchozí stav v kontejnerech, lze tak bezpečně provést přes “pristine” referenci, jelikož při spuštění aplikace lze předpokládat, že v tu chvíli nebudou připojeni žádní klienti. Ti při prvním navázání spojení vždy dostanou plné znění stavu, nehledě na (v tento moment zbytečně) zaznamenávané změny.

Pokud vývojář potřebuje data ve stavu pouze číst bez vedlejších efektů, v kontejneru lze bezpečně použít vlastnost “pristine”.

7.5.2 Manuální zapisování změn

Pokud chce vývojář provést změnu bez automatické detekce, musí v rámci kontejneru přistoupit k vlastnosti “pristine”, jež vystavuje referenci na původní objekt, nikoliv na jeho proxy verzi. Poté manuálně zapíše danou změnu pomocí doprovodných kontejnerových metod, nebo přímou úpravou kontejnerové vlastnosti “merges” nebo “deletes”. Vývojář si musí dát pozor, aby vlastnoručně zanesené rozdíly byly konzistentní se skutečně provedenými změnami, jinak dojde k desynchronizaci.

7.6 Formát synchronizační zprávy

Protože na nejvyšší úrovni je struktura synchronizační zprávy vždy stejná, je reprezentovaná polem, kdy každý index má pevně daný význam. MessagePack pro krátká pole o maximální délce 15 položek využívá pouze jeden bajt pro uvození. Poté následují konkrétní hodnoty, čímž se minimalizuje velikost zprávy. Pokud by se využilo dvojic klíčů a hodnot, musí se ke každé hodnotě přičíst minimálně dva bajty: jeden pro označení začátku znění klíče a alespoň jeden znak s minimální velikosti jednoho bajtu pro identifikaci klíče.

Výhoda dvojic klíčů a hodnot je, že libovolné části zprávy mohou chybět. Pokud použijeme pole a součástí zprávy zrovna není nějaká položka uprostřed, musí se vždy dosadit aspoň prázdná hodnota jako odsazení, aby zůstalo zachováno pořadí hodnot v rámci indexů. Výjimkou je případ, kdy se nepotřebná položka nachází na konci pole. Z tohoto důvodu jsou v synchronizační zprávě položky s nejméně častým výskytem indexovány na konci pole. Kvůli struktuře zprávy stačí, aby její součástí byly alespoň dvě z celkových pěti položek, a už se projeví úspora využití pole oproti mapě[1].

V rámci zprávy se posílají informace o změnách v kontejnerech, vnitřní příkazy pro NewSync (např. informace o odběru kontejneru klientem, vytvoření nového kontejneru atd.), uživatelské zprávy, uživatelské události a informace o slovníku.

7.7 Uživatelská data

NewSync umožňuje vývojáři posílat jakákoli vlastní data libovolné struktury a obsahu, jež se automaticky balíčkují a posílají v rámci synchronizačních zpráv. Pokud je třeba data odeslat okamžitě bez zbytečných prodlev, lze využít variantu pro okamžité odesílání. Zprávy může posílat server klientovi a naopak.

Hlavní výhoda systému uživatelských zpráv spočívá v automatickém formátování zpráv do MessagePacku (popřípadě vlastního kodéru zpráv) a aplikaci slovníku dlouhých klíčů. Vývojář tak může snadno využít komprese objemu dat pro svá vlastní data.

[1] 1 bajt uvození pole + samotné hodnoty (maximálně 4 bajty režie navíc pro odsazení poslední hodnoty v poli) vs. 1 bajt pro uvození mapy + 2krát dva bajty pro uvození 2 klíčů.

Uživatelská data jsou dvojího typu: zpráva a událost. Rozdíl je pouze v rozhraní pro jejich zpracování.

7.7.1 Zprávy

Zprávy využívají rozhraní podobné WebSocketu nebo datového kanálu v WebRTC. Vývojář na instanci NewSync definuje vlastní implementaci metody `onmessage`. Při přijetí zprávy se metoda automaticky zavolá, přičemž jako první parametr obdrží plné znění zprávy. Pokud se jedná o instanci NewSyncServer, metoda navíc jako druhý parametr dostane informace o klientovi, který zprávu odeslal.

7.7.2 Události

Rozhraní události je inspirované knihovnou socket.io. Při odesílání události se jako první specifikuje její typ a pak libovolné množství dalších parametrů. Na události se reaguje registrováním posluchače na instanci NewSync pro konkrétní typ události. Posluchač obdrží všechny parametry v takovém pořadí, jak je odesílatel specifikoval. Pokud se událost zpracovává na straně serveru, posluchač dále obdrží informaci o klientovi, jenž událost vyvolal.

7.7.3 Zprávy s nízkou prioritou

Framework obsahuje systém pro zprávy s nízkou prioritou. Takovými zprávami jsou především ty s velmi krátkou životností. Pokud dojde k její ztrátě, nemá smysl se ji pokusit doručit znova, jelikož v dohledné době bude její informace zastaralá a přepsaná hodnotou z následující zprávy. Například při sledování vozidla na mapě, kdy probíhá synchronizace dostatečně často, nemusí být výpadek jedné zprávy ani zaznamenatelný.

Protože není jisté, že strana klienta obdržela všechny zprávy s nízkou prioritou, nelze na jejich základě efektivně sestavovat rozdíly. Nemá smysl pokoušet se optimalizovat změny prováděné v poli relativizací vůči jeho současné podobě (např. místo celého pole poslat zprávu o vložení konkrétní hodnoty na konkrétní index), protože u klienta může dané pole vypadat jinak, kvůli ztrátě jakékoliv předchozí zprávy s nízkou prioritou.

Systém nízké priority jde využít pouze pro připojení, které využívá UDP. V prostředí internetových prohlížečů je takovým připojením WebRTC, v němž jde při vytvoření nového datového kanálu nastavit, jestli má fungovat v režimu TCP nebo UDP. Z tohoto důvodu je systém pro WebSocket nefunkční, jakékoliv změny s nízkou prioritou se automaticky přibalí do regulérní synchronizační zprávy a odešlou se běžným způsobem.

Nízko prioritní zprávy se generují automaticky prováděním změn označených nízkou prioritou. Pro zanesení těchto změn vývojář musí přistoupit k části stavu, kterou chce měnit, pomocí proxy reference v kontejneru a pak využít symbolu pro přístup k vlastnosti LowProxyHandler. Na ní pak volá metodu `set(key, value)`.

7.8 Distribuce

NewSync je v současné verzi dostupný formou veřejného repositáře na stránce GitHub. Pro instalaci frameworku do vlastního projektu je nutné překopírovat jeho zdrojové soubory.

7.9 Struktura zdrojového kódu

Framework se skládá ze dvou částí, klientské a serverové. Protože jsou si tyto části v mnoha případech podobné, sdílejí mezi sebou výrazné množství zdrojového kódu. Toho je docíleno pomocí dědičnosti. Hlavním zdrojem funkcionality jsou serverové varianty tříd, od kterých dědí klientské verze, které přepisují relevantní části kódu. Bohužel dědičnost neumožňuje odebrat metodu definovanou z rodičovské třídy. Proto metody serverového charakteru, které nedávají smysl v klientské části, jsou zdrojem výjimek. V dokumentaci zdrojového kódu jsou tyto metody explicitně označené.

