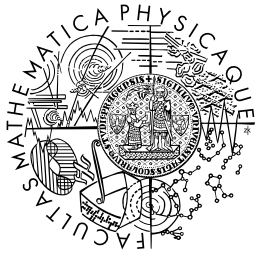


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Lukáš Riedel

Jízdní řády hromadné dopravy

Katedra softwarového inženýrství

Vedoucí bakalářské práce:	RNDr. Filip Zavoral, Ph.D.
Studijní program:	Informatika
Studijní obor:	Programování a softwarové systémy

Praha 2019

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Děkuji panu RNDr. Filipu Zavoralovi, Ph.D. za trpělivé vedení mé práce a za čas, který mi věnoval. Rovněž děkuji lidem, kteří mě po dobu mého studia podporovali.

Název práce: Jízdní řády hromadné dopravy

Autor: Lukáš Riedel

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D., Katedra softwarového inženýrství

Abstrakt: V zájmu lidí užívajících hromadnou dopravu je dostat se na požadované místo co nejrychleji. Za tímto účelem využívají vyhledávače spojení. Většina existujících vyhledávačů se omezuje na předem určenou oblast nebo je lze využívat pouze při stálém připojení k internetu, což může uživatele limitovat. Z tohoto důvodu jsme navrhli přizpůsobitelný systém aplikací, který uživateli umožňuje, za určitých podmínek, vyhledávání i bez nutnosti připojení k internetu. V práci se zabýváme především způsoby vyhledávání spojení v jízdních řádech. Naše řešení využívá inovativní algoritmus, jenž se svým fungováním a časem potřebným k provedení požadavku výrazně odchyluje od většiny existujících řešení. Pro komfort uživatele jsme vyvinuli počítačovou a mobilní aplikaci, které uživateli nabízí přívětivé uživatelské rozhraní, pomocí něhož lze vyhledávat v jízdních řádech a zobrazovat aktuální informace z dopravy.

Klíčová slova: jízdní řády, vyhledávací algoritmus, aktualizace

Title: Public Transport Timetables

Author: Lukáš Riedel

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: It is in the interest of people using public transport to get to the desired location as quickly as possible. That is the reason they use search engines to search journeys. Most existing search engines are limited to a predetermined area or can only be used with a permanent internet connection, which may limit the user. For this reason, we have designed a customizable application system that allows the user, under certain conditions, to search without the need for an Internet connection. In the thesis we analyse mainly with the ways of finding journeys in timetables. Our solution uses an innovative algorithm that, with its operation and time required to execute a request, deviates significantly from most existing solutions. For the convenience of the user, we have developed a desktop and mobile application that provides the user user-friendly interface that can be used to search timetables and display current traffic information.

Keywords: timetables, searching algorithm, updates

Obsah

1. Úvod

Veřejná hromadná doprava je v dnešní době součástí každodenních životů velké části populace. Lidé ji používají k cestám do práce či za zájmovými aktivitami, neboť se jedná o levnější a ekologičtější způsob dopravy, než kdyby si měl každý jedinec zajišťovat dopravu vlastní. Hromadná doprava je komplexní systém, kde každý krok musí být dopředu precizně naplánován, aby byl dopravce schopen poskytnout svým zákazníkům to, co jim slibuje ve formě jízdních řádů. Samotné čtení jízdních řádů a hledání optimálního spojení mezi dvěma body může být náročný proces, obzvlášť když do hry vstupují situace, které odchylují předepsané jízdní řády od skutečnosti, jako například výluky, změny jízdních řádů či mimořádné situace v dopravě, o kterých koncový zákazník nemusí vůbec vědět.

V práci je popsána analýza a implementace počítačové a mobilní aplikace, které mají stejný účel, a sice poskytnout uživateli rychlé a efektivní vyhledávání spojení mezi dvěma body s různými požadavky na spojení. Mezi další znaky se řadí možnost vyhledávat odjezdy ze zadané stanice, zobrazit si mapu dopravní sítě či si vyžádat informace o mimořádných situacích v dopravě a výlukách. Funkčnost celého systému zastřešuje serverová aplikace, která je rovněž zanalyzována. K té se klientské aplikace připojují a dochází v ní k samotnému vyhledávání. Neméně důležitou součástí klientských aplikací je možnost uložit si oblíbená spojení, která následně budou uživateli k dispozici i bez nutnosti přístupu k serveru a která budou pravidelně aktualizována.

Hlavním těžištěm práce je kapitola o samotném vyhledávání spojení, jelikož se předpokládá, že právě tato služba bude pro uživatele hlavní důvod, proč by si měl nainstalovat zmíněné aplikace. V textu bude detailně popsán inovativní algoritmus, který se svým fungováním a časem potřebným k exekuci požadavku výrazně odchyluje od časově závislého modelu, kterým se zabývá většina prací na stejné téma a který bude na konci kapitoly rovněž popsán.

1.1 Cíle práce

Hlavním cílem práce je navrhnout a implementovat serverovou aplikaci a dvě aplikace klientské, konkrétně počítačovou a mobilní. Jedním z dílčích cílů práce je detailně zanalyzovat zvolený vyhledávací algoritmus a popsat další možné alternativy.

Serverová aplikace by měla být schopná:

- Stáhnout, zpracovat potenciálně libovolná data (jízdní řády) ze zadaného zdroje a udržovat je neustále aktuální. Rovněž chceme podporovat možnost zadat více datových zdrojů, načež dojde k jejich integraci a poskytne nám to možnost pracovat nad daty pokrývající libovolně velkou oblast, potažmo i celý svět.[1]

- Poskytovat klientským aplikacím základní data potřebná pro jejich základní funkcionalitu. Mezi tato data se řadí základní informace o zastávkách a linkách.

[1] Tento přístup využívají Google Maps v rámci vyhledávání spojení v hromadné dopravě.

- Přijmout požadavek na vyhledání spojení, kde je možné specifikovat různá omezení na hledané spojení, a odpovědět na něj.
- Přijmout požadavek na vyhledání odjezdů ze stanice, kde je možné omezit se na jednu konkrétní linku, a odpovědět na něj.
- Přijmout požadavek na vyhledání informací o lince a odpovědět na něj.
- Vykonávat všechny zmíněné požadavky asynchronně.

Klientské aplikace by měly být schopné:

- Poskytnout uživateli přívětivé uživatelské rozhraní a přehledné zobrazování výsledků.
- Poskytnout uživateli možnost odeslat na server požadavek na vyhledání spojení a zobrazit nalezené výsledky.
- Poskytnout uživateli možnost odeslat na server požadavek na vyhledání odjezdů ze stanice a zobrazit nalezené výsledky.
- Poskytnout uživateli možnost odeslat na server požadavek na vyhledání informací o lince a zobrazit nalezené výsledky.
- Poskytnout uživateli možnost si přidat spojení, stanici či linku mezi oblíbené položky. Tyto oblíbené položky budou následně přístupné i bez nutnosti připojení k serveru a kdykoliv se aplikace připojí k serveru, budou aktualizovány.
- Zobrazit mapu dopravní sítě, kde budou vykreslené všechny zastávky, přičemž po kliknutí na zastávku dojde k vyhledání nejdřívějších odjezdů z dané zastávky. Rovněž budeme podporovat zobrazení trasy nalezených spojení.
- Poskytnout uživateli informace o výlukách a mimořádných událostech v dopravě tak, jak je poskytuje dopravce.
- Poskytnout uživateli lokalizaci v požadovaném jazyce.

