FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Webová vzdálená správa
Student:	Filip Dolník
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Cílem práce je realizovat softwarové řešení pro vzdálenou správu webových aplikací. Tuto správu budou využívat provozovatelé (například helpdesk, vývojáři apod) pro vzdálené ovládání webové aplikace na na klientském (vzdáleném) zařízení.
Postupujte v následujících krocích:

1. Analyzujte současné možnosti vzdálené správy webových aplikací.
2. Na základě analýzy navrhněte vhodné vlastní řešení, zaměřené na webové aplikace.
3. Realizujte funkční prototyp navrženého řešení.
4. Realizovaný prototyp podrobte vhodným testům.
5. Upravte prototyp dle zjištěných nedostatků.
6. Pokuste se nasadit řešení do minimálně jedné reálné webové aplikace.
7. Zhodnoťte výsledné řešení, navrhněte možné úpravy do budoucna.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 6. února 2019

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství

Bakalářská práce

Webová vzdálená správa

Filip Dolník

Vedoucí práce: Ing. Jiří Hunka

15. května 2019

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Jiřímu Hunkovi za čas a ochotu, kterou mi věnoval při tvorbě této práce. Také děkuji za to, že vymyslel toto zajímavé téma a umožnil mi ho pod jeho vedením realizovat.

Bc. Oldřichovi Malcovi chci poděkovat za pomoc s LaTeX. Děkuji Bc. Pavlovi Kovářovi a Ing. Filipovi Glazarovi za asistenci při nasazení výsledného řešení do produkčního prostředí. Dále také děkuji kolegům z práce, jmenovitě Tadeáši Křížovi a Mgr. Bc. Michaelu Chlubnovi, za zkušenosti, které jsem získal při práci s nimi, a také za technickou a morální podporu.

Na závěr děkuji své rodině a dalším přátelům za podporu během studia a tvorby této práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 15. května 2019 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

DOLNÍK, Filip. *Webová vzdálená správa*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce se zabývá vzdálenou správou webových aplikací. Cílem práce je vytvořit návrh a na jeho základě prototyp, který má umožnit správci vzdáleně ovládat webovou aplikaci. Při návrhu řešení je hlavní důraz kladen na to, aby uživatel nemusel instalovat a používat žádný dodatečný software. V prostředí webu se o takovéto vzdálené správě hovoří jako o co-browsingu.

V teoretické části práce jsou popsány vlastnosti co-browsingu a také různé metody, jak lze realizovat. Z nich jsem zvolil metodu *synchronizace výstupu JavaScriptového enginu*. Hlavním rozdílem oproti jiným existujícím řešením je využití částečné *peer-to-peer* architektury ke zlepšení odezvy.

Výsledkem práce je prototyp umožňující správci ovládat uživatelovu webovou stránku. V práci je popsána architektura prototypu a principy na jakých funguje. Dále jsou zde návrhy, jak vyřešit jeho současná omezení. Tyto informace lze použít při tvorbě plnohodnotného nástroje pro vzdálenou webovou správu.

Klíčová slova co-browsing, vzdálená správa, vzdálená podpora, vzdálená plocha, sdílení obrazovky, vzdálené ovládání, webová aplikace

Abstract

This thesis deals with remote administration of web applications. The aim of this thesis is to create a draft and a prototype based on it, which will allow the administrator to remotely control the web application. The main focus is on ensuring that no additional software is needed to be installed and used by the user. In the web environment, such remote administration is referred to as co-browsing.

The theoretical part of this thesis describes the properties of co-browsing as well as various methods of how to realize it. I chose *JavaScript engine output synchronization* method. The main difference from other existing solutions is the use of partial *peer-to-peer* architecture to improve response time.

The result of this thesis is a prototype allowing the administrator to control the user's website. The thesis describes the prototype architecture and the principles on which it works. There are also suggestions on how to solve its current limitations. This information can be used to create a full-featured remote web administration tool.

Keywords co-browsing, remote administration, remote support, remote desktop, screen sharing, remote control, web application

Obsah

Seznam ukázek kódu

Seznam obrázků

Úvod

Webové aplikace jsou velmi oblíbené mezi vývojáři i uživateli. Pro vývojáře představují jednoduchý způsob, jak zprostředkovat počítačový program široké veřejnosti. Uživatel na druhé straně může aplikaci používat téměř bez jakýchkoliv odborných znalostí. Jako všechny počítačové programy i webové aplikace obsahují chyby. Často také není jejich uživatelské rozhraní příliš srozumitelné. Proto jejich správci poskytují uživatelům podporu v případě, že se setkají s problémem, který oni sami neumí vyřešit.

Práce bude sloužit správcům webových aplikací, kteří chtějí poskytovat na dálku podporu svému produktu. Často ale nemohou nebo nechtějí použít řešení, které vyžaduje po uživateli instalaci externího programu. Právě instalace takového programu je pro méně zkušeného uživatele velmi obtížná. Může být pro něho dokonce i nemožná, pokud například používá mobilní telefon nebo firemní počítač.

S tímto tématem mě seznámil můj pozdější vedoucí Ing. Jiří Hunka, který potřeboval vyřešit přesně tento problém pro aplikaci OptiLynx [1]. Téma jsem přijal hlavně proto, že v případě úspěchu bude výsledek použitelný v praxi. Také se mi líbilo, že se jedná o poměrně těžké zadání, které poskytuje velký prostor pro vymyšlení vlastního řešení.

Hlavním cílem práce je navrhnout, jak realizovat vzdálenou správu webových aplikací, tak aby uživatel nemusel používat žádný další software. Realizovatelnost návrhu má být demonstrována vytvořením prototypu. Na základě testování prototypu v reálné aplikaci je třeba vyhodnotit, jak je návrh a prototyp v praxi použitelný, a navrhnout možné úpravy do budoucna. V rámci

teoretické části práce je třeba analyzovat existující nástroje umožňující vzdálenou správu a prozkoumat technologie, které jsou potencionálně využitelné v praktické části.

V první kapitole se zabývám analýzou zadání. Dále zde popisuji vzdálenou webovou správou, co-browsing a další technologie, které jsem použil při návrhu vlastního řešení. V druhé kapitole je návrh finální architektury řešení. V další kapitole je zachycen postup návrhu této architektury a tvorby prototypu. Také v ní uvádím vybrané implementační detaily a různé problémy, na které jsem v průběhu vývoje narazil. V poslední kapitole hodnotím, jak dobře funguje prototyp v reálné aplikaci. Také je zde seznam možných budoucích vylepšení prototypu.

Kapitola 1

Analýza

V této kapitole se nejdříve podívám na existující nástroje umožňující vzdálenou správu. Poté vysvětlím pojem *co-browsing* a uvedu jeho možná komerční využití. Na to navážu přehledem různých způsobů realizace co-browsingu spolu s popisem webových komunikačních protokolů. Dále představím existující nástroje pro co-browsing. V poslední sekci popíšu funkční a nefunkční požadavky kladené na vlastní řešení. Také uvedu jeho případy užití.

Na začátek potřebuji vysvětlit význam několika pojmů, které používám ve zbytku textu. Jako *uživatele* nazývám lokálního účastníka vzdálené správy. Vzdálenému účastníkovi, který se připojuje k uživateli, říkám *správce*. Uživatele nebo správce označuji jako *klienta*. Podle kontextu myslím těmito názvy buďto člověka obsluhujícího počítač, nebo program na tomto počítači.

1.1 Vzdálená správa

Vzdálená správa umožňuje správci ovládat počítač na dálku přes internet. Program pro vzdálenou správu by tedy měl umět alespoň sdílet obrazovku a vykonávat u uživatele akce provedené správcem. Kromě toho může mít celou řadu dalších funkcí, jako je například podpora pro klávesové zkratky, přenos souborů nebo videohovor.

Pro praktickou část není klasická vzdálená správa příliš zajímavá. Žádný program, který ji realizuje, totiž nesplňuje požadavek, aby uživatel nemusel používat jiný software než webový prohlížeč. Přesto jsem v rámci práce několik takových programů testoval. Na začátku jsem si totiž potřeboval udělat

představu o tom, co umí a jak fungují. Takto získané znalosti jsem později využil při vytváření požadavků na vlastní řešení.

Při testování každého programu jsem se zajímal o jeho instalaci a spuštění. Právě to je pro běžného uživatele největší problém. Dále jsem zkoumal postup nutný pro navázání spojení, a to z toho důvodu, že i v mém programu musím tento problém nějak řešit. Základní funkce jsem testoval při práci s webovými stránkami, u kterých jsem sledoval hlavně odezvu a celkovou plynulost. Neprováděl jsem žádné exaktní měření, šlo mi spíše o to, zda vysoká odezva nebrání v práci.

1.1.1 TeamViewer

TeamViewer [2] je jeden z nejznámějších programů pro vzdálenou správu. Podporuje celou řadu platforem včetně mobilních telefonů. Pro nekomerční použití je zdarma, pro komerční použití je potřeba měsíčně placená licence.

Instalační program lze stáhnout z webových stránek. Postup instalace je stejný jako u běžného programu. Zde je potřeba podotknout, že na macOS je nutné v nastavení povolit speciální oprávnění na ovládání počítače (to je ale potřeba u všech zkoušených programů). Pro uživatele macOS je proto instalace ještě o něco složitější. Musí udělat navíc několik netriviálních kroků, při kterých potřebuje administrátorské heslo.

Spojení je možné navázat dvěma různými způsoby. V tom prvním je potřeba zadat číselný kód (nebo vybrat již známý počítač) a heslo od uživatele. Poté ještě musí uživatel potvrdit v dialogu, že chce povolit příchozí spojení. Tato varianta je vhodná, pokud je u obou počítačů člověk. Pokud u druhého počítače nikdo není, je možné použít druhý způsob. K tomu je potřeba na vzdáleném počítači nastavit, aby povolil připojení od známého počítače i bez hesla a potvrzení.

Samotné ovládání funguje bez problémů, ale z odezvy je poznat, že prohlížeč neběží na lokálním počítači. Nejedná se o velký problém, ale například psaní delšího textu není pro náročnějšího uživatele příliš příjemné.

1.1.2 AnyDesk

AnyDesk [3] je jednodušší alternativou k TeamVieweru. AnyDesk opět podporuje všechny běžné platformy i mobilní telefony. Způsob licencování je také stejný.

Na Windows má AnyDesk výhodu v podobě snadného zprovoznění. Program totiž není nutné instalovat, stačí stáhnout a spustit. Na macOS je opět potřeba přidat oprávnění.

Způsob práce je v podstatě stejný jako u TeamVieweru. Akorát není třeba zadávat heslo, pokud spojení potvrdí uživatel. V opačném případě je heslo potřeba. Také pocit z práce je v podstatě stejný, latence způsobená přenosem dat je srovnatelná.

1.1.3 Chrome Remote Desktop

Chrome Remote Desktop [4] jsem zkoušel z toho důvodu, že se jedná o plugin do webového prohlížeče Chrome [5]. Instalace by tedy mohla být potenciálně snazší než u klasických programů. Tento program poskytuje pouze základní funkcionality, jako je sdílení obrazovky, ovládání myši a psaní na klávesnici.

Přestože se jedná o plugin, je instalace složitější než v případě TeamVieweru. Kromě pluginu je totiž potřeba na vzdáleném počítači stáhnout a nainstalovat dodatečný software (v případě macOS opět povolit v nastavení). Místo jednoho programu, tak musí uživatel v podstatě nainstalovat dva.

Další práce s programem je opět skoro stejná. K přihlášení se používá kód a lze si zvolit heslo, není ale potřeba potvrzovat příchozí spojení.

1.1.4 Shrnutí

Testováním výše zmíněných programů jsem si ověřil význam podmínky o neinstalování externího softwaru. Pro běžného uživatele opravdu nemusí být jednoduché takový software zprovoznit.

Z různých způsobů pro navázání a potvrzení spojení mi přijde nejvhodnější ten, při kterém uživatel příchozí spojení pouze potvrdí. Správce tedy vybírá uživatele podle identifikátoru a nepotřebuje od něho heslo. Heslo slouží spíše k omezení neautorizovaných pokusů o připojení, které by uživatel mohl omylem schválit. Vzhledem k účelu mé práce se dá předpokládat, že správce je důvěryhodná osoba. Mělo by tedy stačit zajistit jeho autentizaci. Stejně tak předpokládám, že u počítače uživatele bude vždy člověk (jinak by mu nemělo smysl poskytovat podporu). Z toho důvodu mi zbylé možnosti potvrzení spojení nepřijdou jako užitečné.

Poslední věcí, která mě při testování zajímala, byla odezva. Ta byla u všech řešení srovnatelná. V podstatě není možné ji zlepšit, protože je způsobená odezvou sítě, po které musí být přeneseny příslušné informace tam a zpět. U pohybu kurzoru a klikání to ještě není takový problém. Při psaní na klávesnici ale dochází ke klasickému efektu pomalého počítače, kdy člověk píše rychleji, než počítač zobrazuje text. To v případě překlepu znamená nutnost vracet se o několik znaků zpět. Z tohoto důvodu se chci v praktické části podívat, zda nelze tento problém v prostředí webu nějak maskovat.

1.2 Co-browsing

V [6] je co-browsing (*collaborative browsing*) popsán jako společné prohlížení webové stránky více lidmi na dálku. Hlavní výhodou co-browsingu je, že využívá pouze webový prohlížeč. Splňuje tedy přesně všechny požadavky kladené na praktickou část. Budu se proto ve zbytku analýzy věnovat výhradně jemu.

Není snadné přesně říct, co vše by měl software splňovat, aby se jednalo o co-browsingové řešení. Z popisu v [7] lze vyvodit, že by měl alespoň sdílet stránku mezi klienty a také umožnit správci naznačit uživateli, jaké akce má provést. Není zde explicitně napsáno, že by měl mít správce možnost webovou stránku sám ovládat. Přesto všechny nástroje, které jsem zkoušel v (1.4), toto částečným způsobem umožňují.

1.2.1 Principy funkce webového prohlížeče

Webový prohlížeč je poměrně složitý program. Proto jeho popis zjednoduším na minimum, které je nutné k pochopení co-browsingu. Pro více informací doporučuji [8]. Odtud jsem také čerpal informace při psaní následujícího textu.

Proces zobrazení stránky Při zobrazení webové stránky začne prohlížeč tím, že načte data z příslušné URL (adresy). Data poskytuje webový server ve formě HTML. Prohlížeč provede syntaktickou analýzu HTML, na základě které vytvoří tzv. DOM. DOM je stromová struktura, která obsahuje objekty reprezentující HTML elementy. Po vytvoření DOM dojde k jeho spojení se styly

získanými z CSS. Vzniklá struktura se poté použije pro výpočet pozic objektů a jejich finální vykreslení na obrazovku.

Vedlejší činnosti prohlížeče V průběhu vytváření DOM může prohlížeč narazit na elementy nebo jejich atributy, které mají speciální význam. U těchto elementů musí prohlížeč udělat nějakou akci navíc. Například u elementu *img* je zapotřebí stáhnout ze zadané URL obrázek, který se poté zobrazí.

CSS Některé elementy a atributy umožňují vložit do stránky CSS. Příkladem je třeba element *style*. CSS je také nutné syntakticky analyzovat. Výsledkem analýzy je sada pravidel stylů (*style rules*), které ovlivňují vzhled stránky. Ty se poté použijí při již zmíněném výpočtu pozic a vykreslování objektů.

JavaScript Dalším důležitým elementem je *script*, který obsahuje JavaScriptový kód. Ten je interpretován pomocí *JavaScriptového enginu*. Úkolem enginu je za běhu provést syntaktickou analýzu JavaScriptu a vykonat jím požadované akce. Důležité je, že ve svém vnitřním stavu má uložené veškeré proměnné a objekty, se kterými JavaScript pracuje. JavaScript umožňuje vývojářům různými způsoby měnit chování webové stránky. Je například možné upravit její obsah nebo měnit a blokovat uživatelské vstupy.

Atributy a vlastnosti objektů Objekty obsažené v DOM mají své *atributy* a *vlastnosti*, které upravují chování a vzhled těchto objektů. Atributy je možné přiřadit objektu přes jeho element v HTML, zatímco vlastnosti lze nastavit pouze přes JavaScript. Atributy a vlastnosti se navzájem ovlivňují, ale nemusí si vždy odpovídat. Existence vlastností má za následek, že obsah webové stránky není určený čistě HTML. Například textový obsah elementu *input* je uložený pouze ve vlastnosti *value*. Existuje sice atribut *value*, ale ten určuje výchozí obsah, nikoliv ten aktuální. Více si je možné přečíst v [9].

1.2.2 Vlastnosti a omezení

Co-browsing má v podstatě jedinou (již zmíněnou) výhodu oproti vzdálené správě. Zato nevýhod existuje celá řada. Důvodem je nutnost použít pro jeho realizaci JavaScript integrovaný do webové stránky. Z toho plyne velké množství různých omezení. Většinu z nich nemůže navíc žádné řešení spolehlivě obejít.

Jejich nejčastější příčinou je sada bezpečnostních opatření webového prohlížeče.

Při co-browsingu je nutné použít JavaScript k ovládání webové stránky a k získání potenciálně citlivých informací. Což je přesně to samé, co by chtěl udělat škodlivý kód. Prohlížeč před tímto brání uživatele například aplikováním *Same-origin policy* [10] nebo blokováním nebezpečných akcí.

V některých případech nejde dané omezení vyřešit jenom kvůli tomu, že není žádné vhodné API. Jeho existence by ale neměla mít na bezpečnost žádný vliv. Zde je dobrou zprávou, že se webové prohlížeče neustále vyvíjejí a přibývají v nich nové funkcionality. Z těchto důvodů si myslím, že v budoucnu bude možné některé problémy řešit lépe.

Níže popíšu konkrétní problémy, které jsem identifikoval ať už během analýzy, návrhu nebo samotné implementace. Není bohužel jednoduché na všechny přijít prostým pohledem do dokumentace. V ní se většinou píše, co je možné udělat, ale málokdy je napsáno, co možné není. Kvůli tomu také nemám stoprocentní jistotu, že uvedené problémy nelze vyřešit i jinak. Opírám se o to, že jsem lepší způsob nenašel. Navíc ani profesionální produkty, kterým se věnuji v (1.4), lepší řešení nemají.

1.2.2.1 Relativní URL

Webová stránka typicky obsahuje odkazy na další *zdroje* (obrázky, styly, atd.), které jsou umístěné na společném serveru. K jejich identifikaci se často používají relativní adresy. Webový prohlížeč vyhodnocuje relativní adresu vůči adrese stránky, čímž dostane správnou absolutní adresu zdroje. Více informací si je možné přečíst v [11].

Stránka, se kterou pracuje správce, typicky nemá stejnou adresu jakou má stránka uživatele. Z toho důvodu jsou u správce všechny relativní adresy neplatné. Řešení je teoreticky poměrně jednoduché: Stačí nahradit relativní adresy těmi absolutními. Prakticky je poměrně složité všechny takové adresy v HTML najít. Navíc v případě, že je k nim přistupováno dynamicky pomocí JavaScriptu, je takové nahrazení nemožné. Pro předejití problémům s relativní adresou, lze upravit webovou aplikaci tak, aby používala pouze absolutní adresy. Existuje také jiné řešení v podobě použití proxy serveru, o kterém podrobněji píšu v (1.3.2).

1.2.2.2 Soukromé zdroje

Pod pojmem *soukromé zdroje* mám na mysli takové zdroje, které jsou serverem poskytnuté pouze omezené skupině klientů. Typickým příkladem jsou třeba data ze serveru, který je přístupný pouze na lokální síti.

Webový prohlížeč správce nemůže takové zdroje získat, čímž dojde k rozdílům v zobrazení webové stránky. Řešením je, aby uživatel tato data přeposlal správci, který je použije namísto původního zdroje.

Zde ovšem přichází do hry *Same-origin policy*. V [10] je popsáno, že z bezpečnostních důvodů není možné, aby JavaScript u uživatele přímo pracoval se zdroji z jiných domén.

Problém ilustruji na stránce s obrázkem z jiné domény, který je vložený pomocí elementu *img*. Obrázek představuje třeba webovou reklamu. JavaScriptem není možné získat obsah obrázku. Uživatel poté nemůže takový obrázek přeposlat správci.

K potlačení *Same-origin policy* slouží tzv. CORS [12]. Pokud server poskytující zdroj zapne CORS, může JavaScript k obsahu zdroje přistupovat, jakoby byl ze stejné domény.

V případě, že zapnutí CORS není možné, je dobré alespoň u správce zobrazit nějaký obrázek signalizující chybu. Je také důležité zachovat stejnou velikost obrázku, aby nedošlo ke změně rozložení stránky. Úplně nejlepším řešením je prostě povolit přístup k soukromým zdrojům i správci. Jenže pokud nelze zapnout CORS, většinou ze stejného důvodu nepůjde ani toto.

1.2.2.3 Cookies

Dle [13] jsou *cookies* nějaká data, která přidělil server v předchozí odpovědi klientovi. Webový prohlížeč klienta si tato data uloží a následně je při každém dalším dotazu pošle zpátky na server. Server díky tomu může identifikovat, že dotaz přišel od toho samého klienta, a upravit tak svoje chování.

Problém s cookies je v podstatě speciálním případem problému se soukromými zdroji. K jeho řešení by teoreticky stačilo přeposlat uživatelovy cookies správci. Správce by s nimi poté mohl běžným způsobem přistoupit ke zdrojům.

Takovéto „zcizení“ cookies je ale z pohledu běžného používání nebezpečné. Proto mu stejně jako v předchozím případě brání *Same-origin policy*. Kvůli ní není možné JavaScriptem číst a zapisovat cookies, které patří k jiné doméně

[10]. Nezbývá tedy nic jiného, než použít stejné řešení se stejnými omezeními, jako v případě soukromých zdrojů.

1.2.2.4 Hover

Hover je pseudo třída v CSS, která se aktivuje, pokud uživatel na příslušný element ukáže kurzorem. Další informace lze najít v [14]. Velmi často se používá k vytvoření víceúrovňového menu, které se postupně zobrazuje.

V současných prohlížečích není možné aktivovat hover pomocí JavaScriptu. To představuje vážný problém, pokud potřebuje správce ukázat uživateli, jak v takovém menu něco najde. Uživatel uvidí pouze pohyb kurzoru bez kontextu daného menu.

Řešením je jedině nahradit ve všech CSS tuto pseudo třídu normální třídou. Tu lze pak JavaScriptem přidělovat jednotlivým elementům podle potřeby. Jedná se o poměrně komplikovaný proces, neboť pro spolehlivou funkci je potřeba pohlídat i dynamicky měněné styly.

1.2.2.5 Vyskakovací okna

Vyskakovací okna (například *alert*) jsou implementována přímo v prohlížečích. Nejsou proto součástí samotné webové stránky. Z toho důvodu není možné je JavaScriptem ovládat. Stejně tak není možné ani zobrazit vyskakovací okno druhému klientovi. Podobně je to i s dialogy pro výběr barvy, data a souboru.

V současné době toto nelze jakkoliv řešit a nepředpokládám, že v budoucnu dojde k nějaké změně.

1.2.2.6 Různý vzhled webových prohlížečů

Každý webový prohlížeč má výchozí styly pro jednotlivé elementy. Jak to tak bývá, neexistuje jednotný standard, jak mají vypadat. Rozdíly můžou být například ve stínech, tvarech, ale hlavně ve velikostech. Přestože jde o pár pixelů, rozdíly se kumulují a ve výsledku může mít stránka úplně jiné rozložení.

Při běžném prohlížení to většinou nevadí. Při nejhorším bude stránka vypadat jinak, než autoři zamýšleli. V co-browsingu ale může dojít k „desynchronizaci“ kurzoru. On sice bude u obou klientů na stejné absolutní pozici, ale vůči elementům na stránce bude jinde. To může vést ke zmatení uživatele, neboť uvidí, že správce kliká na jiné elementy.

K řešení je teoreticky možné přepočítávat pozici kurzoru relativně vůči elementům. Je to ale poměrně složité a zcela jistě to způsobí nepřirozený pohyb kurzoru. Lepší možností je využití *resetovacího CSS*, které sjednotí vzhled stránky. Většina webových stránek již takové CSS stejně používá (pokud ne, přidání je jednoduché), takže nemá význam řešit to na úrovni co-browsingu.

1.2.3 Využití co-browsingu

Běžní uživatelé nemají důvod co-browsing používat. Oproti vzdálené správě má pro ně jenom nevýhody. Jedinou jeho výhodou je absence externího programu. Jenže pokud chce uživatel sám od sebe ovládat vzdálený počítač, tak má motivaci si potřebný program nainstalovat.

Pro firmy provozující webové aplikace je ale situace jiná. Uživatel zpravidla nebude chtít nic stahovat a instalovat. Proces zprovoznění vzdálené správy může být navíc náročnější než samotné řešení problému. V [15] jsou uvedené dva zajímavé scénáře použití ve firemním prostředí, které popisuji níže.

Zákaznický servis V kamenném obchodě může prodejce pomoct zákazníkovi s výběrem zboží a jeho nákupem. Využití co-browsingu umožní prodejci poskytnout stejné služby i na e-shopu. Spíše než na běžné zákazníky cílí toto využití na firmy. To z toho důvodu, že firmy nakupují za vysoké částky a jejich objednávky jsou zpravidla výrazně komplikovanější. Co-browsing také zjednodušuje individuální přístup, který firmy často požadují. Toto neplatí jenom pro klasický e-shop, ale i pro stránky zaměřené na prodej několika produktů nebo služeb.

Interaktivní podpora Webové aplikace mohou být poměrně složité i v případě, že s nimi má pracovat běžný uživatel. V [15] je *e-government* použit jako příklad takového velmi složitého webu. Ten navíc používají běžní lidé neorientující se v jeho problematice. Co-browsing umožňuje zákaznické podpoře lépe řešit uživatelův problém, než při použití běžných postupů, jakým je například telefonní hovor.

1.3 Způsoby realizace co-browsingu

Cílem co-browsingu je zajistit synchronizaci webové stránky mezi dvěma klienty. Jinými slovy oba klienti musí vždy vidět ve svém prohlížeči tu samou stránku. V případě stránky neobsahující JavaScript je situace poměrně jednoduchá: Stačí zajistit synchronizaci DOM mezi oběma klienty. Poté nemůže dojít k porušení její konzistence. V dnešní době ale většina stránek nějaký JavaScript obsahuje.

V okamžiku, kdy do hry vstoupí JavaScript, je řešení podstatně složitější. Synchronizace DOM by sice zajistila stejný vzhled, ale v závislosti na vnitřním stavu JavaScriptového enginu, by se stránka chovala u každého klienta úplně jinak. Ve výsledku by při každé změně docházelo ke kolizi dvou různých stránek. To by sice nevadilo samotnému co-browsingu, ale webová aplikace by byla v podstatě nepoužitelná.

1.3.1 Synchronizace stránky s JavaScriptem

V [6] jsou uvedeny dva rozdílné přístupy pro synchronizaci stránky s JavaScriptem. Každé co-browsingové řešení, které chce takové stránky podporovat, musí jeden z nich implementovat. Oba způsoby mají své výhody i nevýhody a nedává smysl je kombinovat. Z toho důvodu je volba jednoho z uvedených způsobů nejdůležitějším krokem při návrhu co-browsingového řešení.

1.3.1.1 Synchronizace vstupů

Cílem metody synchronizace vstupů je udržet u obou klientů synchronizovaný vnitřní stav JavaScriptového enginu. Díky tomu by se pak stránka chovala u obou klientů stejně. Toho lze dosáhnout synchronizací uživatelských událostí a dat, která prohlížeč získává ze serverů. Tím, že oba JavaScriptové enginy obdrží ve stejném pořadí stejné vstupy (události a data), dojde nepřímo k synchronizaci DOM.

Správná synchronizace dat vyžaduje proxy server. Jeho úkolem je komunikovat s webovými servery a zprostředkovávat získaná data klientům.

JavaScript je tedy vyhodnocován lokálně u obou klientů, díky čemuž není třeba čekat na synchronizaci DOM. To má velkou výhodu v podobě plynulosti stránky. Největší rozdíl je při zobrazování animací a přehrávání videí.

Hlavní nevýhodou tohoto přístupu je vysoká náročnost na implementaci. Je velmi obtížné udržet stejný vnitřní stav enginu. Opětovná synchronizace za běhu je problematická, což znamená, že není možné aktivovat co-browsing bez nového načtení stránky. Toto řešení také neumožňuje, aby klient upravoval stránku prostřednictvím konzole.

1.3.1.2 Synchronizace výstupu

Metoda synchronizace výstupu JavaScriptového enginu přistupuje k problému z opačné strany. Místo snahy o synchronizaci vnitřního stavu dvou enginů, používá pouze jeden. Výsledkem tedy je, že u uživatele běží JavaScript úplně normálně a u správce je zablokovaný. Při tomto přístupu je potřeba přímo synchronizovat DOM se správcem. Od správce se pak posílají vstupy, které se předávají enginu u uživatele. Engine v reakci na ně provede změny na stránce, které se zpátky synchronizují se správcem.

Mezi výhody této metody patří vyšší spolehlivost a snadná opakovaná synchronizace DOM. To také znamená, že při navázání co-browsingu není nutné znovu načítat stránku.

Nevýhodou je větší síťová zátěž kvůli nutnosti synchronizovat navíc celý DOM.

1.3.2 Proxy server

Proxy server slouží jako prostředník mezi klientem a webovými servery. Klienti komunikují pouze s proxy serverem, který jim zprostředkovává požadovaná data. Proxy server si při prvním dotazu načte data z reálného serveru. Tato data si uloží a druhému klientovi poskytne jejich kopii.

Při použití proxy serveru jsou v podstatě zadarmo přesměrovány relativní adresy (1.2.2.1). U správce není třeba tyto adresy nijak upravovat. Relativní adresy vedou automaticky na proxy server, který už si s nimi dokáže poradit. Dochází ale k opačnému problému. Je třeba nahrazovat absolutní adresy, protože ty na proxy server nevedou. Takže ve výsledku proxy server nezjednodušuje práci s adresami.

Proxy server řeší částečně problémy se soukromými zdroji (1.2.2.2), ale ani tak je nedokáže vyřešit úplně. Pouze zajistí, že oba klienti vidí to samé. To ale

neznamená, že zobrazený obsah je správný. Je to z toho důvodu, že proxy server nemůže získat cookies z jiných domén a nemůže přistupovat do lokální sítě.

Proxy server je také možné využít k implementování části logiky potřebné pro synchronizaci stránky. V [6] jsou uvedené možnosti synchronizace popsány ve variantě s proxy serverem i bez něj. Varianta bez proxy serveru v podstatě znamená, že jeho funkcionalita je implementována u uživatele.

Při použití proxy serveru je třeba řešit jeho škálování. S rostoucím počtem klientů bude zapotřebí neustále zvyšovat jeho výkon. Tento problém se týká (byť v menší míře) i varianty bez proxy serveru. Stále totiž může být zapotřebí server, který zprostředkovává komunikaci mezi klienty. To záleží na volbě komunikačního protokolu.

1.3.3 Webové komunikační protokoly

Každé co-browsingové řešení potřebuje nějakým způsobem přenášet informace mezi klienty. Výběr vhodného komunikačního protokolu je důležitým krokem při návrhu řešení, protože významně ovlivňuje celou jeho architekturu. V současné době podporují webové prohlížeče celkem tři komunikační protokoly uvedené níže.

1.3.3.1 HTTP

HTTP je aplikační protokol vytvořený pro přenos dat mezi klientem a webovým serverem. Stejně jako jiné internetové technologie je i HTTP definován v RFC, konkrétně v RFC2616 [16] a dalších.

Protokol je ukázkovým zástupcem architektury *klient-server*. Z toho vyplývá, že komunikaci vždy zahajuje klient tím, že pošle požadavek serveru. Server požadavek zpracuje a pošle klientovi odpověď.

Protokol nemá mechanizmus pro udržení stavu z předchozí komunikace. Z pohledu serveru jsou tedy HTTP dotazy zcela nezávislé na předchozích. Stav je v případě potřeby možné udržet například pomocí cookies.

HTTP neumožňuje navázat přímé spojení mezi dvěma klienty. K tomu je zapotřebí server, který jejich požadavky přeposílá. HTTP ale ze své podstaty neumí zasílat odpovědi ze serveru bez předchozího dotazu od klienta. Proto je třeba ještě řešit, jak samotné přeposlání provést.

Jsou sice způsoby, díky kterým lze takto komunikovat, ale dnes už není moc důvod je používat. K těmto účelům byl totiž vyvinut protokol WebSocket. Starší řešení co-browsingu (například [15]) ale HTTP využívají, protože v době jejich vzniku neexistovala jiná možnost.

1.3.3.2 WebSocket

WebSocket je webový komunikační protokol definovaný v RFC6455 [17]. Slouží jako alternativa k HTTP v situacích, kdy je třeba obousměrně komunikovat mezi klientem a serverem. Jeho standard je po letech vývoje již ustálený a všechny současné webové prohlížeče jej plně podporují [18].

Protokol používá stejně jako HTTP architekturu *klient-server*. API WebSocketů je velmi podobné TCP. Klient i server mají každý svůj příchozí a odchozí socket. To umožňuje serveru zasílat klientovi data v libovolný okamžik. Tyto sockety zůstávají otevřené po celou dobu komunikace, díky čemuž je jednoduché udržet na serveru její stav.

Pro co-browsing jsou WebSockety lepším řešením než HTTP. Stále ale potřebují server jako prostředníka mezi klienty. Pro aplikace vyžadující co nejnižší odezvu je proto vhodnější použít následující protokol.

1.3.3.3 WebRTC

V [19] je WebRTC popsáné jako sada technologií pro komunikaci v reálném čase. WebRTC umožňuje webovým aplikacím zaznamenávat a sdílet zvuk i video. Využití serveru pro zprostředkování komunikace je v takovém případě nepraktické. Proto komunikace mezi klienty probíhá přímo, takzvaně *peer-to-peer*.

WebRTC je stále ve vývoji a jeho API se mění. Organizace W3C vydává dokumenty [20], kterými se postupně snaží o jeho standardizaci. WebRTC z toho důvodu není ve všech prohlížečích podporované stejným způsobem [19].

Přestože původním účelem WebRTC je podpora snadné implementace videohovorů, je pomocí něho možné přenášet i normální data. K tomu slouží API nazvané *RTCDataChannel* [21].

WebRTC je pro co-browsing poměrně zajímavé tím, že je *peer-to-peer*. Při jeho použití se dá očekávat poloviční odezva oproti WebSocketům. Další výhodou je snadné přidání podpory pro videohovor, když už se WebRTC

používá pro přenos dat. Hlavním problémem je zatím nestandardizované API, což značně komplikuje vývoj pro více prohlížečů.

1.4 Existující řešení co-browsingu

K testování existujících co-browsingových řešení jsem zvolil následující produkty: *Acquire* [22], *Surfly* [23] a *Upscope* [24]. Při výběru jsem vycházel z [25]. Všechny produkty jsou zaměřené na firmy a jsou bez výjimky placené. Nabízejí ale zkušební verzi na 14 dní zdarma. Pro její získání je třeba se zaregistrovat na „firemní“ účet a u Upscope dokonce zadat platební údaje.