Modularizace kódu

Transpilace je proces přenesení kódu z jednoho jazyka do druhého, případně v rámci stejného jazyka, ale do jiné verze, přičemž zůstává zachovaná původní funkcionalita.

Dalším problémem mezi serverovou a klientskou části kódu je způsob jeho modularizace. V době svého vzniku JavaScript neumožňoval kód strukturovat do více souborů. Proto vzniklo několik komunitních řešení modularizace, než se vytvořil oficiální standard.

Prohlížeče adoptovaly oficiální standard ESM (EcmaScript Modules), zatímco interpret Node.js využívá komunitní standard CommonJS[2]. Oba tyto způsoby nelze v základu vzájemně kombinovat, tedy není možné bez vnějších nástrojů použít kód napsaný ve standardu CommonJS na platformě, která využívá standard ESM.

Aby bylo možné docílit jednotné kódové základny pro obě části frameworku, celý kód je napsaný ve standardu ESM pro klientskou i serverovou část. Celá serverová část aplikace se poté pomocí nástroje Webpack transpiluje, přičemž se z veškerých použitých souborů vytvoří jeden velký soubor, který do sebe komponuje všechny použité moduly, čímž se předejde potřebě využít jakýkoli způsob načítání modulů. Přeneseně se dá celý proces nazvat “kompilací” za vzniku jednoho spustitelného souboru.

Popsaný proces pomocí nástroje Webpack využívá i klientská část aplikace, ale z jiných důvodů. Protože v minulosti internetové prohlížeče také měly omezenou podporu modularizace kódu, bylo ho nutné nejdříve sloučit. I když dnes jsou moduly podporovány, kód stále prochází sloučením do jednoho

[2]Node.js momentálně nabízí alespoň experimentální podporu modulů ve formátu ESM, ale všechny soubory využívající ESM musí být explicitně pojmenované s příponou .mjs místo standardní .js.

souboru s využitím následné minifikace[3], kdy vznikne kompaktnější soubor a minimalizují se data nutná k přenosu aplikace do prohlížeče uživatele. Pro Node.js se jedná o nestandardní a více komplikovaný proces, jež bylo nutné podstoupit kvůli sdílení zdrojového kódu mezi klientskou a serverovou částí.

[3]Během minifikace se například nahrazují jména funkcí a proměnných na kratší nebo se odstraní formátování, čímž má výsledný kód menší velikost, ale stává se nečitelným.

Kapitola 8
Demo aplikace

8.1 Obecný přehled

Demo aplikace je laická interpretace možné podoby systému pro sledování vozidel policie a záchranných složek. Uživateli je prezentována mapa s pozicí jednotlivých vozidel, nemocnic a policejních stanic. Zobrazit lze i kompletní surová data simulace. Dále je možné v aplikaci pozorovat síťový provoz, tj. konkrétní příchozí zprávy od serveru, jejich obsah a velikost.

Cílem aplikace je demonstrovat efektivitu zmenšení objemu posílaných dat a funkcionalitu frameworku NewSync a dále jeho použití pomocí zveřejněného zdrojového kódu, dostupného v příloze v archivu "newsync.zip", případně v GitHub repositáři přístupném pod url adresou `https://github.com/neuvimat/newsync`.

Synchronizace stavu v rámci dema aplikace probíhá pravidelně každou sekundu.

8.2 Klient

8.2.1 Použité technologie

Klient je webová aplikace napsaná v JavaScriptu, HTML a CSS pomocí UI frameworku Vue. Pro automatickou synchronizaci informací o stavu simulace záchranných složek na klientovi se stavem simulace na serveru je použit framework NewSync.

Protože klient vyžaduje stálé spojení se serverem, využívá architekturu single page aplikace.

Během navigace na internetových stránkách, které nevyužívají single page architekturu, se webový prohlížeč musí pravidelně dotazovat na celkovou novou podobu stránky. To může vyústit v její přenačtení a zpravidla[1] ke ztrátě dosavadního stavu, a především ke ztrátě zdrojů jako otevřeného WebSocket nebo WebRTC spojení, které se poté musí navázat znovu.

[1] Stav je možné explicitně na vyžádaní uložit napříč sezeními nebo přenačítáním stránky do lokálního úložiště a poté zpětně rekonstruovat. Pro stav malého rozsahu lze využít i soubory cookies.

Z tohoto důvodu je klient single page aplikací, u kterých se při prvním přístupu na stránku typicky[2] stáhnou veškeré její komponenty. Poté se uživatelskou interakcí a asynchronními dílčími dotazy dynamicky zobrazují, aktualizují a schovávají její části bez nutnosti žádat server o celý nový HTML dokument. Takto se docílí zachování všech existujících zdrojů a lokálních dat po celou dobu prohlížení stránky.

8.2.2 Interakce se serverem

Kromě automatického přijímání a zpracování synchronizačních zpráv klient také posílá vlastní požadavky na server, kterými ovlivňuje stav simulace nebo části, které budou synchronizovány. Požadavky ovládající simulaci jsou implementovány funkcionalitou uživatelských událostí v rámci NewSyncu.

8.2.3 Distribuce klientské aplikace

Protože klientská část demo aplikace je napsána ve frameworku Vue, v její CLI[3] verzi, je nutné ji nejdříve zkompilovat. Výsledná podoba souborů klientské aplikace se poté musí přenést na server, který ji následně distribuuje.

Snazším způsobem zprovoznění a nasazení klientské části je použít vývojářský server automaticky obsažený ve Vue CLI, jehož jediným úkolem je servírovat danou Vue aplikaci. Konkrétní instrukce pro spuštění pomocí této varianty jsou obsaženy v GitHub repositáři.

8.2.4 Části klienta

Klient je rozdělen do čtyř pohledů. Každý se specializuje na zobrazení nebo práci s konkrétní části dat simulace nebo doprovodných statistik. Protože se jedná především o demonstraci funkcí frameworku NewSync, celkový design UI se vyznačuje minimalistickým přístupem.

Sdílené komponenty

Napříč klientem je přítomná spodní lišta, jež zobrazuje statistiku množství přijatých aplikačních dat od serveru v bajtech. K dispozici je celkem šest kategorií, podle různých možností podoby posílaných dat:

- Kompletní aktualizace ve formátu JSON.
- Rozdílové aktualizace ve formátu JSON.

[2] Pokud by se jednalo o skutečně rozsáhlou single page aplikaci, je možné ji rozdělit na více částí a omezit tak počáteční velikost HTML dokumentu. Komponenty se poté dynamicky načítají a dodávají do již existujícího dokumentu podle potřeby. Stále ale nedochází ke kompletnímu opětovnému načtení stránky.

[3] CLI verze Vue se vyznačuje jiným způsobem práce se zdrojovými soubory. Místo psaní přímo zdrojového JavaScriptového kódu, v CLI variantě jsou jednotlivé části aplikace rozděleny do vlastních souborů se specifikou strukturou. Zdrojový kód je takto více přehledný, ale bez předchozí kompilace nepoužitelný.