Dále chceme u počítačové aplikace podporovat možnost offline režimu, kde aplikace převezme roli serveru a odpovědi na požadavky se budou vyřizovat v rámci téhož procesu. Mobilní aplikaci budeme vyvíjet pouze pro platformu Android, nicméně budeme požadovat, aby aplikace byla psána multiplatformně a bylo možné jednoduše přizpůsobit aplikaci pro běh na iOS či UWP.

1.2 Struktura práce

První kapitola je úvod do problematiky. V druhé kapitole si ukážeme již existující aplikace pro vyhledávání v jízdních řádech. Následuje třetí kapitola, kde detailně zanalyzujeme zvolený vyhledávací algoritmus. Dále ve čtvrté kapitole si ukážeme návrh celého systému a architekturu aplikací na úrovni jednotlivých modulů. V páté kapitole se podíváme na implementaci jednotlivých aplikací. Šestá kapitola se zabývá experimentálním testování implementace algoritmu ze třetí kapitoly. V poslední sedmé kapitole si shrneme výsledky a přínosy této práce.

2. Existující řešení

V této kapitole budou uvedena již existující řešení, která mohou být konkurenty našich aplikací. Zmíníme dva hlavní podobné vyhledávače, ale souvisejících vyhledávačů je celá řada. Mezi další nezmíněné související vyhledávače se řadí například vyhledávač Seznam.cz[1] či vyhledávač IDOS[2], vyvinutý společností CHAPS, která byla Ministerstvem dopravy České republiky pověřena vedením Celostátního informačního systému o jízdních řádech.

Nejdříve ale zmíníme pár základních informacích o zdroji dat, protože se na něj budeme odkazovat v popisu již existujících řešení.

2.1 Zdroj dat

Za formát zdroje dat, který chceme podporovat, jsme zvolili GTFS[3] formát [1], především díky jeho jasně dané struktuře, kterou většina dopravních společností při zveřejňování dat dodržuje. Tento formát bude detailněji popsán v sekci 4.2. Pro naši práci jsme ze zřejmých důvodů zvolili zdroj dat PID [2].

Další uvažovanou variantou byl JDF[4] formát[5] poskytovaný CIS JŘ. Jediná výhoda JDF formátu nad GTFS formátem spočívá ve faktu, že drážní data a data pro regionální autobusy jsou v České republice poskytována pouze v JDF formátu. Daný formát má ale také několik zásadních nevýhod, neboť se využívá pouze v České republice a dochází k jeho neustálým změnám.

2.2 Obecné požadavky na vyhledávač spojení

Aplikace na vyhledávání spojení lze rozdělit na dvě skupiny. První skupinou jsou aplikace, které pro svůj úplný běh vyžadují přístup ke vzdálenému serveru. Sem řadíme například webové a mobilní aplikace. V dnešní době je typické, že pokud existuje webová aplikace, pak má pravděpodobně i svůj ekvivalent pro mobilní zařízení. Druhou skupinou jsou aplikace, které pro svůj běh server vyžadují pouze občas nebo vůbec. Mezi tyto aplikace se řadí počítačové aplikace, kde samotné vyhledávání probíhá na stroji uživatele. Poznamenejme, že naše počítačová aplikace by se měla řadit do obou skupin, díky možnosti přepínání online a offline módu, jak jsme zmínili na konci sekce 1.1 a jak detailně zanalyzujeme v sekci 5.2.1.

Všechny takové aplikace podporují vyhledávání spojení, odjezdů ze stanice a popřípadě i vyhledávání linek. Pokud aplikaci vyvíjí samotný dopravce, pak pravděpodobně bude obsahovat i možnost zobrazit si informace o výlukách, mimořádnostech v dopravě a zprávy od dané dopravní společnosti. Při vyhledávání lze na spojení klást různé požadavky, mezi ně se řadí zejména:

- Možnost vyhledávat podle nejdřívějšího času odjezdu.

[1] `https://www.seznam.cz/jizdnirady/`
[2] `https://www.chaps.cz/cs/products`
[3] General Transit Feed Specification
[4] Jednotný Datový Formát
[5] `https://www.chaps.cz/files/cis/jdf-1.10.pdf`

- Možnost vyhledávat podle maximálního počtu přestupů.
- Možnost vyhledávat podle specifických dopravních prostředků.
- Možnost přizpůsobit si rychlost přestupů mezi jednotlivými zastávkami.
- Možnost vyhledávat pouze bezbariérové spoje.

Dále se možnost filtrování spojení liší v závislosti na aplikaci. Některé aplikace umožňují vyhledávat podle maximálního času příjezdu, jiné umožňují omezit maximální čas přestupů, přidat průjezdní bod či výsledky různě třídit.

2.3 Vyhledávač PID

Vyhledávač PID je přímým konkurentem našich aplikací, neboť naše aplikace by měly být optimalizované pro stejnou datovou sadu [2]. Vyhledávač PID existuje ve formě webové[6] a mobilní[7] aplikace. Mezi zajímavé funkce tohoto vyhledávače se řadí možnost vytvořit si osobní jízdní řád, kde se naleznou všechna spojení mezi zadanými dvěma body pro všechny dny v týdnu a výstup se následně vygeneruje do PDF souboru. My toto chceme podporovat v podobné variantě, která bude popsána v sekci 4.5.3. V ostatních aspektech jsou funkce tohoto vyhledávače totožné s těmi, které by měly podporovat i naše aplikace, jak jsme uvedli v sekci 1.1.

2.4 Vyhledávač Google Maps

V našem výčtu uvádíme vyhledávač Google Maps[8] zejména proto, že společnost Google je tvůrcem referenčního GTFS formátu [1] pro jízdní řády, který by měly umět zpracovat i naše aplikace. Pro zajímavost uveďme, že vyhledávač Google Maps zpracovává více než 800 datových zdrojů[9], které jim poskytují jednotlivé dopravní společnosti. Poznamenejme, že zpracování tolika zdrojů by měla být schopná i naše serverová aplikace, kdybychom k tolika datovým zdrojům měli přístup.