Při testování mě nejvíce zajímalo, jak se jednotlivé produkty potýkají se standardními problémy v co-browsingu (1.2.2). Proto jsem samotné testování odložil, dokud jsem neměl rozpracovanou implementaci vlastního řešení. V té době jsem již měl většinu těchto problémů identifikovanou. Díky tomu jsem mohl snáze analyzovat, jak produkty interně fungují a v čem spočívají jejich slabiny. Samozřejmě jsem si vyzkoušel i běžné použití, při kterém mě zajímala hlavně uživatelská přívětivost.

Protože uvažuji o možnosti výsledný prototyp z praktické části práce dodělat a komercializovat, nahlížel jsem na tyto produkty i jako na konkurenci. Zkušenosti získané z tohoto testování jsem poté použil při úpravách mého řešení a hlavně při navrhování jeho dalších vylepšení.

K testování jsem použil OptiLynx [1] a několik upravených stránek, které jsem potřeboval k odhadnutí principů funkce jednotlivých produktů. OptiLynx jsem zvolil proto, že se jedná o cílovou aplikaci pro moje řešení. Toto řešení budu nakonec na této aplikaci také testovat v (4.1).

Všechny testované produkty jsou si v některých ohledech velmi podobné. Liší se hlavně v samotném co-browsingu, dodatečných funkcích a jejich cílovým trhem. Nejdříve zde popíšu jejich společné charakteristiky a poté se budu věnovat každému zvlášť.

1.4.1 Obecné vlastnosti

Ke každému řešení existuje administrační web, na kterém lze měnit nastavení účtu, licence, placení, atd. Na webu jsou k dispozici také různé další funkcionality, které se u každého produktu liší. O těch ale nepíšu podrobněji, protože

v podstatě vůbec nesouvisí s co-browsingem. Je to například nastavení chatovacího bota, statistiky návštěvnosti, podpora SSO, atd.

Pokyny k integraci produktu do aplikace jsou uvedené v administraci. Stačí vložit na všechny stránky aplikace krátký JavaScriptový kód, který už si další závislosti natáhne sám. Tento kód je možné nastavit různými parametry, z nichž nejdůležitější je identifikátor účtu. Podle něho produkt pozná, o jakou aplikaci a jakého uživatele se jedná.

Další částí administrace je rozhraní pro správce, které je určené k poskytování podpory uživatelům. Zde jsou vidět aktivní spojení a žádosti o podporu. Uživatelé o ni mohou aktivně požádat skrze tlačítko v dolním rohu stránky.

Co-browsing je ve všech případech řešen metodou *Synchronizace výstupu JavaScriptového enginu* (1.3.1.2) a alespoň částečně s pomocí serveru. Veškerá komunikace mezi klienty musí jít přes tento server, což způsobuje vyšší odezvu.

Podobně řešená je práce s kurzorem myši. Oba klienti vidí kurzor toho druhého. Správce má možnost kreslit do stránky jako na tabuli. Tím lze zvýraznit části stránky nebo navést uživatele k provedení nějaké akce. Žádný z produktů neumí řešit různý vzhled webové stránky (1.2.2.6) napříč prohlížeči. Z toho důvodu není zaručené, že oba klienti vidí kurzor na stejném místě relativně k elementům.

Ceny licencí jsou velmi různé, ale princip licencování je stejný. Licence jsou rozdělené do tří úrovní podle dostupných funkcí. Každá z nich je na určitý počet *agentů* neboli různých správců, kteří mohou obsluhovat uživatele. Typicky je jedna licence pro jednoho agenta. Ceny jsou uvedeny bez daně (není sice uvedeno, ale je to u produktů pro firmy zvykem). Platí se měsíčně a při platbě na rok dopředu je poskytnuta sleva.

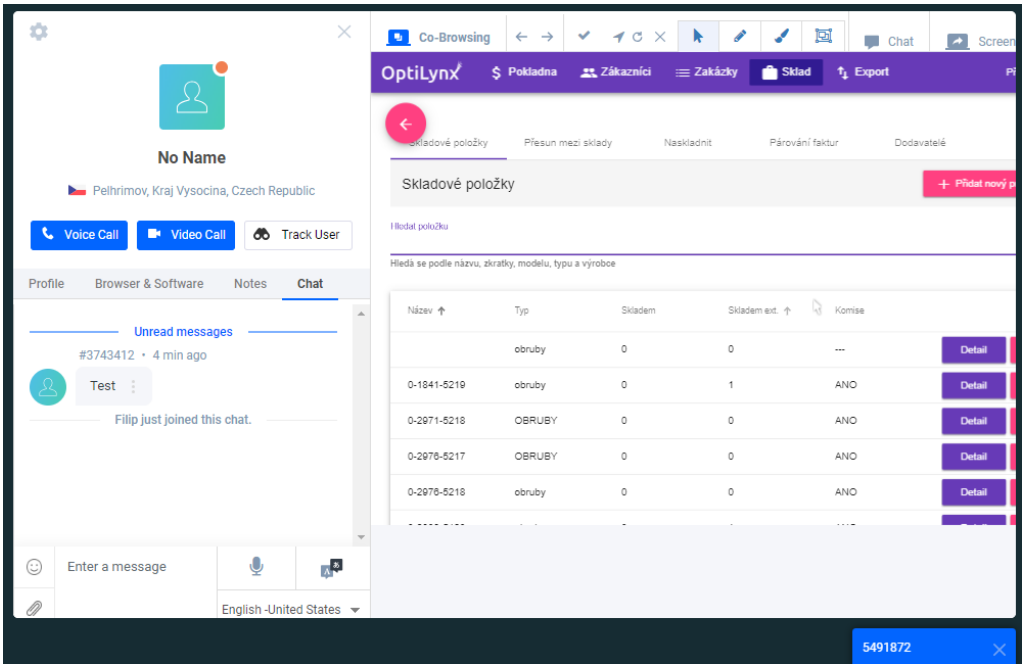
Každý z produktů jsem zkoušel několikrát v různou denní dobu. V průběhu pracovního dne jsem zaznamenal, že načítání stránek u správce trvá výrazně delší dobu (tři až čtyři sekundy) než večer. Také plynulost kurzoru při jeho pohybu byla horší. Z toho usuzuji, že u všech produktů není server dostatečně dobře dimenzovaný.

1.4.2 Acquire

Acquire [22] má z testovaných produktů pocitově nejlepší co-browsing. Jednoznačně obsahuje nejvíce podpůrných funkcí. Také bych ho označil za nejlepší po stránce designu a uživatelské přívětivosti.

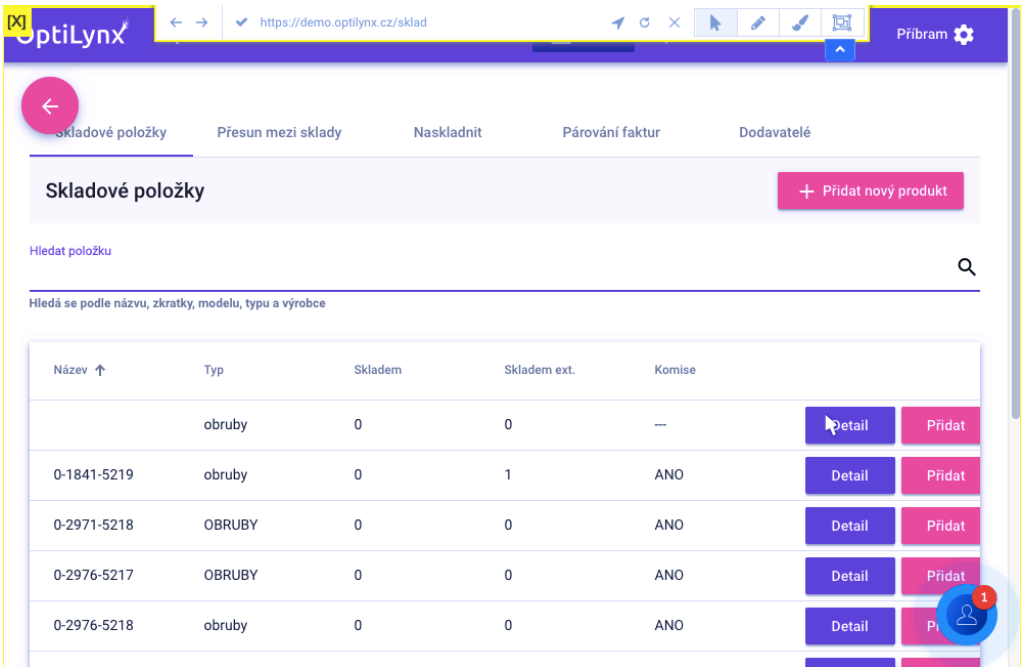
Kvalita a funkce jsou vykoupeny vysokou cenou. Nejlevnější licence stojí měsíčně 35 $ a umožňuje pouze komunikovat s uživatelem a sdílet jeho stránku (bez ovládání). Za 400 $ už je k dispozici plnohodnotný co-browsing a 10 agentů. Pro velké firmy je určená poslední licence bez jakýchkoliv omezení s individuální cenou.

Po načtení webové aplikace se uživateli zobrazí ikona, která otevírá chat. Pokud uživatel potřebuje pomoc, napíše správci zprávu. Tomu se v administraci zobrazí příchozí spojení. Po jeho otevření může odepisovat uživateli, zahájit s ním videohovor nebo ho požádat o zapnutí co-browsingu. Pokud uživatel žádost příjme, přepne se stránka do režimu vzdáleného přístupu. Pohled správce je zobrazen na obrázku (1.1). Pohled uživatele je pak na obrázku (1.2).



Obrázek 1.1: Acquire – pohled správce

Správce vidí stránku uživatele značně zmenšenou, protože kromě ní vidí i řadu nástrojů a chat. Z toho důvodu nemusí být obsah stránky dobře čitelný. Postranní okno navíc nejde zmenšit ani skrýt. Správce může rychle přepínat mezi více uživateli pomocí modrých záložek ve spodní části obrazovky. Ve vrchní části obrazovky je paleta nástrojů. V ní lze přepínat mezi klasickým ovládáním, kreslením nebo zvýrazňováním. Zajímavostí je, že paletu nástrojů má k dispozici i uživatel a tedy i on může například kreslit.



Obrázek 1.2: Acquire – pohled uživatele

Při testování Acquire jsem si nejprve myslel, že nepodporuje ovládání klávesnicí. Později jsem ale zjistil, že tomu tak není a ovládání funguje v jednoduchých stránkách dobře. Problém byl v JavaScriptu, který ovládal zkoušený formulář. Acquire si z nějakého důvodu s tímto JavaScriptem nerozumí. Můj předpoklad je, že Acquire nesimuluje korektně JavaScriptové události. Stejný problém jsem pozoroval i při ovládání myší.

Acquire příliš neřeší omezení zmíněné v (1.2.2). Relativní adresy (1.2.2.1) jsou sice nahrazeny za absolutní, ale ostatní problémy se soukromými zdroji (1.2.2.2) nejsou nijak ošetřeny. Také *hover* (1.2.2.4) nefunguje úplně dobře. Z mě neznámého důvodu nefunguje sdílení vnořených *iframe*, přestože pro to není zjevný důvod a ostatní produkty tento problém nemají. Acquire synchronizuje DOM pouze směrem od uživatele ke správci. Správce tedy nemůže editovat HTML, protože tím dojde k rozbití stránky.

Funkcí, která stojí za zmínku, je sdílení souborů. V co-browsingu nelze ovládat dialog pro nahrání souboru do webové aplikace (1.2.2.5). Acquire proto nabízí sdílení souborů, přes které správce odešle uživateli potřebný soubor. Uživatel může následně tento soubor nahrát do stránky od sebe.

1.4.3 Surfly

Surfly [23] je ze všech produktů nejjednodušší. Obsahuje pouze funkce, které přímo souvisí s poskytování podpory zákazníkům. Nezaměřuje se pouze na firmy s vlastními aplikacemi. Druhou cílovou skupinou jsou lidé, kteří potřebují spolupracovat v rámci jedné aplikace, která není nutně jejich.

Výhodou Surfly je nízká cena. Základní licence stojí pouze 19 $. Není ale určená pro integraci do webové aplikace. Slouží pro týmovou spolupráci. Druhá licence stojí 31 $ a je již určena pro firemní zákazníky. Poslední varianta za 62 $ je cílená na velké firmy a umožňuje omezovat privilegia správců.

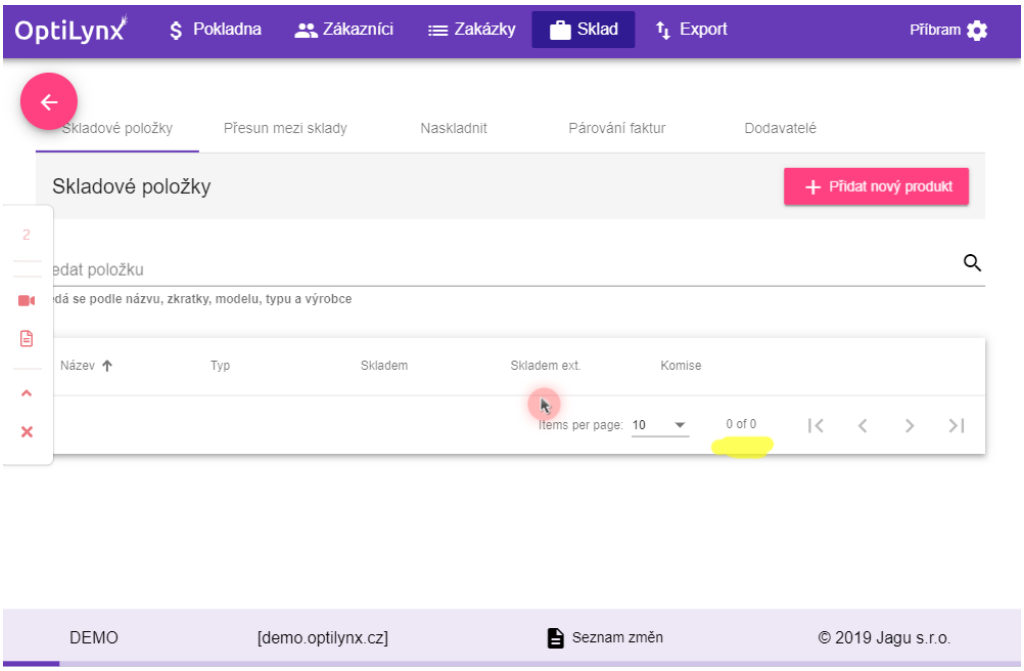
Poměrně zajímavě funguje připojení. Klasický způsob zahrnuje vložení JavaScriptového kódu do webové aplikace. Postup je poté stejný jako u Acquire. Uživatel ale nemusí ještě jednou potvrzovat spojení. Po té co se správce rozhodne připojit, je spojení navázáno okamžitě. Chat a videohovor je dostupný až po zapnutí co-browsingu.

Druhý způsob připojení je unikátní pro Surfly. Z libovolné URL je možné vygenerovat link a ten poslat protistraně. Po jeho otevření se okamžitě aktivuje co-browsing. Tím je možné použít Surfly s libovolnou webovou aplikací, bez nutnosti upravovat její kód.

Při testování v OptiLynx jsem zjistil, že Surfly není s touto aplikací kompatibilní. Ihned po spuštění co-browsingu dojde k desynchronizaci stránky. Rozdíl je možný vidět na obrázcích (1.3) a (1.4). Nikde jinde jsem se s tímto nesetkal. Podle logu z konzole je spouštěčem tohoto problému dynamické načítání položek ze serveru, což odpovídá i stavu stránky. Z toho usuzuji, že nedochází ke správnému přesměrování dynamických dotazů na proxy server, který Surfly používá.

Proxy server je použitý k zajištění všech funkcionalit. To s sebou nese nevýhody již zmíněné v (1.3.2). Při spojení dojde ke znovu načtení stránky, což může smazat její stav, se kterým třeba uživatel potřeboval pomoct. V případě použití vygenerované URL nemůže dojít k předání cookies na proxy server. Z pohledu uživatele se proto stránka tváří, jako by byla otevřena v anonymním prohlížeči. Kvůli tomu se uživatel musí znovu přihlásit.

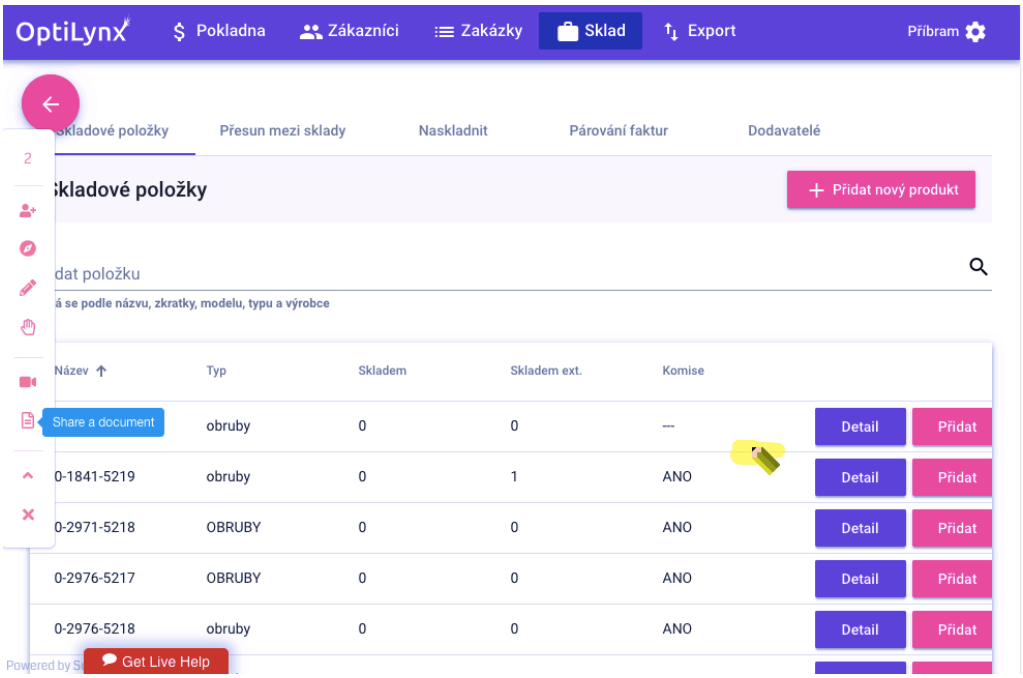
Ovládání stránky není na rozdíl od Acquire problematické. Surfly v tomto ale trochu podvádí. V ovládání se totiž musí klienti střídat. Dochází k synchronizaci vždy jenom jedním směrem. Druhý klient se může pouze dívat a kreslit.



Obrázek 1.3: Surfly – pohled správce

Co se týká omezení z (1.2.2), tak Surfly umí jako jediné dobře řešit *hover* (1.2.2.4). Přítomnost proxy serveru také sjednocuje obsah ze soukromých zdrojů (1.2.2.2). Ve zbytku problémů je na tom stejně jako ostatní. Objevuje se tu ale jiný problém. Surfly neumí detekovat (a synchronizovat) změny vlastností objektů způsobené JavaScriptem. Aplikace, které využívají JavaScript například k úpravám formulářů, nemusí být proto zobrazené u protistrany ve stejném stavu.

Zajímavou funkcí v nejdražší licenci je nastavení privilegií správců. Předpoklad je takový, že velká firma nemusí mít naprostou důvěru v to, že správce nezneužije citlivé údaje uživatelů. K těm by teoreticky mohl mít v průběhu co-browsingu přístup. Stejně tak je možné omezit aktivaci určitých prvků nebo klikání na tlačítka. Neměl jsem v rámci zkušební verze k této funkci přístup. Nejsem si ale jist, jak efektivní toto omezení je. Pokud nedojde k naprostému zablokování vzdáleného ovládání, může správce stále editovat HTML. Tím by do něho mohl podstrčit i element *script* s JavaScriptovým kódem, kterým by požadovaná data stejně získal.



Obrázek 1.4: Surfly – pohled uživatele

1.4.4 Upscope

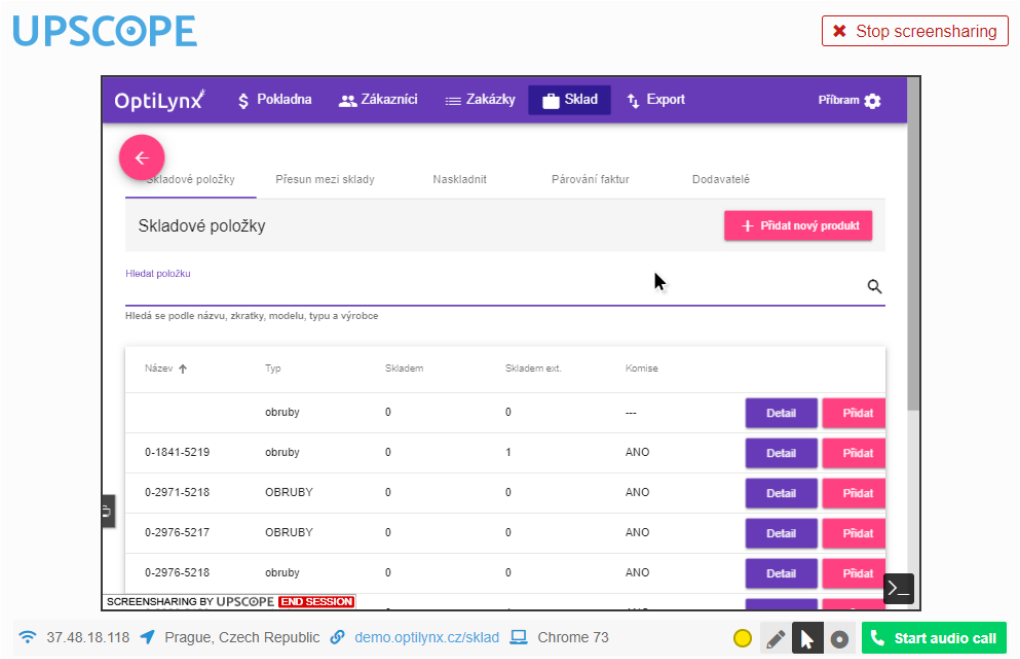
Upscope [24] je funkcemi něco mezi Acquire a Surfly. Necílí ale nutně na stejné zákazníky. Autoři sami popisují svůj produkt v [25]. Zde píšou, že Upscope je určený pro firmy, které potřebují bezpečný co-browsing. Mezi jejich zákazníky jsou banky a zdravotní zařízení.

Cena za základní licenci obsahující v podstatě pouze co-browsing je 14 €. Prostřední licence je za 28 € a umožňuje používat vzdálenou konzoli a lépe konfigurovat správce. Cena za plnou licenci je individuální a je opět určená pro velké firmy. Výše uvedené částky jsou ale zavádějící, protože v popisku pod nimi je uvedené, že je nutné koupit minimálně tři licence. Což rázem činí Upscope pro nejmenší firmy výrazně dražším než Surfly. Netradiční je také účtování dodatečných poplatků za hovor.

Oproti zbylým produktům není vlastně vůbec jasné, kolik musí zákazník zaplatit. Cena za hovor není na stránkách napsaná vůbec. Tu lze zjistit pouze po vytvoření účtu a je 7 $ za 5 hodin. Ještě zajímavější je, že čím víc hodin se provolá, tím je volání dražší.

Upscope nemá vlastní chat ani videohovor. Spoléhá na integraci s dalšími službami. To je pravděpodobně také důvod, proč je vlastní funkce volání placená. Bude totiž řešená přes nějakou IP telefonii.

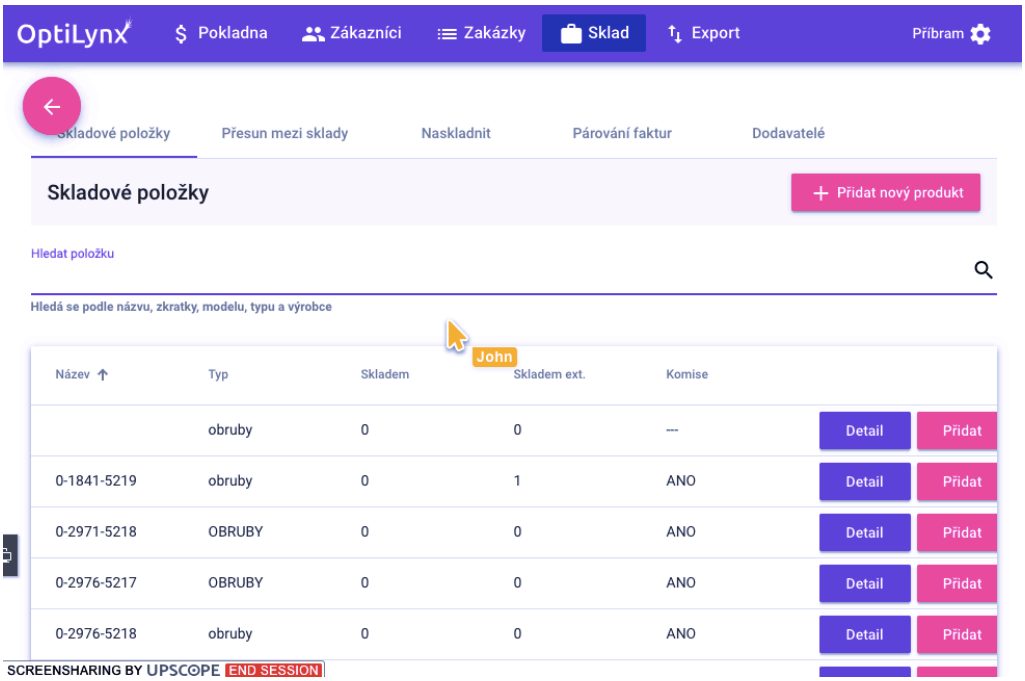
Způsob připojení je identický tomu u Acquire. Podobný je také samotný co-browsing. Na obrázku (1.5) je vidět, že stránka je také zmenšena kvůli liště s nástroji. Není to tak špatné jako u Acquire, protože zde není chat. Uživatel na obrázku (1.6) nemá žádné dodatečné nástroje. Výjimkou je ikona pro správce souborů.



Obrázek 1.5: Upscope – pohled správce

Upscope podobně jako Acquire nepřenáší dobře vstupy od správce. Řekl bych, že dokonce funguje ještě hůře. Při testování se mi několikrát stalo, že najednou nemohl správce klikat na odkazy, přitom ten samý odkaz ještě před chvílí fungoval. K 20. 4. 2019, kdy jsem Upscope testoval, neměl ještě podporu pro ovládání klávesnicí. Autoři ale slibují, že tato funkce bude brzo dostupná. Je však otázka, zda nebude mít stejné problémy jako u Acquire.

Většina omezení z (1.2.2) není také nijak řešena. Upscope ale používá proxy server pro stahování některých souborů. Přítomnost proxy serveru řeší částečně problém se soukromými zdroji (1.2.2.2). Je totiž možné si nakonfigurovat,



Obrázek 1.6: Upscope – pohled uživatele

jak se bude proxy server poskytovatelům těchto zdrojů prokazovat. Pokud jejich administrátor pak takové připojení povolí, bude k nim mít správce stejný přístup jako uživatel.

Funkce, kterou žádný z ostatních produktů nemá, je práce se vzdálenou konzolí. Ta umožňuje správci vidět obsah konzole od uživatele. Bohužel v ní nejsou zprávy, které vznikly před tím, než byla otevřena. Skrz konzoli je také možné na dálku volat JavaScript.

Stejně jako u Surfly, i tady jsem skeptický k funkci zabezpečení. Podle dokumentace je mezi automaticky maskovanými daty i formulář pro heslo. To jsem zkoušel a ano, správce opravdu nevidí heslo v tom formuláři. Není jenom skryté klasickými hvězdičkami, je jimi opravdu nahrazené. Jenže jej lze získat pomocí JavaScriptu přes konzoli. Aby to ale neměl správce tak složité, tak je to heslo přítomné v HTML. Takže kdokoliv, kdo má základní povědomí o funkci webové stránky, nebude mít problém s překonáním této ochrany. Navíc Upscope předpokládá, že správce bude někdo technicky zdatný, protože jinak mu nemá smysl poskytovat vzdálenou konzoli.

1.4.5 Shrnutí

Každý produkt má trochu jinou sadu funkcí, a hodí se proto pro jiné způsoby využití. Od toho se také odvíjí rozdíly v jejich ceně.

Výsledek testování dobře ukazuje, jak obtížné je realizovat co-browsing. I komerčně používané produkty mají spoustu problémů. Sdílená stránka neodpovídá vždy té skutečné. Ještě větší problém je vzdálené ovládání. Obecně největším nepřítelem těchto produktů je JavaScript. Ani jeden z nich neumí pracovat uspokojivě se stránkou, která je ve velké míře dynamicky ovládaná JavaScriptem.

Co považuji za vážný problém je přístup k zabezpečení. Co-browsing z principu není úplně bezpečný a vždy je potřeba věřit tomu, že správce vzdálený přístup nezneužije. Nemůže sice ovládat celý uživatelův počítač, ale i jedna webová stránka stačí k provedení celé řady útoků. Toto je risk, který by měli všichni zúčastnění znát a akceptovat ho. Proto mi nepřijde vhodné řešení Surfly a Upscope, které inzerují, že jsou bezpečné, protože nezobrazují správci citlivá data. V reálu mu k nim maximálně ztíží přístup. Navíc nijak nevarují před různými útoky, které může správce na uživatele provést.

Pro praktickou část jsem vyvodil následující závěry. Je potřeba se zaměřit na bezproblémovou práci s JavaScriptem. Důležité je také korektní zpracování vstupů. Jinak hrozí, že prototyp nebude v OptiLynx vůbec fungovat. Nemá význam snažit se ochránit citlivá data před správcem. Nejde to dobře udělat bez výrazného omezení ostatních funkcí. Je potřeba si dát pozor na zátěž serveru, v reálu to opravdu může způsobovat problémy. Od začátku počítám s tím, že správce bude moc manipulovat s HTML, pokud by například potřeboval dočasně opravit nějakou chybu. Proto se mi líbí funkce vzdálené konzole u Upscope, kterou do prototypu také zakomponuji.

1.5 Podklady pro návrh vlastního řešení

Na závěr této kapitoly zde uvádím požadavky a případy užití pro vlastní řešení co-browsingu (dále označuji jako *program*). Ty slouží jako podklady (rozšíření zadání) praktické části. Sestavil jsem je na základě zadání a konzultací s vedoucím práce. Také jsem v nich zohlednil informace získané v předchozích částech analýzy.

Program je cílený na firmy (dále také zákazníci), kteří mají vlastní webovou aplikaci. Zákazníci těchto zákazníků jsou z mého pohledu uživatelé.

1.5.1 Funkční a nefunkční požadavky

Následující seznam zahrnuje všechny požadavky na vyvíjený program. V rámci této práce má ale kvůli jeho rozsahu a náročnosti vzniknout pouze prototyp. Ten nemusí všechny uvedené požadavky zcela splnit. V takovém případě alespoň navrhnu jak prototyp dodělat.

Pro případné komerční použití by bylo potřeba přidat mnoho dalších požadavků, které by byly zaměřené hlavně na konfigurovatelnost chování programu. Tyto požadavky zde neuvádím, protože nesouvisí přímo se zadáním, a ani je nebudu v rámci této práce realizovat.

Některé požadavky neberou v potaz omezení sepsaná v (1.2.2). Jejich dokonalé splnění tedy nemusí být možné. Cílem je tato omezení vyřešit, jak nejlépe to půjde.

Funkční požadavky

F1: *Sdílení webové stránky* – Správce uvidí webovou stránku ve stejném stavu jako uživatel.

F2: *Ovládání myší* – Správce bude ovládat uživatelovu webovou stránku pomocí myši. Bude moct interagovat s elementy, jako jsou odkazy, tlačítka a formuláře.

F3: *Ovládání klávesnicí* – Správce bude ovládat uživatelovu webovou stránku pomocí klávesnice, tak aby mohl vyplňovat textové formuláře.

F4: *Simultánní ovládání* – Oba klienti budou najednou ovládat webovou stránku, bez nutnosti předávat řízení mezi nimi.

F5: *Editace HTML* – Uživatel i správce budou moct editovat HTML pomocí konzole.

F6: *Podpora JavaScriptu* – Program musí podporovat stránky s JavaScriptem.

F7: *Vzdálená konzole* – Správce bude mít přístup k obsahu konzole uživatele.

F8: *Vzdálené spuštění JavaScriptu* – Správce bude moct na dálku spustit u uživatele JavaScriptový kód.

F9: *Zobrazení kurzoru* – Oba klienti uvidí pozici kurzoru protistrany.

F10: *Zvýraznění kurzoru* – Kurzor bude měnit vzhled podle elementu, na který zrovna ukazuje. Speciálně bude zobrazovat, že druhý klient stiskl tlačítko myši.

F11: *Přechod mezi stránkami* – Program umožní co-browsing i v aplikacích skládajících se z více stránek.

F12: *Žádost o asistenci* – Uživatel bude moct požádat o asistenci. Program zobrazí tuto žádost správci. Uživatel také může žádost zrušit, pokud již problém vyřešil sám.

F13: *Potvrzení spojení* – Před navázáním spojení ho musí protistrana potvrdit.

F14: *Připojení za běhu* – Správce se musí umět připojit k uživateli. Při tom nesmí dojít k novému načtení uživatelovy stránky.

F15: *Ukončení* – Správce i uživatel budou mít možnost kdykoliv ukončit spojení.

F16: *Různé domény* – Program bude podporovat více domén. Jednotlivé domény budou na sobě zcela nezávislé. Takže například klienti z jiných domén nebudou smět mezi sebou komunikovat. Domény existují proto, že program bude nabízen většímu počtu zákazníků. Každý z nich může mít více webových aplikací. Je proto potřeba je od sebe izolovat.

F17: *Správa klientů* – Každá doména bude mít svého administrátora. Ten bude mít možnost vytvářet a spravovat klienty.

F18: *Správa skupin* – Administrátor bude přiřazovat klienty z jedné domény do více různých skupin. Skupiny slouží k jednoduššímu nastavení oprávnění.

F19: *Správa oprávnění* – Každá skupina bude mít určitá práva pro komunikaci s ostatními. Těmito právy lze nastavit, zda můžou klienti z jedné skupiny být správci pro klienty z jiné skupiny. Jinými slovy jedná se o nastavení

práva na spojení s jiným klientem, při kterém se zohledňují role klientů v tomto spojení. Klientům stačí být členem alespoň jedné skupiny, která jim spojení povoluje. Uživatelé v jedné skupině nemají v základu vůči sobě žádné oprávnění.

F20: *Správa pomocí skriptu* – Všechny úkony, které činí administrátor, je nutné zautomatizovat. To je z toho důvodu, že klientů může být velké množství a nebylo by tedy únosné každého v programu vytvářet ručně. Navíc ke změnám klientů může docházet velmi často, například když se v zákazníkově aplikaci registruje nový uživatel. K tomuto účelu napíše administrátor skript, který provede příslušné úpravy konfigurace. Skript může administrátor spouštět ručně nebo může být zabudovaný do webové aplikace zákazníka.