- Rozdílové aktualizace ve formátu JSON a použití slovníku dlouhých klíčů.
- Kompletní aktualizace ve formátu MessagePack.
- Rozdílové aktualizace ve formátu MessagePack.
- Rozdílové aktualizace ve formátu MessagePack a použití slovníku dlouhých klíčů.

Od serveru vždy přichází zprávy ve formátu MessagePack s využitím slovníku dlouhých klíčů. Klient po deserializaci a zpracování zprávy lokálně nasimuluje, jak by zpráva vypadala, kdyby komunikace probíhala způsobem dané kategorie. Poté spočítá její velikost a započítá ji do celkové statistiky.

U každé kategorie je indikátor v procentech reprezentující podíl velikosti zpráv dané kategorie a kompletních aktualizací v JSONu pro snazší porovnání efektivity snížení objemu posílaných dat u jednotlivých kategorií.

Úvodní obrazovka

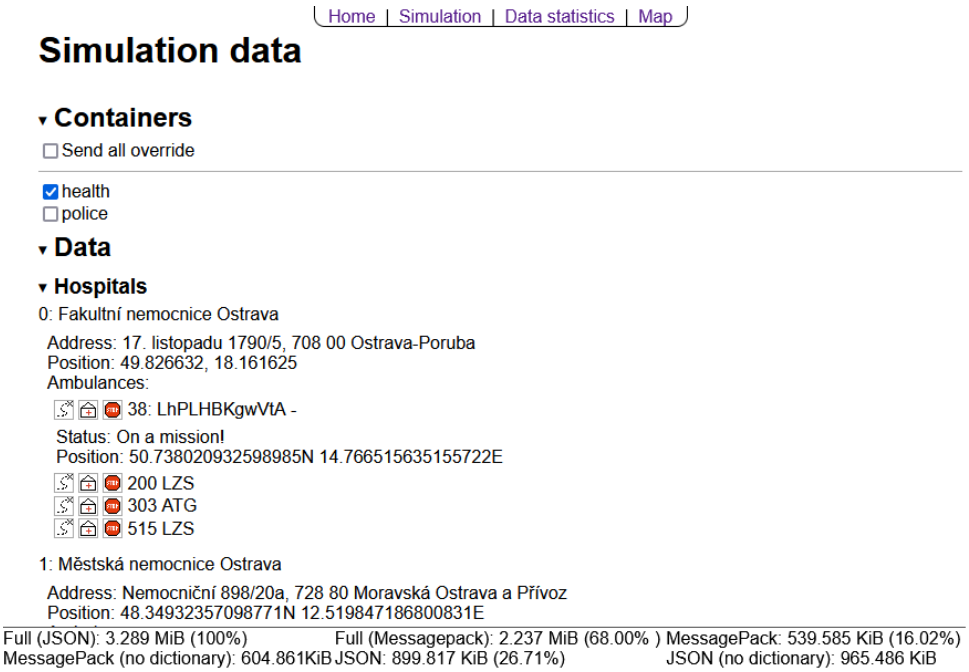
Na úvodní obrazovce si uživatel může vybrat mezi připojením pomocí technologie WebSocket nebo WebRTC. Po vyplnění požadovaných parametrů pro konkrétní připojení a úspěšném navázání komunikace se zobrazí krátké instrukce ohledně používání aplikace a odemkne se možnost přepínat mezi ostatními částmi pomocí navigačního menu.

Data simulace

Na této obrazovce v záložce kontejnerů může uživatel specifikovat, jaká část stavu má být synchronizována.

V záložce požadavků se nachází tlačítka pro ovládání vozidel, jako možnost zastavit všechna vozidla nebo náhodně rozpohybovat dané množství vozidel. Dále lze vozidlo s konkrétním ID zastavit, poslat zpět do přiřazené nemocnice, respektive policejní stanice, nechat náhodně pohybovat nebo poslat na specifické souřadnice.

V záložce dat si lze prohlížet aktuální stav strukturovaný podle různých kategorií (vozidla, nemocnice, ...), nebo jeho surovou kompletní podobu reprezentovanou formátem JSON. U zobrazení informací ohledně jednotlivých vozidel jsou k dispozici ikonky, které fungují jako zkratka pro poslání požadavku o změně jejich chování. Ikonami lze auto zastavit, poslat zpět na základnu nebo náhodně rozpohybovat.



Obrázek 8.1: Ukázka přehledu aktuálního stavu v demo aplikaci.

Přehled zpráv

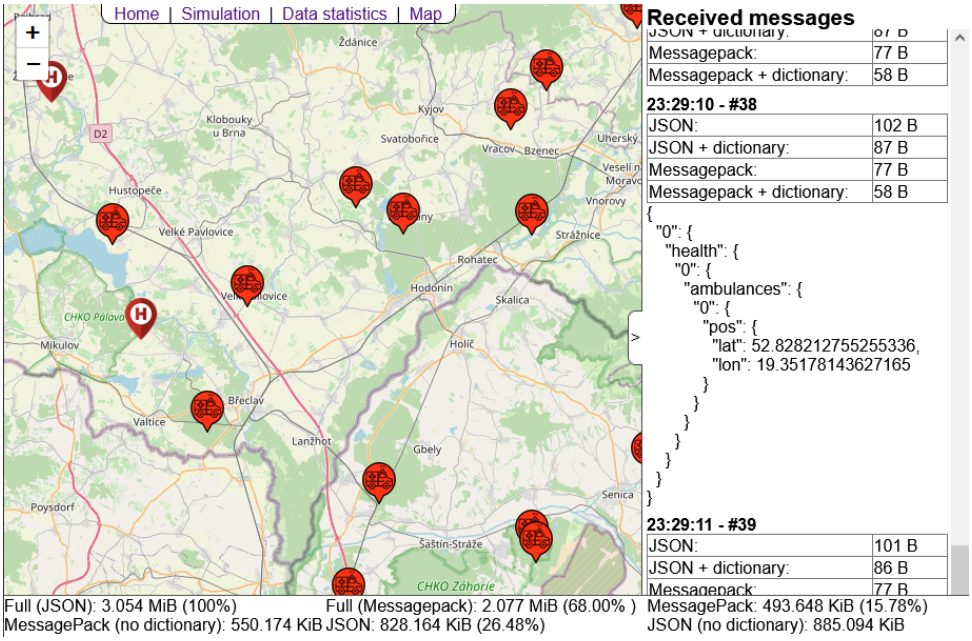
V přehledu zpráv se zaznamenávají všechny zprávy přijaté během sezení. U každé zprávy je ukázáno její pořadí, čas přijetí a velikosti v bajtech pro kategorie rozdílových aktualizací a rozdílových aktualizací se slovníkem dlouhých klíčů pro formát JSON i MessagePack. Po kliknutí na zprávu se zobrazí její obsah v čitelném formátu JSON. Pro snazší interpretaci obsahu zprávy jsou všechny klíče převedeny zpět do své původní dlouhé formy, i když v rámci zprávy byly poslané zkráceně.