[6] `https://pid.cz/`
[7] `https://app.pidlitacka.cz/`
[8] `https://www.google.com/maps`
[9] `https://en.wikipedia.org/wiki/Google_Maps`

3. Vyhledávání

V této kapitole se budeme zabývat způsoby hledání spojení v mapě dopravní sítě. Běžný přístup k řešení tohoto problému je vytvoření grafu, kde zastávky tvoří vrcholy grafu a elementární spojení tvoří hrany grafu, které jsou navíc ohodnocené. Na tento graf se poté spustí algoritmus na nalezení nejkratší cesty, což má za výsledek nalezení nejrychlejšího spojení. Tento přístup má několik nevýhod, mezi ty nejzásadnější se řadí fakt, že s rostoucí velikostí grafu se rychlost vyhledávání spojení rapidně snižuje, v důsledku časové složitosti nejlépe $\mathcal{O}(|E| + |V| \log |V|)$ při použití Dijkstrova algoritmu s Fibonacciho haldou [3, str. 149]. Tento poznatek znemožňuje použití popsaného přístupu v interaktivních aplikacích, neboť dopravní sítě obsahují tisíce, ve velkoměstech až desetitisíce, zastávek, což by vedlo k velké odezvě.

Naše řešení využívá RAPTOR[1] algoritmus [4], který ke své činnosti nepotřebuje prioritní frontu a pracuje v lineární časové složitosti, čímž dosáhneme rapidního zrychlení vůči existujícím řešením. Algoritmus pracuje v kolech, přičemž v k-tém kole se počítá dosažitelnost pomocí $k-1$ přestupů. Příjezdové časy do zastávek v síti se počítají procházením každé linky nejvýše jednou za kolo. Algoritmus se, na rozdíl od jiných algoritmů, nespoléhá na předzpracování dat, což ho činí dynamickým a dokáže se snadno vyrovnat s různými změnami v dopravě, ať už výlukami či změnami tras linek.

3.1 Formalizace problému

Vyhledávací algoritmus požaduje jasně danou strukturu dat, která bude v této sekci formalizována. Nejprve si uveďme základní definici jízdního řádu převzatou z referenčního článku [4].

Definice 1. *Jízdní řád je pětice* $(\Pi, \mathcal{S}, \mathcal{R}, \mathcal{T}, \mathcal{F})$*, kde* $\Pi \subset \mathbb{N}$ *je doba platnosti,* $\mathcal{S}$ *je množina zastávek,* $\mathcal{R}$ *je množina linek,* $\mathcal{T}$ *je množina unikátních jízd všech linek a* $\mathcal{F}$ *je funkce přestupů.*

Množinu Π si lze představit jako časový interval (p_1, p_2) platnosti jízdního řádu, kde $p_1, p_2 \in \mathbb{N}$ jsou časová razítka Unixového času.

Každá zastávka $s \in \mathcal{S}$ se váže k fyzické lokaci v mapě dopravní sítě, kde lze nastoupit či vystoupit z dopravního prostředku.

Linka $r \in \mathcal{R}$ je posloupnost zastávek, typicky je nějakým způsobem označena a je obsluhována pouze jedním druhem dopravního prostředku. Počet zastávek linky r budeme značit jako $|r|$.

Unikátní jízda linky je dvojice $(t, r) \in \mathcal{T}$, kde $r \in \mathcal{R}$ je linka a t je zobrazení, které každé zastávce s v trase linky r přiřadí čas příjezdu do dané zastávky, značíme $\tau_{arr}(t, s) \in \Pi$, a čas odjezdu z dané zastávky, značíme $\tau_{dep}(t, s) \in \Pi$. Navíc musí platit, že $\tau_{arr}(t, s) \leq \tau_{dep}(t, s)$. První a poslední zastávka v jedné unikátní jízdě má nedefinovaný čas příjezdu, respektive odjezdu.

Funkce přestupů $\mathcal{F}$ je zobrazení $\mathcal{F} : \mathcal{S} \to \mathcal{S} \to \mathbb{N}_0$ a určuje čas chůze ze zastávky $s_1 \in \mathcal{S}$ do zastávky $s_2 \in \mathcal{S}$. Pro zastávky $s_1, s_2 \in \mathcal{S}$ platí, že $\mathcal{F}(s_1, s_2) =$

[1] Round-bAsed Public Transit Optimized Router

$\mathcal{F}(s_2, s_1)$ a $\mathcal{F}(s_1, s_1) = 0$. V samotné implementaci bývá, pro ušetření paměti a času, čas přestupů mezi zastávkami typicky omezen nějakou konstantou. Pro takové dvě zastávky, kde by čas přestupu mezi nimi převýšil danou konstantu, je funkce nedefinovaná. Počet všech přestupů budeme pro jednoduchost značit $|\mathcal{F}| = |\{\mathcal{F}(s_1, s_2) : s_1, s_2 \in \mathcal{S}\}|$.

Definice 2. *Spojení $\mathcal{J}$ je posloupnost unikátních jízd a přestupů. Každé unikátní jízdě jsou navíc přiřazeny dvě zastávky dané linky, které označují místa nástupu a výstupu z dopravního prostředku.*

Každé spojení má tedy jednoznačně daný čas odjezdu z výchozí zastávky a čas příjezdu do cílové zastávky. Z toho se odvíjí celková doba trvání spojení, mezi další kritéria spojení se řadí například celkový počet zastávek ve spojení, doba čekání na přípoje nebo celková doba trvání přestupů. Našim cílem je najít pouze optimální spojení, tedy ta nejrychlejší a pro uživatele nejpohodlnější. Proto pro dvě spojení $\mathcal{J}_1, \mathcal{J}_2$ definujme relaci $\prec$, kde $\mathcal{J}_1 \prec \mathcal{J}_2$ právě tehdy, když $\mathcal{J}_1$ je ve zmíněných kritériích lepší než $\mathcal{J}_2$.

Definice 3 (Problém nejdřívějšího příjezdu [4])**.** *Mějme výchozí zastávku s_s, cílovou zastávku s_t a čas odjezdu τ. Problém nejdřívějšího příjezdu řeší nalezení spojení ze zastávky s_s s odjezdem nejdříve v čas τ takové, že se do zastávky s_t dostane co nejdříve.*

Tedy s co nejmenším časem příjezdu. Tento problém řeší vyhledávací algoritmy. Problém můžeme zobecnit na více-kritériový problém s tím, že zachováme původní problém a přidáme nová kritéria. Příkladem kritéria může být například počet přestupů n, kde řešíme tentýž problém s maximálně n přestupy. Dalším kritériem může být například požadavek na specifické dopravní prostředky atp. Kritéria lze libovolně kombinovat.

Stěžejní funkci algoritmu tvoří funkce pro nalezení nejdřívější unikátní jízdy linky, na kterou můžeme v dané zastávce nastoupit. Smysl této funkce bude detailněji rozebrán v sekci 3.2.1. Nyní si ji pouze formálně zadefinujeme.