Nefunkční požadavky

N1: *Žádný externí software* – Po uživateli nesmí být vyžadováno instalování ani používání žádného dodatečného softwaru. Pro správce toto omezení neplatí.

N2: *Podporované webové prohlížeče* – Program musí fungovat v *Chrome* [5] verze 73. Podpora dalších prohlížečů není pro prototyp vyžadována.

N3: *Bezpečnost dat* – K datům od klientů nesmí mít přístup nikdo jiný kromě nich samotných. Výjimku po dobu trvání co-browsingu má druhý klient.

N4: *Databáze* – V případě, že server bude vyžadovat databázi, se musí jednat o *PostgreSQL* [26].

N5: *Integrace do aplikace* – Pro základní integraci programu do webové aplikace musí stačit vložení JavaScriptového kódu do každé její stránky. Nesmí být vyžadována další úprava zdrojového kódu aplikace.

1.5.2 Případy užití

Z pohledu případů užití a jejich účastníků lze program rozdělit na dvě části. V první části je popsaná administrace neboli konfigurace domén, klientů a oprávnění. Ve druhé části je samotný co-browsing a ostatní případy užití, které nejsou v administraci.

1.5.2.1 Administrace

Hlavním účelem administrace je umožnit administrátorovi nastavit oprávnění. K tomu je potřeba nejprve vytvořit domény, přidat do nich klienty a ty ještě rozřadit do skupin. Těmto skupinám je pak možné nastavit tato oprávnění.

Nyní popíšu jednotlivé případy užití administrace, k nimž se vztahuje diagram na obrázku (1.7).

U1: *Správa domén* – Příslušný administrátor spravuje doménu pro zákazníka. Každý zákazník potřebuje alespoň jednu svoji doménu, ale může jich mít v případě potřeby i více. Správa domén zahrnuje jejich vytváření a mazání.

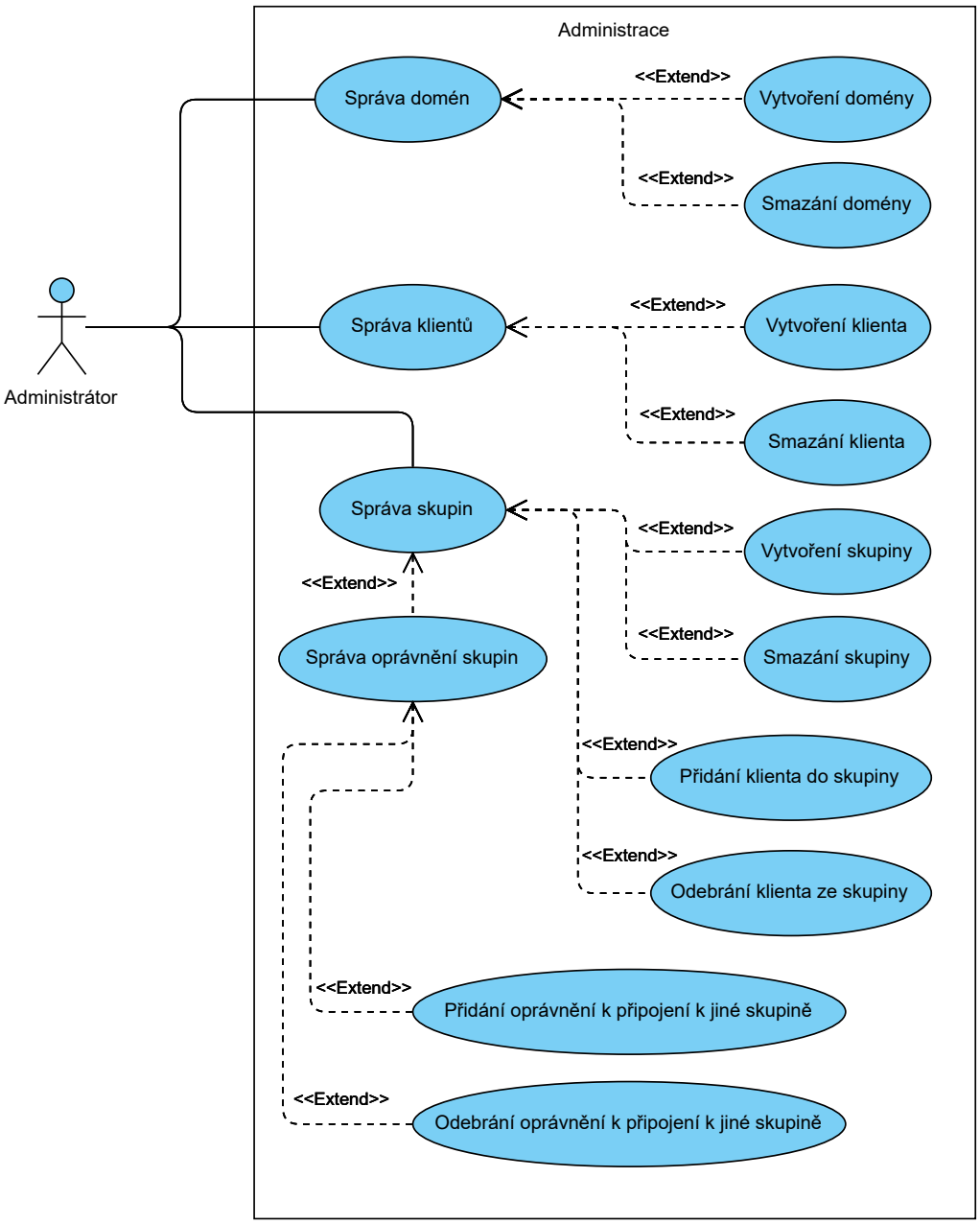
U2: *Vytvoření domény* – Administrátor vytváří domény pro nové zákazníky, aby v ní mohl provést požadovanou konfiguraci klientů. Novou doménu taky může potřebovat stávající zákazník, pokud má více aplikací, které chce mít od sebe oddělené.

U3: *Smazání domény* – Administrátor může smazat doménu spolu s veškerým jejím obsahem (klienti, skupiny, oprávnění). Nejčastěji k tomu dojde v situaci, kdy se zákazník rozhodne ukončit používání programu své webové aplikaci. Tím, že při smazání domény dojde k zneplatnění veškeré konfigurace, přestanou mít klienti z této aplikace možnost používat program.

U4: *Správa klientů* – Administrátor v rámci jedné domény přidává a odebírá klienty. Většinou bude jeden klient pro každého uživatele ze zákazníkovy aplikace. Mezi klienty jsou také správci, které si zákazník vybere.

U5: *Vytvoření klienta* – Administrátor vytvoří klienta například v momentě, kdy se do zákazníkovy aplikace zaregistruje nový uživatel, který má mít možnost používat program. Vytvořit klienta je potřeba také pro nového správce. V případě, že má s programem pracovat i neregistrovaný uživatel, je třeba k němu přiřadit dočasného klienta, který je později zase smazán.

U6: *Smazání klienta* – V okamžiku, kdy už nebude klient potřebovat pracovat s programem, ho může administrátor smazat. Při zrušení klienta dojde k jeho odstranění i ze skupin, ve kterých byl členem.



Obrázek 1.7: Diagram případu užití – administrace

U7: *Správa skupin* – Administrátor spravuje skupiny, aby mohl nastavit oprávnění klientům. Správa skupin zahrnuje jejich vytváření a mazání, přiřazování klientů do skupin a správu jejich oprávnění.

U8: *Vytvoření skupiny* – Administrátor vytvoří skupinu, pokud potřebuje nějakým klientům přiřadit oprávnění. Skupina je potřeba i v případě, že v ní klient bude sám. Přiřadit oprávnění přímo klientovi není pro zjednodušení možné.

U9: *Smazání skupiny* – Administrátor může smazat skupinu, pokud už není potřeba (například v ní není žádný klient). Spolu se skupinou dojde ke smazání oprávnění, která jsou k ní nastavená. To vede k přerušení nově neplatných spojení, podobně jako kdyby administrátor odebral oprávnění přímo.

U10: *Přidání klienta do skupiny* – Administrátor přidává existující klienty do skupin, aby jim nastavil společná oprávnění. Není možné přidat klienta do stejné skupiny, ve které již je členem. Je ale možné, aby byl klient členem několika skupin naráz.

U11: *Odebrání klienta ze skupiny* – Administrátor odebere klienta ze skupiny, pokud již klient nemá mít oprávnění vyplývající z členství v té skupině. Zároveň s tím opět dojde k přerušení nově neplatných spojení.

U12: *Správa oprávnění skupin* – Administrátor spravuje oprávnění skupin, aby mohl nastavit, kteří klienti se mohou (a v jaké roli) spojit s jinými klienty. V podstatě existuje jenom jeden typ oprávnění, které je vždy spojené se dvěma skupinami. Celá správa tak představuje přidávání a odebírání tohoto oprávnění.

U13: *Přidání oprávnění k připojení k jiné skupině* – Administrátor přidá oprávnění skupině, aby se klienti z této skupiny mohli v roli správce spojit s klienty z druhé skupiny (v roli uživatele).

U14: *Odebrání oprávnění k připojení k jiné skupině* – Administrátor odebere oprávnění skupině, aby zabránil klientům z této skupiny spojit se v roli správce s klienty z druhé skupiny (v roli uživatele). Tím také dojde k přerušení všech spojení, která přestanou být povolená (nejsou povolená ještě dalšími skupinami).

1.5.2.2 Co-browsing

Následující případy užití zachycené diagramem (1.8) se týkají co-browsingu a v podstatě všeho dalšího, co není součástí administrace. Vystupuje v nich správce a uživatel. Přestože je teoreticky možné, aby byl jeden člověk zároveň správcem i uživatelem, není to automatické. Proto jejich případy užití od sebe odděluji.

U15: *Zobrazení uživatelů žádajících o podporu* – Správce si zobrazí seznam uživatelů, kteří žádají o podporu. Ze seznamu si může vybrat, zda s nějakým z nich zahájí spojení.

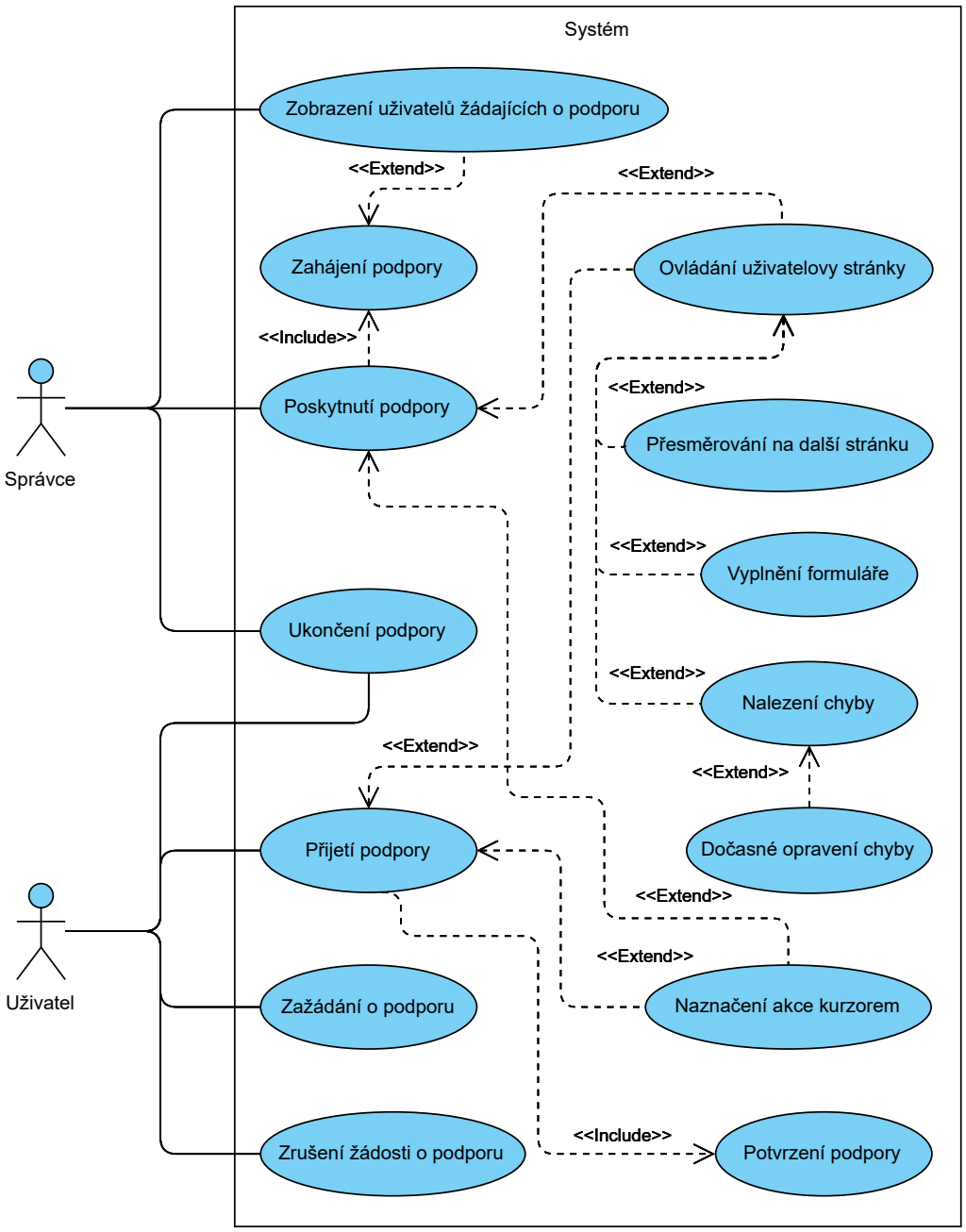
U16: *Poskytnutí podpory* – Správce poskytuje podporu uživateli tím, že ovládá jeho stránku nebo mu myší naznačuje, co má udělat. Před tím, než je možné poskytovat podporu, je nutné ji zahájit.

U17: *Zahájení podpory* – Před poskytnutím podpory musí nejprve správce vybrat uživatele a s ním následně zahájit podporu. Nejčastěji jej bude vybírat ze seznamu uživatelů, kteří žádají o pomoc. Je ale možné uživatele vybrat čistě podle jeho identifikátoru (jména). Uživatel musí podporu potvrdit před tím, než dojde k jejímu zahájení.

U18: *Ovládání uživatelovy stránky* – Správce i uživatel mohou v rámci podpory ovládat uživatelovu stránku. Ovládání zahrnuje možnost kliknutí na odkaz (přesměrování na další stránku aplikace), vyplnění formuláře a řešení chyb.

U19: *Přesměrování na další stránku* – Správce i uživatel mohou při ovládání stránky kliknout na odkaz. Tím dojde k načtení jiné stránky. Pokud je tato stránka součástí podporované webové aplikace, co-browsing pokračuje. V opačném případě dojde k přerušení co-browsingu do té doby, než se uživatel vrátí zpátky do aplikace.

U20: *Vyplnění formuláře* – Správce i uživatel mohou při ovládání stránky vyplnit formulář nebo pracovat s jiným podobným interaktivním prvkem stránky. Nejčastěji se jedná o vyplnění textového pole, výběr možností ze zaškrtávacího seznamu, výběr data, barvy a souboru. Součástí formulářů jsou také tlačítka, která se chovají stejně jako odkazy.



Obrázek 1.8: Diagram případu užití – co-browsing

U21: *Nalezení chyby* – Správce může při pokusu o ovládání stránky narazit na chybu, kvůli které uživatel potřeboval pomoct. V takovém případě mu prozkoumání zdrojového kódu stránky a JavaScriptové konzole pomůže

chybu najít. Hledat takto chybu může i uživatel, ale není to častý případ, protože to vyžaduje odborné znalosti. Může k tomu ale dojít například během vývoje a testování aplikace.

U22: *Dočasné opravení chyby* – V případě, že při hledání chyby je správce úspěšný, může být výhodné dočasnou editací HTML tuto chybu opravit. Díky tomu by uživatel mohl dokončit svoji akci bez čekání na aktualizaci aplikace. Podobně jako u hledání chyby i tady se může zapojit sám uživatel.

U23: *Naznačení akce kurzorem* – Správce i uživatel vidí pozici kurzoru toho druhého. Díky němu si mohou navzájem naznačovat, s čím je problém nebo co je třeba udělat.

U24: *Ukončení podpory* – Správce i uživatel mohou jednostranně (bez potvrzení druhého klienta) ukončit podporu v momentě, kdy již není potřeba.

U25: *Přijetí podpory* – Přijetí podpory je uživatelův pohled na *poskytnutí podpory* od správce. Jedná se o celý ten proces během, kterého správce pomáhá uživateli. Uživatel tedy přijímá jeho podporu. Na rozdíl od správce ovládá uživatel během podpory svou vlastní stránku. A také podporu nezahajuje, ale potvrzuje.

U26: *Potvrzení podpory* – Uživatel musí potvrdit podporu v reakci na její zahájení od správce. Bez tohoto kroku není možné přijmout podporu.

U27: *Zažádání o podporu* – Uživatel dává zažádáním o podporu najevo správcům, že potřebuje pomoct. Tuto žádost vidí všichni správci, kteří jsou k uživateli přiřazení pomocí skupin. Ti pak na ni mohou zareagovat a poskytnout uživateli podporu.

U28: *Zrušení žádosti o podporu* – Může se stát, že uživatel vyřeší problém dříve, než se mu správce stihne věnovat. V takovém případě uživatel zruší žádost o podporu.

Kapitola 2

Návrh architektury

Tato kapitola se věnuje návrhu architektury vyvinutého prototypu (programu) pro co-browsing. Nejprve uvedu a zdůvodním návrhová rozhodnutí, která jsem musel na začátku udělat. Poté představím samotnou architekturu, která se skládá z několika komponent. U jednotlivých komponent popíšu jejich úkol a závislosti na ostatních komponentách. Následně vysvětlím komunikační schéma mezi komponentami z pohledu funkčních částí programu. Také se budu věnovat návrhu zabezpečení, které je při co-browsingu obzvláště důležité. Za součást architektury považuji i základní principy, na kterých jednotlivé komponenty fungují. Z toho důvodu je také popisuji v rámci této kapitoly.

Kód prototypu je psán anglicky. Aby návrh architektury co nejvíce odpovídal kódu, budu uvádět při jeho popisu i termíny z kódu (anglické). Češtinu budu nadále používat v případě výrazů, které s kódem přímo nesouvisí. Například názvy tabulek a sloupců databáze jsou anglicky, ale třeba názvy komponent jsou v češtině.

2.1 Základní návrhová rozhodnutí

Před tím, než jsem mohl začít s návrhem architektury, jsem musel udělat několik důležitých návrhových rozhodnutí. Při nich jsem vycházel z analýzy provedené v (1).

2.1.1 Metoda synchronizace stránky

Z metod uvedených v (1.3.1) jsem se rozhodl pro metodu *synchronizace výstupu JavaScriptového enginu* (1.3.1.2). Tím, že zadání práce není o vytvoření nástroje pro společné sledování videa, nemají výhody druhého řešení tak velký přínos. Navíc nemám představu o tom, jak rozumně implementovat spolehlivou *synchronizaci vstupů do JavaScriptového enginu* (1.3.1.1).

Mezi funkčními požadavky (1.5.1) je uvedený požadavek, který říká, že i správce může editovat HTML. Kvůli tomu je nutné synchronizovat DOM i druhým směrem. Původní metoda počítá pouze se synchronizací od uživatele ke správci.

2.1.2 Události

Standardní posílání akcí od správce není dostatečné k zajištění správné funkcionality webové stránky. V ní může být například JavaScript, který reaguje na pohyb myši. Při něm ale vzniká celá řada událostí a JavaScript může záviset na kterékoliv z nich. V případě, že by tato událost nebyla správně vyvolána, se JavaScript bude chovat nečekaným způsobem.

Z toho důvodu budou od správce posílány všechny události. To by mělo zajistit výrazně lepší ovládání stránky, což je v současnosti největší slabinou existujících co-browsingových řešení testovaných v (1.4).

V průběhu testování programů pro vzdálenou správu v (1.1) jsem zjistil, že při psaní textu je nepříjemné, pokud program reaguje se zpožděním (větší odezvou). Proto jsem se rozhodl vylepšit mechanizmus reakcí na události. Na následujícím příkladu vysvětlím základní princip jeho funkce. Jednotlivá písmena se správci zobrazí v momentě, kdy správce zmáčkne klávesu. Současně s tím je k uživateli poslána příslušná událost. JavaScript ji má možnost zablokovat. Pokud k tomu dojde, je změna u správce vrácena zpět. Na rozdíl od běžného řešení tedy správce vidí výsledky některých svých akcí před tím, než dojde k jejich aplikaci u uživatele.

Nevýhodou tohoto celkového přístupu k událostem je zvýšení datového toku a větší zátěž pro JavaScript u obou klientů. Správce totiž musí všechny události zpracovat a odeslat. Poté, co je uživatel příjme, musí být vyhodnoceny. Navíc je ještě potřeba každou událost potvrdit správci.

2.1.3 Komunikace

Pro komunikaci mezi klienty jsem zvolil protokol WebRTC. WebRTC umožňuje přímou komunikaci mezi klienty. Tím se částečně vykompenzuje hlavní nevýhoda zvolené metody synchronizace, kterou je vysoká odezva na změny stránky. Použití WebRTC také výrazně snižuje zátěž serveru, což znamená, že jej bude možné v případě potřeby výrazně lépe škálovat.

Z mnoha různých důvodů se může stát, že komunikace přes WebRTC nebude fungovat. Prohlížeč klienta například vůbec nemusí WebRTC podporovat. Častěji spíš bude mít jinou verzi API, se kterou se při vývoji nepočítalo. Klient také může mít WebRTC zablokované. Proto jako záložní řešení použiji WebSockety.

2.1.4 Proxy server

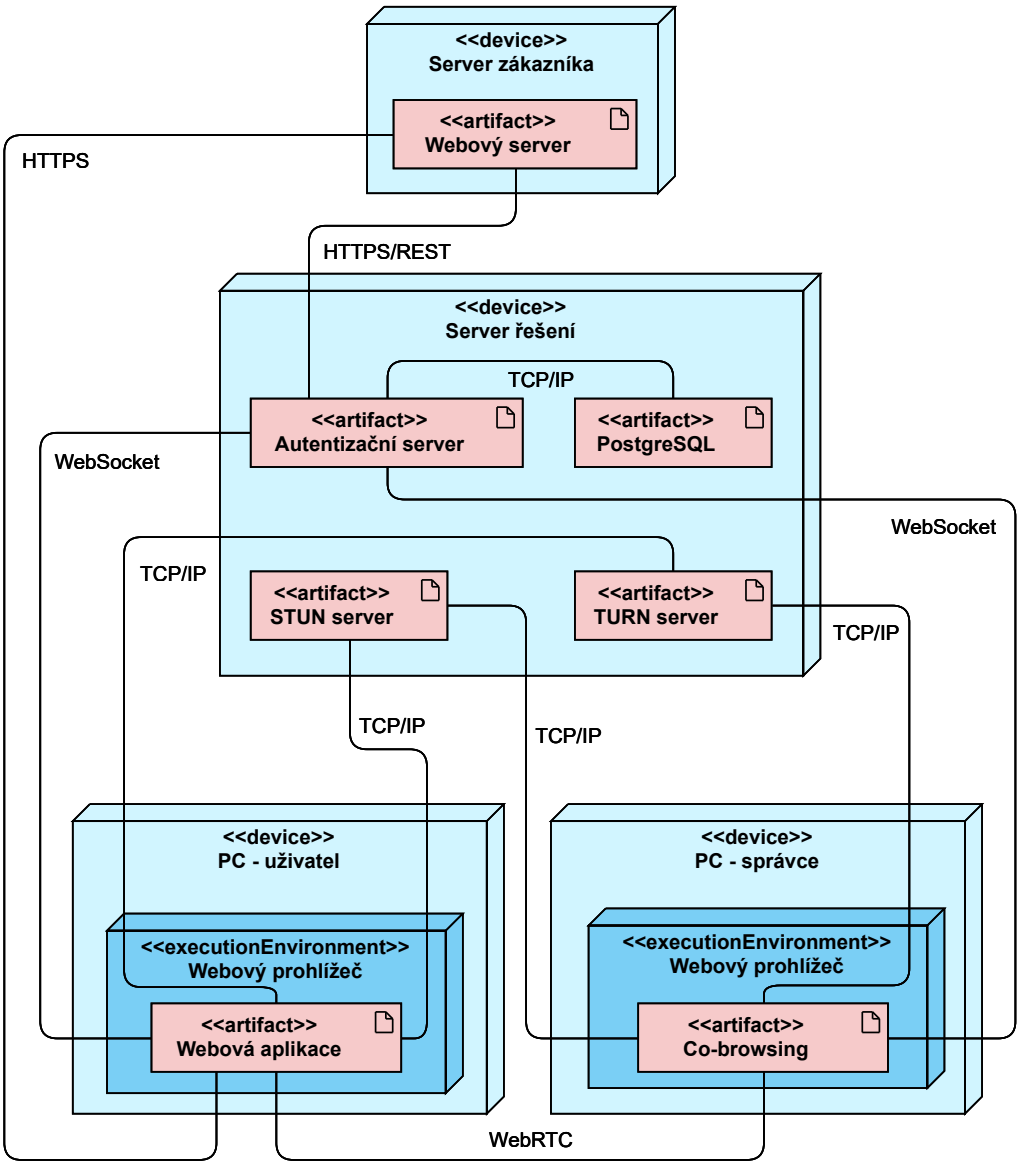
Kvůli použití WebRTC nedává přítomnost klasického proxy serveru, tak jak je vysvětlený v (1.3.2), příliš smysl. V podstatě by z velké části negoval výhody WebRTC. Proto jeho funkcionalitu budu implementovat u uživatele. Jedná se tedy o variantu bez proxy serveru.

2.1.5 Autentizační server

V (1.5.1) je požadována podpora pro domény, administraci klientů a nastavení jejich oprávnění. K tomu bude sloužit autentizační server, který bude mít na starost propojení klientů. Také přes něj bude potřeba řešit žádosti o podporu a potvrzování spojení. Další úlohou serveru bude zajištění náhradní komunikace přes WebSockety.

2.2 Komponenty a komunikace

Celé řešení se skládá z poměrně velkého počtu komponent, které spolu komunikují několika různými protokoly. Protože celá architektura je poměrně hodně složitá, uvádím ji pro přehled na diagramu nasazení (2.1).



Obrázek 2.1: Diagram nasazení

2.2.1 Server zákazníka

Každý zákazník má svůj vlastní server, na kterém je spuštěný webový server poskytující uživatelům aplikaci. Program proto nespolupracuje pouze s jedním webovým serverem. Jejich počet není nijak omezen. Reálně jich bude alespoň tolik co různých zákazníků.

Webový server komunikuje s uživatelem prostřednictvím HTTPS protokolu, stejně jako u běžných webových aplikací. Kromě toho může pomocí REST API konfigurovat autentizační server, například za účelem dynamického vytváření klientů.

Žádná součást samotného programu nijak nezávisí na webovém serveru zákazníka. Ani JavaScript realizující co-browsing u uživatele nepředpokládá od serveru žádné speciální chování. Závislost vzniká pouze obráceně na konfiguračním REST API autentizačního serveru.

2.2.2 Server řešení

Server řešení se skládá z celkem čtyř komponent. Nemusí být všechny provozovány na stejném fyzickém zařízení. STUN a TURN server dokonce nemusí být přítomny vůbec, případně je může poskytovat třetí strana.

2.2.2.1 Autentizační server

Jak jeho název napovídá, primárním účelem autentizačního serveru je zajistit autentizaci klientů. Klienti také přes tento server mohou komunikovat prostřednictvím WebSocketů. Kromě toho server zajišťuje i předávání žádostí o podporu od uživatelů ke správcům.

Server také řeší stav spojení mezi klienty. Umožňuje jim spojení navázat, potvrdit, zrušit, atd. Při navázání spojení server podle konfigurace rozhoduje, zda je spojení vůbec povolené. Server předává data mezi klienty pouze v případě, že je spojení mezi nimi aktivní.

Dalším úkolem autentizačního serveru je sloužit jako signalizační server pro WebRTC. Signalizační server je potřeba k navázání WebRTC spojení. Více si je o něm možné přečíst v [27]. Autentizační server neví o tom, že je k takovému účelu použit, protože informace nutné k signalizaci mezi klienty jsou přenášeny pomocí WebSocketů. Tím pádem je před navázáním WebRTC spojení potřeba navázat klasické spojení pro WebSockety a díky tomu vlastně server řeší autentizaci i pro WebRTC.

Chování serveru je možné nastavit pomocí konfigurace. K této konfiguraci poskytuje server REST API, které používá administrátor nebo webový server zákazníka. K uložení konfigurace potřebuje server nějakou databázi. S ní komunikuje pomocí protokolu založeném (většinou) na TCP/IP.

Autentizační server je závislý pouze na své databázi. Naopak na něm závisí server zákazníka a oba pluginy klientů.

2.2.2.2 PostgreSQL

Autentizační server potřebuje k uložení své konfigurace nějakou databázi. V podstatě by bylo úplně jedno jakou, ale v požadavcích z (1.5.1) je uvedený PostgreSQL [26]. Důvodem tohoto omezení je, že na serveru, na který se bude nasazovat autentizační server, už PostgreSQL je. Nemá proto smysl mít tam ještě jinou databázi.

Na databázi závisí pouze autentizační server, který navíc nevyžaduje žádné její speciální vlastnosti. V případě potřeby by tedy měla být její výměna poměrně jednoduchá.

2.2.2.3 STUN server

STUN server je další nezbytnou součástí k navázání WebRTC spojení. O jeho přesné roli si lze přečíst v [27]. Ve zkratce slouží k vytvoření tunelu skrze NAT. Jeho úkolem je tedy umožnit spojení i mezi klienty, kteří nemají veřejnou IP adresu. STUN server pouze pomáhá navázat spojení, ale neslouží k přenosu dat.

Existuje poměrně velké množství veřejných STUN serverů, takže není úplně nutné provozovat si vlastní. Na druhou stranu pro komerční aplikace je to vhodnější, protože u těch veřejných není zaručená dostupnost.

2.2.2.4 TURN server

TURN server je v podstatě záložní řešení ke STUN serveru a to v případě, že k navázání spojení STUN server z nějakého technického důvodu nestačí. TURN server slouží jako prostředník mezi klienty. Jeho použití tím pádem degraduje některé výhody WebRTC. Detailnější popis je opět v [27].

Protože přes TURN server je potenciálně přenášeno velké množství dat, musí mít dostatečný výkon. Z toho důvodu nejsou k dispozici veřejné neplacené TURN servery. Je možné si případně provozovat vlastní. Ale pokud je potřeba opravdu velký výkon a datový tok, může být výhodnější zaplatit si server poskytovaný třetí stranou.

Pro účely tohoto programu není přítomnost TURN serveru důležitá. Lepší je prostě použít WebSockety. TURN server by měl smysl, pouze pokud by bylo potřeba používat další technologie z WebRTC (například videohovor).

2.2.3 Plugin uživatele

Jediným programem, který je zapotřebí u uživatele, je webový prohlížeč. V něm je spuštěna webová stránka, která pochází od příslušného webového serveru. Součástí této stránky je JavaScriptový kód, který zajišťuje všechny potřebné funkce pro správnou práci mého programu. Tento kód je jednou z komponent programu a budu ho nazývat *plugin uživatele*. Nejedná se však o klasický plugin z pohledu prohlížeče (na rozdíl od pluginu pro správce).

Plugin se například stará o komunikaci s autentizačním serverem, o navázání spojení se správcem nebo o samotný co-browsing. Při co-browsingu slouží mimo jiné ke synchronizaci stránky a k reagování na vstupy od správce.

Webová aplikace používá HTTPS k získání HTML a dalších zdrojů z webového serveru. Kromě toho může plugin z webového serveru načítat zdroje a přeposílat je správci. Vlastně se tak chová jako proxy server.

Plugin potřebuje komunikovat hlavně s autentizačním serverem a se správcem. Data mezi uživatelem a serverem jsou přenášená přes WebSockety. V případě potřeby je možné využít WebSockety i pro zprostředkování komunikace se správcem. K přenosu dat mezi uživatelem a správcem se ale za normálních okolností používá WebRTC. Při použití WebRTC může také podle potřeby dojít ke komunikaci se STUN a TURN servery.

Z pohledu programu je plugin závislý na rozhraní autentizačního serveru a pluginu správce. Navenek poskytuje pouze rozhraní, které používá plugin správce.

2.2.4 Plugin správce

Komponentou programu u správce je plugin do webového prohlížeče. Tento plugin při co-browsingu otevírá speciální webovou stránku, jejíž obsah poté synchronizuje s uživatelem. Plugin dále obstarává i ostatní činnosti nutné pro realizaci co-browsingu. Funkcemi je proto plugin velmi podobný pluginu uživatele.

Hlavní důvod pro použití pluginu oproti normální webové stránce je nutnost přesměrovávat adresy dotazů. To může dělat pouze plugin. Je sice možné nahradit tyto adresy přímo v kódu stránky, je to ale výrazně složitější na implementaci. Proto je tento problém v rámci prototypu řešen tímto způsobem.

Správce má na rozdíl od uživatele možnost navázat spojení. Také vidí seznam uživatelů žádajících o pomoc. K tomu potřebuje nějaké uživatelské rozhraní, které také zajišťuje plugin.

Komunikace je více méně identická jako u uživatele. Stejně tak i závislosti na ostatních komponentách jsou stejné.

2.3 Zabezpečení

Zabezpečení je u co-browsingu velmi důležité, protože při něm dochází k přenosu dat z webové stránky uživatele. V ní mohou být citlivé údaje, jako je například heslo nebo číslo platební karty. Také samotné ovládání je z hlediska bezpečnosti problematické. Kdyby kdokoliv mohl převzít kontrolu nad stránkou uživatele, mohl by se za něho vydávat, což by mu umožnilo si jménem uživatele třeba objednat nějaké zboží.

Na správné zabezpečení je třeba myslet od začátku návrhu, protože hodně souvisí se samotnou architekturou. Samozřejmě jenom zajištění bezpečnosti v návrhu nestačí. Je nutné správně provést i její implementaci.

Při návrhu zabezpečení jsem kladl důraz na zaručení, že k uživateli a jeho datům nebude mít přístup nikdo neoprávněný. V případě tohoto programu existuje několik různých oblastí, které je třeba řešit zvlášť. Jejich popis v této části představuje pouze takový obecný náhled, ve kterém uvádím, co je třeba zabezpečit a jak to zhruba udělat. Nepopisuji zde detailní implementaci konkrétních bezpečnostních opatření, protože to už příliš nesouvisí se samotnou prací.

Bezpečnost dat není jedinou oblastí bezpečnosti, která je třeba řešit u aplikací přístupných z internetu. Existuje celá řada útoků s jiným cílem, kterým se zde nevěnuji, protože obrana proti nim není specifická pro tento program. Navíc velkou část z nich nemá význam z ekonomický důvodů příliš řešit. To platí alespoň do té doby, než mají potenciál způsobit velkou škodu. Příklad takového útoku je třeba DDoS.