Mapa

Na mapě jsou pomocí ikon zobrazena všechna vozidla, nemocnice a policejní stanice. Kliknutím na jednotlivé ikony se zobrazí lišta poskytující detaily o vybrané entitě stejným způsobem jako v záložce dat v pohledu "data simulace", včetně ikon pro ovládání, je-li momentálně vybráno vozidlo.

Vozidla, která jsou momentálně zaparkovaná na základně, se nezobrazují.

Pro snazší porovnání dopadu příchozích zpráv na stav simulace v mapě lze v pravé části pohledu otevřít a schovat kompaktní verzi pohledu "přehled zpráv".



Obrázek 8.2: Ukázka pohledu na mapu v demo aplikaci.

8.3 Server

8.3.1 Použité technologie

Serverová část je napsána v JavaScriptu uzpůsobeném pro spuštění v interpretu Node.js s využitím frameworku Express, jehož cílem je usnadnit vytvoření HTTP serveru.

8.3.2 Úkoly serveru

Úkolem serverové části je udržovat stav simulace v paměti a pravidelně jej aktualizovat. Serverová varianta NewSyncu poté automaticky rozesílá aktualizace klientům. Server v sobě dále obsahuje implementaci WebSocket serveru, sloužícího jako jeden z druhů komunikace. Pro navázání komunikace pomocí WebRTC je na serveru dále implementace socket.io knihovny, která slouží pro předávání signálních informací pro WebRTC. Nakonec je server schopný obsluhovat HTTP požadavky a servírovat statické HTML soubory, jako například výkonnostní testy formátů určených pro prohlížeč nebo klientskou aplikaci.

Po spuštění serveru se automaticky vytvoří nová simulace, která se naplní vždy stejným množstvím náhodných dat. Simulace je zprvu statická, tj. neprobíhají v ní žádné změny. Až připojení klienti pomocí svých příkazů, implementovaných formou NewSync uživatelských událostí, zadají simulaci instrukce, které vedou k pravidelným změnám. Zadají-li klienti protichůdné příkazy (např. zastavit a rozpohybovat konkrétní vozidlo), projeví se pouze poslední z protichůdných požadavků.

Server dále plně obsluhuje síťová připojení, monitoruje jejich stav a registruje je do seznamu klientů v NewSyncu, respektive při ztrátě spojení odebírá.

8.3.3 Omezení

Přestože server v sobě obsahuje kód pro vytváření spojení pomocí WebSocketu i WebRTC, v daný moment může být aktivní pouze jedna z možností, viz sekce 7.2 Omezení. V jakém režimu má server fungovat se stanoví argumentem v příkazové řádce při jeho spuštění.

8.4 Dokumentace

Dokumentace zdrojového kódu klientské i serverové části se zaměřuje především na práci s NewSync instancí, synchronizaci, práci s daty a komunikaci formou JSDoc komentářů.

V klientské části zdrojového kódu u jednotlivých částí UI, jeho architektury a způsobu interakce s UI prvky není dokumentace, jelikož je pro práci s NewSyncem irelevantní, a protože UI v rámci demo aplikace spadá pod standardní využití frameworku Vue. Navíc vývojář má volnost využít pro UI jakoukoliv technologii.

V GitHub repositáři jsou dále technické pokyny pro vývojáře ohledně zprovoznění aplikace, základní principy používání a příklady kódu.

Kapitola 9

Testování frameworku NewSync

Kompletní naměřená data pro všechny testy v této kapitole jsou k dispozici v příloze v souboru *demoTests.xlsx*, kde jsou navíc statistické údaje, jako minima, maxima a intervaly spolehlivosti.

9.1 Paměťové testy

Testování efektivity zmenšení objemu přenášených dat probíhalo v prostředí demo aplikace popsané v kapitole 8 Demo aplikace.

9.1.1 Metodika testování

Během všech testů se synchronizoval pouze kontejner obsahující sanitky a nemocnice. Cílem testování je porovnání efektivity snížení objemu přenášených dat technikami použitých v NewSyncu s kompletními aktualizacemi ve formátu JSON a vyhodnotit přínos jednotlivých testovaných technik v závislosti na množství měnícího se stavu.

Datový model sanitky má následující atributy:

- id – textový řetězec – id sanitky v rámci systému
- sign – textový řetězec – volací znak (jméno) sanitky
- hospital – textový řetězec – id nemocnice, ke které je sanitka přiřazená
- pos – vnořený objekt reprezentující GPS pozici sanitky
  - lat – číslo – zeměpisná výška
  - lon – číslo – zeměpisná délka

Datový model nemocnice má následující atributy:

- id – textový řetězec – id nemocnice v rámci systému
- name – textový řetězec – jméno nemocnice
- address – textový řetězec – adresa nemocnice

- ambulances – pole textových řetezců – seznam id sanitek, které jsou přiřazené k nemocnici
- pos – vnořený objekt reprezentující GPS pozici nemocnice
  - lat – číslo – zeměpisná výška
  - lon – číslo – zeměpisná délka

V testech se pracovalo se simulací obsahující 100 nemocnic a 600 sanitek. Celkově proběhly čtyři testy, v každém se po celou dobu testování pohyboval neměnný počet sanitek. V prvním testu se jich pohybovalo 50, v druhém 300 a v třetím i čtvrtém 600.

V prvních třech testech datový model odpovídá výše uvedené specifikaci, pro čtvrtý test byly klíče "pos", "lat" a "lon" v modelu sanitky nahrazeny delšími variantami "position", "latitude" a "longitude". Smysl čtvrtého testu je pozorovat vliv délky klíčů na efektivitu komprese slovníkem (viz sekce 7.4.5).

Testy byly ukončeny, jakmile celkový objem přenesených dat pro kompletní aktualizace ve formátu JSON přesáhl hranici 50 MiB.

9.1.2 Naměřené hodnoty

Paměťová náročnost se testovala pro formáty MessagePack i JSON. Pro každý formát byl zaznamenán objem přenesených dat pro kompletní aktualizace (sloupec "celý stav"), rozdílové aktualizace (sloupec "rozdíly stavu") bez slovníku a rozdílové aktualizace se slovníkem (sloupec "rozdíly stavu + slovník").

Všechny hodnoty uvedené v tabulkách 9.1, 9.2, 9.3 a 9.4 vyjadřují velikost v jednotkách MiB. V závorce je v procentech uveden poměr výsledné velikosti dané techniky a formátu vůči kompletním aktualizacím v JSONu. Poslední sloupec porovnává, o kolik jsou rozdílové aktualizace menší, použije-li se navíc slovník dlouhých klíčů.