Definice 4. *Nechť $r \in \mathcal{R}$, $s \in \mathcal{S}$ a $\tau \in \Pi$. Pak zobrazení $et : \mathcal{R} \to \mathcal{S} \to \Pi \to \mathcal{T}$ je dáno předpisem $et(r,s,\tau) = \min\{(t,r) \in \mathcal{T} : \tau_{dep}(t,s) \geq \tau\}$.*[2]

Pro unikátní jízdy $(t_1,r), (t_2,r) \in \mathcal{T}$ je porovnávací relace dána předpisem $(t_1,r) < (t_2,r) \iff \tau_{dep}(t_1,s) < \tau_{dep}(t_2,s)$, kde $s \in \mathcal{S}$ je pevně daná zastávka v trase linky. Pokud neexistuje žádná unikátní jízda splňující kritérium na čas odjezdu, je funkce pro tyto hodnoty nedefinována.

3.2 Algoritmus na vyhledávání spojení

V této sekci bude, na základě referenčního článku [4], detailně rozebrán vyhledávací algoritmus ve své základní podobě, dále budou zmíněna možná vylepšení a rozšíření algoritmu.

[2] Označení *et* je zkratka anglického označení *earliest trip*, což značí nejdřívější unikátní jízdu linky.

Nechť $(\Pi,\mathcal{S},\mathcal{R},\mathcal{T},\mathcal{F})$ je jízdní řád, $s_s \in \mathcal{S}$ výchozí zastávka, $s_t \in \mathcal{S}$ cílová zastávka, $\tau \in \Pi$ čas odjezdu a $n \in \mathbb{N}_0$ maximální počet přestupů. Pro $k \in \{0,\ldots,n\}$ definujme množiny J_k následujícím předpisem:

$$J_k = \{\mathcal{J} : \mathcal{J} \text{ je spojení z } s_s \text{ do } s_t \wedge \tau_{dep}(\mathcal{J}) \geq \tau \wedge \#\text{přestupů}(\mathcal{J}) = k\}$$

Cílem algoritmu je najít spojení $\mathcal{J} \in \bigcup_{k=0}^{n} J_k$ takové, že platí následující výrok:

$$\forall k \in \{0,\ldots,n\} \forall \mathcal{J}' \in J_k : \mathcal{J} \preceq \mathcal{J}'$$

3.2.1 Princip fungování

Algoritmus pracuje v kolech. V k-tém kole počítá nejrychlejší spojení ze zastávky s_s do všech zastávek $s \in \mathcal{S}$ tak, aby spojení obsahovalo nejvýše k unikátních jízd, tzn. nejvýše $k-1$ přestupů. Je zcela běžný jev, že spojení z s_s do s, mající nejvýše $k-1$ přestupů, neexistuje. Počet kol je shora omezen konstantou n, která značí maximální počet přestupů. Později ukážeme, že algoritmus nalezne nejrychlejší spojení a skončí, i když je maximální počet přestupů nedefinovaný, neboli $n \to \infty$.

Algoritmus přiřazuje každé zastávce $s \in \mathcal{S}$ $(n+2)$-tici $(\tau_0(s),\tau_1(s),\ldots,\tau_{n+1}(s))$, kde $\tau_i(s)$ značí nejdřívější možný příjezd do zastávky s tak, že nalezené spojení obsahuje i unikátních jízd, tedy $i-1$ přestupů. Výchozí hodnoty $\tau_i(s)$ jsou nastaveny na ∞.

Při spuštění algoritmu se nastaví $\tau_0(s_s) = \tau$. Platí invariant, že na začátku k-tého kola jsou záznamy $\tau_0(s),\ldots,\tau_{k-1}(s)$ pro $\forall s \in \mathcal{S}$ korektní, tzn. známe nejrychlejší spojení do všech zastávek v sítí, využívající nejvýše $k-1$ přestupů. Připomeňme, že pokud $\tau_i(s) = \infty$ pro $i \leq k$, pak spojení do zastávky s, využívající $i-1$ přestupů, neexistuje.

Cílem k-tého kola je spočítat záznamy $\tau_k(s)$ pro $\forall s \in \mathcal{S}$. Činnost jednoho kola lze rozdělit na tři části:

1. Počáteční inicializace. Nastaví se $\tau_k(s) = \tau_{k-1}(s)$ pro $\forall s \in \mathcal{S}$. Tím získáme horní mez nejdřívějšího příjezdu do zastávky s s nejvýše $k-1$ přestupy. V dalších částech se algoritmus snaží tento čas vylepšit.

2. Procházení tras linek. Každá linka $r \in \mathcal{R}$ je zpracovávána právě jednou. Nechť $s_1,\ldots,s_{|r|}$ jsou zastávky v trase linky r v pořadí, v jakém je linka projíždí. Linku bude algoritmus zpracovávat tím způsobem, že bude procházet její zastávky $s_1,\ldots,s_{|r|}$, dokud nenalezne zastávku s_i, kde $i \in \{1,\ldots,|r|\}$, takovou, že je funkce $et(r,s_i,\tau_{k-1}(s_i))$ definována. Až se tak stane, získá z funkce unikátní jízdu $(t,r) \in \mathcal{T}$, která určí čas $\tau_{dep}(t,s_i)$. Tedy čas, kdy můžeme na danou linku nastoupit. Pokračuje se v procházení zastávek s_j, kde $j \in \{i+1,\ldots,|r|\}$, a pro každou zastávku s_j se nastaví $\tau_k(s_j) = \tau_{arr}(t,s_j)$, tedy čas příjezdu do zastávky s_j užitím jízdy linky (t,r). Spojení lze snadno zrekonstruovat pomocí ukazatelů na nástupní zastávku s_i a aktuální zastávku s_j.

   Při procházení trasy se může stát, že algoritmus narazí na zastávku s_j takovou, že $\tau_{k-1}(s_j) < \tau_{arr}(t,s_j)$. Tedy zastávku, do které jsme se v nějakém z předchozích kol dostali rychleji než použitím této jízdy linky. V takovém případě je potřeba aktualizovat t přepočítáním funkce $et(r,s_j,\tau_{k-1}(s_j))$. Poté lze pokračovat v procházení trasy.

3. Zahrnutí přestupů. Pro každé dvě zastávky $s_i, s_j \in \mathcal{S}$, pro něž je funkce $\mathcal{F}(s_i, s_j)$ definována, se nastaví $\tau_k(s_j) = \min\{\tau_k(s_j), \tau_k(s_i) + \mathcal{F}(s_i, s_j)\}$. Tím se přidají do spojení přestupy mezi zastávkami.

Pokud v k-tém kole nedojde k vylepšení $\tau_k(s)$ pro žádnou ze zastávek $s \in \mathcal{S}$, nedošlo by k vylepšení ani v $(k+1)$-ním kole. V tento okamžik můžeme algoritmus zastavit.

Tvrzení 1. *Algoritmus vyřeší problém nejdřívějšího příjezdu v konečném čase.*

Důkaz. Nejprve dokážeme konečnost. V jednom kole prochází algoritmus všechny linky právě jednou, těch je konečný počet. Pokud je nastaven maximální počet přestupů, máme hotovo, neboť počet kol je shora omezen tímto číslem plus jedna. Pokud ne, je třeba dokázat, že existuje takové kolo k, ve kterém nedojde k vylepšení $\tau_k(s)$ pro nějakou zastávku $s \in \mathcal{S}$. To je také zřejmé, neboť mapa dopravní sítě není nekonečná. Můžeme tedy počet kol shora omezit $k \leq |\mathcal{S}|$.