2.3.1 Autentizace

Základním obraným mechanizmem snad každé aplikace je omezení přístupu na základě nějaké autentizace. O její zajištění se v tomto případě stará autentizační server.

Ostatní komponenty (například TURN server nebo databáze) mohou mít také vlastní autentizaci. O těchto komponentách dále nepíšu, jelikož jejich způsob autentizace není pro celkové zabezpečení programu důležitý a navíc je řešený třetí stranou. Pro zajištění bezpečnosti těchto komponent stačí jejich autentizaci správně nastavit a používat.

V programu jsou celkem tři role: administrátor, uživatel a správce. Každá z rolí má svá oprávnění. Existuje pouze jedna akce, která v současnosti nevyžaduje žádné oprávnění. Tou je vytvoření nové domény. Všechny ostatní akce jsou podmíněné přihlášením se za jednu z rolí. Vytvoření nové domény odpovídá registraci, a proto je volně přístupné.

Administrátor je role, která využívá REST API serveru k jeho konfiguraci. Každá doména má svého administrátora, který vznikne při jejím vytvoření. Při vytvoření domény je třeba zadat jméno této domény a heslo. Tyto údaje může následně administrátor použít k přihlášení, které probíhá pomocí *HTTP basic auth* [28].

Kromě administrátora jsou v programu účty pro klienty, které jsou vázané na doménu. Těm administrátor přiřazuje role uživatele a správce. Účet může mít žádnou, jednu nebo obě role.

Uživatel a správce jsou si z hlediska oprávnění podobní. Přestože teoreticky mají oba jinou sadu funkcí, je možné je realizovat pro oba stejně. Konkrétní role se ověřuje pouze na vyžádání u některých akcí. Rozdíl mezi rolemi je například při navazování spojení, kdy se kontroluje příslušné oprávnění nastavené administrátorem. Toto oprávnění určuje, kdo může být uživatel a kdo může být správce. Autentizační server ale sám neřeší, kdo z klientů spojení zahajuje. Je tedy možné, aby i uživatel zahájil spojení, ale jeho plugin tuto funkci nepodporuje.

K účtu klienta se vážou dva důležité údaje. Tím prvním je *token*, který slouží jako tajné heslo pro přihlášení. Druhý údaj slouží k identifikaci účtu ostatními klienty (je tedy veřejný) a nazývám ho *ID klienta* (v kódu *clientID*). Tyto údaje jsou vygenerované administrátorem při vytváření účtu. Token je z důvodů bezpečnosti nutné generovat pomocí CSPRNG. Také musí mít rozumnou délku

(alespoň 128 bitů), aby nebyl problém s kolizemi. ID klienta stačí generovat jako *UUID*, neboť je veřejné.

Klienti komunikují s autentizačním serverem výhradně přes WebSockety. Autentizace probíhá tak, že při otevření socketu pošle klient serveru svůj token. Server ověří jeho platnost a následně si přiřadí klienta k tomuto socketu. Není tedy nutné posílat token opakovaně.

Token klienta je potřeba poskytnout jeho pluginu, který se má následně připojit k serveru. Správce si může token zadat do svého pluginu sám, ale u uživatele něco takového není reálně proveditelné. Z toho důvodu mu jej musí poskytnout webový server zákazníka. Nejjednodušší je token dynamicky vložit do kódu pluginu, který je poslán uživateli jako součást webové stránky.

I přes všechna bezpečnostní opatření existuje reálné riziko, že dojde k odcizení tokenu. K nim totiž musí mít přístup uživatelé, jejichž chybou může k odcizení dojít. Pro omezení tohoto rizika je třeba tokeny pravidelně resetovat. Toto ale nemůže dělat autentizační server sám od sebe, protože server zákazníka může mít tokeny uložené. Je tedy na administrátorovi, aby zhodnotil míru rizika u svojí aplikace a podle toho zvolil vhodný interval resetování tokenů. Možnou realizaci tohoto zabezpečení představuje například generování nového tokenu při každém přihlášení uživatele do aplikace.

2.3.2 Uložení dat

K zabezpečení uložených dat se dá přistupovat dvěma způsoby. Tím prvním je zajištění, že k datům mají přístup pouze oprávněné osoby. Ten druhý řeší minimalizování následků zcizení dat. Jako v jakékoliv jiné části bezpečnosti je vhodné oba přístupy kombinovat.

V mém programu je oprávněný přístup k datům řešen hlavně v rámci autentizace. Dále je třeba vyřešit klasické problémy s fyzickým zabezpečením serveru a správnou konfigurací jeho softwaru. Ty však souvisí spíše až s nasazením serverových komponent (nikoliv jejich vývojem), takže je podrobněji nerozepisuji.

Druhý zmíněný způsob zabezpečení dat je také důležitý, protože například útoky vedoucí k ukradení obsahu databáze jsou mimo jiné kvůli *SQL Injection* poměrně časté. Pokud budou takto získaná data zašifrovaná nebo zahashovaná, bude mít útočník výrazně více práce s jejich zneužitím.

Jedinou komponentou programu, ve které jsou permanentně uložená data, je databáze. Konkrétní struktura databáze je popsáná až v (2.4). Pro účely

jejího zabezpečení ale stačí vědět, že obsahuje přihlašovací údaje administrátora a klientů (v podobě tokenů). Jiná citlivá data v ní nejsou. Dále obsahuje ještě nějakou konfiguraci oprávnění a klientů. To ale za citlivá data nepovažuji, protože klienti nejsou nijak identifikovatelní a nastavení jejich oprávnění se těžko dá považovat za důležité tajemství.

Při návrhu zabezpečení uložených hesel a tokenů jsem postupoval podle doporučení v [29]. Jako hashovací funkci jsem zvolil *PBKDF2* v kombinaci s *SHA512*. Hlavním důvodem její volby bylo, že je přítomná v programovacím jazyku použitém na serveru a splňuje doporučení z článku. Jiné podobné funkce by šly použít stejně dobře, na konkrétní volbě v tomto případě příliš nezáleží.

Důležitým aspektem všech těchto funkcí je, že lze nastavit obtížnost jejich výpočtu. To zvyšuje čas nutný k prolomení hashe. Podstatné je také přidat před zahashováním k heslu sůl, která brání v použití předpočítaných tabulek k rychlému nalezení vzoru hashe. Mezi další použitá doporučení patří porovnávání dvou hashí pomocí algoritmu s konstantní časovou složitostí. To brání útokům založených na různé prodlevě mezi odpověďmi serveru.

Způsob uložení tokenů se mírně liší od hesel administrátorů. Tokeny jsou totiž generované pomocí CSPRNG, takže slovníkový nebo brute-force útok na ně není v podstatě možný. K tomu musí mít dostatečnou délku, což je se 128 bity splněno. Není proto nezbytně nutné při jejich hashování používat nějakou dlouho trvající funkci a sůl. Stále je ovšem potřeba token alespoň nějak zahashovat, aby ho útočník nemohl rovnou použít. V tomto případě jsem využil stejnou hashovací funkci jako pro hesla, akorát jsem odebral sůl a nastavil nejrychlejší režim funkce.

Některá data z databáze jsou za běhu serveru načtená v jeho paměti. Zde jsou po krátkou dobu přítomna i hesla v nezahashované podobě. Útočník by tedy teoreticky mohl tato data z paměti získat. Je sice možné šifrovat i data v paměti, ale je poměrně obtížné to správně realizovat. V praxi pokud má útočník přístup k paměti, tak už žádná další obrana stejně nemá velkou šanci na úspěch. Proto ji v rámci práce neřeším, ale pro nějaké opravdu kritické systémy může mít význam. A pokud by někdy měl být program použitý například v bankovním sektoru, je rozhodně dobré toto zvážit.

2.3.3 Přenos dat

Téměř všechna komunikace mezi jednotlivými komponentami probíhá po síti. Ta obecně není zabezpečená proti odposlechu. Je proto potřeba použitím správných protokolů zajistit šifrování posílaných dat. Protože jednotlivé komponenty používají celou řadu různých protokolů, je nutné každý z protokolů ošetřit zvlášť.

Webový server – Uživatel Pro přenos dat z webového serveru se používá protokol HTTP. Ten ale není šifrovaný, a proto k němu existuje šifrovaná varianta známá jako HTTPS. HTTPS brání útočníkovi v odposlechu přenášených dat a také v jejich úpravě. Více o významu HTTPS je uvedeno v [30]. Použití šifrování je čistě záležitostí serveru, který v tomto případě patří zákazníkovi. Proto je za tuto část zabezpečení odpovědný zákazník.

Webový server – Autentizační server Autentizační server poskytuje navenek REST API, které využívá protokol HTTP. Situace je tedy stejná jako u webového serveru zákazníka. Je nutné použít HTTPS. V tomto případě je tedy potřeba při nasazení správně nakonfigurovat autentizační server. Tím je zabezpečení tohoto API vyřešené.

Autentizační server – Databáze Databáze je většinou přítomná na stejném fyzickém serveru jako autentizační server. Není proto potřeba řešit zabezpečení při přenosu dat. Pokud by tomu tak ale nebylo, je možné použít postup z [31] k aktivaci TLS, které zajistí šifrování.

Autentizační server – Klient Mezi autentizačním serverem a klienty se používá protokol WebSocket. Jeho nezabezpečená i zabezpečená verze je definovaná v RFC6455 [17]. Nezabezpečená verze má v URL zkratku „ws“, zabezpečené verzi patří zkratka „wss“. WebSockety jsou úzce spojené s HTTP, takže typicky zapnutí HTTPS zároveň zajistí použití zabezpečené verze WebSocketů.

Klient – STUN a TURN server Komunikace mezi klientem a STUN nebo TURN serverem je plně pod kontrolou implementace WebRTC. V [32] je napsáno, že WebRTC si zajišťuje šifrování při této komunikaci samo. V pluginu klientů tedy není potřeba toto zabezpečení řešit.

Uživatel – Správce Data mezi uživatelem a správcem se přenášejí pomocí WebSocketů a WebRTC. Přenos dat přes WebSockety využívá spojení mezi klientem a serverem, jehož zabezpečení již bylo vyřešeno v jednom z předchozích kroků. Zabezpečení WebRTC je teoreticky poměrně složité, ale podle [32] si většinu problémů řeší jeho implementace. Na programátorovi je pouze zajištění bezpečného přenosu signalizačních dat. Ta jsou přenášena dle návrhu pomocí WebSocketů. I u WebRTC je tedy bezpečnost zajištěná.

2.3.4 Data uživatele

Při co-browsingu je veškerý obsah stránky uživatele posílán po síti ke správci. Samotný přenos je vůči třetí straně zabezpečený pomocí předchozích kroků. Zbývá data ochránit proti provozovateli co-browsingového řešení.

Při použití WebRTC je správce přímým příjemcem dat od uživatele. U WebSocketů tohle ale neplatí. Autentizační server pracuje s daty v nezašifrované podobě, protože WebSockety zajišťují šifrování pouze při přenosu dat. Server si sice data neukládá, ale to neznamená, že tomu musí zákazník důvěřovat.

Server nepotřebuje znát obsah přenášených dat. Je proto možné je šifrovat a dešifrovat u klientů. Takové šifrování nemůže implementoval sám program. To by totiž vůbec neřešilo problém s důvěrou zákazníka. Z toho důvodu pluginy umožňují zákazníkovi implementovat si vlastní vrstvu pro šifrování, která zajistí, že server nebude rozumět přenášeným datům.

Zákazník také nemusí věřit tomu, že pluginy neobsahují škodlivý kód. To lze ale ověřit analýzou kódu a síťové komunikace.

K bezpečnosti dat uživatele je ještě potřeba dodat, že při co-browsingu k nim má správce plný přístup. Je velmi obtížné tento přístup selektivně omezit, protože to nutně znamená zablokovat většinu funkcí co-browsingu. Správce by tedy měl být někdo, komu druhá strana věří. Z pohledu uživatele toto není takový problém, protože provozovatel webové aplikace jeho data už má. Z pohledu zákazníka ale záleží, jak má vyřešenou spolehlivost správce, který by třeba mohl chtít získaná data zneužít. Také záleží na povaze aplikace. Většinou se správce nedostane k výrazně většímu množství citlivých údajů, než k jakým by měl přístup, kdyby poskytoval podporu přes telefon.

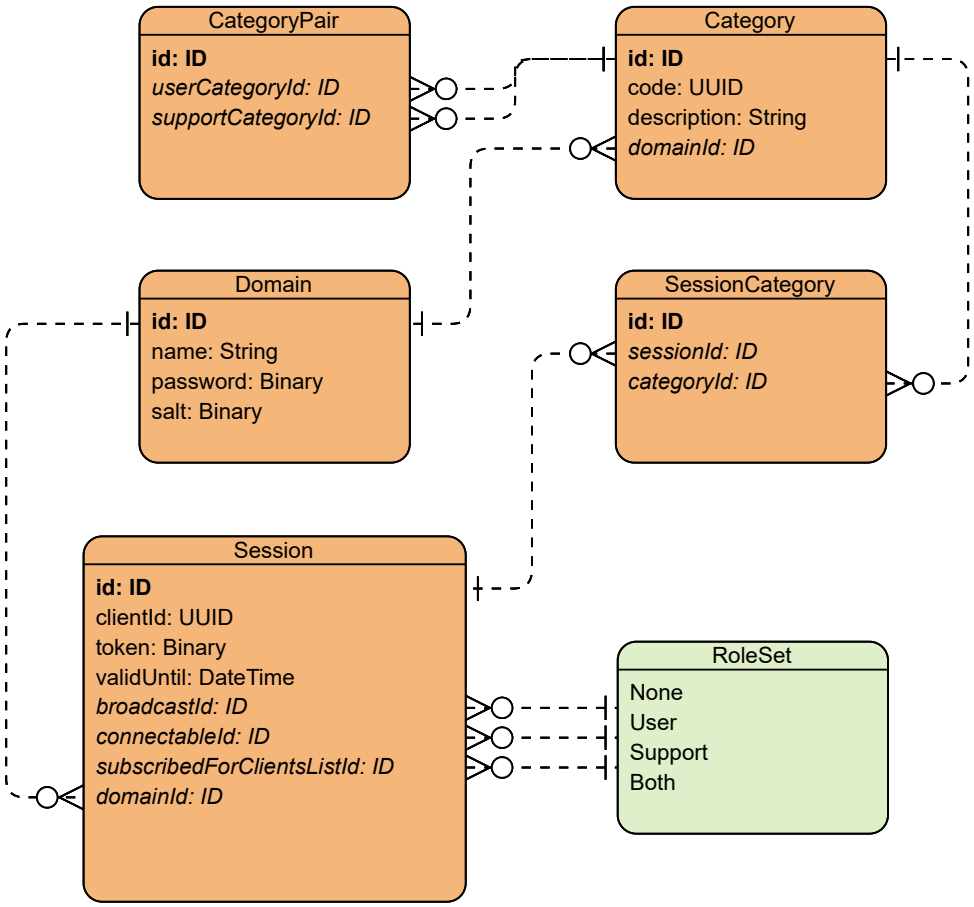
2.4 Návrh databáze

Jak již bylo řečeno výše, autentizační server potřebuje ke své činnosti databázi. Databáze byla navržena hned na začátku vývoje, kdy ještě nebyly ustálené funkční požadavky. Více o vzniku návrhu píšu v kapitole (3). Databáze proto obsahuje podporu pro některé funkce, které se v aktuální implementaci prototypu nepoužívají. Ve výsledku je databáze navržená poměrně univerzálně, což může být výhoda při budoucím rozvoji prototypu. Aby návrh odpovídal reálné databázi, uvádím i sloupce, které nejsou nezbytně nutné pro realizaci požadovaných funkcí.

Relační model databáze je na diagramu (2.2). Databáze má poměrně jednoduchou strukturu, protože autentizační server nemusí poskytovat příliš velké množství funkcí.

Ve zbytku této sekce jsou stručně popsané jednotlivé tabulky a její sloupce. Neuvádím explicitně popis primárních klíčů (sloupec ID), které jsou na diagramu znázorněny tučně. Primární klíče v tomto případě slouží pouze k propojení tabulek a nejsou používány mimo databázi a server. Cizí klíče jsou na diagramu zvýrazněny kurzívou a končí příponou *Id*. Výjimkou je sloupec *clientId*, který slouží k identifikaci účtu klienta, ale cizím klíčem není.

- **Domain** – Reprezentuje doménu z funkčních požadavků. Slouží k rozdělení dat z ostatních tabulek do oddělených domén. Obsahuje přihlašovací údaje administrátora domény.
    - *name* – název domény a přihlašovací jméno administrátora
    - *password* – zahashované heslo administrátora
    - *salt* – sůl použitá pro hash hesla
- **Category** – Odpovídá skupinám klientů. Obsahuje informace, které se používají při práci se skupinami pomocí REST API.
    - *code* – unikátní identifikátor skupiny pro REST API
    - *description* – textový popis skupiny; Tento popis existuje, aby měl správce lepší představu o tom, co která skupina znamená.
    - *domainId* – cizí klíč domény obsahující tuto skupinu



Obrázek 2.2: Relační model databáze

- **CategoryPair** – Ukládá oprávnění skupin. Každý záznam odpovídá právě jednomu oprávnění. To říká, že klienti se skupinou *supportCategoryId* mohou být správci pro klienty se skupinou *userCategoryId*. Obě skupiny musí být součástí stejné domény.
  - *userCategoryId* – cizí klíč skupiny uživatelů
  - *supportCategoryId* – cizí klíč skupiny správců
- **RoleSet** – Jedná se o výčet reprezentující potenční množinu rolí (správce a uživatel). Místo sloupců proto uvádím hodnoty. Používá se v některých nastaveních v *Session*, ke kterým lze přiřadit obě role zároveň.
  - *None* – žádná role

    - *User* – role uživatele
    - *Support* – role správce
    - *Both* – role správce i/nebo uživatele

- **Session** – Obsahuje informace vztahující se k účtu klienta. Mezi ně patří identifikační a přihlašovací údaje. K implementaci funkce žádosti o podporu slouží *broadcastId* a *subscribedForClientsListId*.

    - *clientId* – veřejný identifikátor klienta; Používají ho ostatní klienti hlavně při navazování spojení.
    - *token* – hash tokenu; Token slouží k autentizaci podobně jako heslo.
    - *validUntil* – platnost účtu; Omezuje dobu platnosti účtu. Záznamy, jejichž doba platnosti vypršela, jsou odstraněny a účet je tak znepřístupněn.
    - *broadcastId* – cizí klíč *RoleSet*; Určuje, v jaké roli je klient viděn ostatními. Lze interpretovat jako žádost o pomoc. Pokud uživatel nežádá o pomoc je nastaven na *None*. Pokud o pomoc žádá, má hodnotu *User*. Tímto způsobem je možné zobrazit i aktivní správce. To se ale momentálně nepoužívá.
    - *connectableId* – cizí klíč *RoleSet*; Slouží k automatickému zablokování příchozích spojení. Je to další mechanizmus, jak omezit, kdo může s kým navazovat s pojení. Pouze spojení, ve kterých vystupuje klient v určené roli, jsou povoleny. Jeho použití zase o trochu zvyšuje bezpečnost. Typické nastavení je *None* pro správce a *User* pro uživatele.
    - *subscribedForClientsListId* – cizí klíč *RoleSet*; Omezuje, které klienty s nastaveným *broadcastId* klient vidí. Pokud chce správce zobrazit uživatele žádající o pomoc, nastaví tuto hodnotu na *User*.
    - *domainId* – cizí klíč domény účtu

- **SessionCategory** – Přiřazuje skupiny účtům. Účet musí být ve stejné doméně jako skupina.

    - *sessionId* – cizí klíč účtu
    - *categoryId* – cizí klíč skupiny

2.5 Rozhraní autentizačního serveru

Autentizační server lze konfigurovat pomocí REST API. To znamená, že server přijímá HTTP dotazy s různými metodami na různých URL, které interpretuje na příslušné akce. Většina akcí přímo odpovídá jednoduchým CRUD operacím nad databází. Proto je nebudu znovu detailně popisovat. O databázi si je možné přečíst v (2.4).

Všechny akce kromě vytvoření nové domény vyžadují administrátorská práva. Přihlášení používá mechanizmus *HTTP basic auth* [28]. Přihlašovací údaje jsou získány při vytvoření domény. Přihlášením také dojde k rozlišení, v rámci které domény konfigurace probíhá. Není tedy možné činit změny v jiné doméně.

Pro přenos dat mezi administrátorem a serverem se používají takzvaná DTO ve formátu JSON. Data jsou uložená v těle HTTP dotazu. Některé akce nepotřebují žádná data v těle dotazu. Jiné akce sice JSON vyžadují, ale všechny jeho vlastnosti jsou volitelné. V takovém případě je potřeba poslat na server alespoň prázdný objekt.

Akce, které se vztahují ke konkrétnímu záznamu v databázi, používají k identifikaci tohoto záznamu nějaké ID. To je přenášené jako součást příslušné URL. V následujícím popisu jednotlivých dotazů je možné takovou akci poznat tím, že má v URL nějaký výraz ve složených závorkách. Tento výraz včetně složených závorek je potřeba při dotazu nahradit správnou hodnotou.

Pro signalizaci výsledku dotazu používá server standardní stavové kódy HTTP. Jejich specifikaci je možné najít například v [33]. Pokud jsou součástí odpovědi nějaká data, jsou podobně jako u dotazu v jejím těle ve formátu JSON.

Pro popis jednotlivých DTO budu používat notaci podobnou standardnímu JSON, akorát místo hodnot jsou uvedené jejich typy. Volitelné hodnoty lze poznat tak, že mají otazník u jejich datového typu. Volitelné hodnoty nemusí být přítomny v objektu, což je ekvivalentní hodnotě *null*. Pokud je hodnota typu pole, je typ jednotlivých prvků obalen hranatými závorkami. Některé klíče mohou mít výchozí hodnotu. V takovém případě je uvedena za typem, oddělená pomocí znaku „=“. Klíč s výchozí hodnotou také nemusí být v objektu přítomný.

2.5.1 Nastavení domén

Vytvoření nové domény Dotaz **POST /domain** vytvoří doménu. Vyžaduje *DomainDTO*, jehož formát je uvedený v (2.1). Jako jediná akce nevyžaduje přihlášení.

```
{
    "name": String,
    "password": String
}
```

Ukázka kódu 2.1: DomainDTO

Změna přihlašovacích údajů Dotaz **PUT /domain** aktualizuje jméno a heslo domény, ke které je administrátor přihlášený. Stejně jako při vytvoření domény vyžaduje *DomainDTO*.

Smazání domény Dotaz **DELETE /domain** smaže doménu. Doména je určena podle přihlášení, není tedy potřeba posílat serveru žádná dodatečná data. Smazání domény vede k odstranění všech dat, která do ní patří.

2.5.2 Nastavení účtů

Výpis účtu Dotaz **GET /session/{clientId}** vrátí účet s odpovídajícím *clientId*. Data účtu jsou ve formátu *SessionDTO*, který je popsaný v ukázce kódu (2.2). Výčet *RoleSet* má stejné hodnoty jako v databázi. Tento dotaz nevrací token účtu.

```
{
    "clientId": String,
    "token": String,
    "categoryCodes": [String],
    "duration": Int?,
    "broadcast": String = "None", // výčet RoleSet
    "connectable": String = "None",  // výčet RoleSet
    "subscribedForClientsList": String = "None" // výčet RoleSet
}
```

Ukázka kódu 2.2: SessionDTO

Výpis všech účtů Dotaz **GET /session** je podobný výpisu jednoho účtu, ale vrací pole všech *SessionDTO* v dané doméně.

Vytvoření účtu Dotaz **POST /session** slouží k vytvoření účtu. Jako data vyžaduje *SessionDTO* bez vlastností *clientId* a *token*, které jsou vygenerované až na serveru. V odpovědi posílá zpátky celé *SessionDTO*, tentokrát i s těmito údaji.

Změna nastavení účtu Dotaz **PUT /session/{clientId}** je opět podobný vytvoření. Vyžaduje navíc parametr *clientId*, podle kterého se určí, který účet je aktualizován. Dotaz vrací výsledné nastavení účtu v podobě *SessionDTO*, akorát bez tokenu.

Smazání účtu Dotaz **DELETE /session/{clientId}** smaže účet se zadaným *clientId*.

Resetování tokenu Dotaz **POST /session/{clientId}/token** vygeneruje nový token pro účet s příslušným *clientId*. Tento token poté vrátí v odpovědi.

2.5.3 Nastavení skupin

Výpis popisu skupiny Dotaz **GET /category/{code}** slouží k získání popisu skupiny s příslušným kódem.

Výpis všech skupin Dotaz **GET /category** vrací pole se všemi skupinami, které patří do domény administrátora. Využívá *CategoryDTO*, jehož formát je v ukázce kódu (2.3).

```
{
    "code": String,
    "description": String?
}
```

Ukázka kódu 2.3: CategoryDTO

Vytvoření skupiny Dotaz **POST /category** vytvoří novou skupinu. Vyžaduje *CategoryDTO* bez kódu, který se generuje na serveru. V odpovědi je *CategoryDTO* již s přiděleným kódem.

Změna popisu skupiny Dotaz **PUT /category/{code}** slouží ke změně popisu skupiny s příslušným kódem. Popis je zadaný ve stejném formátu jako při vytvoření skupiny.

Smazání skupiny Dotaz **DELETE /category/{code}** smaže odpovídající skupinu. Tím zároveň dojde ke smazání přidělených oprávnění.

2.5.4 Nastavení oprávnění

Výpis všech oprávnění Dotaz **GET /categoryPair** vrací pole všech oprávnění v dané doméně. Jednotlivá oprávnění jsou ve formátu *CategoryPairDTO*, který je v ukázce kódu (2.4).

```
{
    "userCategoryCode": String,
    "supportCategoryCode": String
}
```

Ukázka kódu 2.4: CategoryPairDTO

Přiřazení oprávnění Dotaz **POST /categoryPair** vytvoří oprávnění mezi dvěma skupinami. Kódy skupin jsou zvolené pomocí *CategoryPairDTO*.

Odebrání oprávnění Dotaz **DELETE /categoryPair** odebere oprávnění. Konkrétní oprávnění je identifikováno pomocí stejného *CategoryPairDTO*, které bylo použité pro jeho vytvoření.

2.6 Komunikace mezi klientem a serverem

Jak jsem již uvedl při popisu komponent, pro komunikaci mezi klientem a serverem jsou použité WebSockety. Klient přes ně posílá serveru dotazy,

pomocí kterých se přihlašuje a odhlašuje. Dále používá dotazy pro nastavení vlastního účtu, podobně jako to může provést administrátor. Ale nejdůležitější jsou dotazy pro navázání spojení s jiným klientem a následný přenos dat mezi nimi. Server klientovi posílá informace o změnách stavu spojení s jinými klienty, seznam uživatelů žádajících o podporu a data od jiných klientů.

WebSockety běžně nepoužívají režim dotaz a odpověď. V tomto případě je ale výhodné posílat ze serveru ke každé zprávě od klienta nějakou odpověď, podobně jako tomu je u HTTP. V odpovědi je stavový kód signalizující výsledek dotazu a někdy také data. Díky tomuto potvrzení může klient reagovat na chyby. Některé dotazy také využívají potvrzení k získání dat ze serveru. V opačném směru se potvrzení neposílá, protože není potřeba.

Příchozí a odchozí sockety jsou na sobě nezávislé. Není tedy jednoduché rozlišit, ke kterému dotazu patří jaká odpověď. Proto klient posílá spolu s daty i ID dotazu. Server v odpovědi použije stejné ID. To klientovi umožňuje spárovat dotazy a odpovědi.

Pro normální komunikaci se používá WebSocket v textovém režimu a JSON. Při předávání dat mezi klienty se ale používá binární režim se speciálním formátem, který popíšu později.

WebSockety se při načtení nové stránky rozpojí. Po přechodu ze stránky na stránku je proto potřeba se znovu přihlásit k serveru. Zároveň s tím také dojde ke ztrátě stavu u klienta. Ten by sice bylo možné si uložit, ale jednoduší je, když server při přihlášení pošle nutné informace znovu.

2.6.1 Navázání spojení

Proces navázání spojení slouží k autentizaci klienta. Autentizace musí být provedena pomocí prvního dotazu, protože všechny ostatní akce autentizaci vyžadují. Dotaz se nazývá *ConnectToSession* a klient přes něj posílá serveru svůj token.

Server si přiřadí klienta k WebSocketu, z něhož získal jeho token. V následující komunikaci proto není potřeba token znovu posílat. Zároveň tím server ví, na který WebSocket má posílat data určená klientovi.

Jeden klient může mít najednou přiřazený pouze jeden WebSocket. Pokud by došlo k tomu, že se více klientů přihlásí stejným tokenem, server přeruší spojení s předchozím WebSocketem. To se může stát například otevřením nové záložky prohlížeče.

2.6.2 Nastavení účtu

Klient posílá serveru celou řadu dotazů sloužících k úpravě nastavení jeho účtu. Na rozdíl od administrátora může měnit pouze svůj účet, ale má navíc možnost měnit stav spojení s jiným klientem. O tom píšu více v (2.7).

Nastavení účtu slouží hlavně k požádání o podporu a k získání seznamu uživatelů žádajících o podporu. Což jsou vlastnosti z (2.5.2) se jménem *broadcast* respektive *subscribedForClientsList*. Také je teoreticky možné měnit *connectable*, ale v praxi tuto vlastnost nastaví na začátku každému klientovi administrátor. Takže klient už ji poté nemá důvod měnit. K nastavení účtu slouží dotaz *UpdateSession*. Pro získání současného nastavení spolu s *clientID* lze použít dotaz *RequestSessionInfo*.

2.6.3 Přeposílání dat

Pro přeposílání dat mezi klienty je nejprve zapotřebí mít mezi nimi aktivní spojení. Způsob navázání tohoto spojení je vysvětlený až v (2.7.2). Pokud spojení není aktivní, server odmítne data přeposlat.

Pokud chce poslat jeden klient data ke druhému klientovi, použije k tomu dotaz *RelayData*. Je to jediný dotaz, který využívá binární režim. Jeho pakety mají následující formát. První čtyři bajty představují ID dotazu, dalších 36 bajtů je UUID spojení mezi klienty v *Base64* a zbytek jsou samotná data. Tato data mohou mít jakýkoliv obsah a formát. Aktuálně se ale používá pouze JSON v kódování UTF-8. Binární režim se využívá proto, aby bylo možné v budoucnu použít místo JSON jiný prostorově úspornější formát.

K druhému klientovi přijdou data také v binárním režimu. Tentokrát již v paketu není ID dotazu, protože klient nepotvrzuje serveru přijetí paketu. Paket tedy obsahuje pouze UUID spojení a data.

2.6.4 Ukončení spojení

Spojení je ukončeno buďto rozpojením WebSocketu nebo zasláním dotazu *DisconnectFromSession*. K rozpojení WebSocketu dochází nejčastěji při síťové chybě nebo opuštěním stránky. Ukončením spojení dojde k přerušení veškeré komunikace ze strany serveru. Klient také nebude zobrazen ostatním klientům jako aktivní, přestože by to jeho nastavení účtu dovolovalo.

Rozpojení pomocí dotazu umožňuje řešit situaci, kdy se uživatel odhlásí z aplikace. U běžných aplikací dojde k rozpojení už tím, že se při odhlášení změní stránka. Existují ale jednostránkové aplikace, které tuto možnost potřebují.

2.7 Komunikace mezi klienty

Komunikace mezi klienty slouží k posílání dat nutných pro sdílení stránky a pro vzdálené ovládání. Staví na všech předešlých komponentách, jejichž společný cíl je právě zajistit tuto komunikaci.

K identifikaci konkrétního spojení slouží UUID, které autentizační server vygeneruje při vytvoření žádosti o nové spojení. Mezi dvěma klienty může být v jeden okamžik více aktivních spojení. Z toho důvodu je nutné poslat serveru UUID jako součást každého dotazu, který se vztahuje k určitému spojení. Při běžném použití má každý klient maximálně jedno spojení.

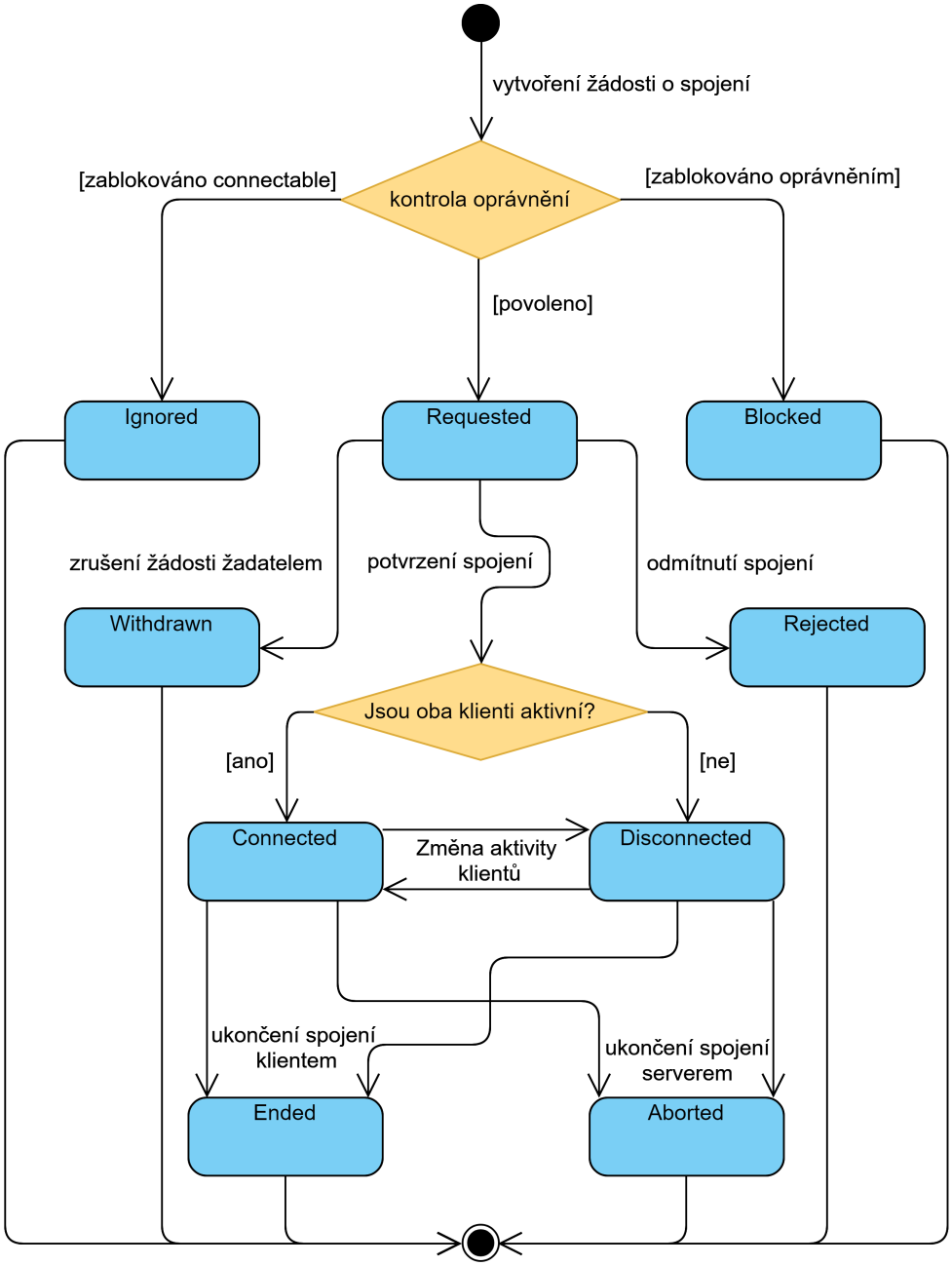
Posílání dat mezi klienty představuje bezpečnostní riziko. Přesně z toho důvodu existuje autentizační server, který zajišťuje, že komunikace může probíhat pouze po jejím oboustranném potvrzení. Autentizační server to realizuje tak, že blokuje přeposílání dat přes WebSockety, pokud mezi klienty není aktivní spojení.