Formát	Celý stav	Rozdíly stavu	Rozdíly stavu + slovník	Komprese slovníkem
JSON	50,036 (100%)	2,037 (4,07%)	1,930 (3,86%)	5,26%
MessagePack	34,077 (68,10%)	1,256 (2,51%)	1,149 (2,30%)	8,49%

Tabulka 9.1: Paměťová náročnost pro 50 sanitek v pohybu, v jednotkách MiB

Formát	Celý stav	Rozdíly stavu	Rozdíly stavu + slovník	Komprese slovníkem
JSON	50,026 (100%)	11,832 (23,65%)	11,150 (22,29%)	5,77%
MessagePack	34,077 (68,12%)	7,282 (14,56%)	6,600 (13,19%)	9,36%

Tabulka 9.2: Paměťová náročnost pro 300 sanitek v pohybu, v jednotkách MiB

Formát	Celý stav	Rozdíly stavu	Rozdíly stavu + slovník	Komprese slovníkem
JSON	50,008 (100%)	23,617 (47,23%)	22,208 (44,41%)	5,97%
MessagePack	34,077 (68,14%)	14,549 (29,09%)	13,140 (26,28%)	9,68%

Tabulka 9.3: Paměťová náročnost pro 600 sanitek v pohybu, v jednotkách MiB

Formát	Celý stav	Rozdíly stavu	Rozdíly stavu + slovník	Komprese slovníkem
JSON	50,084 (100%)	26,445 (52,80%)	19,952 (39,81%)	24,55%
MessagePack	35,760 (71,40%)	18,293 (36,52%)	11,800 (23,56%)	35,49%

Tabulka 9.4: Paměťová náročnost pro 600 sanitek v pohybu (s delšími klíči), v jednotkách MiB

9.1.3 Analýza naměřených hodnot

V tabulkách 9.1, 9.2, 9.3 můžeme pozorovat přímou souvislost efektivity rozdílových aktualizací v závislosti na množství statického stavu. Přestože se v třetím testu pohybovaly všechny sanitky v simulaci, zprávy obsahující pouze rozdíly ve stavu stále byly o více jak polovinu menší než zprávy s celým stavem pro daný formát, protože se neposílaly zbytečně informace o nemocnicích nebo ostatních vlastnostech sanitek, jako jejich id nebo volací znak.

Dále si lze všimnout drobného rozdílu v absolutní velikosti celého stavu pro formát JSON, přestože by na vliv kompletních aktualizací počet pohybujících se sanitek napříč testy neměl mít žádný vliv. Protože se pozice sanitek měnila přičítáním náhodných čísel, občas se stalo, že desetinná čísla reprezentující GPS souřadnice měla jiný počet desetinných míst. Protože JSON zapisuje čísla pomocí číslovek jako text, velikost výsledných zpráv se mohla různě lišit.

U formátu MessagePack se absolutní velikost zpráv nezměnila a zůstala po dobu všech prvních tří testů 34.077 MiB, protože desetinná čísla jsou v něm vždy reprezentovaná čtyřmi bajty.

V prvních třech testech můžeme pozorovat mírný růst efektivity komprese slovníkem s přibývajícím počtem sanitek v pohybu. Synchronizační zpráva vždy obsahuje fixní počet bajtů navíc kvůli své struktuře na nejvyšší úrovni a kontrolním sekvencím. S vyšším počtem užitných dat (změn) vzniká více prostoru pro využití slovníku a zároveň se snižuje poměr fixní velikosti dat kontrolních sekvencí vůči celkové velikosti zprávy.

U čtvrtého testu s delšími klíči je vidět celkový nárůst velikosti celého stavu i rozdílových aktualizací kvůli delším klíčům. Je nutné si uvědomit, že kvůli metodice testování můžou být absolutní hodnoty v tabulce 9.4 zavádějící. Test byl ukončen, jakmile bylo přijato celkem 50 MiB v kompletních aktualizacích ve formátu JSON. Tedy v rámci čtvrtého testu bylo celkově přijato méně zpráv než v prvních třech testech, přesto bylo dosaženo stejné hranice 50 MiB. Nárůst objemu dat kvůli delším klíčům lze pozorovat v relativním vyjádření v procentech v ostatních buňkách. Výjimkou je sloupec rozdílů stavu s využitím

slovníku. Zprávy v této kategorii zůstaly ve skutečnosti stejně veliké, ale zvětšil se objem ostatních zpráv, proto se snížil jejich relativní objem. Z tohoto důvodu lze pozorovat výrazně vyšší efektivitu komprese slovníkem. Konkrétně pro JSON je 4,11× vyšší, pro MessagePack 3,67× vyšší. Absolutní velikost zpráv v této kategorii je menší než v tabulce 9.3 kvůli celkově nižšímu počtu přijatých zpráv.

9.1.4 Závěr

Pouze změna formátu z JSONu na MessagePack dosáhla téměř třetinové úspory dat. Další významné komprese objemu dosáhla technika rozdílových aktualizací, i v nepříznivém případě, kdy se pohybovaly všechny sanitky (tabulka 9.3), porovnáním objemu velikosti dat sloupce “celý stav” a “ rozdíly stavu” v rámci jednoho řádku došlo ke snížení posílaných dat o 52,77 % pro JSON a 57,31 % pro MessagePack. V nejpříznivějším testovaném případě (tabulka 9.1), bylo dosaženo úspory 95,93 % pro JSON a 96,31 % pro MessagePack.

Z výsledků měření je zřejmé, že efektivita technik je závislá na konkrétním případě použití a struktuře dat v simulaci. Každopádně ve všech testovaných případech lze důvěryhodně tvrdit, že kombinací všech technik lze dosáhnout značné úspory dat, přičemž největší zásluhu má přechod na formát MessagePack a použití rozdílových aktualizací, kdy použití MessagePacku vykazuje nejvíce konzistentní výsledky.

Jediný případ, kdy rozdílové aktualizace neušetří žádná data, je velice nepravděpodobná situace, kdy se změní celý stav aplikace mezi synchronizačními cykly. Ale i tehdy by mohl pomoci slovník dlouhých klíčů, pokud by se nejednalo o extrémní případ, kdy by struktura aplikace byla vyjádřena velice krátkými klíči nebo pouze pomocí indexovaných polí.

9.2 Výkonnostní testy

Z hlediska výkonnosti byla testována speciální, minimální verze aplikace, která mění svůj stav s využitím NewSyncu, a pak její druhá verze bez použití frameworku. Množství dat a jejich struktura v aplikaci byla shodná s aplikací testovanou v paměťových testech.

Zvlášť byl testován dopad Proxy API na rychlost práce s objekty s úmyslem vyhodnotit zpomalení způsobené jeho používáním.

Všechny hodnoty v tabulkách vycházejí z průměrných hodnot z 10 měření. Před začátkem měření nejdříve proběhly testy stokrát “naprázdno”, aby se mohly aplikovat optimalizace ve formě JIT kompilace.

9.2.1 Celková rychlost aplikace

Protože největší vliv NewSyncu na rychlost aplikace se nachází na serverové straně automatickým sledováním změn, byla testována pouze tato část aplikace.

Metodika testování

Pro testy byla vyčleněna a měřená pouze ta část kódu, jež v simulaci provádí změny, vytváří synchronizační zprávy a serializuje je.

Prvně byla testována aplikace plně využívající NewSync s rozdílovými aktualizacemi pomocí automatické detekce změn, slovník dlouhých klíčů a zprávy ve formátu MessagePack. Poté druhá verze aplikace, jež pouze provedla změny a veškerý stav převedla do formátu JSON.