Korektnost dokážeme matematickou indukcí dle počtu kol. Pro $k = 0$ pracuje algoritmus správně, neboť užitím žádné linky se můžeme dostat z s_s pouze do s_s, tedy máme pouze $\tau_0(s_s) = \tau$, což se nastaví při spuštění algoritmu. Nyní dokažme indukční krok $k \to k+1$. V $(k+1)$-ním kole prochází algoritmus trasy všech linek a pokouší se vylepšit příjezdové časy užitím k přestupů. Na konci kola platí, že záznamy $\tau_{k+1}(s)$ pro $\forall s \in \mathcal{S}$ jsou nejlepší možné příjezdové časy do zastávky s užitím k přestupů. Kdyby existovala taková zastávka $s \in \mathcal{S}$, pro kterou by $\tau_{k+1}(s)$ nebyl nejlepší možný příjezdový čas s užitím k přestupů, pak by musela existovat taková jízda $(t,r) \in \mathcal{T}$ linky $r \in \mathcal{R}$, která by příjezdový čas $\tau_{k+1}(s)$ zlepšila. To ale není možné kvůli definici funkce *et* a tomu, že algoritmus zpracoval všechny linky. Tedy musí být špatně nastaveny hodnoty $\tau_k(s)$ pro nějaké $s \in \mathcal{S}$, což je spor s indukčním předpokladem. □

Tvrzení 2. *Časová složitost algoritmu je* $\mathcal{O}\left(n\left(\sum_{r\in\mathcal{R}}|r| + |\mathcal{T}| + |\mathcal{F}|\right)\right)$. *[4]*

Důkaz. Důkaz budeme vést ve stejném duchu, jak je uveden v referenčním článku [4]. V každém kole se prochází trasa linky $r \in \mathcal{R}$ právě jednou, celkový počet navštívených zastávek je roven $\sum_{r\in\mathcal{R}}|r|$. Během procházení si algoritmus pamatuje výsledky z funkce $et(r,\_,\_)$, tudíž každou jízdu linky zpracovává nejvýše jednou. Zbývá zohlednit přestupy počítané na konci každého kola, těch je $|\mathcal{F}|$. Celková doba běhu jednoho kola je tedy lineární a rovna $\sum_{r\in\mathcal{R}}|r| + |\mathcal{T}| + |\mathcal{F}|$, celkem dostáváme $\mathcal{O}\left(n\left(\sum_{r\in\mathcal{R}}|r| + |\mathcal{T}| + |\mathcal{F}|\right)\right)$, kde n je počet kol, neboli maximální počet přestupů mínus jedna. □

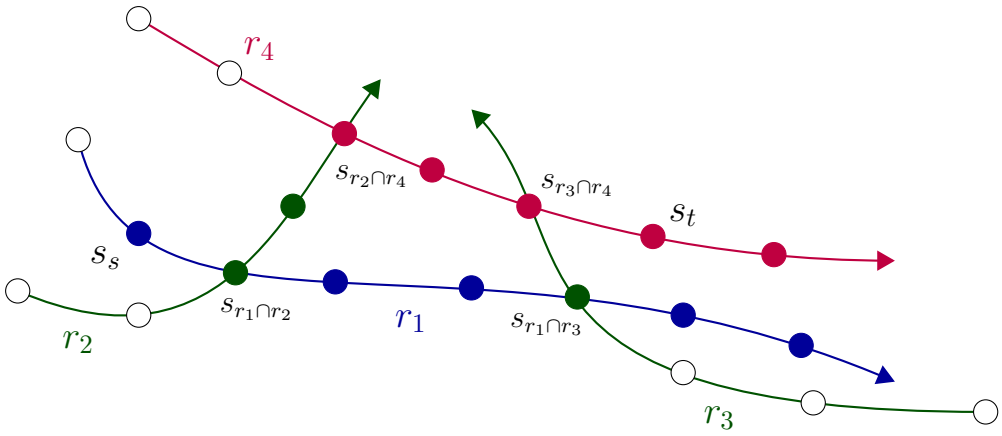
3.2.2 Zrychlení algoritmu

Při popisu základní verze algoritmu v předchozí sekci jsme uvedli, že v každém kole je třeba zpracovávat všechny linky $r \in \mathcal{R}$ v mapě dopravní sítě. Většina požadavků na spojení reálně pracuje pouze s malým zlomkem celé sítě, tudíž zpracovávat pokaždé všechny linky by bylo neefektivní. Proto není potřeba zpracovávat takové linky, které nesdílí žádnou zastávku $s \in \mathcal{S}$ s linkami zpracovávanými v předchozím kole, neboť neexistuje způsob, jak nastoupit na nějakou z jejich jízd.

Pro objasnění předchozího odstavce si uveďme příklad. Mějme nějakou linku $r \in \mathcal{R}$, kde příjezdový čas do nějaké ze zastávek její trasy byl vylepšen v kole

$k < n$. V kole $k + 1 < n$ byla zpracovávána znovu, nicméně žádný z příjezdových časů nebyl během tohoto zpracovávání zlepšen. Pak nemá smysl ji procházet v kole $k + 2 < n$ a ani v žádném z dalších kol, dokud nebude zlepšen nějaký z příjezdových časů vlivem jiné linky, neboť linka r již nemůže příjezdové časy vylepšit.

Implementaci tohoto vylepšení provedeme následovně. Během $(k-1)$-ho kola si bude algoritmus značit všechny zastávky $s \in \mathcal{S}$, jejichž příjezdový čas $\tau_{k-1}(s)$ se zlepšil. Na začátku k-tého kola zpracuje všechny označené zastávky $s \in \mathcal{S}$ tím způsobem, že nalezne všechny linky $r \in \mathcal{R}$, jejichž trasa obsahuje zastávku s. Označená zastávka s je navíc přesně ta, ve které můžeme nastoupit na danou linku r v kole k. Tím ušetříme další čas, neboť trasu $s_1, \ldots, s, \ldots, s_{|r|}$ linky r není potřeba procházet již od zastávky s_1, stačí až od zastávky s. Algoritmus bude obsahovat množinu Q a do ní bude přidávat dvojice (r,s), kde r je linka projíždějící označenou zastávkou s. Po zpracování označené zastávky s danou zastávku s odznačí. Ve druhém kroku kola k nyní neuvažuje všechny linky $r \in \mathcal{R}$, ale pouze linky $(r,s) \in Q$ s počátkem procházení trasy v s (viz Obrázek 3.1). Třetí krok kola upravíme tím stylem, že bude algoritmus procházet pouze označené zastávky s a pro každou takovou zastávku $s' \in \mathcal{S}$, pro niž je funkce $\mathcal{F}(s,s')$ definovaná, se pokusí vylepšit příjezdový čas $\tau_k(s')$ užitím tohoto přestupu. Ukončovací podmínku stačí upravit na kontrolu počtu označených zastávek na konci každého kola. Pokud žádná zastávka není označená, nemá smysl pokračovat ve vyhledávání.