2.7.1 Stavy spojení

Každé spojení během své existence projde několika stavy, které jsou zachycené na diagramu (2.3). Změnu stavu může vyvolat jeden z klientů tím, že zašle příslušný dotaz serveru. Stav také může změnit server sám od sebe. O změně stavu vždy server informuje pomocí zprávy *ConnectionEvent* klienta, který změnu stavu nezpůsobil. O některých změnách stavu jsou informování oba klienti. To je typické pro stavy, o kterých rozhoduje server.

Spojení rozlišuje klienty ze dvou pohledů. Tím prvním je role klientů. To znamená, kdo je uživatel a kdo správce. Server řeší role kvůli kontrole oprávnění. Z hlediska klientů nehraje role při komunikaci žádný význam. Používá se pouze pro rozhodnutí, jak se mají klienti chovat při co-browsingu. Druhý způsob rozlišení klientů určuje, kdo spojení založil. V závislosti na tom může každý klient ovlivnit stav spojení jiným způsobem. Například přijmout

spojení může pouze pozvaný klient, zatímco zakládající klient může zrušit svoji žádost o spojení.



Obrázek 2.3: Stavový diagram spojení mezi klienty

Requested Tento stav znamená, že server vytvořil spojení na základě žádosti od zakládajícího klienta. Toto spojení navíc opravdu může být navázáno, protože mu nebrání nastavení autentizačního serveru. Pokud by nebylo možné spojení navázat, bylo by místo toho ve stavu *Blocked* nebo *Ignored*. Pozvaný klient se může rozhodnout, zda spojení příjme či ne. Podle toho dojde ke změně stavu. V případě, že klient spojení potvrdí, je v závislosti na tom, zda jsou oba klienti aktivní, změněn stav na *Connected* nebo *Disconnected*. Pokud klient spojení odmítne, je nastaven stav *Rejected*.

Blocked Tento stav signalizuje nemožnost navázání spojení, protože k tomu nemá klient oprávnění. Absence oprávnění může znamenat buďto, že pozvaný klient neexistuje, nebo je z jiné domény, nebo mezi klienty není povolené spojení ve zvolené roli. O spojení v tomto stavu se pozvaný klient vůbec nedozví. Slouží pouze k signalizaci chyby k zakládajícímu klientovi. Po doručení informace o tomto stavu k zakládajícímu klientovi spojení zaniká.

Ignored Tento stav je chováním stejný jako *Blocked*. Liší se akorát významem chyby. Znamená, že toto spojení by bylo možné navázat, ale pozvaný klient ho ignoruje nastavením *connectable* ve svém účtu. Tato chyba je rozlišená, aby bylo jednodušší najít problém se špatně nastaveným *connectable*.

Withdrawn Tento stav vyjadřuje, že zakládající klient zrušil spojení před tím, než se stihl pozvaný klient rozhodnout, jak na žádost o spojení zareaguje. O zrušení žádosti je pozvaný klient informován. Poté spojení zaniká.

Rejected Tento stav znamená, že pozvaný klient odmítl žádost o spojení. Zakládající klient je o tomto stavu informován a následně spojení zaniká.

Connected Toto je hlavní stav spojení. Znamená, že spojení bylo potvrzeno pozvaným klientem. Oproti stavu *Disconnected* jsou navíc oba klienti aktivní, a tedy i samotné spojení je aktivní. Přenos dat mezi klienty je možný pouze v tomto stavu. Při změně na tento stav jsou informování oba klienti.

Disconnected Tento stav stejně jako *Connected* vyjadřuje, že pozvaný klient spojení potvrdil. V tomto případě je ale alespoň jeden z klientů neaktivní. To

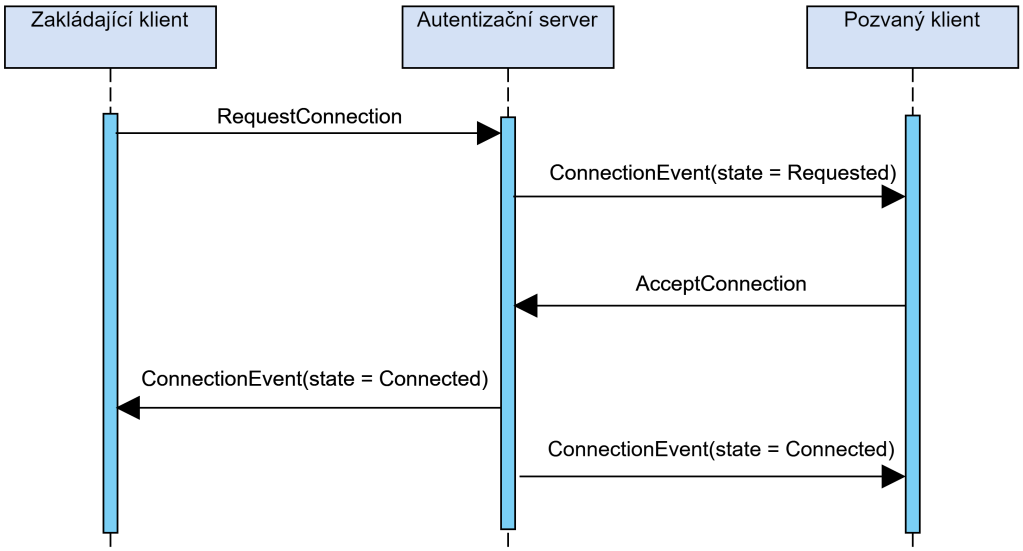
znamená, že spojení sice není zrušené po tom, co klient odejde ze stránky, ale není mu možné poslat data. Změna mezi stavy *Connected* a *Disconnected* se používá při znovu zahájení co-browsingu.

Ended Oba klienti mohou jednostranně ukončit spojení, čímž dojde k jeho přepnutí do stavu *Ended*. Informace o změně na tento stav se posílá oběma klientům. Spojení je následně odstraněno.

Aborted Tento stav také znamená, že došlo k ukončení spojení. V tomto případě jej ale ukončil server. K tomu může dojít, pokud administrátor změní jeho konfiguraci. Typickou změnou, která stav *Aborted* způsobí, je odebrání potřebného oprávnění nebo smazání jednoho z klientů.

2.7.2 Navázání spojení

Navázání spojení je nutné provést před tím, než je možné posílat mezi klienty data. Při navázání spojení si musí klienti vyměnit s autentizačním serverem několik zpráv. Na diagramu (2.4) je zachyceno, jak vypadá nejběžnější sekvence těchto zpráv. Zprávy (a jejich význam) odpovídají jednotlivým stavům spojení.



Obrázek 2.4: Sekvenční diagram navázání spojení

Spojení zahajuje jeden z klientů tím, že pošle zprávu *RequestConnection*. V této zprávě uvede, s jakým klientem se chce spojit pomocí jeho *clientId* a také v jaké roli chce ve spojení vystupovat. Pokud má k této akci oprávnění, dostane pozvaný klient žádost o spojení. Pozvaný klient může žádost přijmout nebo odmítnout. K tomu existují zprávy *AcceptConnection* a *RejectConnection*. Pokud si zahajující klient spojení rozmyslí, může ji stáhnout pomocí zprávy *WithdrawConnection*.

2.7.3 Přenos dat

Poté, co je spojení úspěšně vytvořené, mohou klienti mezi sebou začít komunikovat pomocí WebSocketů. Z pohledu návrhu komunikace není obsah takto posílaných dat nijak omezen. Je tedy teoreticky možné použít stejný návrh i pro libovolnou jinou aplikaci, která vyžaduje asymetrickou potvrzenou komunikaci mezi dvěma webovými prohlížeči.

Přestože je možné posílat data výhradně přes WebSockety, je hlavním zamýšleným způsobem přenosu dat protokol WebRTC. WebRTC potřebuje k vytvoření datového kanálu vyměnit několik zpráv mezi klienty a kontaktovat STUN a případně i TURN server. K výměně zpráv slouží právě WebSockety, proto je potřeba nejdříve vytvořit spojení přes ně. Vytvoření datového kanálu trvá kvůli výměně zpráv určitou dobu. To znamená, že ho není možné ihned po navázání spojení použít.

Čekání na WebRTC by zbytečně prodlužovalo dobu nutnou k zahájení co-browsingu. Proto lze v mezičase použít WebSockety i pro přenos ostatních dat. Na WebRTC se přepne automaticky, jakmile je jeho datový kanál vytvořený.

WebRTC se podobně jako WebSockety rozpojí, pokud dojde k načtení nové stránky. Proto je potřeba také datový kanál pokaždé vytvořit znovu.

Data jsou při přenosu zabalená do paketů. Přenos dat je spolehlivý a při doručení zachovává pořadí paketů. Není ale zaručeno, že klient dostane všechna odeslaná data, pokud dojde k přepnutí stavu spojení na *Disconnected*. To signalizuje, že se jeden z klientů odpojil ze serveru. Poté, co se opět připojí a spojení se přepne zpátky do stavu *Connected*, již nedostane zbývající data. V některých situacích je proto může být potřeba znovu poslat. Při co-browsingu toto není problém, protože po načtení nové stránky jsou stejně všechna předchozí data neaktuální.

2.7.4 Ukončení spojení

Spojení mohou ukončit oba klienti nezávisle na tom, v jakém stavu se nachází. K tomu slouží několik zpráv. Nejčastěji se používá zpráva *EndConnection*, která ukončí spojení ve stavu *Connected* nebo *Disconnected*.

Ukončením spojení dojde k zabránění v přenosu dalších dat přes WebSockety. WebRTC tím není přímo ovlivněné. Bylo by tedy možné jej používat i nadále. Proto si oba klienti hlídají konec spojení, a pokud k němu dojde, ukončí ze své strany i WebRTC.

Pokud jeden z klientů zašle na server zprávu *DisconnectFromSession* nebo načte novou webovou stránku, dojde k zániku jeho spojení se serverem. Spojení mezi klienty tím ale nekončí. Pokud se ale klient dostatečně brzo znovu nepřipojí, dojde ke změně stavu z *Connected* na *Disconnected*.

K ukončení spojení může dojít i ze strany serveru (stav *Aborted*). Server spojení ukončí, pokud byl jeden z klientů smazán nebo bylo administrátorem odebráno oprávnění ke spojení. Server spojení sám ukončuje, protože životnost spojení není jiným způsobem omezená. Mohlo by se tedy stát, že nepovolené spojení zůstane funkční, dokud nedojde k restartování serveru.

2.8 Sdílení stránky a vzdálené ovládání

Po navázání spojení mezi oběma klienty je možné zahájit co-browsing. Úkolem co-browsingu je mimo jiné sdílet stránku uživatele se správcem a také umožnit správci tuto stránku ovládat. Co-browsing je z pohledu návrhu vyčleněný do samostatné vrstvy.

Co-browsing využívá služby vrstvy komunikace, která je popsaná v předchozí sekci. Co-browsing požaduje od komunikace pouze zajištění spolehlivého přenosu mezi klienty, zaručené pořadí doručených paketů a také, že nedojde k fragmentaci paketů. Nezávisí na konkrétní implementaci komunikace. Tu je tak možné při dodržení těchto podmínek měnit. Díky tomu je i celý návrh podstatně jednodušší.

Co-browsing je ze své podstaty asymetrický. To znamená, že klienti mají dvě různé role, kterými jsou uživatel a správce. V mém návrhu je dominantní rolí uživatel. Jeho stránka je ta, která se sdílí, a také je u něho spuštěný JavaScript. Uživatel tedy pracuje se svou stránkou naprosto plnohodnotně, zatímco správce je omezený přirozenými limitacemi co-browsingu. Správce nekomunikuje na

přímo se serverem webové aplikace, takže z pohledu tohoto serveru vždy platí stav stránky uživatele. To je důležité, pokud dojde k desynchronizaci. Při opětovné synchronizaci tak rozhoduje obsah stránky uživatele.

Protože jsou funkce obou rolí různé, je různý i jejich návrh a implementace. Vrstva co-browsingu se proto dělí na tři moduly. Těmi jsou modul správce, uživatele a klienta. Modul klienta obsahuje funkce společné pro moduly uživatele i správce. Ty jsou proto na něm závislé.

Při co-browsingu je velmi důležité a zároveň těžké zajistit synchronizaci mezi klienty. Pro synchronizaci přes síť neexistuje jednoduchý mechanizmus podobný třeba zamykání u vícevláknových programů. Každá funkční část co-browsingu proto musí řešit synchronizaci zvlášť. Jednotlivé postupy jsou uvedené u konkrétních částí.

2.8.1 Zahájení

Úspěšné navázání spojení na komunikační vrstvě ještě nemusí nutně znamenat, že vrstva co-browsingu je u obou klientů připravená ke spuštění. Například se může stát, že jeden klient začne posílat data před tím, než druhý klient začne tato data očekávat. Z toho důvodu je třeba zajistit počáteční synchronizaci mezi klienty. Po jejím proběhnutí je zaručeno, že oba klienti o sobě vědí a jsou připraveni zahájit co-browsing.

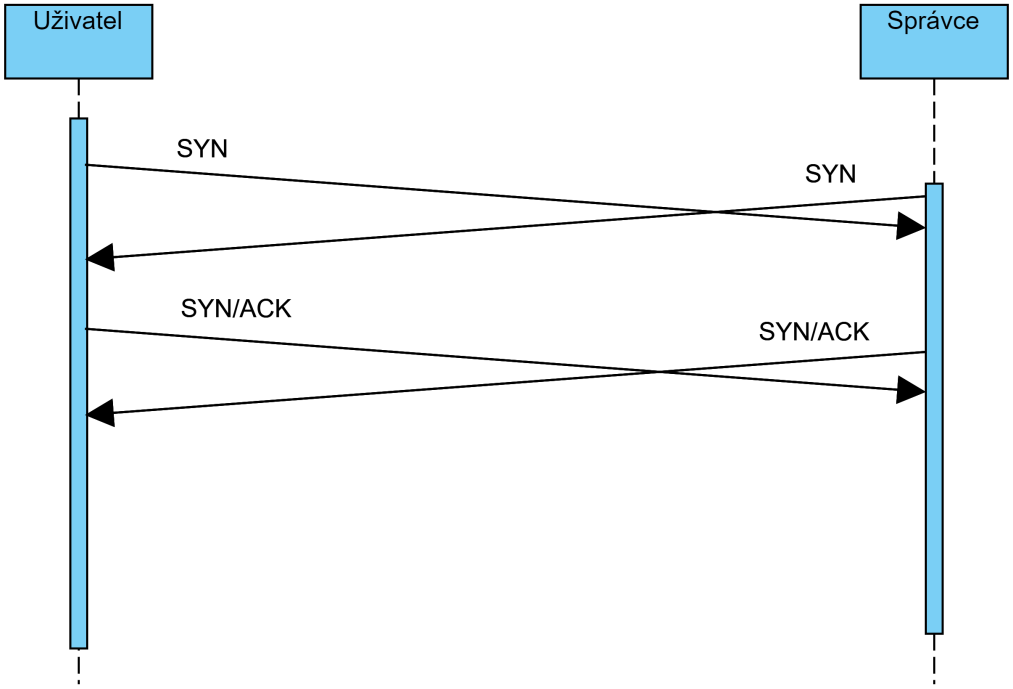
Pro tuto synchronizaci jsem použil takzvaný *three way handshake*, který je známý z protokolu TCP. Ten je popsaný spolu s TCP v [34]. Musel jsem ho mírně upravit, protože na rozdíl od TCP zde mohou prvotní synchronizaci zahájit oba klienti najednou. To je významný rozdíl, protože nelze předpovědět, který z klientů bude připravený jako poslední. Z pohledu posledního klienta je to právě on, kdo synchronizaci zahajuje.

Při počáteční synchronizaci si oba klienti vyměňují tři typy zpráv. Zprávou SYN oznamuje klient protistraně svou existenci a zároveň ji vyzývá, aby taky zahájila synchronizaci. V případě, že již protistrana synchronizaci zahájila, ji musí přerušit a začít od začátku posláním zprávy SYN/ACK. Zpráva SYN/ACK je kombinací SYN a ACK, ale liší se tím, že příjemce nezahajuje po jejím přijetí novou synchronizaci. Zpráva ACK je potvrzením o přijetí SYN. Jejím odesláním klient signalizuje, že je připravený na přijetí dat pro co-browsing.

Po přijetí zprávy SYN/ACK nebo ACK může klient bezpečně zahájit posílání dat pro co-browsing. Nepředpokládá při tom, že druhý klient má uložený nějaký

předchozí stav stránky. Například uživatel tedy pošle celou stránku bez ohledu na to, co za data již správci odeslal před tím.

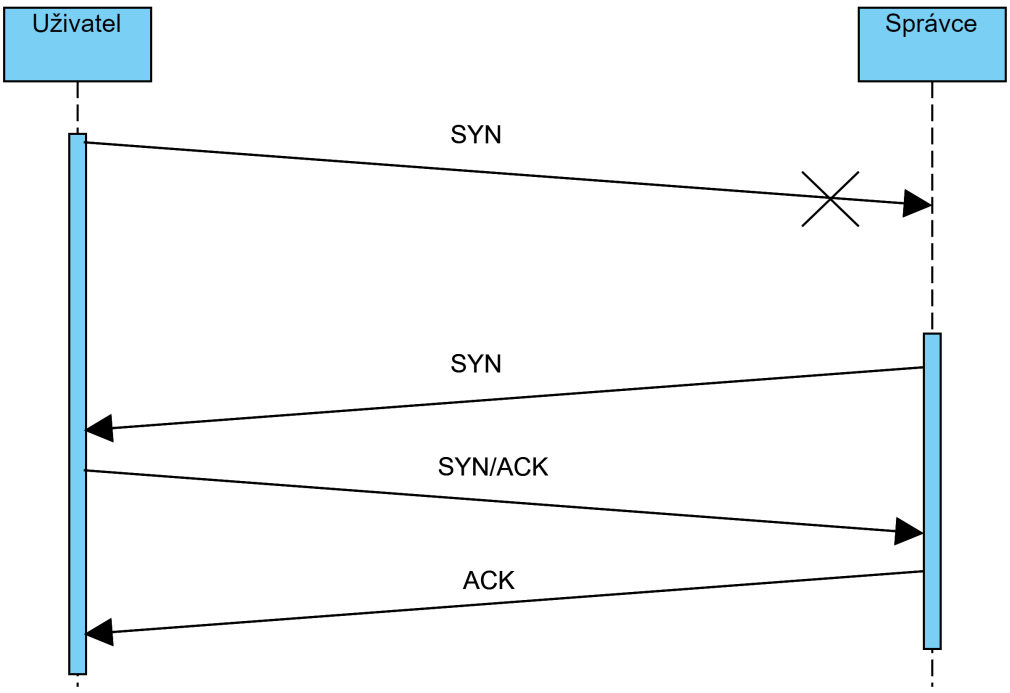
Existují dvě základní varianty, jak posílání zpráv vypadá. Ty jsou zachyceny na diagramech (2.5) a (2.6). Varianta A představuje situaci, kdy oba klienti existují před tím, než dostanou SYN od protistrany. Naproti tomu ve variantě B jeden z klientů nezaznamená příchozí SYN. Diagramy jsou úmyslně nakreslené tak, aby mezi reakcemi na zprávy byly různě velké prodlevy, tak jako tomu při reálném použití.



Obrázek 2.5: Three way handshake – varianta A

Varianta A se nejčastěji stane hned po zahájení komunikace. Prodleva sítě je většinou natolik velká, že se program u obou klientů stihne na příjem zpráv připravit. Může se však například stát, že JavaScript na stránce uživatele vyčerpá zdroje počítače. Potom nemusí zbýt dost prostředků pro včasnou přípravu co-browsingu a zachycení prvního SYN od správce. V takovém případě dojde k variantě B.

Na diagramu (2.6) je správce tím, kdo nepřijme SYN. Naprosto stejná situace (akorát zrcadlově otočená) se ale může stát i u uživatele. Kromě



Obrázek 2.6: Three way handshake – varianta B

případu popsaném v předchozím odstavci se také varianta B použije při restartování co-browsingu. K restartování nejčastěji dochází po načtení nové stránky u uživatele. Je totiž nutné o změně stránky informovat správce a tuto stránku mu poslat. Správce poté musí u sebe smazat veškerý obsah předchozí stránky. Toho lze nejjednodušeji docílit právě tím, že uživatel zahájí znovu počáteční synchronizaci.

Stejný způsob také umožňuje správci odpojit se a poté se znovu připojit. V tomto případě je to on, který znovu zahajuje synchronizaci.

2.8.2 Ukončení

Životnost vrstvy co-browsingu je úzce spjatá s komunikační vrstvou. Vzniká až po navázání spojení mezi klienty a zaniká nejpozději v momentě jeho ukončení. Spojení mohou ukončit oba klienti, proto není potřeba, aby byl na této vrstvě nějaký další speciální mechanizmus pro její ukončení.

Po ukončení co-browsingu zůstane stránka uživatele v tom stavu, v jakém je. Uživatel tedy může nerušeně pokračovat ve své práci. U správce dojde ke smazání obsahu stránky. Což mu zabrání v další manipulaci s touto stránkou.

Může se stát, že jeden z klientů spojení pouze přeruší. Typickým příkladem je, že uživatel otevře stránku, na které není kód zajišťující co-browsing. Podobnou situaci způsobí zavření prohlížeče. Z pohledu co-browsingu je to podobné, jako by bylo spojení ukončeno, protože zkrátka nemůže posílat data. Nedojde však ke standardnímu ukončení spojení přechodem do stavu *Ended*, a proto musí co-browsing detekovat přerušení jiným způsobem. K tomu slouží funkce autentizačního serveru, která při přerušení spojení mění jeho stav na *Disconnected*. Pro co-browsing se stav *Disconnected* liší od *Ended* tím, že předpokládá, že spojení bude brzy obnoveno. Nedochází tedy k uvolnění prostředků, pouze se pozastaví veškeré funkce.

2.8.3 Sdílení stránky

Sdílení stránky je zahájeno ihned po dokončení počáteční synchronizace. Sdílení stránky funguje oběma směry. Správce tedy může provádět libovolné změny obsahu stránky, jako by on sám byl uživatelem. Oba klienti také mohou stránku měnit současně.

Existuje několik typů možných změn. Jejich kombinací lze libovolně měnit obsah stránky. Velmi často dojde k několika změnám najednou. V takovém případě se posílají a vykonávají společně. Mezi tyto změny patří: přidání a odebrání elementu, změna vlastnosti nebo změna atributu. Změna pozice elementu lze složit z jeho odebrání a následného přidání na jiné místo.

Pro ovládání co-browsingu je zapotřebí přidat do stránky některé elementy. Příkladem je třeba tlačítko pro ukončení spojení nebo obrázek kurzoru druhého klienta. Tyto elementy se nesmí s druhým klientem sdílet. Kvůli tomu existuje atribut *wrs-ignore*. Element s tímto atributem a všichni jeho potomci jsou ze sdíleného obsahu stránky odebrány.

2.8.3.1 Elementy

Obsah stránky je sdílený po jednotlivých HTML elementech. To umožňuje synchronizovat obsah stránky výrazně efektivněji, než kdyby se posílalo pokaždé

celé HTML. U každého elementu je potřeba poslat jeho název, atributy, vlastnosti a pozici v hierarchii stránky. Zatímco atributy je vždy nutné poskytnout všechny, u vlastností toto neplatí. Většina vlastností představuje odvozená data, která není potřeba posílat, protože jsou vygenerovaná prohlížečem.

Každý element obsahuje jiné vlastnosti, které je potřeba sdílet. Proto se s každým typem elementu musí pracovat individuálně. Speciálním případem je element *input*. U něj tyto vlastnosti závisí ještě na obsahu atributu *type*. Elementy *input*, *option*, *select* a *textarea* jsou momentálně jediné, o kterých vím, že mají nějaké vlastnosti ke sdílení. Předpokládám ale, že vzhledem ke složitosti HTML je těchto elementů více.

Zvláštní kapitolou jsou elementy, které vytvořil programátor. Standard HTML umožňuje dodefinovat nové typy elementů se zcela jiných chováním. Je na programátorovi, jaké vlastnosti bude takový element mít. Je tedy například možné vytvořit element podobný elementu *input* s vlastností *content* namísto *value*. S něčím takovým nemůžu počítat, takže v takovém případě nebude sdílení obsahu fungovat správně. Jediným řešením je dodatečně přidat podporu pro každý takový element. Nedokážu úplně vyhodnotit, jak častý problém toto bude. Vlastní elementy jsou často používané frameworky, málokdy si je programátor píše sám. Těch nejpoužívanějších frameworků je pouze několik, takže by mělo být možné jejich vlastní elementy v základu podporovat.

V HTML existují kromě elementů i uzly (*node*). Zmiňuji se o nich proto, že třeba text není v HTML reprezentován elementem ale textovým uzlem. Pro zjednodušení zahrnuji pod pojem element i tyto uzly. Uzly jsou ale ve skutečnosti nadtřídou elementů a správně by to tedy mělo obráceně. Z pohledu návrhu jsou textové uzly pouze dalším typem elementu, který je třeba řešit zvlášť. Typicky nemají atributy ani vlastnosti, ale obsahují textovou hodnotu. Podobný případ jsou i komentáře. Ty sice není potřeba sdílet, ale je to výhodné, aby měl správce k dispozici co nejvěrohodnější obraz zdrojového kódu.

2.8.3.2 ID elementu

Pro identifikaci jednotlivých elementů u obou klientů je zapotřebí, aby měly identitu. Identifikovat element je nutné při aktualizaci jeho atributů a vlastností nebo při smazání elementu. V HTML žádná taková identita není, proto je potřeba ji zajistit pomocí nějakého umělého ID. Toto ID je přiřazeno každému

elementu při jeho vzniku a je posláno druhému klientovi spolu se zbytkem informací o elementu.

ID také slouží k určení pozice elementu v rámci hierarchie stránky. Každý element má svého rodiče a předchozího sourozence. To neplatí pro kořenový element a prvního potomka. K jednoznačnému určení pozice elementu stačí ID těchto dvou elementů. Pokud tyto elementy neexistují má příslušné ID nulovou hodnotu. Pozici elementu stačí určit pouze při jeho vytvoření.

Element lze odebrat z DOM a následně ho přidat na jiné místo. V takovém případě dostane element přidělené nové ID. Z pohledu JavaScriptu se ale bude jednat o stále stejný element. ID je totiž svázané jak s elementem, tak s jeho pozicí, aby ho bylo možné použít pro určení pozice nových elementů.

2.8.3.3 Synchronizace změn

Synchronizace změn probíhá na úrovni jednotlivých atributů a vlastností. Tento způsob je efektivní pro většinu elementů. U textových vlastností a atributů je ale potřeba posílat při každé změně celý text, což může být v případě elementu *textarea* poměrně hodně dat. Je možné posílat pouze změny textu, stejně jako se posílají pouze změny elementů. Algoritmus na nalezení změn v textu je poměrně jednoduchý. Je ale velmi obtížné v takovém případě udržet data synchronizovaná. V prototypu se kvůli zjednodušení implementace pracuje s textem v celku.

Na celém co-browsingu je algoritmicky nejtěžší udržet stránku u obou klientů synchronizovanou. K desynchronizaci může dojít například tím, že uživatel provede změnu A, ale než se správce o změně A dozví, tak provede změnu B. Pokud změna A a změna B není kompatibilní, dojde po jejich naivní aplikaci u obou klientů k desynchronizaci. Uživatel totiž uvidí stránku, na které byla nejprve provedena změna A a až poté změna B. Zatímco u správce by pořadí změn bylo obráceně.

Některé kombinace změn z principu nemohou desynchronizaci vyvolat. Mezi ně patří například odebrání dvou různých elementů. I kdyby byl jeden z elementů rodičem toho druhého, budou nakonec u obou klientů odebrány stejné elementy. Obecně problémem jsou změny vlastností a atributů a také změny ID elementů.

Jediným způsobem, jakým se může změnit nějaké ID, je odebrání elementu s tímto ID. To může způsobit následující problém. Jeden z klientů přidá nový

element. Jeho pozice v hierarchii stránky je určená pomocí ID jeho předchozího sourozence. Pokud druhý klient odebere tohoto sourozence před tím, než se dozví o přidání elementu, už nebude vědět kam má tento element umístit.

Podobný problém nastává při současném přidání dvou elementů. Když oba klienti přidají na stejné místo nový element, budou mít oba stejného předchozího sourozence. Ale poté, co dojde k přidání tohoto elementu u druhého klienta, bude zařazen mezi původního sourozence a druhý nový element. To znamená, že u uživatele budou elementy v pořadí: původní, nový od správce, nový od uživatele. Ale u správce bude pořadí: původní, nový od uživatele, nový od správce.

K vyřešení těchto problémů s ID je zapotřebí zajistit, aby ID nezanikalo spolu s elementem. Tím odpadá problém při smazání elementu. Problém s křížovým přidáním elementů lze řešit tím, že uživatel používá k identifikaci pozice předchozí element, zatímco správce používá následující element. To v podstatě způsobí, že při kolizi se elementy od uživatele přidají ze začátku a elementy správce od konce. Tím je zaručené, že jejich pořadí bude u obou klientů stejné.

Posledním problémem s ID je zaručení jeho unikátnosti. K tomu stačí, aby uživatel přiděloval kladná ID a správce záporná. Přetečení datového typu není potřeba řešit, protože i pro 32 bitové číslo má každý klient možnost vytvořit přes dvě miliardy elementů. Při každé změně stránky jsou ID vyresetované, takže jejich vyčerpání je při normálním použití v podstatě nemožné.

Kolizi atributů a vlastností nelze zabránit žádným podobný trikem. Lze však zajistit, že pokud k ní dojde, bude výsledek stejný u obou klientů. Protože rozhodující je obsah stránky u uživatele, je potřeba v takovém případě opravit stránku u správce. K tomu je možné využít potvrzování změn. Každou změnu atributů a vlastností, kterou provede správce, mu uživatel potvrdí. Po přijetí potvrzení od uživatele si správce u sebe aplikuje jím provedené změny ještě jednou. Princip je v tom, že je zaručené pořadí zpráv přenesených komunikační vrstvou. Díky tomu budou vždy změny u správce aplikovány ve stejným pořadí jako u uživatele. Některé změny mohou být aplikovány u správce dvakrát, ale to nemá na výsledek vliv.

2.8.4 Vzdálené ovládání

Vzdálené ovládání umožňuje správci manipulovat se sdílenou stránkou tak, jako by to byla normální stránka. Zatímco sdílení stránky je velmi podobné

u uživatele i správce, o vzdáleném ovládání to neplatí. U správce je potřeba zachytávat události a u uživatele tyto změny simulovat pomocí JavaScriptu. Události od uživatele není nutné správci vůbec posílat.

Na stránce bez JavaScriptu by v podstatě nebylo potřeba události řešit. Stačilo by od správce sdílet výsledné změny stránky a posílat příkazy ke spuštění odkazů. JavaScript ale může reagovat na události například tím, že úplně změní obsah stránky. Bez simulace těchto událostí by tak správce nemohl stránku plnohodnotně ovládat.

2.8.4.1 Typy událostí

Existují tři typy událostí, které jsou významné pro ovládání stránky. Těmi jsou vstupy z myši, vstupy z klávesnice a změna vlastnosti *focus*. U myši patří mezi nejdůležitější události její pohyb a stisknutí tlačítka. U klávesnice je to stisknutí klávesy. *Focus* určuje, který element na stránce momentálně reaguje na vstupy uživatele. Jeho změna je téměř vždy vyvolaná jinou událostí. Je ale obtížné dopočítat, že tato událost bylo opravdu vyvolána. Proto je lepší tuto událost rovnou zachytit a zpracovat. Podobně je tomu i s ostatními odvozenými událostmi. Takovou událostí je například *MouseOver*. Ta vyjadřuje, že kurzor myši ukázal na nějaký element.

Každý typ události má jiné JavaScriptové vlastnosti (obsahuje jiné informace). U myši je to například její pozice. U klávesnice to může být číslo stisknuté klávesy. Je proto potřeba k jednotlivým událostem přistupovat individuálně, podobně jako je tomu při sdílení elementů. Zároveň je významná část těchto vlastností zastaralá (*deprecated*) nebo nestandardizovaná. Každý prohlížeč může k těmto vlastnostem přistupovat jiným způsobem. Toto může způsobovat problémy, pokud JavaScript na stránce uživatele na nějaké z těchto vlastností závisí.

2.8.4.2 Zpracování výchozích akcí

Některé události mají k sobě přiřazené výchozí akce. Například při stisknutí klávesy dojde k napsání písmene do vybraného textového vstupu. U některých událostí není možné při simulaci události vynutit i tuto výchozí akci. Důvodem je, že událost vytvořená JavaScriptem není takzvaně *trusted*. Neboli některé výchozí akce jsou provedeny, pouze pokud je událost vyvolaná člověkem.

Řešením je opět individuální přístup ke každé události a případné aplikování výchozí akce jiným způsobem.

Výsledek výchozích akcí je potřeba nějakým způsobem sdílet s protistranou. U uživatele není žádný problém, změna je prostě poslána správci v rámci sdílení obsahu stránky. U správce to ale není tak jednoduché. Jak už jsem psal v (2.1.2), je vhodné zobrazit výchozí akce správci okamžitě, bez čekání na zpracování události u uživatele. To vytváří jeden problém. JavaScript totiž může událost zablokovat zavoláním metody *preventDefault*.

Zablokováním události říká JavaScript prohlížeči, že nemá na tuto událost reagovat. V takovém případě prohlížeč neprovede výchozí akci. Díky tomu lze realizovat například textový vstup, který dovoluje zadat pouze velká písmena. U správce je veškerý JavaScript zablokovaný, a proto zde k zablokování události nedojde. To může mít za následek, že by správce mohl se stránkou pracovat úplně jiným způsobem než uživatel.