Časová náročnost posílání zpráv nebyla součástí měření.

Celkově bylo provedeno 6 testů, zvlášť pro 50, 300 a 600 sanitek v pohybu pro obě verze aplikace. V každém testu se měřila doba trvání provedení 100 iterací simulace.

Naměřené hodnoty

Kromě celkového trvání testu se samostatně měřilo, kolik času zabere do aplikace změny zanést, a poté, jak dlouho trvá serializace synchronizační zprávy.

Pro NewSync je v tabulce 9.5 součástí sloupce "Změna stavu" také čas nutný pro vytvoření synchronizační zprávy (ve verzi bez NewSyncu se serializuje přímo stav aplikace, není třeba vytvářet speciální zprávu pouze se zaznamenanými změnami).

Sanitek v pohybu	NewSync	Změna stavu [ms]	Serializace [ms]	Celkem [ms]
50	Ano	5,010	8,667	13,640
50	Ne	0,194	69,656	69,924
300	Ano	28,101	45,204	73,250
300	Ne	0,592	70,114	70,705
600	Ano	61,972	130,312	192,132
600	Ne	1,342	69,858	71,200

Tabulka 9.5: Doba trvání synchronizačního cyklu pro aplikaci využívající NewSync vs. aplikace s kompletními JSON aktualizacemi.

Analýza naměřených hodnot

Z hodnot v tabulce je pro NewSync jednoznačně vidět přímá souvislost mezi počtem změn a časem nutným pro jejich zanesení do simulace i serializaci výsledné zprávy pro iteraci simulace.

Doba provádění změny stavu se úměrně zvedá s počtem sanitek v pohybu pro obě verze aplikace, ale pro verzi bez frameworku tvoří doba provádění změn z celkového času nevýraznou část. Vliv počtu změn na dobu trvání synchronizace v této verzi aplikace sice existuje, ale projevuje se minimálně. Mnohem větší vliv zde má celková velikost stavu, na rozdíl od verze s NewSyncem, kde se na celkovém času významně podílí doba provádění změn i serializace.

Ve verzi aplikace bez NewSyncu s kompletními aktualizacemi, podle očekávání, trvá serializace zprávy přibližně stejnou dobu, nehledě na počet

pohybujících se sanitek.

Porovnáním doby nutné pro změnu stavu v rámci obou verzí aplikace pro daný počet sanitek lze pozorovat, že varianta s NewSyncem je pro 50 sanitek 28,82× pomalejší, pro 300 sanitek 47,47× pomalejší a pro 600 sanitek 46,18× pomalejší.

Závěr

Přestože je NewSync zatížen používáním pomalejšího formátu a Proxy API, jestliže se mění pouze část stavu aplikace, NewSync může kromě zmenšení objemu dat dokonce snížit i časovou náročnost synchronizace.

Při provedení velkého počtu změn během synchronizačního cyklu bohužel NewSync nepřináší časovou úsporu, naopak celý proces zpomaluje. Na zpomalení se společně podílí nutnost dané změny zaznamenávat a detekovat, a postupně narůstající velikost synchronizačních zpráv serializovaných do formátu MessagePack pomocí knihovny msgpackr, jež je pomalejší než nativní JSON.

9.2.2 Výkonnost proxy

Jelikož jádro frameworku je postavené na využití Proxy API, testy výkonnosti proxy mají za cíl zjistit, jak velký podíl z režie navíc použitím frameworku v aplikaci je způsobený jenom tím, že framework používá Proxy API.

Dále je cílem testování zjistit efektivitu optimalizace pomocí obalování metod v polích (viz sekce 7.5.1 Optimalizace proxy)

Metodika testování

Všechny testované případy probíhaly na běžném JavaScriptovém objektu (řádek “Nativní”) a pak na objektu, k jemuž se přistupovalo pomocí proxy reference (řádek “Proxy”).

Naměřené hodnoty

V tabulce 9.6 bylo pro určení vlivu proxy na rychlost provádění metod v poli testována specializovaná verze proxy, označená jako “Smart Proxy”, jež používá navrhovanou optimalizaci obalování metod, kdy se uvnitř obalené metody přistupuje k poli přes původní, neobalenou referenci. Kromě tohoto v handlerech u proxy nebyla použita žádná logika navíc, jako např. zaznamenávání změn.

Ve sloupcích tabulky 9.6 jsou zmíněné konkrétní testované metody pole. Metody *sort* a *filter* byly v rámci testu spuštěny jednou na poli o 150 000 prvcích. Výsledkem metody filtr bylo odstranění poloviny prvků v poli. Metoda *push* byla ve smyčce zopakována 75 000krát, kdy do pole vždy přidala jeden prvek.

Verze pro metodu *push* s využitím cache (sloupec “Array.push (cache)”) místo, aby opakovaně ve smyčce přistupovala k metodě a tím spouštěla past

v proxy, si referenci na metodu uložila a ve smyčce volala danou metodu přes tuto referenci.

Typ	Array.sort	Array.filter	Array.push	Array.push (cache)
Nativní	183,553	2,191	0,490	-
Proxy	277,098	16,919	115,034	109,616
Smart Proxy	187,096	2,473	1,608	0,467

Tabulka 9.6: Vliv proxy na rychlost operací v polích, v jednotkách ms.

V tabulce 9.7 jsou výsledky testování následujících operací na objektu: přístup k vlastnosti, přiřazení konkrétní nové hodnoty do vlastnosti, přičtení náhodného čísla k vlastnosti a přičtení náhodného čísla k vlastnosti pomocí operátoru "+=".

Typ	Přístup	Přiřazení	Přičtení	Přičtení (+=)
Nativní	0,021	0,629	0,671	0,674
Proxy	1,882	29,987	33,957	34,431

Tabulka 9.7: Vliv proxy na rychlost operací v objektu, v jednotkách ms.

Analýza naměřených hodnot

V tabulce 9.6 lze pozorovat zpomalení u proxy na všech testovaných metodách, ovšem vždy v jiné intenzitě.

Pokud se porovná zpomalení způsobené proxy vůči nativní verzi, nejvyšší je v testu pro metodu *push*, kdy bylo až 234násobné, kdy ani varianta s cache zpomalení výrazně nesnížila.

Smart Proxy se od nativní varianty z relativního hlediska výrazně nelišila, až na test pro metodu *push*, kde se projevilo zpomalení kvůli opakovanému spouštění pasti v každé iteraci smyčky. Toto zpomalení odstranila varianta s cache.

Bez znalosti vnitřní implementace interpretu nelze přesně určit, proč se u řádku "Proxy" zpomalení tak výrazně lišilo napříč metodami. Jednou z možností je, že přes proxy referenci jsou jakékoliv operace měnící počet prvků pole zatížené výraznou režií navíc. Metoda *sort*, která je nejméně zpomalená použitím proxy reference, jako jediná nemění počet prvků v poli, ale jen jejich pořadí.

Z tabulky 9.7 vyplývá, že přístup k objektu pomocí proxy v porovnání s řádkem "nativní" je až 89,61× pomalejší, přiřazení konkrétní hodnoty nebo přičtení k ní pak přibližně 50× pomalejší.