Obrázek 3.1: Vyhledávání spojení z s_s do s_t. V nultém kole se nastaví požadovaný čas odjezdu pro s_s. Linka r_1 se prochází v prvním kole ze zastávky s_s. Linky r_2 a r_3 se prochází ve druhém kole ze zastávky $s_{r_1 \cap r_2}$, respektive $s_{r_1 \cap r_3}$. Linka r_4 se prochází ve třetím kole ze zastávky $s_{r_2 \cap r_4}$. Nevybarvené zastávky nejsou nikdy navštíveny, neboť neexistuje způsob, jak se do nich dostat.

V základní verzi algoritmu jsme rovněž uvedli, že při inicializaci kola se pokaždé kopírují hodnoty $\tau_k(s) = \tau_{k-1}(s)$ pro $\forall s \in \mathcal{S}$. Tímto krokem lze ztratit spoustu času. Proto si bude algoritmus pamatovat hodnotu $\tau^*(s)$ pro $\forall s \in \mathcal{S}$, která značí nejdřívější možný příjezd do zastávky s. Formálně by tato hodnota šla definovat předpisem $\tau^*(s) = \min\{\tau_i(s) : 0 \leq i \leq k \leq n\}$, kde k je aktuální kolo, n je maximální počet přestupů a $s \in \mathcal{S}$. Jelikož nás zajímají spojení s nejdřívějším příjezdem, při procházení linek nebude algoritmus označovat všechny zastávky $s \in \mathcal{S}$, pro něž $\tau_k(s) < \tau_{k-1}(s)$, ale pouze ty, pro něž $\tau_k(s) < \tau^*(s)$. Zamezíme tak zbytečnému kopírování, neboť hodnoty τ^* jsou pro všechna kola sdílená.

V této sekci uvedeme ještě jedno triviální zrychlení algoritmu. Algoritmus ve své základní podobě hledá spojení z s_s do všech zastávek $s \in \mathcal{S}$, ale našim cílem je nalézt spojení pouze do s_t. Proto, při procházení k-tého kola, nebude algoritmus označovat takové zastávky $s \in \mathcal{S}$, jejichž příjezdový čas by byl v k-tém kole horší než nejlepší příjezdový čas do s_t. Bude označovat pouze takové zastávky $s \in \mathcal{S}$, pro něž platí $\tau_k(s) < \tau^*(s_t)$.

Nyní uvedeme kód algoritmu se všemi zmíněnými vylepšeními, který kopíruje referenční verzi [4]. V této verzi je algoritmus implementován v našich aplikacích. Budeme kalkulovat pouze s příjezdovými časy, spojení lze snadno zrekonstruovat pomocí ukazatelů na zastávky, jak jsme uvedli při popisu algoritmu. Algoritmus pracuje nad jízdním řádem $(\Pi,\mathcal{S},\mathcal{R},\mathcal{T},\mathcal{F})$, který má načtený v operační paměti.

Algoritmus 1 RAPTOR

Vstup: výchozí a cílová zastávka $s_s, s_t \in \mathcal{S}$, čas odjezdu $\tau \in \Pi$ a maximální počet přestupů $n \in \mathbb{N}_0$

Výstup: nejlepší možný příjezdový čas do s_t uložený v $\tau^*(s_t)$, zrekonstruované spojení z s_s do s_t

for all i **do**
    $\tau_i(\_) \leftarrow \infty$
$\tau^*(\_) \leftarrow \infty$
$\tau_0(s_s) \leftarrow \tau$
Označ s_s
for $k \leftarrow 1,\ldots,n+1$ **do** ▷ Jedna iterace cyklu je jedno kolo.
    $Q = \emptyset$
    for all označená zastávka s **do** ▷ První fáze kola.
        for all linka r obsahující zastávku s **do**
            if $(r,s') \in Q$ pro nějakou zastávku s' **then**
                Nahraď (r,s') dvojicí (r,s) v Q, pokud s předchází s' v lince r
            else
                Přidej (r,s) do Q
        Odznač s
    for all $(r,s) \in Q$ **do** ▷ Druhá fáze kola.
        $(t,r) \leftarrow \perp$ ▷ Aktuální jízda linky.
        for all zastávka $s_i \in s,\ldots,s_{|r|}$ (části) trasy linky r **do**
            if $(t,r) \neq \perp \wedge \tau_{arr}(t,s_i) < \min\{\tau^*(s_i), \tau^*(s_t)\}$ **then**
                $\tau_k(s_i) \leftarrow \tau_{arr}(t,s_i)$
                $\tau^*(s_i) \leftarrow \tau_{arr}(t,s_i)$
                Označ s_i
            if $(t,r) = \perp \vee \tau_{k-1}(s_i) \leq \tau_{dep}(t,s_i)$ **then**
                $(t,r) \leftarrow et(r,s_i,\tau_{k-1}(s_i))$
    for all označená zastávka s **do** ▷ Třetí fáze kola.
        for all definovaný přestup $\mathcal{F}(s,s')$ **do**
            $\tau_k(s') \leftarrow \min\{\tau_k(s'), \tau_k(s) + \mathcal{F}(s,s')\}$
            Označ s'
    if žádná zastávka není označená **then return**

Pokud máme požadavek na vyhledání $n \in \mathbb{N}$ spojení, pak stačí spustit algoritmus n-krát, abychom dostali požadované výsledky. Při každé iteraci algoritmu posuneme nejdřívější čas odjezdu o jednu sekundu od času odjezdu nejrychlejšího spojení z předchozí iterace. Pokud chceme vyhledávat podle maximálního času příjezdu $\tau_{max} \in \Pi$, pak stačí spouštět algoritmus podobným způsobem do doby, dokud nevrátí spojení $\mathcal{J}$ takové, že $\tau_{arr}(\mathcal{J}) \geq \tau_{max}$. Pokud bychom chtěli vyhledávat spojení, která obsahují jen specifické dopravní prostředky, pak bychom do kódu Algoritmu 1 přidali rozšiřující podmínku na vlastnosti linky na řádek 9. Pokud bychom chtěli vyhledávat pouze bezbariérová spojení, pak bychom museli upravit Definici 4 tak, aby funkce vracela pouze bezbariérové jízdy linek.

3.2.3 Další možná vylepšení

V této sekci uvedeme další možná vylepšení algoritmu, která již v našem řešení implementována nebyla. Může se tak jednat o podnět čtenářům k rozšíření funkcionality aplikací.