Tento problém je vyřešen potvrzováním událostí, jako je to u jinak vyvolaných změn atributů a vlastností. Rozdílem je, že pokud JavaScript u uživatele událost opravdu zablokuje, dojde k poslání negativního potvrzení. Pokud správce obdrží negativní potvrzení a nedokáže sám vrátit výchozí akci, požádá uživatele o zaslání všech vlastností elementů. Stačí znovu načíst jenom vlastnosti, protože nic jiného není výchozími akcemi ovlivněno.

Je potřeba dodat, že celý mechanizmus vzdáleného ovládání je teoreticky velmi jednoduchý. Jedinou složitější částí je blokování událostí. Co činní vzdálené ovládání velmi náročným na realizaci a způsobuje velkou chybovost, je nutnost pracovat s každou událostí jiným způsobem. Některé události se velmi liší v tom, jak fungují. Navíc některé elementy reagují na stejnou událost různým způsobem. Příkladem může být třeba zaškrtávací políčko (*checkbox*). U toho je potřeba k zablokování výchozí akce zabránit události *MouseUp*. U jiných elementů to je typicky *MouseDown* nebo *Click*. Ve výsledku je tedy potřeba pro bezchybné fungování vzdáleného ovládání zkontrolovat všechny kombinace událostí, elementů a teoreticky vzato i prohlížečů.

2.8.5 Sdílení ukazatele

Sdílení ukazatele se skládá ze dvou částí. Tou první je přenos informace o pozici a stavu kurzoru mezi klienty. Tou druhou je samotné zobrazení kurzoru.

Zobrazení kurzoru funguje u obou klientů stejným způsobem, ale přenos informací se liší.

Směrem od správce k uživateli jsou již všechny potřebné informace přenášeny v rámci vzdáleného ovládání. Není tedy nutné posílat je ještě jednou. Stačí sledovat příchozí události o pohybu myši a stisknutí tlačítka.

Opačným směrem (od uživatele) ale vzdálené ovládání není, takže je potřeba některé informace dodat. Je zbytečné k tomu použít posílání všech událostí se všemi daty, jako tomu je u správce. Stačí posílat pouze pozici myši a informaci o stisknutém tlačítku.

K zobrazení kurzoru slouží element *img*, jehož pozice se mění podle aktuální pozice kurzoru druhého uživatele. Aby nedocházelo ke sdílení tohoto elementu s protistranou, má nastavený atribut *wrs-ignore*. Konkrétní zobrazený obrázek (typ kurzoru) závisí na elementu, na který zrovna kurzor ukazuje. To je možné ve většině případů JavaScriptem odvodit. Výjimkou je textový uzel, který není elementem. Pro znázornění stisknutí tlačítka myši je přidaný speciální typ kurzoru, který je zobrazený po dobu stisknutí tohoto tlačítka.

2.8.6 Vzdálená konzole

Vzdálená konzole nahrazuje správci vývojovou konzoli poskytovanou webovým prohlížečem. Zatímco uživatel může plnohodnotně pracovat s touto vývojovou konzolí, správce tuto možnost nemá. On by totiž manipuloval pouze se stránkou, ve které je zobrazená stránka, se kterou chce pracovat. Navíc u správce je zablokovaný JavaScript, takže by v něm nemohl pomocí konzole volat žádné funkce. Tento problém řeší vzdálená konzole, která umožňuje správci ovládat konzoli u uživatele.

Vzdálená konzole poskytuje dvě základní funkce. Tou první je čtení obsahu konzole. Princip funkce je velmi jednoduchý, stačí obsah konzole u uživatele přečíst, poslat ke správci a tam vhodně zobrazit. Poté už je jenom potřeba zajistit, aby se průběžně posílaly i nové záznamy.

Druhou funkcí je volání JavaScriptu. To funguje na podobném principu jako RPC. Správce pošle uživateli kód, který chce vykonat. Uživatel jej vykoná a pošle zpátky případnou výslednou hodnotu. Díky synchronizaci obsahu stránky a konzole to z pohledu správce vypadá, jako by pracoval s JavaScriptem lokálně.

Kapitola 3

Realizace

V této kapitole popíšu, jak jsem postupoval při tvorbě této práce. Dále zde zdůvodním volbu použitých technologií. Spolu s tím uvedu největší problémy, na které jsem při realizaci narazil, a také jak jsem je řešil. Problémů byla celá řada, takže se omezím jenom na ty opravdu nejzajímavější.

V prvních dvou kapitolách jsem text psal, jako kdybych postupoval podle klasického vodopádového modelu – to znamená, že nejdříve by měla být analýza a poté návrh. Ve skutečnosti jsem ale tyto činnosti dělal postupně, po částech. Také jsem se často vracel v momentě, kdy jsem narazil na chybu v návrhu. V této kapitole se budu při popisu snažit co nejvíce přiblížit skutečnému pořadí, v jakém jsem dané činnosti vykonával.

Rozhodl jsem se realizaci práce rozdělit do několika etap. Princip těchto etap spočívá v tom, že jsem na začátku každé etapy zvolil určitou funkcionalitu. Poté jsem udělal analýzu této funkcionality. Velká část analýzy zahrnovala čtení dokumentace a testování krátkých programů v JavaScriptu. Mým cílem vždy bylo zjistit, jaké jsou teoretické možnosti a omezení. Následně jsem udělal návrh, jak danou funkcionalitu implementovat.

Při implementaci jsem vždy začal s co nejjednodušším funkčním prototypem. Typicky jsem díky němu našel spoustu dalších problémů, o kterých jsem před tím nevěděl. Poté jsem provedl úpravu jeho návrhu a udělal novou implementaci. To často znamenalo upravit nebo alespoň refaktorovat již existující kód.

Při tvorbě práce bylo zcela jistě největší překážkou nalezení všech problémů, které bylo nutné zohlednit při návrhu. V dokumentaci většinou není uvedený

seznam věcí, které nelze udělat. To ve výsledku velmi často vedlo k tomu, že vymyšlený postup nebyl realizovatelný kvůli relativní drobnosti, ve které mě nenapadlo hledat problém. V momentě, kdy jsem příslušný problém dostatečně dobře identifikoval, ho už většinou nebylo těžké vyřešit. Podobným způsobem jsem našel i problémy z (1.2.2), které z větší části není možné řešit vůbec.

Celá práce probíhala stylem prototypování, častého upravování a přepisování kódu. Byl to v podstatě jediný způsob, jak se vypořádat s tím, že nebylo možné udělat na první pokus plně funkční návrh. Pro představu, finální prototyp má v součtu přibližně 12 000 řádků kódu v Kotlinu. Podle statistik z verzovacího systému bylo do repozitáře přidáno celkem 28 000 řádků a odebráno 14 500 řádků kódu. Reálný počet změněných řádků je samozřejmě mnohem větší, protože ne každá změna byla verzována. Celý prototyp jsem tak dohromady v podstatě napsal dvakrát.

Proto jsem se také rozhodl nepsat automatické testy. Předpokládal jsem, že jejich tvorba a následná údržba by mě jenom zdržovala. Zpětně musím s tímto rozhodnutím souhlasit. Většina chyb, na které jsem při vývoji narazil, byla v návrhu nikoliv v implementaci.

3.1 Tvorba základního návrhu

Na úplném začátku práce jsem si nejprve rozmyslel výše zmíněný postup. Není možné začít pracovat tímto způsobem bez jakékoliv představy o požadovaném výsledku. Proto jsem nejprve na základě konzultací s vedoucím práce sestavil základní požadavky a případy užití. Vycházel jsem ze zkušeností z používání klasických programů pro vzdálenou správu a konkrétních požadavků pro aplikaci OptiLynx. Požadavky se v průběhu vývoje už příliš neměnily. Z těch několika změn je nejvýznamnější přidání vzdálené konzole.

V této době jsem ještě neznal termín *co-browsing*. Ten se mi podařilo náhodou nalézt až později v průběhu vývoje. Takže jsem nemohl na začátku použít v podstatě žádné existující informace včetně analýzy jiných produktů.

Poté, co jsem měl sestavené požadavky, jsem udělal základní návrh architektury. Návrh byl poměrně jednoduchý udělat, protože se zabývá hlavně komunikací. Od začátku jsem tak měl například vymyšlené, jaké komponenty bude systém mít. Současný stav popsaný v (2.2) se v podstatě během dalšího vývoje nezměnil.

Již při vytváření zadání jsem předpokládal, že ve sdílení stránky a vzdáleném ovládání bude nejvíc problémů. Proto jsem se v základním návrhu omezil pouze na to, že sdílení stránky bude fungovat pomocí synchronizace HTML a vzdálené ovládání bude řešené simulováním událostí od správce. Neřešil jsem žádné konkrétní postupy ani implementační detaily. Pouze jsem věděl, že na teoretické úrovni je možné toto udělat. Podobným způsobem totiž fungují frameworky pro automatické testování webových stránek.

3.2 Autentizační server

Jako první jsem se rozhodl vytvořit autentizační server. Chtěl jsem nejprve zprovoznit celou komunikaci a až poté, co bude dobře fungovat, na ní postavit co-browsing. Protože pro komunikaci mezi klienty je potřeba nejprve vytvořit komunikaci mezi klientem a serverem, začal jsem právě tím serverem.

3.2.1 Volba technologií

Nejvíce zkušeností mám s programovacími jazyky Swift a Kotlin (Java), a proto jsem se rozhodl vytvořit server v jednom z nich. Nejsem velkým příznivcem dynamicky typovaných jazyků, jako je například JavaScript nebo PHP, takže jsem je ve výběru vůbec nezvažoval.

Původně jsem chtěl server psát ve Swiftu s použitím frameworku Kitura [35]. Jedná se o poměrně nové technologie, takže ještě nejsou tak propracované jako například Java. Nepovažoval jsem ale autentizační server za příliš složitý, takže jsem si myslel, že mi budou stačit.

Při implementaci jsem zjistil, že server sice nemá příliš mnoho funkcionalit, ale za to jsou některé poměrně komplexní. Navíc práce s databází v Kitura není zrovna jednoduchá. Implementace serveru v této technologii se tak proměnila spíše v boj s touto technologií.

Proto jsem se po několika dnech rozhodl na místo toho použít Kotlin spolu s frameworkem Ktor [36]. Konceptuálně není mezi těmito dvěma frameworky příliš velký rozdíl. V obou je možné programovat explicitním způsobem. Naproti tomu stojí třeba Spring [37], který využívá anotace a reflexi.

Teoreticky by měl explicitní způsob programování přinést větší kontrolu nad tím, jak kód funguje. Nevýhodou takového přístupu obecně je, že u větších

projektů je potřeba napsat výrazně více kódu. Tuto nevýhodu jsem považoval za zanedbatelnou, protože autentizační server je poměrně malý.

Při práci s Ktor jsem již neměl problém s možnostmi technologie jako takové. Bylo v něm možné udělat všechno, co jsem potřeboval. Problém jsem měl s jinou jeho nevýhodou – nedefinuje příliš architekturu a je tedy na programátorovi, jak bude svůj kód členit. Pro někoho s dostatkem zkušeností s tvorbou serverů to nemusí být takový problém. Já jsem ale dostatek zkušeností neměl, takže jsem musel svůj kód několikrát upravovat.

Přestože jsem nakonec s Ktor uspěl, volil bych pro realizaci podobného serveru už jenom Spring. Kdybych tuto volbu udělal hned napoprvé, ušetřil bych si ve výsledku poměrně velké množství práce.

3.2.2 Architektura kódu

Použil jsem klasickou třívrstvou architekturu s datovou, aplikační a prezenční vrstvou. Volba architektury byla jasná, protože to je standardní architektura pro servery. S její realizací už to bylo horší. Jak jsem již uvedl, Ktor nevynucuje žádnou architekturu. Členění kódu a rozhraní mezi vrstvami je proto na programátorovi.

Právě návrh správného rozhraní mi působil největší problémy. Potřeboval jsem totiž udržet databázové transakce a synchronizaci mezi vlákny. Také jsem musel vymyslet vlastní způsob kontroly oprávnění.

Ve výsledku nic z toho není těžké v Ktor udělat. Většinou stačilo napsat několik pomocných funkcí. Tou obtížnou částí je jejich vymyšlení. A to je přesně výhoda například Spring, ve kterém už je všechno toto vyřešeno.

3.2.3 Datová vrstva

Pro práci s relační databází (PostgreSQL) jsem zvolil knihovnu Exposed [38]. Exposed se stará o spojení s databází a o vykonávání dotazů.

Exposed podporuje dva režimy práce s databází. Tím prvním je klasické SQL, při kterém Exposed poskytuje pouze API obalující SQL dotazy. Pokročilejší režim navíc funguje i jako ORM.

Nejprve jsem zkoušel pracovat s tímto ORM, ale velmi rychle jsem zjistil, že je příliš omezující. Hlavním problémem bylo, že jím vytvořené entity nesmí opustit databázovou transakci, jinak hrozí vyvolání výjimky. Navíc tyto entity

v sobě obsahují celou řadu metod spojených s databází, které umožňují měnit její obsah. To ve výsledku znamená, že by i prezenční vrstva věděla o konkrétní implementaci datové vrstvy a mohla ji omylem ovlivnit bez zásahu aplikační vrstvy. To jsem nechtěl připustit, a proto jsem od použití ORM upustil.

Namísto toho jsem si vytvořil ruční mapování na vlastní entity. K tomuto účelu jsem pro každou entitu vytvořil DAO. Jeho úkolem je poskytovat rozhraní mezi datovou a aplikační vrstvou. Ve výsledku se tak datová vrstva skládá z definice databázových tabulek, entit a DAO.

K definici tabulek jsem použil DSL z Exposed. Jak taková tabulka vypadá, je možné vidět v ukázce kódu (3.1). Objekt z příkladu slouží k vytvoření tabulky v databázi, ale používá se i při sestavování SQL dotazů.

```
object CategoryTable: EntityTable() {
    override val id = long("id").primaryKey().autoIncrement()
    val code = text("code").uniqueIndex()
    val description = text("description").nullable()
    val domainId = reference("domainId", DomainTable.id)
}
```

Ukázka kódu 3.1: Definice tabulky

Ke každé tabulce existuje entita, která představuje jeden záznam v této tabulce. Jedná se o obyčejnou datovou třídu, jako je v ukázce kódu (3.2). Na rozdíl od entit vytvořených přes ORM, jsem zde reference na jiné entity řešil pouze pomocí jejich ID. ORM místo ID pracuje přímo s odkazovanou entitou, kterou ale kvůli výkonu načítá z databáze až se zpožděním. To je přesně ten důvod, proč pak entita nesmí opustit transakci.

```
data class Category(
    override var id: Long?,
    var code: String,
    var description: String?,
    var domainId: Long
): DatabaseEntityWithEquals<CategoryTable>()
```

Ukázka kódu 3.2: Entita

Pokud potřebuje aplikační vrstva pracovat s nějakým záznamem z databáze, požádá o něj příslušné DAO. Tyto objekty mají na starost obousměrné mapování entit do reprezentace databáze. Příklad tohoto mapování je v ukázce kódu

(3.3). Tyto mapovací funkce se poté volají při načtení dat, respektive při jejich uložení.

```
fun CategoryTable.resultRowToEntity(it: ResultRow) = Category(
    it[id],
    it[code],
    it[description],
    it[domainId]
)

fun CategoryTable.entityToStatement(entity: Category,
                                    it: UpdateBuilder<Int>) {
    it[code] = entity.code
    it[description] = entity.description
    it[domainId] = entity.domainId
}
```

Ukázka kódu 3.3: Ruční mapování entit

V DAO jsou také funkce, které představují SQL dotazy. Například v ukázce kódu (3.4) je klasický dotaz typu SELECT s podmínkou.

```
fun findByCodeInDomain(code: String, domainId: Long): Category?
    = databaseService.transaction {
        CategoryTable.select {
            (CategoryTable.code eq code) and
            (CategoryTable.domainId eq domainId)
        }
        .takeOne()
        ?.let(this@CategoryDAO::resultRowToEntity)
    }
```

Ukázka kódu 3.4: SQL dotaz

3.2.4 Aplikační vrstva

Aplikační vrstva se skládá z objektů typu *service*. Tyto objekty obsahují doménovou logiku. Aplikační vrstva poskytuje služby prezenční vrstvě a zároveň využívá datovou vrstvu k uložení dat.

Na aplikační vrstvě se také řeší oprávnění. Pokud danou akci nemůže aktuální administrátor provést, dojde k vyhození výjimky. Tu prezenční vrstva interpretuje a na základě ní odešle zpět příslušný stavový kód.

Některé metody v objektech jsou poměrně jednoduché. V podstatě pouze zprostředkovávají výstup z DAO. Například v ukázce kódu (3.5) je načtení skupiny s ověřením domény administrátora. Pokud skupina v jeho doméně není, dojde k vyhození výjimky, která reprezentuje stavový kód 403.

```
fun tryFindByCodeInDomain(code: String, domainId: Long): Category =
    dependencies.categoryDAO.findByCodeInDomain(code, domainId)
    ?: throw ForbiddenException
```

Ukázka kódu 3.5: Nalezení skupiny v rámci domény

Jsou zde také složitější metody. Jedna z nich je v ukázce kódu (3.6). Jejím úkolem je smazat skupinu. Při tom nejprve ověří, že mazaná skupina opravdu patří do administrátorovy domény. Dále zavolá metody na jiných objektech typu *service*, aby mohly na smazání skupiny příslušně zareagovat. Nakonec pomocí DAO danou skupinu opravdu odstraní z databáze.

```
fun tryDeleteByCodeInDomain(code: String, domainId: Long) =
dependencies.databaseService.transaction {
    tryFindByCodeInDomain(code, domainId).let { category ->
        dependencies.categoryPairService
            .deleteAllContainingCategory(category)
        dependencies.sessionService.deleteCategory(category)

        dependencies.categoryDAO.delete(category)
    }
}
```

Ukázka kódu 3.6: Smazání skupiny

3.2.5 Prezenční vrstva

Prezenční vrstva obsahuje řadiče (*controller*), které obsluhují rozhraní pro jiné systémy. Server poskytuje rozhraní v podobě REST a také přes WebSockety. Z toho důvodu jsou i dva typy řadičů.

REST řadiče jsou rozdělené podle první části URL, kterou obsluhují. V ukázce kódu (3.7) je definice jednotlivých koncových bodů řadiče, který spravuje skupiny. Ostatní řadiče jsou udělané podobným způsobem.

```
override fun route(routing: Routing) {
    routing.authenticate {
        route("/category") {
            getCategories()
            getCategory()
            createCategory()
            updateCategory()
            deleteCategory()
        }
    }
}
```

Ukázka kódu 3.7: Definice rozhraní REST

Každý koncový bod je reprezentován vlastní metodou, jako je v ukázce kódu (3.8). Typicky funguje tak, že získá data z dotazu, předá je na zpracování do aplikační vrstvy a následné odešle výsledek.

```
private fun Route.getCategory() = get("{code}") {
    requiresAuthentication { domainId ->
        requiresParameter("code") { code ->
            val categoryDTO = dependencies.categoryService
                .tryFindByCodeInDomain(code, domainId)
                .let(::categoryToDTO)

            call.respond(HttpStatusCode.OK, categoryDTO)
        }
    }
}
```

Ukázka kódu 3.8: Implementace koncového bodu REST

Pro WebSockety existuje jediný řadič, protože u nich se nerozlišuje typ dotazu podle URL. Namísto toho záleží na tom, jaká data dotaz obsahuje. Jinak je jeho kód velmi podobný jako u REST řadičů.

3.3 Pluginy klientů

Poté, co jsem měl hotový autentizační server, jsem začal pracovat na pluginech. Implementaci pluginů jsem rozdělil do tří částí. Nejprve jsem udělal komunikaci, poté sdílení stránky a nakonec vzdálené ovládání. Výhodou tohoto postupu bylo, že jednotlivé části na sebe dobře navazovaly. Tento postup byl ale trochu

riskantní, protože se mohlo stát, že by sdílení stránky nebo vzdálené ovládání vůbec nebylo možné udělat. V takovém případě bych už měl vytvořenou velkou část programu, který by nebylo možné nijak využít.

Tohoto rizika jsem si byl vědom, ale přesto jsem se rozhodl takto postupovat. Bez naprosto spolehlivé komunikace mezi klienty by totiž nebylo v podstatě možné sdílení stránky vyvíjet. Zpětně bych ale neimplementoval veškerou funkcionalitu komunikace jako první. Například bych vynechal podporu pro WebRTC. Sice by to znamenalo, že bych musel přepsat později více kódu, ale zase by se tím značně snížilo toto riziko.

3.3.1 Volba technologií

Z důvodů již uvedených v (3.3.1) jsem se rozhodl použít místo JavaScriptu Kotlin. Pro Kotlin existuje kompilátor, který je schopný jeho kód převést do JavaScriptu. Je tedy možné Kotlin použít pro psaní skriptů do webových stránek. Kromě Kotlinu jsem nepotřeboval žádné knihovny. Lépe řečeno knihovny, které bych potřeboval, pro Kotlin v kombinaci s JavaScriptem nejsou.

Například bych potřeboval knihovnu na zpracování JSON. Ale v současné době neexistuje žádná dostatečně kvalitní. Z toho důvodu jsem musel napsat převod mezi JSON a Kotlin objekty ručně.

V průběhu implementace jsem zjistil, že Kotlin se po zkompilování do JavaScriptu poměrně špatně ladí. Naštěstí je vygenerovaný JavaScript dobře čitelný. Není proto velký problém ladit kód s použitím standardních nástrojů pro JavaScript.

3.3.2 Komunikace

Vrstva komunikace poskytuje službu posílání dat mezi klienty. Tento zdánlivě jednoduchý úkol je ve skutečnosti poměrně komplexní. Mezi klientem a autentizačním serverem musí proběhnout celá řada zpráv a při poslání každé z nich může dojít k chybě. Abych zjednodušil implementaci co-browsingu co nejvíce, rozhodl jsem se rozdělit komunikaci do tří podvrstev. Tyto podvrstvy se v kódu jmenují *WrsWebSocket*, *Session* a *Connection*. Co-browsing pracuje jenom s tou poslední a ty ostatní jsou obsluhovány kódem, který ovládá co-browsing.

WrsWebSocket řeší základní práci s WebSockety, jako je například navázání a ukončení spojení, posílání a přijímání dat a také chybové stavy způsobené sítí.

Jeho dalším úkolem je zpracovávat odpovědi se stavovými kódy, které posílá server.

Session představuje spojení mezi klientem a serverem. Obsahuje autentizaci pomocí tokenu, nastavení účtu klienta a rozděluje příchozí data ze serveru podle jejich typu. Také umožňuje navázat spojení s jiným klientem. Využívá vrstvu *WrsWebSocket* k posílání a přijímání dat ze serveru.

Vrstva *Connection* slouží pro přenos dat mezi klienty. Je odstíněná od navazování spojení a dalších problémů, které mohou nastat při přenosu dat mezi klientem a serverem. Používá vrstvu *Session* pro přenos dat skrze server, ale přenos dat pomocí WebRTC je implementovaný přímo v ní.

3.3.2.1 DTO

V podstatě všechny zprávy mají formu nějakého DTO ve formátu JSON. Kvůli absenci knihovny pro práci s JSON, jsem musel převod mezi JSON a DTO napsat ručně. V ukázce kódu (3.9) je příklad, jak vypadá serializace do JSON. Využívá triku, že v JavaScriptu není mezi objektem a JSON příliš velký rozdíl. Proto stačí převést objektu v Kotlinu na objekt v JavaScriptu. Deserializace funguje přesně obráceně.

```
data class RequestConnectionDTO(val clientId: String, val role: Role):
    OutgoingSessionDTO() {

    override fun serializeInner(): dynamic = jsObject {
        this.clientId = clientId
        this.role = role.serialize()
    }
}
```

Ukázka kódu 3.9: Ukázka serializace do JSON

Přítomností mnoha vrstev dochází k nutnosti vkládat DTO z vyšší vrstvy do DTO z nižší vrstvy. Inspirací mi zde byla architektura TCP/IP. V ukázce kódu (3.10) je příklad takovéhoto obalu, který používá vrstva *WrsWebSocket*. Slouží k přidání ID ke každému poslanému dotazu na server. Toto ID se poté používá pro spárování odpovědi.

```
class OutgoingDTOIdWrapper(val dtoId: Int,
    val dto: Serializable): Serializable {

    override fun serialize(): dynamic {
        val jsObject = dto.serialize()
        jsObject.dtoId = dtoId
        return jsObject
    }
}
```

Ukázka kódu 3.10: Přidání ID k DTO

3.3.2.2 Řešení chybových stavů

Nejtěžším úkolem při implementaci komunikace bylo navrhnout, jak pracovat s chybovými stavy. To není tak jednoduché, protože při komunikaci přes sít může dojít k velkému počtu různých chyb. Některé mohou být vyvolané samotnou sítí, jiné pocházejí od serveru, atd. Použití více vrstev výrazně zjednodušuje práci s chybovými stavy, neboť každá vrstva je odpovědná pouze za chyby, které vzniknou u ní.

Existují dva základní typy chyb. Jedny jsou způsobené nezávisle na činnosti programu (výpadek internetového připojení). Do druhé skupiny patří chyby, které vznikly v reakci na odeslání dat.

Pro řešení chybových stavů jsem použil několik návrhových vzorů. Těmi jsou *observer*, *dispose*, *promise* a *result*. Tyto návrhové vzory elegantně řeší problémy způsobené asynchronním chováním sítě.

Při vývoji jsem vyzkoušel několik různých postupů, než jsem došel k finálnímu řešení, které je pro oba typy chyb v ukázkách kódu (3.11) a (3.12). Tyto metody má příslušný objekt v každé vrstvě a volající objekt je může využít pro reakci na chyby.

```
fun addOnErrorListener(listener: (Event) -> Unit): Disposable
```

Ukázka kódu 3.11: Reakce na chybu spojení

Pro tyto návrhové vzory také zatím neexistují žádné knihovny. Musel jsem proto udělat velmi jednoduchou vlastní implementaci. To byla ale i výhoda.

```
fun send(serializableData: Serializable): Promise<Result<Unit>>
```

Ukázka kódu 3.12: Reakce na chybu při odeslání dat

Například u návrhového vzoru *result*, který je zachycený v ukázce kódu (3.13), jsem díky tomu mohl přidat podporu pro stavové kódy HTTP.

```
sealed class Result<T> {

    abstract val code: Int
    abstract val message: String?

    data class Success<T>(val data: T, override val code: Int,
        override val message: String?): Result<T>()

    data class Failure<T>(override val code: Int,
        override val message: String?): Result<T>()
}
```

Ukázka kódu 3.13: Result

3.3.2.3 WebRTC

Implementace WebRTC byla podstatně složitější než implementace WebSocketů. Hlavním důvodem je, že tato technologie je výrazně složitější než WebSockety. Kromě přenosu dat totiž obsahuje podporu pro spoustu dalších funkcí. Také ještě není standardizovaná, což má za následek, že v ní chybí důležité části API.

Hlavní problém s WebRTC je způsob, jakým funguje datový kanál. Ten sice zajišťuje spolehlivost přenosu, ale chybí mu mechanizmus zabraňující zahlcení. Lépe řečeno, pokud je posláno v jeden okamžik příliš velké množství dat, dojde k chybě ve spojení. Není úplně snadné zjistit, kolik dat je možné bezpečně poslat. Experimentálně jsem došel k číslu 16 KiB na jeden paket. Navíc je omezená velikost vyrovnávací paměti, která pojme zhruba 100 paketů.

V současnosti není možné zjistit, kolik paketů ještě nebylo odesláno. Ve výsledku je v podstatě nemožné spolehlivě zabránit zahlcení, protože nelze získat dostatek informací od WebRTC. Je sice možné detekovat chybu, ale v ten moment dojde ke ztrátě neodeslaných dat.

Proto jsem vytvořil algoritmus, který zajišťuje potvrzování a řazení paketů. Při jeho návrhu jsem se inspiroval algoritmem, který používá TCP. K vyřešení problému už stačí jenom fragmentovat příliš velké pakety a automaticky obnovovat spojení při jeho chybě.

Výhodou tohoto postupu zároveň je, že je možné velmi jednoduše posílat data více způsoby najednou. To je umožněno tím, že u příjemce jsou nakonec data seřazena do správného pořadí. Díky tomu lze použít WebSockety pro přenos dat v momentě, kdy je WebRTC nedostupné. Navíc přepnutí mezi těmito dvěma protokoly proběhne naprosto plynule.

3.3.3 Sdílení stránky

Sdílení stránky jsem implementoval v několika iteracích. V každé z nich jsem narazil na nějaké technické problémy. Ty jsem následně vyhodnotil a v další iteraci opravil. Také jsem průběžně v iteracích přidával další funkce.

V první iteraci jsem si ještě nebyl zcela jist, jak moc je spolehlivá vrstva komunikace. Nechtěl jsem v ten moment řešit sdílení stránky příliš složitě, abych neměl problém s hledáním chyb ve dvou různých vrstvách. První sdílení stránky proto bylo co nejjednodušší. Při jakékoliv změně se vždy poslalo celé HTML v textové podobě. Nezískal jsem tak žádné podstatné informace pro realizaci sdílení stránky, ale s tím jsem dopředu počítal. V této iteraci jsem tedy hlavně testoval komunikaci. Ta fungovala nad očekávání dobře, ale přesto v ní nějaké drobné chyby byly.

Další iterace už byly výrazně složitější, proto se jejich jednotlivým aspektům věnuji zvlášť později. Zde je pouze stručný přehled. Ve druhé iteraci jsem již začal s implementací reálného algoritmu pro synchronizaci obsahu. Ale zatím jsem udělal synchronizaci pouze směrem od uživatele ke správci, což je výrazně jednodušší než oběma směry zároveň. Tato iterace sloužila hlavně k otestování detekce změn v obsahu stránky.

Ve třetí iteraci jsem pokračoval ve vylepšování práce s jednotlivými elementy a v odstraňování chyb. Nejdůležitější novinkou byla podpora pro detekci změn u správce. To s sebou přineslo celou řadu problémů s tím, jak udržet stránku synchronizovanou.

V závěrečné iteraci jsem vytvořil plugin správce jako opravdový plugin. Doposud byl tento kód součástí normální webové stránky. Také jsem zprovoznil sdílení obsahu konzole a přidal grafické uživatelské rozhraní pro pluginy. Žádná

další iterace z časových důvodů už neproběhla. To ale neznamená, že by nebyla potřeba. V prototypu jsou stále nedořešené některé problémy a chyby. Více o tom píšu v kapitole (4).

Většinu principů, na kterých prototyp funguje, jsem již popsal v kapitole (2.8). Nyní tedy budu popisovat spíše implementační detaily, různé problémy a obecně cestu k tomuto finálnímu návrhu. Také ukážu výslednou podobu uživatelského rozhraní obou pluginů.

3.3.3.1 Detekce změn stránky

Můj úplně první nápad na detekci změn spočíval v tom, že budu ukládat staré HTML a porovnávat ho s novým HTML. Pokud by byly rozdílné, použil bych nějaký složitější algoritmus, který by vytvořil seznam rozdílů na úrovni elementů.

Při pročítání dokumentace jsem našel *MutationObserver*, který přesně toto umí. V ukázce kódu (3.14) je vidět, jak se *MutationObserver* používá. Takto nastavený sleduje všechny změny v potomcích elementu *root* a při každé změně zavolá funkci *onDOMMutation*.

```
MutationObserver { records, _ ->
    onDOMMutation(records)
}.also {
    it.observe(root, MutationObserverInit(
        attributes = true,
        childList = true,
        subtree = true,
        characterData = true
    ))
}
```

Ukázka kódu 3.14: MutationObserver

Tímto způsobem je možné sledovat změny elementů a jejich atributů. Nelze takto detekovat změny vlastností, které je potřeba vyřešit dodatečně. Takto získaný seznam změn není pro další použití úplně vhodný, protože jeho obsah závisí na pořadí, v jakém byly změny provedeny. Proto seznam ještě upravuji pomocí dodatečného algoritmu, který jej převede do kanonické formy.

Pro detekci změn u vlastností není žádné API ze strany prohlížeče. Existuje ale několik událostí, které znamenají, že k nějaké změně vlastnosti mohlo

dojít. Pokud je taková událost vyvolána, je zapotřebí zkontrolovat příslušnou vlastnost. Některé události mohou znamenat změnu více vlastností. Například u přepínače (*radio*) znamená jedna událost aktivaci jednoho elementu a zároveň deaktivaci jiného. Je proto jednodušší zkontrolovat všechny vlastnosti na stránce.

Kontrolovaných vlastností není zase tak velký počet, jak by se mohlo na první pohled zdát. Většina kontrolovaných vlastností patří formulářovým prvkům. I na velké stránce jich proto bude řádově desítky. Je ale stále důležité efektivně tyto vlastnosti hledat. Není možné pokaždé kontrolovat všechny elementy, protože těch může být i více než tisíc. Takový počet už by měl znatelný vliv na výkon.

Změny způsobené JavaScriptem vyžadují speciální přístup. Teoreticky je potřeba takové změny řešit stejně, jako by je provedl člověk. Problémem je detekce těchto změn. U elementů a atributů stále funguje *MutationObserver*. JavaScript ale nevyvolá žádnou událost při změně vlastnosti. Není možné žádným způsobem zjistit přesný moment, kdy JavaScript změnu vlastnosti provedl. Jediná možnost je periodicky kontrolovat všechny vlastnosti. Při vhodné volbě periody (například 100 ms) není prodleva dostatečně dlouhá na to, aby ji správce poznal. Zároveň to pro moderní počítač nepředstavuje výpočetně takovou zátěž. JavaScript je spuštěný pouze u uživatele, takže tento odstavec se správce netýká.

3.3.3.2 ID elementu

Implementace ID pro elementy je hodně obtížná, protože musí splňovat požadavky z (2.8.3.3). V ukázce kódu (3.15) je rozhraní třídy, která přidává do DOM podporu pro ID. V ukázce jsou některé metody, které zajišťují mapování mezi ID a elementem. Také v ní jsou metody na přidání a odebrání elementu. Metoda *getSiblingId* se používá pro získání ID sousedního elementu. Toto ID je potřeba pro identifikaci pozice elementu v rámci stránky.

Každý element má přidělené ID po celou dobu své existence v DOM. Po odebrání elementu je ID zachováno, ale element musí být z paměti uvolněn. Jinak by došlo k úniku paměti. Není proto možné získat zpětně ID elementu, který byl odebrán z DOM, ale stále existuje jako objekt. Některé jiné třídy, si ale na základě ID udržují další pomocné objekty. Po odebrání elementu potřebují toto ID získat, aby i ony mohly uvolnit paměť. V takovém případě si musí

```
interface DOM {

    ...

    fun getNodeId(node: Node): NodeId?

    fun getNode(nodeId: NodeId): Node?

    fun getSiblingId(nodeId: NodeId): NodeId?

    fun add(node: Node, nodeId: NodeId,
            parentId: NodeId, siblingId: NodeId?)

    fun append(node: Node, parentId: NodeId, nodeId: NodeId? = null)

    fun remove(node: Node)

    ...
}
```

Ukázka kódu 3.15: Rozhraní třídy pro mapování ID a elementu

vytvořit vlastní hashovací tabulku mezi elementem a jeho ID. Z mnoha důvodů toto není zrovna dobré řešení, ale v současnosti neznám žádné lepší.