Závěr

Bohužel testování prokázalo, že použití proxy má výrazný vliv na rychlost provádění operací na objektu, i přestože proxy neobsahuje vlastní logiku.

Optimalizace pro smart proxy u metod polí funguje až nativní rychlostí, protože zpomalení vzniklé užitím proxy se projeví pouze jednou, a to při

přístupu k dané metodě. Proxy poté vrátí obalenou verzi metody, jež pracuje na původní referenci na objekt, což znamená v nativní rychlosti, ale zároveň bez automatické detekce změn. Protože cíleně obalujeme konkrétní metody pole, známe jejich účel a efekt, který budou na pole mít. Změny tak napřímo zaznamená obalená metoda, nikoliv metody reagující na spouštění pastí v proxy. Např. metoda *push*, která by byla volaná na proxy referenci, by ve své neobalené verzi vždy spustila past pro přiřazení hodnoty na daný index a past pro úpravu vlastnosti délky pole. Obalená *push* metoda místo toho může automaticky zapsat změnu do kontejneru bez spuštění jakýchkoliv pastí.

Kapitola 10
Prostor pro zlepšení

10.1 Více druhů připojení současně

Přestože NewSync není závislý na konkrétních typech připojení, momentálně je nelze v základu libovolně kombinovat. Obsluhu síťového připojení obstarávají ovladače, viz sekce 7.4.3 Síťový ovladač (driver). Díky návrhu interakce NewSyncu s ovladači je možné, aby si vývojář napsal vlastní obecný ovladač. Ten na základě vývojářem dodaných parametrů během volání metody pro přidání NewSync klienta dosadí do modelu popisující klientské spojení další vlastnosti, které umožní identifikaci konkrétního typu spojení daného klienta. Ovladač pak může správně reagovat na různé varianty spojení.

V ideálním stavu by již v základu byla k dispozici implementace ovladače, jež by fungoval jako abstrakční vrstva. Pomocí kompozice by do sebe ovladač integroval konkrétní implementace různých jiných ovladačů a automaticky mezi nimi přepínal pro specifické klienty.

10.2 Zachování identity objektů

Kvůli omezení, kdy není zachována identita objektů během přenosu, dochází k redundanci zápisu změn, jestliže se někde ve stavu objevují mnohočetné reference na jeden konkrétní objekt. Stejná změna se do zprávy zapisuje několikrát, a to na všechna místa, kde se referencovaný objekt nachází. Kdyby se podařilo zachovat identitu během přenosu, namísto zápisu změn do všech lokací by pouze stačilo předat informaci, že došlo ke změně v konkrétním objektu podle specifického identifikátoru. Klientská část frameworku by poté sama nalezla všechna místa, kam danou změnu dosadit.

Pro tento způsob přenášení identity by bylo nutné generovat unikátní identifikátory pro všechny nově vzniklé objekty a odesílat je klientům, aby pak mohlo dojít k následnému párování změn. Klient by si pak musel udržovat mapu těchto identifikátorů a na nich navázaných objektů.

Přenos těchto identifikátorů by znamenal přenášení dalších dat navíc. Pokud by ale stav aplikace běžně pracoval s několika referencemi na jeden objekt, objem dat ušetřený na opakovaném zanášení změn převáží nad velikostí identifikátorů. Jako další optimalizace se nabízí k objektu přiřadit identifikátor

a propagovat změny přes něj až potom, co se detekuje více referencí na objekt ve stavu aplikace.

Další z možných problémů by bylo určování hranic začátku a konce instancí objektů konkrétních tříd při jejich vzájemném vnořování. Například v demo aplikaci je objekt třídy “ambulance”, která v sobě obsahuje vnořený objekt “pozice”, celkově se ale stále jedná o jednu entitu v simulaci. NewSync by tuto skutečnost musel rozpoznat a správně identifikovat oba tyto objekty jako jeden celek.

10.3 Pokročilé filtrování stavu

Filtrování stavu je momentálně podporované pouze na nejvyšší úrovni specifikováním kontejnerů, o které má klient v daný okamžik zájem. Tento systém by šlo rozšířit o možnost různých filtrů.

Základní filtry by vyřazovaly konkrétní části stavu na základě cest s více úrovňovou podporou. Například bychom ze zpráv vyřadily změny stavu obsažené pod klíčem “hospitals” a poté vnořeným klíčem “cz”. Pokročilé filtry by mohly dále ovlivňovat přítomnost konkrétních dat obsažených pod daným klíčem. Například ze stavu pod cestou “hospitals.cz” vynechat všechny záznamy, které nesplňují konkrétní podmínky, třeba požadovaný počet přiřazených sanitek. Ovšem u takto pokročilých filtrů by bylo nutno sledovat zátěž na výkon celého systému, pokud by větší množství klientů specifikovalo až příliš přísné filtrovací podmínky.

10.4 Historie změn

Pokud klient krátkodobě ztratí spojení, nebo na chvíli přestane vyžadovat změny stavu pro kontejner, po opětovném zapnutí synchronizace se musí poslat kompletní snímek stavu. Pokud by si server pamatoval stav více do minulosti, a jaký klient má momentálně u sebe jakou verzi stavu, mohl by místo kompletní aktualizace poslat soubor změn napříč několika synchronizačními cykly za předpokladu, že jsou v dohledatelné historii.

10.5 Synchronizace konkrétních uživatelů na vyžádání

Přidáním historie změn, jak popisuje předchozí sekce, by se zároveň umožnilo synchronizovat klienty v rozdílných intervalech, případně pouze po jejich explicitním požadavku o synchronizaci.

Protože v současné implementaci není k dispozici historie, synchronizace konkrétního klienta by propsala do aktuálního stavu simulace na serveru všechny provedené změny, jejich seznam odeslala klientovi a poté jej vyprázdnila pro nový synchronizační cyklus. Ostatní klienti by se o těchto změnách nedozvěděli.

10.6 Distribuce frameworku

Současný těžkopádný systém distribuce přímým kopírováním zdrojových souborů frameworku do projektu není ideální. Nejlépe by byl framework dostupný v NPM s možností jednotlivé části importovat způsoby ESM nebo CommonJS na základě případu použití vývojáře.

Kapitola 11

Závěr a shrnutí

V práci jsem popsal výzvy spojené s aktualizací stavu v reálném čase pomocí rozdílových aktualizací, vytipoval části aplikace, kterým je nutno věnovat zvýšenou pozornost při návrhu i implementaci, a zmínil, jaké technologie a postupy využít, abychom dosáhli vytyčeného zadání.

Podrobněji jsem se zaměřil na testování různých formátů dat a jejich rychlosti a efektivitu komprese v JavaScriptu. Na základě pozitivních výsledků jsem se rozhodl využít pro implementaci frameworku NewSync formát MessagePack, který vynikal vysokou mírou komprese dat a výkonností. Pro nejvyšší možnou výkonnost a úsporu paměti jsem zvolil konkrétní implementaci MessagePacku: msgpackr, který lze využít na serverové i klientské straně.