Paralelizace algoritmu

Nejvíce procesorového času stráví algoritmus ve druhé fázi kola, procházení linek. Linky jsou na sobě nezávislé a jsou zpracovávané v libovolném pořadí, tudíž je možné si pro číslo $n \in \mathbb{N}$ rozdělit množinu Q na n podmnožin a každou takovou podmnožinu zpracovávat v separátním vlákně. Při vylepšování příjezdových časů se zapisuje do sdílených struktur, kde mohou vznikat race conditions. Těmto situacím lze zabránit použitím základních synchronizačních primitiv, kde ale může vzniknout netriviální režie, a tudíž by vícevláknový běh mohl být méně efektivní než běh jednovláknový. V této sekci si uvedeme dvě lock-free varianty implementace algoritmu tak, jak jsou popsané v referenčním článku [4].

- První varianta se spoléhá na fakt, že zápisy do $\tau_k(s)$ jsou atomické, kde $k \in \mathbb{N}$ je aktuální kolo a $s \in \mathcal{S}$. V takovém případě lze „naslepo“ zapisovat hodnoty do $\tau_k(s)$ s jistotou, že na odpovídající adrese v paměti vždy nalezneme validní horní mez na čas příjezdu do s, ať už dané vlákno hodnotu přepsalo či nikoliv. Během procházení si každé vlákno bude vést *update log*, kam si bude zapisovat všechny případné aktualizace $\tau_k(s)$, které se pokusí provést. Hlavní vlákno poté sekvenčně projde všechny logy, zkontroluje všechny záznamy a případně opraví chyby vzniklé sdíleným přístupem.

- Druhá varianta spočívá ve vytvoření *precedenčního grafu* při inicializaci dat. Pokud dvě linky $r_1, r_2 \in \mathcal{R}$ nesdílí ve své trase žádnou zastávku, pak je bezpečné každou z nich procházet paralelně v jiném vlákně. Pro precedenční graf $G = (V,E)$ bude platit, že $V = \mathcal{R}$ a $E = \{(r_1, r_2) : r_1, r_2 \in V \wedge (\exists s \in \mathcal{S} : s \in r_1 \wedge s \in r_2)\}$. Poté obarvíme vrcholy grafu tak, aby žádné dva sousední vrcholy neměly stejnou barvu. Vrcholy, respektive linky, které mají stejnou barvu, lze pak zpracovávat nezávisle paralelně. Detaily efektivní implementace této varianty jsou k dispozici v referenčním článku [4].

Vyhledávání podle tarifních pásem

Až doposud jsme vyhledávali spojení pouze podle nejdřívějšího času příjezdu a až na výjimky jsme jiné požadavky na spojení nekladli. Na konci sekce 3.2.2 jsme si ukázali, jak snadno lze rozšířit algoritmus, přidáním pouze jedné podmínky do těla algoritmu. Nyní si na příkladu tarifních pásem ukážeme další rozšíření, které se již bez většího zásahu do algoritmu neobejde.

Dopravní společnosti obvykle každé zastávce $s \in \mathcal{S}$ přiřazují jedno nebo více tarifních pásem z množiny $\mathcal{Z}$. Celková cena spojení se poté odvíjí od druhů pásem $z \in \mathcal{Z}$, jimiž dané spojení projíždí, a času, který je potřebný na projetí daného tarifního pásma. Počítání ceny spojení přímo v algoritmu by mohlo být složité, spousta údajů by musela být uvedena přímo v algoritmu ve formě konstant a při změně cenové politiky společnosti by mohlo dojít k omezení funkčnosti algoritmu. Proto tyto věci nebudeme řešit a v algoritmu budeme uvažovat pouze tarifní pásma, která jsou ve spojení zahrnuta. Následná cena spojení se vypočítá až v postprocessingu.

Dříve byl hodnotám $\tau_k(s)$, pro $0 \leq k \leq n$ a $s \in \mathcal{S}$, přiřazován nejdřívější čas příjezdu do zastávky s užitím $k-1$ přestupů, tedy hodnota $\tau_{arr}(t,s)$ pro nějakou jízdu linky $(t,r) \in \mathcal{T}$. Nyní do těchto hodnot bude algoritmus ukládat dvojici $(\tau_{arr}(t,s), Z)$, kde $Z \subseteq \mathcal{Z}$ bude podmnožina všech tarifních pásem, které jsme navštívili, abychom se do dané zastávky s dostali pomocí $k-1$ přestupů. Porovnávací relaci rozšíříme tak, že $(\tau_{arr}(t,s), Z) \leq (\tau_{arr}(t',s), Z')$ právě tehdy, když $\tau_{arr}(t,s) \leq \tau_{arr}(t',s)$ a zároveň $Z \subseteq Z'$. Inicializační hodnotě $\tau_0(s_s)$ pro výchozí zastávku $s_s \in \mathcal{S}$ přiřadí algoritmus dvojici $(\tau, \zeta(s_s))$, kde $\tau \in \Pi$ je čas odjezdu a ζ je zobrazení $\zeta : \mathcal{S} \to \mathcal{Z}$, které zastávce $s \in \mathcal{S}$ přiřadí podmnožinu $Z \subseteq \mathcal{Z}$ tarifních pásem, pod něž spadá. Při zlepšování hodnoty $\tau_k(s) = (\tau_{arr}(t,s), Z)$ hodnotou $(\tau_{arr}(t',s), Z')$ nyní bude algoritmus do $\tau_k(s)$ zapisovat hodnotu $(\tau_{arr}(t',s), Z \cup Z')$, aby byla zohledněna všechna tarifní pásma v daném spojení. Implementaci podmnožiny tarifních pásem $Z \subseteq \mathcal{Z}$ můžeme efektivně provést pomocí bitových operací. Jeden bit bude reprezentovat jedno tarifní pásmo, bitové AND bude test na podmnožinu a bitové OR bude sjednocení tarifních pásem. Bitovou masku použijeme, pokud budeme chtít vyhledávat spojení pouze v určitých tarifních pásmech.

3.2.4 Návrh datových struktur

Uvedený Algoritmus 1 pracuje nad jízdním řádem $(\Pi,\mathcal{S},\mathcal{R},\mathcal{T},\mathcal{F})$, který musí mít po dobu svého běhu načtený v paměti. Rovněž je potřeba zajistit, aby se výsledky z algoritmu pohodlně předaly uživateli. V této sekci si ukážeme jednu z možných implementací datových struktur, která je použita i v našem řešení, přičemž uvedeme jen to nezbytně nutné a implementační detaily ponecháme stranou.