Základní implementace je řešená pomocí dvou hashovacích tabulek pro elementy a jejich ID. Také je potřeba speciální stromová struktura, která si pamatuje pozici ID v hierarchii stránky. Implementace si sama řeší přiřazení ID k nově vzniklým elementům.

K přidání elementu získaného od druhého klienta je možné použít metodu *add*, která element přidá na správnou pozici určenou rodičem a sousedem. Existují dvě různé implementace pro uživatele a správce, které se liší tím, jestli pracují s předchozím nebo následujícím sousedem.

Po odebrání elementu ze stránky není potřeba volat metodu *remove*. Uvolnění paměti proběhne automaticky. Metoda *remove* slouží k odstranění elementu, který byl odebrán druhým klientem.

3.3.3.3 Typy elementů

Jak jsem již dříve uvedl, existují určité typy elementů, se kterými je potřeba pracovat jinak než s ostatními. Příkladem je třeba element *input*. V ukázce kódu

(3.16) jsou různé podtypy elementu *input*. K těmto podtypům jsou přiřazené důležité vlastnosti a události, které jsou při změně těchto vlastností vyvolány.

```
val trackedProperties = when (type) {
    "button", "file", "hidden", "image", "reset", "submit" ->
        emptyList()
    "checkbox", "radio" -> listOf("checked")
    else -> listOf("value")
}

val trackedEvents = when (type) {
    "button", "file", "hidden", "image", "reset", "submit" ->
        emptyList()
    else -> listOf("input")
}
```

Ukázka kódu 3.16: Sledované vlastnosti elementu *input*

Aby bylo jednodušší přidávat postupně podporu pro nové typy elementů, má každý typ elementu přiřazenou třídu. Ta určuje, jakým způsobem se budou elementy tohoto typu sdílet. Implementace podpory pro nové typy elementů je díky tomu velmi jednoduchá. Problém je ale takové elementy najít. Je to časově velmi náročná činnost a vyžaduje hodně experimentování s různými webovými stránkami.

Existují tři elementy, které se chovají velmi odlišně od ostatních. Jsou to elementy *html*, *head* a *body*. Za normálních okolností jsou tyto elementy unikátní. Ale hlavně je po jejich odebrání většinou prohlížeč vytvoří znovu. Proto je nazývám persistentní. Toto chování nepříjemně komplikuje algoritmy, které pracují s DOM. Pokud se kterýkoliv z těchto elementů změní, je zapotřebí mu znovu nastavit původní ID, které je na rozdíl od ostatních elementů konstantní. Kód starající se o kontrolu těchto elementů je v ukázce kódu (3.17). Situace je ještě složitější kvůli tomu, že se nelze spolehnout na to, že budou tyto elementy včas vytvořeny prohlížečem. Je proto lepší je v takovém případě vytvořit za něj.

Další speciální skupinou elementů jsou takzvané *frames*. Většina z nich se v dnešní době nepoužívá. Jediným poměrně častým elementem z této skupiny je *iframe*. Pro sdílení je problematický, protože obsahuje svůj vlastní DOM. Bylo by tedy potřeba spustit celý algoritmus pro sdílení zvlášť na tento vnořený DOM. Kvůli zjednodušení implementace prototypu jsem toto zatím neudělal.

```
private fun checkKnownNodes() {
    if (idContainer.getNode(HtmlNodeId) != root) {
        idContainer.reassignIdToNode(root, HtmlNodeId,
            RootNodeId, null)
    }

    if (idContainer.getNode(HeadNodeId) != document.head ||
        document.head == null) {

        val head = document.head ?:
                   document.createElement("head")
                       .also { root.appendChild(it) }

        idContainer.reassignIdToNode(head, HeadNodeId,
            HtmlNodeId, null)
    }

    if (idContainer.getNode(BodyNodeId) != document.body ||
        document.body == null) {

        val body = document.body ?:
                   document.createElement("body")
                       .also { root.appendChild(it) }

        idContainer.reassignIdToNode(body, BodyNodeId,
            HtmlNodeId, HeadNodeId)
    }
}
```

Ukázka kódu 3.17: Řešení problému s perzistentními elementy

Při testování na stránce, která obsahovala element *svg*, jsem zjistil, že se tento element nezobrazuje u správce správně. Při detailnějším přezkumu jsem dále objevil, že některé běžné elementy mají trochu odlišný vzhled. Nejprve jsem si myslel, že problém spočívá v tom, že používám element *iframe* k zablokování JavaScriptu u správce. Můj předpoklad byl, že webový prohlížeč pracuje s obsahem *iframe* jiným způsobem než s běžnou stránkou. Ukázalo se ale, že celý problém je způsobený použitím špatného jmenného prostoru (*namespace*) při vytváření elementů u správce. Oba problémy byly vyřešeny tím, že jsem začal sdílet spolu s typem elementu i jeho jmenný prostor.

Snažil jsem najít řešení pro nové typy elementů vytvořených programátorem, o kterých jsem psal v (2.8.3.1). Jedna možnost je sdílet vlastnosti na základě *blacklistu* a nikoliv *whitelistu*, jako je tomu nyní. Bylo by tedy potřeba vypsat všechny vlastnosti elementů, které není potřeba sdílet. Stále to znamená mít

tento seznam pro každý typ elementu, ale byla by větší šance, že neznámý typ elementu by fungoval správně. Velkou nevýhodou je, že *blacklist* musí být kvůli výpočetní složitosti co největší. Jeho sestavení je tak výrazně pracnější než u *whitelistu*. *Blacklist* navíc neposkytuje žádnou jistotu, že bude neznámý element fungovat správně.

Pro vlastní typy elementů se mi tedy nepodařilo najít žádné rozumné řešení. Každý takový element musí být zkrátka podporován zvlášť. Může být proto potřeba spolupracovat se zákazníkem na zprovoznění prototypu v jeho aplikaci. Co však lze udělat, je usnadnit implementaci podpory pro tento element.

3.3.3.4 Obousměrné sdílení stránky

Cílem obousměrného sdílení je umožnit správci měnit obsah sdílené stránky. K takovýmto změnám je možné použít konzoli. Je také výhodné tímto způsobem implementovat vzdálené ovládání. Implementaci vzdáleného ovládání tedy tato funkce zjednodušuje, ale zároveň přináší nový problém při sdílení obsahu. Je totiž nutné rozlišit, které změny jsou způsobené lokálně a které jsou převzaté od druhého klienta.

Pokud při tomto rozlišení nastane chyba, dojde k nekonečné aplikaci změn. Například první klient přidá element A. Druhý klient obdrží informaci o přidání elementu A a také si ho přidá. Pokud tato změna bude považována za lokální, bude interpretována jako přidání elementu B. Po odeslání elementu B k prvnímu klientovi může dojít ke dvěma situacím. V lepším případě je tuto změnu první klient schopný správně rozlišit a dojde pouze k duplikaci elementu A. V horším případě dojde k poslání elementu C ke druhému klientovi. V ten moment se algoritmus zacyklí a bude do nekonečna přidávat elementy. Na obrázku (3.1) je ukázka, jak se může tato chyba projevit.

(a) Originál (b) Duplikace textu

Obrázek 3.1: Příklad chyby v obousměrném sdílení

Podobný problém se týká i atributů a vlastností. Zacyklení v tomto případě způsobí „pouze“ větší zátěž pro síť a procesor, zatímco stránka bude vypadat

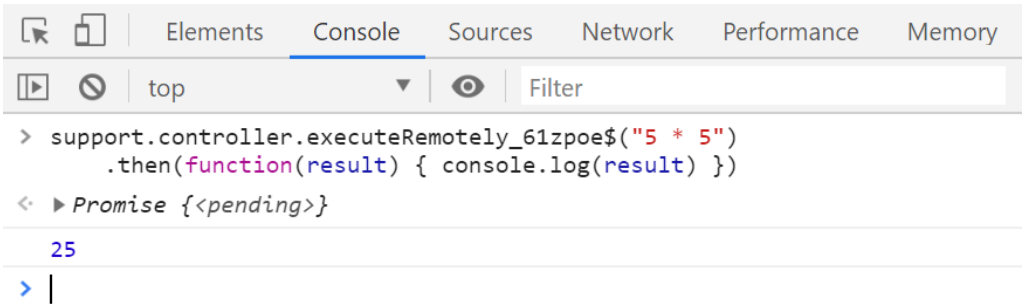
normálně. U atributů a vlastností je také výrazně větší pravděpodobnost, že dojde pouze ke dvojnásobné aplikaci změn. Při přidávání elementů je ale pravděpodobnější nekonečné zacyklení.

Kvůli tomu, jak funguje *MutationObserver*, není jednoduché změny správně rozlišit. Obrázek (3.1) dokazuje, že v současné implementaci prototypu se chyby tohoto typu stále vyskytují, i když už jsem několik z nich opravil. Nikdy není záruka, že po opravení jedné této chyby, ještě nějaká další nezbývá. Bylo by proto potřeba vytvořit automatické testy, které by usnadnily hledání a opravu těchto chyb.

3.3.3.5 Vzdálená konzole

Vzdálená konzole byla nejjednodušší funkcí, kterou jsem v rámci co-browsingu implementoval. Přesto jsem se ani zde zcela nevyhnul problémům. Při testování přenosu většího množství záznamů z konzole, jsem například zjistil, že WebRTC má omezenou velikost přenášených paketů.

První částí vzdálené konzole je vzdálené volání JavaScriptu. To je možné vidět v reálném použití na obrázku (3.2). Na tomto obrázku je konzole správce, ve které může uvedeným způsobem spustit libovolný kód u uživatele. Druhý řádek kódu slouží pouze k vytisknutí odpovědi z *promise*. Podivná syntaxe volání metody *executeRemotely* je způsobená tím, že veškerá implementace je v Kotlinu. Při kompilaci do JavaScriptu dochází k takzvanému *name mangling*, který způsobuje změnu názvu metody. Do budoucna bylo by vhodné obalit tuto metodu do nějaké JavaScriptové funkce s lepším názvem.

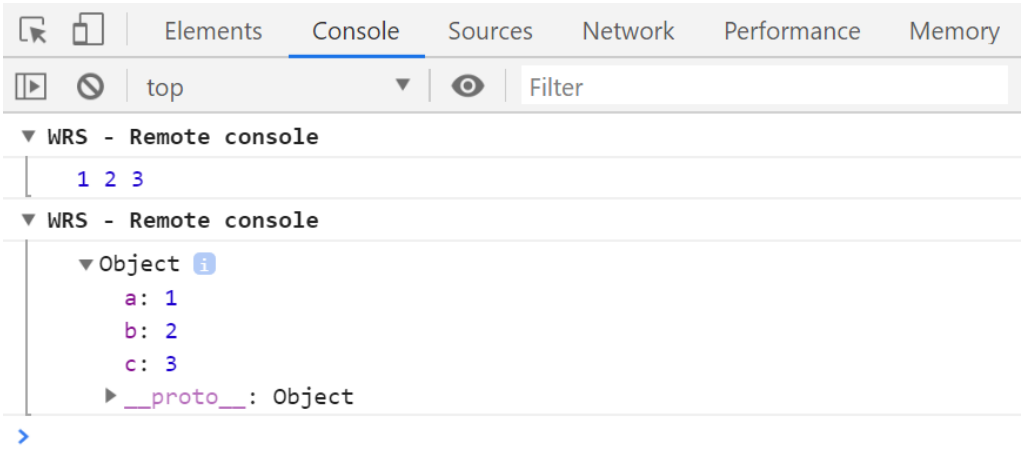


Obrázek 3.2: Vzdálené volání JavaScriptu

Druhou částí vzdálené konzole je zobrazení jejího obsahu. Na obrázku (3.3) jsou ukázány dva záznamy z uživatelovy konzole. Vzdálené záznamy jsou pro větší přehlednost od těch lokálních oddělené pomocí odsazení.

Není možné číst obsah konzole přímo. K získání záznamů používám reflexi, pomocí které nahrazuji implementaci metod objektu *console*. Každou z těchto metod upravuji tak, aby byl při jejím zavoláním navíc proveden kód, který pošle správci informaci o zavolání této metody i s jejími argumenty. U správce je poté možné zobrazit volání těchto metod libovolným způsobem.

Toto provedení má dvě limitace. První limitací je, že nelze získat záznamy, které vznikly před tím, než byl upraven objekt *console*. To je možné řešit tak, že se tato úprava provede před tím, než se na stránce spustí další skripty. Druhá limitace se týká záznamů, které pocházejí z prohlížeče. Předpokládám, že prohlížeč má možnost zapisovat do konzole i jinak než přes objekt *console*. Vycházím z toho, že ne všechny záznamy se mi podařilo zachytit. Tímto způsobem nelze tedy například získat chyby způsobené porušením bezpečnosti, kdy prohlížeč zablokuje nějakou akci a do konzole napíše proč.



Obrázek 3.3: Zobrazení obsahu vzdálené konzole

K přenosu argumentů pro vzdálenou konzoli se vztahuje jedna nepříjemná vlastnost serializace v JavaScriptu. Ne všechny vlastnosti objektů jsou automaticky serializovány. Dobře je to vidět například u libovolného elementu. Při výpisu v konzoli má celou řadu vlastností, ale výsledkem jeho serializace je

prázdný objekt. To představuje vážný problém, protože by se při přenosu argumentů ztratila významná část záznamu z konzole. Řešením je implementovat si vlastní serializaci.

Dalším důvodem pro vytvoření vlastní serializace jsou cyklické reference. JSON nemá pro cyklické reference podporu, takže standardní serializace vyhodí při detekování cyklu výjimku. To opět není vhodné chování pro sdílení obsahu konzole. Ideálním řešením by bylo cykly detekovat a zakódovat je do JSON. To je poměrně obtížné, takže jsem zvolil jednodušší způsob. Ten spočívá v omezení, kolik vlastností se maximálně serializuje. Po vyčerpání tohoto limitu se již další vlastnosti nezahrnou do výsledného JSON. Stejným způsobem lze limitovat počet vnořených objektů.

Je vhodné nějak rozumně omezit počet serializovaných vlastností. Serializace a přenos JSON o několika megabajtech trvá řádově několik sekund. V prototypu používám omezení na tisíc vlastností a deset zanoření.

3.3.3.6 Další problémy

Při realizaci sdílení stránky jsem postupně nalézal problémy, které jsem již popsal v (1.2.2). Zde jsem také uvedl k jakým řešením jsem došel. V prototypu jsem uvedené způsoby řešení nepoužil, protože byly časově příliš náročné na implementaci.

Jedinou výjimkou je úprava relativních adres, bez kterých by prototyp v podstatě nefungoval. K tomu jsem využil API pluginů v Chrome, které dovoluje měnit adresy dotazů před tím, než je prohlížeč pošle na server.

Nepříjemnou vlastností tohoto API je, že vyžaduje, aby změny URL byly provedeny synchronně. To efektivně znemožňuje komunikaci mezi kódem, který mění URL, a zbytkem pluginu. Jenže pro změnu URL je potřeba vědět, jaká je absolutní adresa serveru. To je informace, která lze získat pouze od uživatele. Tento problém jsem nedokázal vyřešit jinak, než opakovaným načtením obsahu stránky správce. Při druhém načtení je již adresa serveru asynchronně předána, takže je možné URL upravit. Dvojité načtení stránky je potřeba pouze, pokud je prohlížečem vyžádána relativní adresa před tím, než se stihne předat adresa serveru. Nedochází k tomu tedy příliš často.

Tento způsob řešení relativních URL nelze rozumně použít v produkčním nasazení. Mnohem lepší řešení je nahrazení URL přímo v kódu. To je ale výrazně složitější, takže pro prototyp jsem se rozhodl použít plugin.

Původně jsem chtěl podobným způsobem řešit i soukromé zdroje. Vymyšlené jsem to měl tak, že nahradím každý dotaz datovou URL, kterou si před tím vyžádám od uživatele. Uživatel by datovou URL získal tak, že by se dotázal serveru na data zdroje. Z těchto dat by poté vytvořil datovou URL pomocí kódování *Base64*. Tuto funkcionalitu jsem i implementoval, ale zjistil jsem, že je potřeba minimalizovat počet takto přenášených zdrojů.

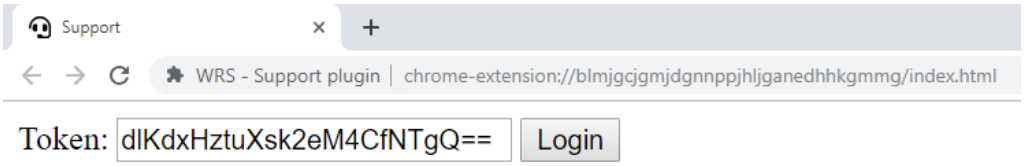
Větší webová stránka obsahuje klidně i několik desítek megabajtů dat. Díky kešování ale nemusí prohlížeč tato data stahovat při každém načtení stránky. Moje implementace žádné kešování nepoužívala, protože je obtížné zjistit, jaké zdroje je potřeba znova načíst a jaké ne.

Přenos takto velkého množství dat při každém načtení nové stránky způsoboval u správce několika sekundovou prodlevu. Z výkonových důvodů jsem tedy přeposílání všech zdrojů vypnul do té doby, než přidám algoritmus, který bude efektivně rozhodovat o tom, které zdroje jsou opravdu potřeba. Ideální stav je, aby se přenášely pouze zdroje, ke kterým správce nemá přístup a které ještě v minulosti od uživatele nezískal.

3.3.3.7 Plugin správce

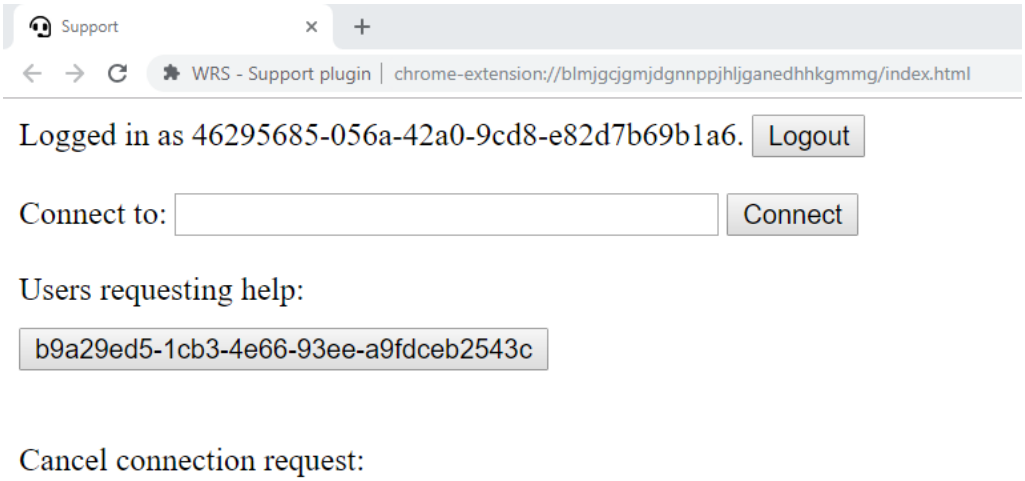
Tento plugin poskytuje uživatelské rozhraní pro správce. V rámci prototypu jsem neřešil grafický návrh ani jednoho z pluginů. Mým cílem bylo pouze udělat co nejjednodušší grafické rozhraní, aby klienti nemuseli používat k ovládání prototypu konzoli.

Po otevření stránky uložené v pluginu, se zobrazí jednoduchý přihlašovací formulář. Do něho je potřeba zadat token správce, který byl vytvořen spolu s účtem. Ukázka tohoto formuláře je na obrázku (3.4).



Obrázek 3.4: Přihlášení do pluginu správce

Po přihlášení uvidí správce obrazovku podobnou obrázku (3.5). Zde může správce zahájit spojení s nějakým uživatelem. K tomu buďto vyplní jeho ID nebo klikne na příchozí žádost o podporu.



Obrázek 3.5: Hlavní stránka pluginu správce

Adresa autentizačního serveru je v pluginu určená napevno. Nelze tedy změnit prostřednictvím grafického rozhraní. To odpovídá i reálnému použití, protože tento plugin je sice určený různým zákazníkům, ale autentizační server je pro všechny společný.

Pro použití pluginu jsem se rozhodl pouze kvůli jednoduššímu řešení relativních adres. V momentě, kdy budou tyto adresy řešeny jiným způsobem, už nebude žádný důvod pro použití pluginu. Poté bude lepší udělat místo pluginu klasickou webovou stránku.

3.3.3.8 Plugin uživatele

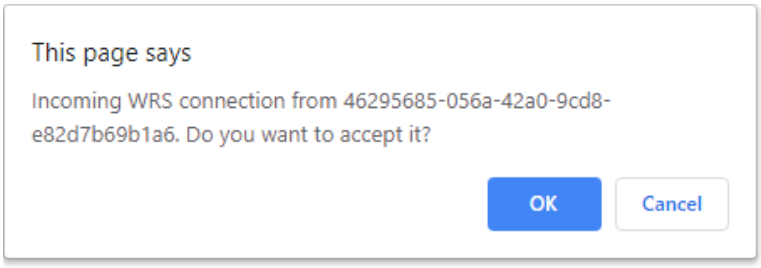
Plugin uživatele je udělaný tak, aby byl součástí webové stránky, kterou dostane uživatel od serveru. Z hlediska grafického rozhraní je výrazně jednodušší než plugin správce, aby příliš nenarušoval původní stránku.

Přítomnost pluginu v aplikaci je možné poznat podle modře ohraničeného rámečku v levém spodním rohu stránky. Na obrázcích (3.6) jsou dvě varianty tohoto rámečku. První varianta obsahuje tlačítko, pomocí kterého může uživatel požádat správce o podporu. Po jeho stisknutí dojde k přepnutí na druhou variantu, jejíž tlačítko umožňuje žádost zrušit.

(a) Požádání o podporu (b) Zrušení žádosti o podporu

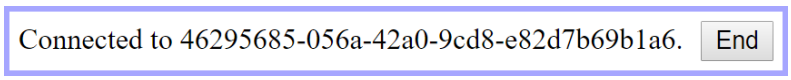
Obrázek 3.6: Ovládání žádosti o podporu

Součástí grafického rozhraní pluginu je i několik dialogů. Z nich nejdůležitější je na obrázku (3.7) a slouží k potvrzení žádosti o spojení. Další dialogy jsou pouze informační.



Obrázek 3.7: Dialog pro potvrzení žádosti o spojení

Při probíhajícím co-browsingu nedává smysl, aby mohl uživatel nadále žádat o podporu. Místo toho je nutné, aby mohl co-browsing zrušit. Proto se po potvrzení spojení změní obsah modrého rámečku do podoby zachycené na obrázku (3.8).



Obrázek 3.8: Tlačítko pro ukončení spojení

3.3.4 Vzdálené ovládání

Ze všech tří částí byla implementace vzdáleného ovládání ta nejjednodušší. Pro představu má pouze několik set řádků kódu. Návrh principů funkce, tak jak je vysvětlený v (2.8.4), se mi podařilo udělat na první pokus. Na rozdíl od sdílení stránky jsem tak nemusel postupovat iterativně.

Implementačně není vzdálené ovládání vůbec složité. Jedinou výjimkou je detekce výchozích akcí událostí. Tato detekce je potřeba, aby bylo možné zjistit, jakou akci vrátit zpět při zablokování události. Problémem je, že některé

výchozí akce je možné zablokovat pomocí dvou událostí. Například událost *MouseDown* a *Click* jsou spolu propojené. Takže když dojde k zablokování *MouseDown*, tak se *Click* vůbec nevyvolá. Při znalosti tohoto chování už není takový problém vymyslet řešení.

Vzdálené ovládání je velmi obtížné udělat stoprocentně správně. Na vině je programátor webové aplikace. Ten totiž může napsat JavaScript, který bude reagovat na události a měnit tak chování stránky. Jenže je poměrně hodně skoro ekvivalentních způsobů, jak lze s událostmi pracovat. Pokud programátor použije nějaký jiný způsob, než se kterým se počítá, nebude vzdálené ovládání fungovat. Nezbývá nic jiného, než testovat co nejvíce různých aplikací, které používají různé frameworky, a hledat chyby.

Tento problém umocňuje fakt, že se události za dobu vývoje HTML značně změnily. Takže existuje celá řada zastaralých událostí a jejich vlastností. Některé vlastnosti jsou také podporovány jenom v některých prohlížečích. Jeden konkrétní případ, se kterým jsem se setkal, je událost *keypress*. Ta je označená jako zastaralá, ale já jsem se ji v rámci zpětné kompatibility rozhodl podporovat. V prohlížeči Chrome ale tato událost způsobovala dvojitou aplikaci výchozí akce, takže jsem ji musel opět odebrat. Je otázka, jak se tato událost chová v jiných prohlížečích. Její absence ale zcela jistě může způsobit problémy v nějaké starší aplikaci.

V návrhu jsem zmínil, že pro správné zpracování událostí je potřeba brát v potaz i typ elementu, ke kterému se událost váže. V ukázce kódu (3.18) je funkce, která se stará o zablokování akcí u správce pro události týkající se myši. Jejím úkolem je zabránit v provedení akcí, které se musí vykonat pouze u uživatele. Příkladem je třeba kliknutí na odkaz. Na druhou stranu u elementu typu *input* je hodně situací, kdy je žádoucí akci provést, protože otevírá nějaký dialog.

Tato funkce je nejlepší ukázkou toho, jak na sobě závisí elementy a události. Naštěstí je to co do počtu výjimek ta nejsložitější funkce. Existuje ještě několik dalších míst, kde je v kódu potřeba rozlišit typ události a elementu, ale ty jsou podstatně jednodušší.

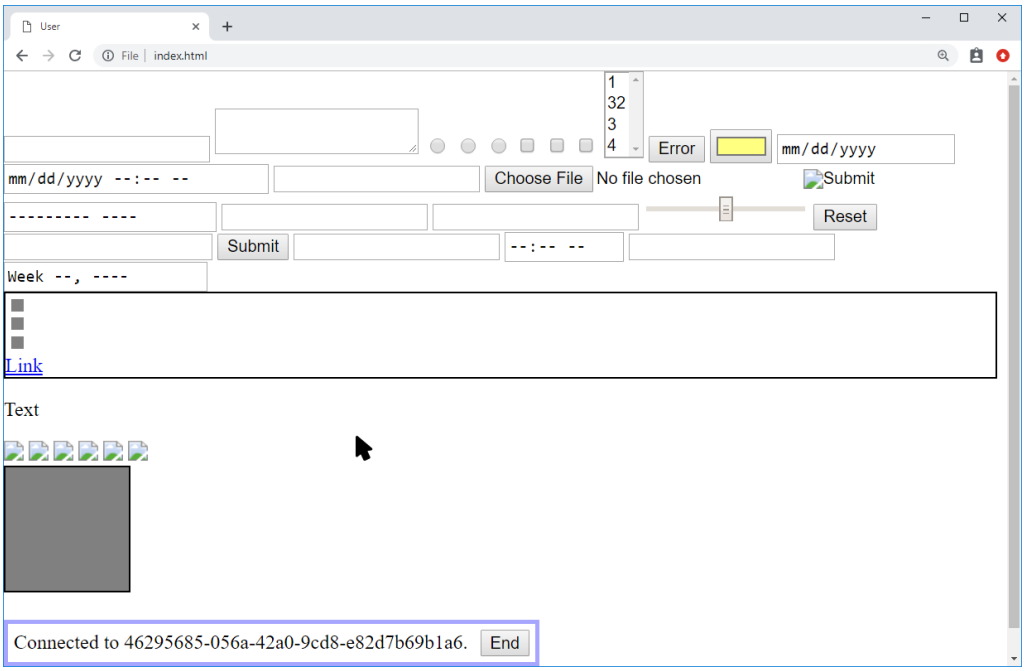
```
fun preventDefaultIfNecessary(event: dynamic) {
    if (event.button in 2..10) {
        event.preventDefault()
        return
    }

    val target = event.target
    val targetName = event.target?.nodeName?.toLowerCase()

    if (event.type == "mouseup" && targetName == "input") {
        val type = getAttribute(target, "type")?.toLowerCase()
        when (type) {
            "checkbox", "radio" -> event.preventDefault()
        }
    }

    if (event.type != "click") return

    when (targetName) {
        "input" -> {
            val type = getAttribute(target, "type")?.toLowerCase()
            when (type ?: "text") {
                "text", "color", "date", "datetime-local",
                "email", "month", "number", "password",
                "search", "tel", "time", "url", "week" -> return
            }
        }
        "textarea" -> return
    }

    event.preventDefault()
}
```

Ukázka kódu 3.18: Blokování výchozí akce podle typu události a elementu

3.4 Ladění a oprava chyb

Při vývoji všech částí jsem testoval a opravoval chyby průběžně. K tomu jsem si vytvořil speciální stránku. Její obsah jsem udělal tak, aby co nejlépe ověřovala funkčnost prototypu. Během vývoje jsem do této stránky postupně přidával další funkce. Výsledný vzhled této testovací stránky je na obrázku (3.9).

Na testovací stránce jsou všechny typy formulářových elementů, které jsou důležité při testování vzdáleného ovládání. Spolu s odkazy jsou formuláře v podstatě jedinými interaktivními elementy v HTML. Také je zde JavaScript, který reaguje na různé události. Například šedý čtverec v levém spodním rohu stránky je naprogramovaný tak, aby změnil barvu, pokud na něj zrovna ukazuje



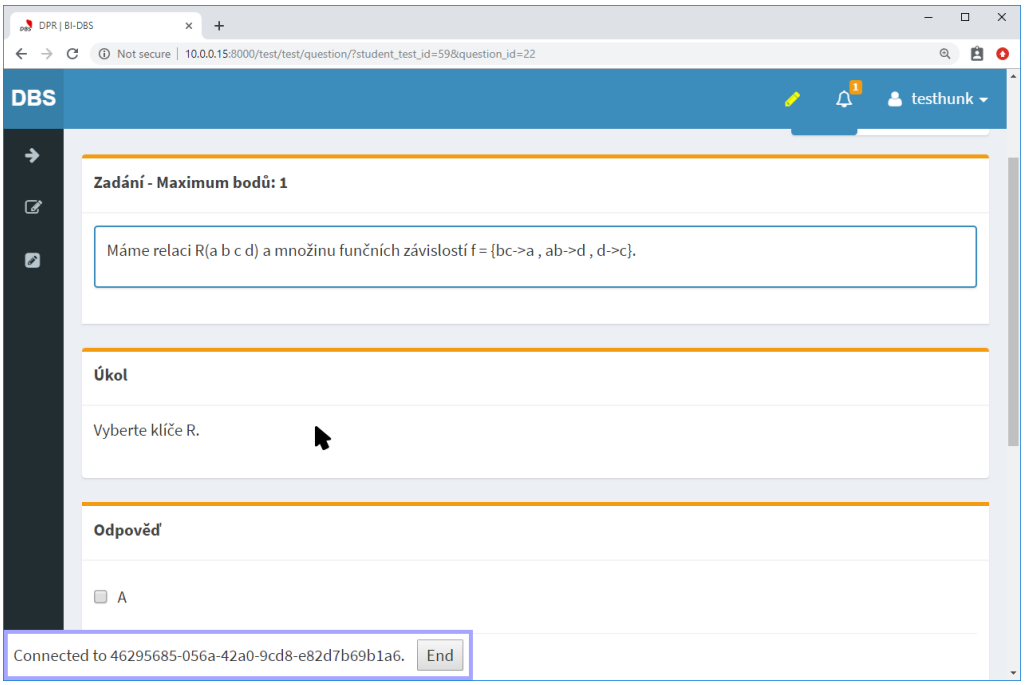
Obrázek 3.9: Speciální stránka pro testování

kurzor. Další JavaScript slouží k ověření, zda se u správce správně blokuje jeho vykonání.

Díky této stránce se celkem dobře testuje základní sdílení obsahu a vzdálené ovládání u malých stránek, které neobsahují příliš velké množství JavaScriptu. Také jsem ale potřeboval otestovat, jak prototyp funguje ve větších a podstatně složitějších aplikacích, které se skládají z mnoha stránek. K tomuto účelu jsem zvolil portál DBS [39].

Portál DBS je velmi specifický tím, že obsahuje velké množství formulářů. Každá jeho stránka je tedy velmi interaktivní. Zároveň jsou tyto stránky napsané tak, že využívají JavaScript téměř ke všemu. Portál DBS proto docela dobře testuje možnosti prototypu. Ukázka z testování v portálu je na obrázku (3.10).

Mohlo by se zdát zvláštní, že jsem pro testování nepoužil i OptiLynx, pro který je tato práce ve výsledku určená. Důvodem je, že jsem chtěl OptiLynx využít až při závěrečném vyhodnocení v kapitole (4). Kdybych na něm prototyp testoval od samého začátku, tak bych na konci neměl možnost zhodnotit, jak dobře by prototyp fungoval v aplikaci, pro kterou nebyl speciálně upraven.



Obrázek 3.10: Ukázka testování v portálu DBS

V průběhu testování jsem našel a opravil celou řadu chyb, o kterých jsem již psal v předchozích částech této kapitoly. Kromě těchto poměrně závažných problémů jsem našel i velké množství „drobností“, které jsem zvlášť nerozepisoval. Pro představu jich pár uvedu zde.

V portálu DBS jsem například zjistil, že kurzor myši druhého klienta měl špatně nastavenou pozici v ose z. V testovací stránce s tím nebyl žádný problém, ale v portálu byl kurzor zakrytý dialogy.

U kurzoru byl problém i s pozicí v ostatních osách. To se projevovalo tak, že uživatel viděl, jak správce kliká na jedno zaškrtávací pole, ale ve skutečnosti označil pole sousední. Problém nebyl ve výpočtu pozice události ale v tom, že každý obrázek ukazatele má jiné rozměry. Takže při změně typu kurzoru (ukazoval na ovladatelný element) došlo k optickému posunutí tohoto kurzoru.

Do třetice zmíním vlastnost *keycode* u události *keydown*. V portálu DBS nefungovaly kvůli absenci této vlastnosti některé textové formuláře. Ty jsou ovládané JavaScriptem, který tuto vlastnost používá. Na tuto chybu jsem nepřišel dříve, protože událost *keydown* má jiné ekvivalentní vlastnosti, které *keycode* nahrazují.

Výše sepsané chyby zde uvádím pro ilustraci toho, kolik detailů je potřeba při co-browsingu řešit. Ani důkladné testování v jedné nebo dvou aplikacích nezaručuje, že budou všechny takovéto drobné chyby nalezeny. Jejich projevení závisí čistě na tom, jak je zrovna stránka naprogramovaná. Neexistuje proto lepší způsob, jak tyto chyby opravit, než při reálném použití.

Kapitola 4

Vyhodnocení

V této kapitole popíšu závěrečné testování prototypu v aplikaci OptiLynx [1]. Na základě výsledků z testování provedu vyhodnocení použitelnosti prototypu. Před tím, než je možné prototyp použít, je potřeba ho nasadit do cílové aplikace. Do vyhodnocení proto zahrnu i obtížnost tohoto nasazení.