Dále jsem zmínil hotová řešení, která splňují nebo pomáhají docílit zadání práce. Žádné řešení se ovšem nehodí pro projekty malého rozsahu nebo již existující aplikace, protože migrace na tato řešení by s nejvyšší pravděpodobností znamenala přechod na jiné technologie nebo přepracování způsobu získávání a ukládání dat nebo práci s nimi. Navíc nepočítají se sledováním simulace, jejichž změny je nutno pravidelně synchronizovat v reálném čase.

Na základě požadavků jsem vytvořil framework NewSync, který vývojáři umožňuje vyčlenit stav aplikace, jejž automaticky detekuje programátorem prováděné změny a propaguje je klientům. Pro pokročilé uživatele frameworku je k dispozici několik metod a postupů pro asistování detekce změn, kterými lze zvýšit výkonnost práce s daty. Framework dále umožňuje programátorovi posílat vlastní typy zpráv, které využívají implementované kompresní techniky.

Následně jsem vytvořil aplikaci založenou na vzniklém frameworku NewSync. Aplikace se skládá z Node.js serveru, který distribuuje svůj aktuální stav, a klientské webové aplikace využívající UI framework Vue. Klient stav přijímá ve formě kompletních i rozdílových aktualizací a vypisuje statistiky sledující objem posílaných dat pro různé metody. Pro propojení serverové a klientské části lze využít WebSocket nebo WebRTC.

Nakonec jsem otestoval rychlost provádění změn v simulaci a efektivitu snížení objemu posílaných dat, stará-li se o jejich synchronizaci NewSync v porovnání s aplikací, která by stejný stav simulace a na něm prováděné změny rozesílala kompletními aktualizacemi ve formátu JSON.

Testování prokázalo, že posílání stavu pomocí rozdílových aktualizací je

velice efektivní technika synchronizace dat. Časová investice zprovoznit tento způsob synchronizace se pro aplikace se sdíleným stavem, jež je většího rozsahu, a který se musí pravidelně posílat klientům, vyplatí. Vždy ale musíme brát v potaz případ použití a délku životnosti dat. Čím větší část stavu aplikace je spíše statického charakteru s dlouhou životností, tím více je přechod na rozdílové aktualizace efektivní.

Příloha A

Zdrojový kód

Zdrojový kód je obsažen v příloze v archivu *sourcecode.zip*. Dostupný je i v GitHub repositáři na url `https://github.com/neuvimat/newsync`. Složky ve zdrojovém kódu mají následující strukturu:

```
\dist     - složka pro výsledný transpilovaný kód demo aplikace
\lib      - zdrojový kód frameworku
\perftest - výkonnostní testy
\src      - zdrojový kód demo aplikace
\test     - transpilované unit testy
\test_src - zdrojové soubory unit testů
```

Součástí zdrojového kódu je dokumentace formou JSDoc komentářů. V repositáři je dále *readme.md* soubor obsahující ukázky kódu, příklady použití frameworku a instrukce pro programátory, jak spustit demo aplikaci.

Příloha B

Výsledky měření

Podrobnější výsledky měření prezentovaných v kapitole 6 Porovnání formátů v prostředí JavaScript a 9 Testování frameworku NewSync, včetně dalších statistických údajů, jako minimum, maximum a interval spolehlivosti, jsou k dispozici v přiložených souborech *formatTests.xlsx*, respektive *demoTests.xlsx*.

Příloha C

Literatura

[1] Stack overflow trends. https://insights.stackoverflow.com/trends?tags=json%2Cxml%2Ccsv%2Csoap%2Cavro. Zobrazeno: 7.1. 2022.

[2] Json. https://www.json.org/json-cz.html. Zobrazeno: 7.1. 2022.

[3] Bson (binary json): Faq. https://bsonspec.org/faq.html. Zobrazeno: 7.1. 2022.

[4] Messagepack specification. https://github.com/msgpack/msgpack/blob/master/spec.md. Zobrazeno: 24.4. 2022.

[5] Cbor — concise binary object representation | overview. https://cbor.io/. Zobrazeno: 8.1. 2022.

[6] Rfc 8949 - concise binary object representation (cbor). https://datatracker.ietf.org/doc/html/rfc8949. Zobrazeno: 25.4. 2022.

[7] Messagepack vs cbor. https://diziet.dreamwidth.org/6568.html. Zobrazeno: 8.1. 2022.

[8] Internet assigned numbers authority. https://www.iana.org/. Zobrazeno: 28.4. 2022.

[9] Concise binary object representation (cbor) tags. https://www.iana.org/assignments/cbor-tags/cbor-tags.xhtml. Zobrazeno: 28.4. 2022.

[10] @shelacek/ubjson - npm. https://www.npmjs.com/package/@shelacek/ubjson. Zobrazeno: 8.1. 2022.

[11] Thoughts on es6 proxies performance. https://thecodebarbarian.com/thoughts-on-es6-proxies-performance. Zobrazeno: 9.1. 2022.

[12] Long polling - concepts and considerations. https://ably.com/topic/long-polling#how-does-long-polling-work. Zobrazeno: 28.4. 2022.

[13] Questions about firebase's service. https://groups.google.com/g/firebase-talk/c/Doi4MP_A81k. Zobrazeno: 28.4. 2022.

[14] Best practices for cloud firesotre | firebase documentation. https://firebase.google.com/docs/firestore/best-practices. Zobrazeno: 10.1. 2022.

[15] Pricing | firestore | google cloud. https://cloud.google.com/firestore/pricing. Zobrazeno: 28.4. 2022.

[16] Is cloud storage gdpr compliant? - harper james. https://harperjames.co.uk/article/cloud-storage-and-gdpr/. Zobrazeno: 28.4. 2022.

[17] Introduction | meteor guide. https://guide.meteor.com/. Zobrazeno: 9.1. 2022.

[18] Couchbase documentation | couchbase docs. https://docs.couchbase.com/home/index.html. Zobrazeno: 10.1. 2022.

[19] Github - forbesmyester/syncit: Syncit is a library to enable you to easily add synchronization to your (offline / phonegap) web apps. https://github.com/forbesmyester/SyncIt. Zobrazeno: 9.1. 2022.

[20] sync-it npm. https://www.npmjs.com/package/sync-it. Zobrazeno: 9.1. 2022.

[21] performance.now() - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now#reduced_time_precision. Zobrazeno: 28.4. 2022.

[22] Real-time text, sip and webrtc | meetecho blog. https://www.meetecho.com/blog/realtime-text-sip-and-webrtc/. Zobrazeno: 2.5. 2022.

[23] Rfc 6455: The websocket protocol. https://www.rfc-editor.org/rfc/rfc6455. Zobrazeno: 2.5. 2022.

[24] Rfc 3629 - utf-8, a transformation format of iso 10646. https://datatracker.ietf.org/doc/html/rfc3629. Zobrazeno: 2.5. 2022.

[25] Looking at all 13 javascript proxy traps | digitalocean. https://www.digitalocean.com/community/tutorials/js-proxy-traps. Zobrazeno: 3.5. 2022.