V druhé části kapitoly uvedu seznam různých vylepšení. Tato vylepšení jsou potřeba udělat, aby bylo možné prototyp použít i komerčně. Seznam jsem sestavoval průběžně při tvorbě práce. Jsou v něm díky tomu zahrnuté poznatky z vývoje, z testování existujících produktů pro co-browsing i z testování mého prototypu.

4.1 Závěrečné testování v reálné aplikaci

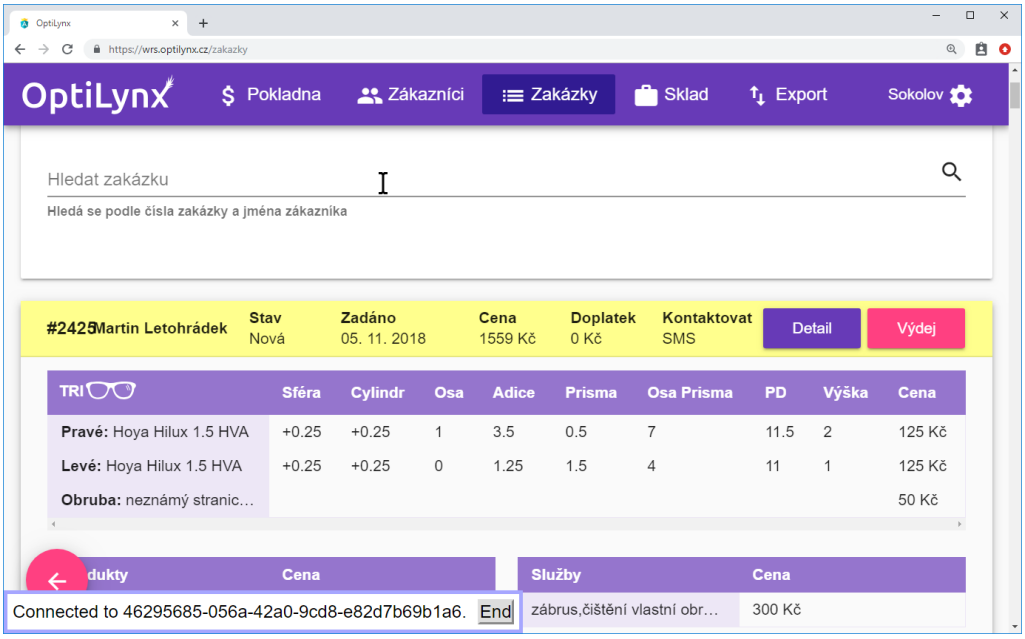
Pro závěrečné testování jsem zvolil aplikaci OptiLynx, protože pro ni je výsledný prototyp určený. Při nasazení autentizačního serveru a pluginu uživatele do aplikace mi asistovali Bc. Pavel Kovář a Ing. Filip Glazar, kteří se starají o provoz této aplikace. Díky nim proběhlo nasazení velmi hladce.

Autentizační server jsem nasadil s pomocí technologie Docker [40]. Vytvoření takzvaného *Docker image* bylo jednoduché a jedinou překážkou bylo nalezení správné verze Javy. Na serveru už byla databáze PostgreSQL připravená, protože se používá i pro jiné projekty. Stačilo v ní tedy vytvořit nového uživatele a databázi. Při nasazení bylo jediným problémem správně nastavit komunikaci autentizačního serveru a proxy, která se stará o zajištění HTTPS.

Integrace pluginu do stránky byla také jednoduchá. Stačilo přidat soubory obsahující zkompilovaný kód Kotlinu a několik obrázků pro zobrazení vzdáleného kurzoru. Posledním krokem bylo přidat odkaz na soubory do hlavičky HTML. Díky architektuře aplikace to bylo možné udělat změnou jednoho souboru.

Pro testování byla vytvořena speciální subdoména, aby testování neovlivnilo běžný provoz aplikace. K tomuto účelu byl také vytvořen jeden účet, kterému jsem přes autentizační server přiřadil token. Nasazení jsem si tedy trochu zjednodušil, protože jsem nepřidal podporu pro více účtů.

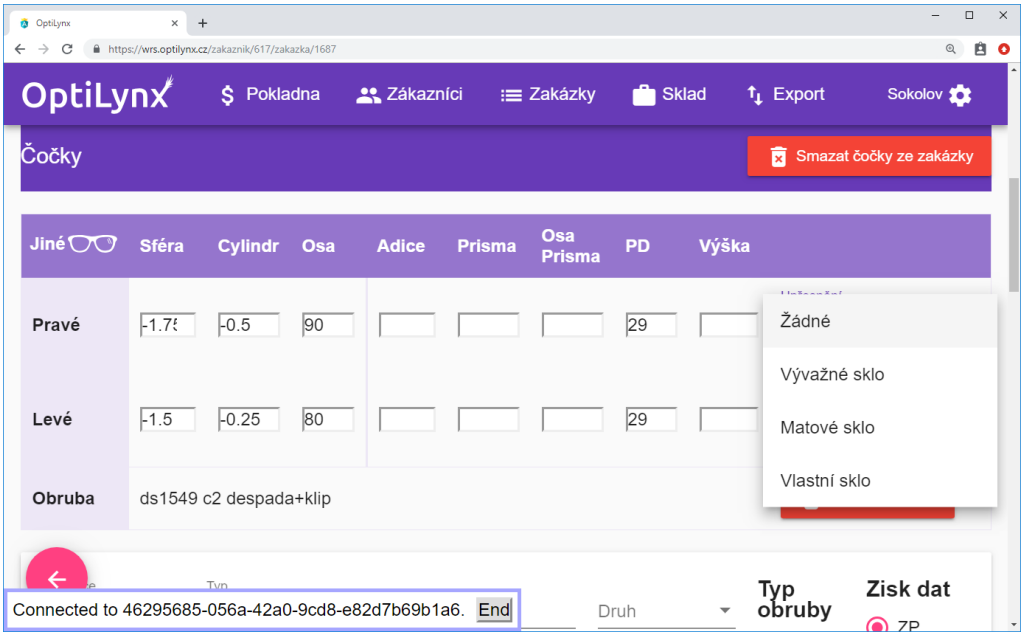
Testování jsem zahájil tím, že jsem coby uživatel prošel velkou část aplikace a porovnával jsem, jak funguje sdílení stránky. Ukázka z testování je na obrázku (4.1). Sdílení stránky bylo v pořádku. Akorát u stránky *Zakázky* z obrázku (4.1), jsem pozoroval, že její načtení trvá u správce přibližně tři sekundy. Problém není v prototypu jako takovém, ale ve velikosti té stránky. Má více než 5 000 elementů a její načtení trvá podobně dlouhou dobu i bez zapnutého sdílení.



Obrázek 4.1: Ukázka testování v aplikaci OptiLynx

Při testování vzdáleného ovládání jsem narazil na poměrně velké problémy. V první nasazené verzi prototypu nebylo možné stránku dostatečně dobře ovládat. V OptiLynx se používá velké množství dialogových oken k výběru

položky ze seznamu. Příklad takového dialogu je na obrázku (4.2). Tyto dialogy nemohl správce otevírat. Přesněji řečeno v dialogu, který otevřel správce, nemohl ani jeden z klientů vybírat položky. Dialog nebylo možné ani zavřít. S dialogem, který otevřel uživatel, mohl pracovat i správce.

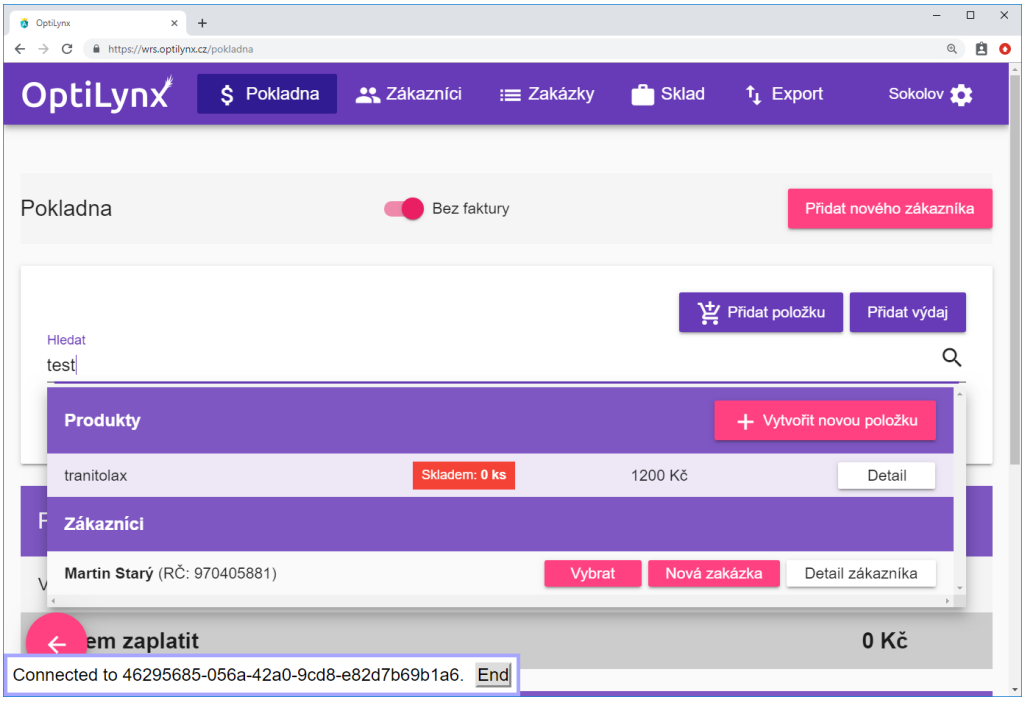


Obrázek 4.2: Dialog pro výběr ze seznamu

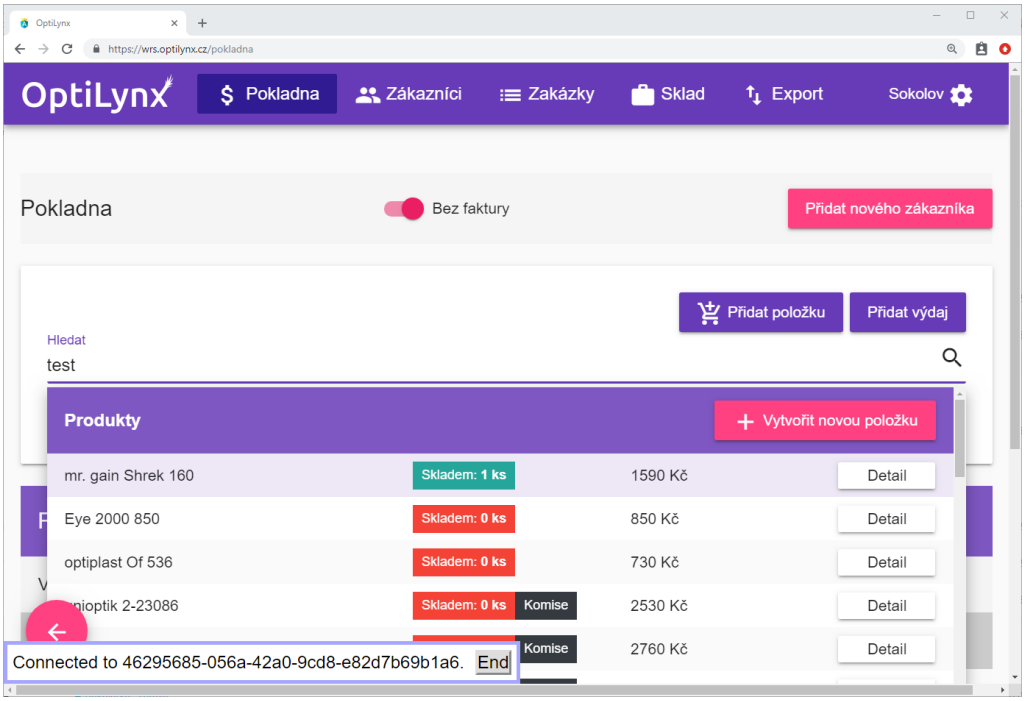
Tuto chybu jsem opravil a pokračoval v testování. Byla způsobena tím, že pokud dialog otevřel správce, došlo k překrytí tohoto dialogu jeho kopií. Tato kopie nebyla vytvořena příslušným JavaScriptem, takže ji žádný JavaScript neovládal, a tudíž nereagovala na klikání. Toto byla opět klasická chyba zduplikování přidaného elementu, o které jsem psal v (3.3.3.4).

Druhý problém, který jsem nalezl, byl spojený s ovládáním klávesnicí. Stránka se sice tvářila, že do ní správce může psát, ale ve skutečnosti nefungovala správně. Problém byl, že se obsah textových formulářů nesynchronizoval s JavaScriptem. Toto je dobře vidět na obrázcích (4.3) a (4.4). Na prvním obrázku je výsledek vyhledávání, které provedl uživatel. Na druhém obrázku je vyhledán tentýž text, ale tentokrát ho napsal správce. Formulář sice zobrazil správný text, ale výsledek vyhledávání je úplně jiný.

Tuhle chybu jsem při prvním testování úplně přehlédl. Přišel jsem na ni náhodou, když jsem se neúspěšně snažil přihlásit do aplikace. Původně jsem



Obrázek 4.3: Výsledek vyhledávání uživatelem



Obrázek 4.4: Výsledek vyhledávání správcem

si ji nevšiml ani při testování Acquire v (1.4.2). Proto jsem zkusil podobný test i v Acquire a tím jsem zjistil, že ani v něm nefunguje psaní textu správně.

Tato chyba je způsobená tím, že je aplikace napsaná ve frameworku Angular [41], který automaticky mapuje hodnoty z textových polí do datového modelu v JavaScriptu. Při tom musí řešit problém se sledováním změn vlastností elementů. Je to ten stejný problém, který jsem řešil při implementaci sdílení stránky.

Angular sledování vlastností neřeší žádným automatizovaným způsobem. Pokud chce programátor měnit v Angular obsah formulářů externím JavaScriptem, je zodpovědný za to, že na konci vyvolá událost *input*. Po zjištění této informace jsem již neměl problém prototyp upravit tak, aby událost při každé změně vyvolal. Ve výsledku tedy tato chyba vznikla kombinací toho, jak funguje vzdálené ovládání a Angular.

4.2 Vyhodnocení použitelnosti prototypu

Na základě testování v aplikaci OptiLynx jsem dospěl k následujícím poznatkům. Nasazení prototypu do aplikace je dostatečně jednoduché. S krátkým návodem jej zvládne každý programátor. Autentizační server je pro všechny zákazníky stejný, takže o jeho provoz se nemusí starat.

Jako největší slabinu prototypu vidím to, že ho po nasazení může být nutné speciálně upravit pro aplikaci. Při testování v OptiLynx jsem našel v prototypu dvě nové závažné chyby, které znemožňovaly aplikaci správně ovládat.

První chyba souvisela s implementací. Měla být proto nalezena už při testování. Tento typ chyb lze do budoucna minimalizovat lepším testováním.

Druhá chyba byla z velké části způsobena frameworkem, ve kterém je aplikace napsáná. Tento typ chyb nelze dopředu dobře podchytit. Kvůli nim nemusí prototyp při prvním nasazení vůbec fungovat. Stejným problémem trpí i existující produkty popsané v (1.4). Jejich autoři to řeší tím, že aktivně nabízejí pomoc při integraci jejich produktu s aplikací.

Poté, co jsem odstranil všechny nalezené chyby, už bylo možné prototyp používat pro vzdálenou správu v OptiLynx. Prototyp splňuje požadavky, které na něj klade zadání. Z tohoto pohledu je tedy dobře použitelný.

Nabízí se srovnání s konkurenčními produkty. Ty jsou samozřejmě mnohem více propracované a navíc nabízí další funkce. Ale pokud budu srovnávat pouze

sdílení stránky a vzdálené ovládání, nevychází můj prototyp vůbec špatně. V OptiLynx funguje moje řešení v podstatě nejlépe. To je hlavně díky tomu, že jsem ho mohl na míru upravit. Předpokládám, že autoři Acquire by byli také schopni opravit chybu s nefunkčním ovládáním pomocí klávesnice. Surfly nefunguje v OptiLynx vůbec a myslím si, že vzhledem k tomu, jak používá proxy server, jej není možné jednoduše opravit. Sdílení stránky v Upscope je srovnatelné s mým prototypem a Acquire. Upscope ale nemá vůbec podporu pro klávesnici, takže přes něj není možné aplikaci plnohodnotně ovládat.

Je nutné dodat, že prototyp není úplně připravený pro produkční nasazení. Je ho potřeba ještě výrazně lépe otestovat a vytvořit automatické testy. Také by to chtělo vylepšit grafické rozhraní.

4.3 Návrhy na budoucí vylepšení

V průběhu práce jsem stále vymýšlel nové způsoby, jak prototyp zlepšit. Některá zlepšení jsem rovnou do prototypu přidal. Musel jsem ale někde udělat hranici, protože některá vylepšení už jsou hodně mimo zadání a hlavně na ně nezbyl čas.

V předchozích kapitolách jsem u vybraných problémů naznačil lepší řešení, než jaké v současnosti používám. Pro přehlednost zde uvádím ty nejdůležitější znovu. Spolu s nimi jsou zde i návrhy na přidání nových funkcí. Většinu z nich jsem převzal z existujících produktů pro co-browsing, o nichž jsem psal v (1.4).

Návrhy jsem rozdělil do čtyř kategorií. V každé z nich je několik nejdůležitějších zástupců. Rozhodně však nejde o vyčerpávající seznam, protože prostor pro další zlepšení je opravdu velký.

Vylepšení implementace Kód prototypu jsem psal s tím, že se jedná spíše o technologické demo a nikoliv o reálný produkt. Proto je potřeba do budoucna některé jeho aspekty zlepšit.

- Automatické testy a refaktoring – Kvůli způsobu, jakým jsem prototyp vyvíjel, nebylo vhodné psát automatické testy současně s kódem. Pro snížení počtu chyb a zvýšení spolehlivosti jsou ale potřeba dodělat. Díky nim bude snazší refaktorovat existující kód, což je také potřeba udělat.

- Odebrání pluginu správce – Plugin správce se momentálně používá pro zjednodušení nahrazování relativních URL. Bylo by lepší tyto URL nahradit přímo v HTML a plugin odebrat.
- Přepis autentizačního serveru do Spring – Pro současné použití je autentizační server dostatečný. Pokud by ale do něj bylo potřeba přidat další funkce, bylo by vhodné jej namísto refaktorování rovnou přepsat do technologie Spring.
- Vlastní STUN a TURN server – Provozovat vlastní STUN a hlavně TURN server má smysl pouze v kombinaci s přidáním videohovoru. Vlastní STUN server může být potřeba, pokud se ukáže, že ty veřejně poskytované nejsou dostatečně spolehlivé.

Optimalizace Při návrhu formátu posílaných dat jsem příliš nebral v potaz velikost výsledného datového toku. Spíše jsem se snažil, abych mohl na základě odeslaných a přijatých zpráv dobře ladit pluginy. Správnou optimalizací je možné snížit datový tok několikanásobně. To také bude mít pozitivní vliv na rychlost při načítání nových stránek u správce.

- Slovníková komprese – Velká část přenášených dat je pokaždé stejná. Například názvy většiny elementů, jejich atributů, vlastností a jmenných prostorů jsou dané standardem HTML. Je tedy možné na tyto názvy použít vlastní slovníkovou kompresi. Slovník pro kompresi může být u obou klientů, takže ho navíc není potřeba přenášet s daty.
- Nahrazení JSON – Formát JSON není příliš efektivní z hlediska množství dat, které je potřeba přenést navíc. Na místo něho by bylo lepší použít vlastní binární formát, který by nemusel obsahovat metadata.
- Synchronizace změn textu po částech – Textové atributy a vlastnosti jsou synchronizovány v celku. To znamená, že pokud se změní jeden znak v textu s tisíci slovy, je potřeba znovu poslat celý text. Lepším řešením je najít části textu, které se změnily, a ty synchronizovat zvlášť.

Zlepšení co-browsingu Následující zlepšení se týkají sdílení stránky a vzdáleného ovládání. Tato zlepšení nejsou příliš důležitá, takže je ani existující produkty většinou nepodporují. Přesto si myslím, že se je vyplatí přidat.

- Soukromé zdroje – Soukromé zdroje jsem se už snažil v prototypu vyřešit. Zatím jsem ale nedokázal dostatečně dobře rozhodovat o tom, které zdroje se mají přenášet od uživatele. Přenášení všech zdrojů zbytečně zpomalovalo sdílení stránky, a proto jsem ho v prototypu vypnul. Pro dodělání této funkce stačí vyřešit kešování zdrojů a detekci, které zdroje jsou opravdu soukromé.

- Hover – Pseudo třída *hover* slouží ke změně vzhledu elementů, v závislosti na pozici kurzoru myši. Tuto pseudo třídu není možné JavaScriptem aktivovat, a proto při vzdáleném ovládání nemusí být u druhého klienta stránka správně zobrazená. Je potřeba *hover* nahradit ve všech výskytech v HTML normální třídou, kterou pak bude JavaScript přiřazovat elementům.

- Výběr aktivního elementu – Správce aktuálně nemůže pro uživatele označit formulář, do kterého má psát. Problematičtější je, když se na stránce nachází JavaScript, který mění aktivní element. U správce potom není tento element aktivován, takže se správci se stránkou hůř pracuje. Řešením je sdílet mezi oběma klienty informaci o tom, který element je aktivní.

- Výběr textu ve formuláři – Označení textu souvisí s výběrem elementu. Je to opět nutné řešit kvůli JavaScriptu, který bez toho nemůže správci usnadňovat ovládání stránky. Výběr textu je specifikovaný vlastnostmi *selectionStart* a *selectionEnd*. Stačí tedy v kódu označit tyto vlastnosti jako sdílené.

- Rolování stránky – Mezi klienty se nesynchronizuje rolování stránky, takže uživatel u delší stránky neví, se kterou její částí správce pracuje. Je proto potřeba přidat synchronizaci rolování, aby oba klienti viděli stejnou část stránky.

Nové funkce Co-browsing je pouze jednou z částí řešení pro komerčně použitelnou vzdálenou správu. Následující funkce jsou tím hlavním rozdílem mezi mým prototypem a existujícími produkty. Po jejich přidání bude mé řešení konkurenceschopné těmto produktům.

- Web pro konfiguraci – Konfigurace autentizačního serveru pomocí REST API je pro zákazníka složitá, protože k ní potřebuje programátora. Web s grafickým rozhraním by umožnil stejnou činnost dělat i manažerovi. Součástí webu by navíc mohl být i návod pro nasazení co-browsingu do aplikace. Dále by zde mohla být stránka pro správce, která je v současné době umístěná v pluginu.

- Jednořádkové nasazení – Aktuální plugin uživatele vyžaduje, aby se při jeho nasazení přidalo do aplikace několik souborů. Tento krok lze ale zjednodušit napsáním krátkého skriptu, který si požadované soubory stáhne například z autentizačního serveru. Zákazníkovi by potom stačilo vložit tento skript do aplikace.

- Jména klientů – Autentizační server neposkytuje o klientech žádné údaje kromě jejich ID. To je ale pro člověka nečitelné. Při navazování spojení by bylo lepší zobrazit nějaké relevantnější údaje, jako je jméno klienta. Aplikace zákazníka si údaje může dodat sama, ale výrazně jednodušší by bylo, kdyby to řešil autentizační server.

- Přenos souborů – Kvůli jedné z limitací co-browsingu nemůže správce nahrávat do stránky soubory. Toto omezení lze obejít tak, že správce pošle soubor uživateli. Uživatel si pak může tento soubor vložit do stránky sám. Samotný přenos souboru je velmi jednoduchý. Je akorát potřeba udělat grafické rozhraní.

- Chat a videohovor – Pokud má být co-browsing použitý pro poskytování vzdálené správy, musí podporovat dorozumívání mezi uživatelem a správcem. Pokud je co-browsing určený spíše pro spolupráci ve firmě, není tato funkce potřeba. Záleží tedy na způsobu využití. V případě potřeby videohovoru, je velkou výhodou, že prototyp již používá WebRTC. Díky tomu je implementace videohovoru výrazně jednodušší.

Závěr

Cílem této práce bylo navrhnout a následně vyrobit prototyp, který by umožňoval na dálku spravovat webové aplikace. Hlavním požadavkem bylo, aby si uživatel nemusel instalovat dodatečný software, jako je tomu u běžných nástrojů na vzdálenou správu.

Během analýzy jsem zjistil, že požadavky ze zadání přesně popisují takzvaný *co-browsing*. Pro účely analýzy jsem otestoval tři produkty, které umožňují přidat podporu pro co-browsing do webových aplikací. V analýze jsem také vysvětlil, jak co-browsing funguje a jak lze realizovat.

Pro vlastní řešení co-browsingu jsem zvolil metodu *synchronizace výstupu JavaScriptového enginu*. Tuto metodu jsem trochu upravil, abych vykompenzoval některé její nevýhody. Pro komunikaci mezi klienty používám technologii WebRTC, která umožňuje provozovat co-browsing v *peer-to-peer* režimu. Mé řešení se také zaměřuje na zlepšení kompatibility s JavaScriptem webové stránky, což je dle mého názoru největší nedostatek testovaných co-browsingových produktů.

Výsledný prototyp umožňuje sdílet stránku se vzdáleným webovým prohlížečem. Ze vzdáleného prohlížeče je možné stránku i ovládat. Prototyp jsem otestoval na reálné webové aplikaci. K testování jsem zvolil aplikaci OptiLynx [1]. Poté jsem na základě výsledků testování vyhodnotil použitelnost prototypu. Po opravení všech nalezených chyb, funguje prototyp v aplikaci dobře.

Výsledná práce je rozsahem výrazně větší, než je u bakalářské práce běžné. Důvodem k tomu je, že zadání je poměrně obtížné a rozsáhlé. Také jsem práci

věnoval více času, než by bylo nezbytně nutné. Prací jsem se zabýval 4 měsíce a celkem jsem odpracoval přes 600 hodin.

Všechny cíle práce byly splněny. Z finálního testování vyplývá, že zvolený způsob řešení je opravdu možné použít. Také jsem navrhl, jak je třeba upravit prototyp, aby ho bylo možné použít i komerčně.

Zdroje

1. JAGU S.R.O. *OptiLynx* [online]. 2019 [cit. 2019-04-25]. Dostupné z: https://www.optilynx.cz.
2. TEAMVIEWER. *TeamViewer* [software]. 2019. Dostupné také z: https://www.teamviewer.com.
3. ANYDESK SOFTWARE. *AnyDesk* [software]. 2019. Dostupné také z: https://anydesk.com.
4. GOOGLE. *Chrome Remote Desktop* [software]. 2019. Dostupné také z: https://chrome.google.com/webstore/detail/chrome-remote-desktop/gbchcmhmhahfdphkhkmpfmihenigjmpp.
5. GOOGLE. *Chrome* [software]. 2019. Dostupné také z: https://www.google.com/chrome.
6. LOWET, Dietwig; GOERGEN, Daniel. Co-browsing dynamic web pages. *Proceedings of the 18th international conference on World wide web – WWW 09*. 2009. Dostupné z DOI: 10.1145/1526709.1526836.
7. ORACLE. *Co-browse Deployment and Use Guide* [online]. 2017 [cit. 2019-04-19]. Dostupné z: https://docs.oracle.com/cloud/february2017/servicecs_gs/FASCG/cobrowse_deployment_guide.htm#FASCGth_CobrowseOverview.
8. GARSIEL, Tali; IRISH, Paul. *How Browsers Work: Behind the scenes of modern web browsers – HTML5 Rocks* [online]. 2011 [cit. 2019-04-20]. Dostupné z: https://www.html5rocks.com/en/tutorials/internals/howbrowserswork.

9. MOZILLA. *Content versus IDL attributes* [online]. 2019 [cit. 2019-04-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes#Content_versus_IDL_attributes.

10. MOZILLA. *Same-origin policy* [online]. 2019 [cit. 2019-04-19]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

11. MOZILLA. *What is a URL?* [online]. 2019 [cit. 2019-04-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL.

12. MOZILLA. *Cross-Origin Resource Sharing (CORS)* [online]. 2019 [cit. 2019-04-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

13. MOZILLA. *HTTP cookies* [online]. 2019 [cit. 2019-04-16]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

14. MOZILLA. *:hover* [online]. 2019 [cit. 2019-04-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/:hover.

15. THUM, Christian; SCHWIND, Michael. Synchronite – A Service for Real-Time Lightweight Collaboration. *2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. 2010. Dostupné z DOI: 10.1109/3pgcic.2010.36.

16. FIELDING, Roy T.; GETTYS, James; MOGUL, Jeffrey C.; NIELSEN, Henrik Frystyk; MASINTER, Larry; LEACH, Paul J.; BERNERS-LEE, Tim. *Hypertext Transfer Protocol – HTTP/1.1*. RFC Editor, 1999. Č. 2616. ISSN 2070-1721. Dostupné také z: http://www.rfc-editor.org/rfc/rfc2616.txt.

17. FETTE, I.; MELNIKOV, A. *The WebSocket Protocol*. RFC Editor, 2011. Č. 6455. ISSN 2070-1721. Dostupné také z: http://www.rfc-editor.org/rfc/rfc6455.txt.

18. MOZILLA. *The WebSocket API (WebSockets)* [online]. 2019 [cit. 2019-04-16]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

19. MOZILLA. *WebRTC API* [online]. 2019 [cit. 2019-04-21]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API.

20. NARAYANAN, Anant; JENNINGS, Cullen; ABOBA, Bernard; BRUAROEY, Jan-Ivar; BURNETT, Daniel; BERGKVIST, Adam; BRANDSTETTER, Taylor. *WebRTC 1.0: Real-time Communication Between Browsers*. 2018. Candidate Recommendation. W3C.

21. MOZILLA. *RTCDataChannel* [online]. 2019 [cit. 2019-04-21]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel`.

22. ACQUIRE. *Acquire* [software]. 2019. Dostupné také z: `https://acquire.io`.

23. SURFLY. *Surfly* [software]. 2017. Dostupné také z: `https://www.surfly.com`.

24. UPSCOPE LIMITED. *Upscope* [software]. 2019. Dostupné také z: `https://upscope.io`.

25. KULLAR, Pardeep. *3 Best Co-browsing Software Companies* [online]. 2019 [cit. 2019-04-26]. Dostupné z: `https://blog.upscope.io/best-cobrowsing-software`.

26. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL* [software]. 2019. Dostupné také z: `https://www.postgresql.org`.

27. DUTTON, Sam. *WebRTC in the real world: STUN, TURN and signaling – HTML5 Rocks* [online]. 2013 [cit. 2019-04-27]. Dostupné z: `https://www.html5rocks.com/en/tutorials/webrtc/infrastructure`.

28. MOZILLA. *HTTP authentication* [online]. 2019 [cit. 2019-04-28]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication`.

29. DEFUSE SECURITY. *Salted Password Hashing – Doing it Right* [online]. 2018 [cit. 2019-04-29]. Dostupné z: `https://crackstation.net/hashing-security.htm`.

30. BASQUES, Kayce. *Why HTTPS Matters* [online]. Google, 2019 [cit. 2019-04-29]. Dostupné z: `https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https`.

31. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *Secure TCP/IP Connections with SSL* [online]. 2019 [cit. 2019-04-29]. Dostupné z: `https://www.postgresql.org/docs/9.1/ssl-tcp.html`.

32. NTT COMMUNICATIONS. *A Study of WebRTC Security* [online]. 2015 [cit. 2019-04-29]. Dostupné z: https://webrtc-security.github.io.

33. FIELDING, R.; RESCHKE, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC Editor, 2014. Č. 7231. ISSN 2070-1721. Dostupné také z: http://www.rfc-editor.org/rfc/rfc7231.txt.

34. POSTEL, Jon. *Transmission Control Protocol*. RFC Editor, 1981. Č. 793. ISSN 2070-1721. Dostupné také z: http://www.rfc-editor.org/rfc/rfc793.txt.

35. IBM. *Kitura* [software] [cit. 2019-05-08]. Dostupné z: https://www.kitura.io.

36. JETBRAINS. *Ktor* [software] [cit. 2019-05-08]. Dostupné z: https://ktor.io.

37. PIVOTAL SOFTWARE. *Spring* [software] [cit. 2019-05-08]. Dostupné z: https://spring.io.

38. JETBRAINS. *Exposed* [software] [cit. 2019-05-08]. Dostupné z: https://github.com/JetBrains/Exposed.

39. ČVUT FIT. *Portál pro výuku BI-DBS* [online] [cit. 2019-04-14]. Dostupné z: https://dbs.fit.cvut.cz.

40. DOCKER. *Docker* [software] [cit. 2019-05-14]. Dostupné z: https://www.docker.com.

41. GOOGLE. *Angular* [software] [cit. 2019-05-14]. Dostupné z: https://angular.io.

Dodatek A

Seznam použitých zkratek

API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator
CSS	Cascading Style Sheets
DAO	Data Access Object
DDoS	Distributed Denial of Service
DOM	Document Object Model
DSL	Domain-Specific Language
DTO	Data Transfer Object
ID	Identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
NAT	Network Address Translation
ORM	Object-Relational Mapping
REST	Representational State Transfer
RFC	Request For Comment
RPC	Remote Procedure Call
SQL	Structured Query Language

SSO	Single Sign On
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
TLS	Transport Layer Security
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium

Dodatek B

Slovník pojmů

administrátor	Osoba, která pro zákazníka spravuje nastavení autentizačního serveru.
autentizační server	Program spuštěný na serveru, který zajišťuje autentizaci klientů při co-browsingu.
co-browsing	Společné prohlížení a ovládání webové stránky více klienty najednou. Je to speciální případ vzdálené správy pro webové aplikace, která nevyžaduje použití jiného softwaru než webového prohlížeče.
co-browsingový produkt	Softwarové řešení, které přidává podporu pro co-browsing do webové stránky.
klient	Uživatel a/nebo správce.
plugin správce	Program v podobě pluginu do Chrome, který používá správce. Tento program zajišťuje sdílení stránky od uživatele a její ovládání.
plugin uživatele	Program, který je vložený do webové stránky uživatele. Tuto stránku se správcem program sdílí a umožňuje ji na dálku ovládat.
správce	Účastník vzdálené správy, který na dálku ovládá uživatelův počítač. Správce může být osoba nebo program na jejím počítači, který vzdálenou správu zajišťuje.

uživatel	Účastník vzdálené správy, jehož počítač na dálku ovládá správce. Uživatel může být osoba nebo program na jejím počítači, který vzdálenou správu zajišťuje.
vzdálená správa	Ovládání lokálního počítače ze vzdáleného počítače, většinou s využitím internetu.
zákazník	Nejčastěji firma (zástupce firmy) s vlastní webovou aplikaci, ve které je co-browsingový produkt.

Dodatek C

Obsah přiloženého CD

- `readme.md` ..... stručný popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `BP_Dolnik_Filip_2019.pdf` ..... text práce ve formátu PDF