Vstupní struktury

Těžištěm algoritmu je procházení linek. Aby mohla být linka $r \in \mathcal{R}$ zpracována efektivně, je vyžadována posloupnost její zastávek v pořadí, v jakém je projíždí. Dále je nutné mít seřazené jízdy linky od nejdřívější po nejpozdější, abychom mohli efektivně implementovat funkci pro nalezení nejdřívější jízdy linky. Zastávky v posloupnosti jsou nezávislé na lince a mohou být sdílené pro více linek,

zatímco jízda linky $(t,r) \in \mathcal{T}$ bez linky r existovat nemůže. Proto je vhodné linku r v paměti reprezentovat tak, aby obsahem instance jedné linky byly dvě pole, jedno pro položky $\forall t : (t,r) \in \mathcal{T}$ a druhé pole pro ukazatele na zastávky v trase linky r. Tedy pokud zanikne instance linky, pak zaniknou i všechny její jízdy, ale zastávky budou existovat i nadále. Instance všech linek lze v paměti ukládat libovolně, neboť se k nim nikdy nepřistupuje přímo, což se dozvíme v následujících odstavcích. Tím pádem je nejvhodnější je uchovávat v poli, neboť pole nemá žádnou režii na paměť.

Jízda linky je definována jako zobrazení, což je pro reprezentaci v paměti nepraktické. Proto si zaveďme strukturu zastavení jízdy linky, která bude obsahovat ukazatel na zastávku $s \in r$ a časy $\tau_{arr}(t,s) \in \Pi$ a $\tau_{dep}(t,s) \in \Pi$ pro jízdu linky $(t,r) \in \mathcal{T}$. Instance jízdy linky v paměti pak bude posloupnost všech zastavení dané jízdy linky, seřazená podle času odjezdu jednotlivých zastavení. Tuto posloupnost budeme z pochopitelných důvodů implementovat pomocí pole, neboť pak můžeme v čase $\mathcal{O}(1)$ zjistit čas příjezdu či odjezdu z i-té zastávky jízdy linky, pro $1 \leq i \leq |r|$.

Poslední potřebná struktura je struktura pro zastávky. S odvoláním na řádek 9 Algoritmu 1 je nutné, aby zastávka $s \in \mathcal{S}$ obsahovala kolekci ukazatelů na všechny linky $r \in \mathcal{R}$, jejichž trasa obsahuje zastávku s. Toto je důvod, proč jsme v prvním odstavci této podsekce zmínili, že se k linkám nikdy nepřistupuje přímo. Přistupuje se k nim pouze skrze zastávky užitím této kolekce, což je patrné z kódu Algoritmu 1. Tuto kolekci je vhodné implementovat tak, aby byla snadno iterovatelná. Proto postačí obyčejné pole.

Až doposud jsme neuvedli žádnou strukturu pro přestupy. Přestupy totiž budeme uchovávat přímo v zastávkách. Z řádku 25 Algoritmu 1 je zřejmé, že se opět bude muset jednat o snadno iterovatelnou kolekci. Obsahem této kolekce budou dvojice $(s', \mathcal{F}(s,s'))$, kde $s' \in \mathcal{S}$, pro něž je funkce $\mathcal{F}(s,s')$ definovaná.

Struktury pro algoritmus

Algoritmus vyžaduje ke své činnosti čtyři struktury:

- Struktura pro nejlepší časy příjezdu $\tau^*(s)$ do zastávek $s \in \mathcal{S}$. Horní mez velikosti této struktury bude $|\mathcal{S}|$ prvků. Jako vhodná struktura se zde nabízí pole o velikosti $|\mathcal{S}|$ s časem přístupu $\mathcal{O}(1)$ ke každé položce. Nicméně díky řádku 18 v Algoritmu 1 se zpracuje pouze zlomek zastávek, tudíž většina místa v poli zůstane nevyužitá. Další možností je varianta s červeno-černým stromem, kde lze provést všechny základní operace ve složitosti $\mathcal{O}(\log n)$ [3, str. 206]. Obě varianty jsou přípustné, při samotné implementaci je nutné pamatovat na procesorovou cache, jak odhadovaná velikost $|\mathcal{S}|$ může ovlivnit výpadky cache v jedné či druhé variantě, a najít vhodný kompromis.

  My jsme implementovali obě varianty[3] a na základě profilace algoritmu vyšlo najevo, že varianta s polem je až o 40% rychlejší, ovšem za cenu několikanásobně vyšší spotřeby paměti.

[3]Mezi variantami lze v implementaci přepínat pomocí makra v hlavičkovém souboru s definicemi maker v jádru.

- Struktura pro nejlepší časy příjezdu $\tau_k(s)$ do zastávek $s \in \mathcal{S}$ v k-tém kole. Zde je nejvhodnější použít dynamické pole, přičemž jedna položka v poli bude struktura z předchozího bodu.

- Struktura pro označené zastávky $s \in \mathcal{S}$. Při označování zastávky je nutné zajistit, aby každá zastávka byla v této struktuře nejvýše jednou. Při implementaci dynamickým polem by tento test trval $\mathcal{O}(n)$, což není optimální. Proto jsme zde zvolili variantu s červeno-černým stromem, kde test na výskyt prvku proběhne ve složitosti $\mathcal{O}(\log n)$. Ve struktuře nebudeme uchovávat instance zastávek, nýbrž pouze ukazatel na ně.

- Struktura Q pro dvojice (r,s), kde $r \in \mathcal{R}$ a $s \in \mathcal{S}$. Zde platí totéž co v předchozím případě, každá linka $r \in \mathcal{R}$ může být v Q pouze jednou.

Výstupní struktury

Dle Definice 2 je spojení posloupnost jízd linek a přestupů. Každému prvku této posloupnosti budeme říkat *segment spojení*. Jeden segment je produktem jednoho kola algoritmu. Při zahájení vyhledávání spojení nejsme schopni předem určit, kolik bude mít hledané spojení segmentů. Počet segmentů je sice možné shora omezit číslem $2n + 1$, kde $n \in \mathbb{N}_0$ je maximální počet přestupů, nicméně při implementaci polem by tato varianta pro velká n mohla být zbytečně drahá. Proto se zde nabízí možnost implementace spojení pomocí spojového seznamu, kde každý segment bude obsahovat ukazatel na svého předchůdce. Tím můžeme jednoduše iterativně přidávat segmenty v průběhu algoritmu, přičemž výsledný spojový seznam na konci vyhledávání obrátíme a získáme tím hledané spojení. Výsledný spojový seznam bude typicky obsahovat malý počet segmentů, tudíž hovořit o neefektivitě spojového seznamu vůči procesorové cache není v tomto případě relevantní.

Každý segment má navíc dané dvě zastávky $s_s, s_t \in \mathcal{S}$, které značí výchozí zastávku daného segmentu, respektive cílovou zastávku daného segmentu. Poslední společný rys všech segmentů je pevně daný čas $\tau \in \Pi$ příjezdu do své cílové zastávky. S těmito informacemi nyní můžeme algoritmus upravit tak, že do hodnot $\tau_k(s)$, pro zastávku $s \in \mathcal{S}$, se nebudou ukládat pouze nejdřívější příjezdové časy do dané zastávky s užitím $k-1$ přestupů, ale celé segmenty, které budou i nadále reprezentovat nejdřívější příjezdový čas a zároveň z nich můžeme zrekonstruovat celé spojení.

Existují dva druhy segmentů:

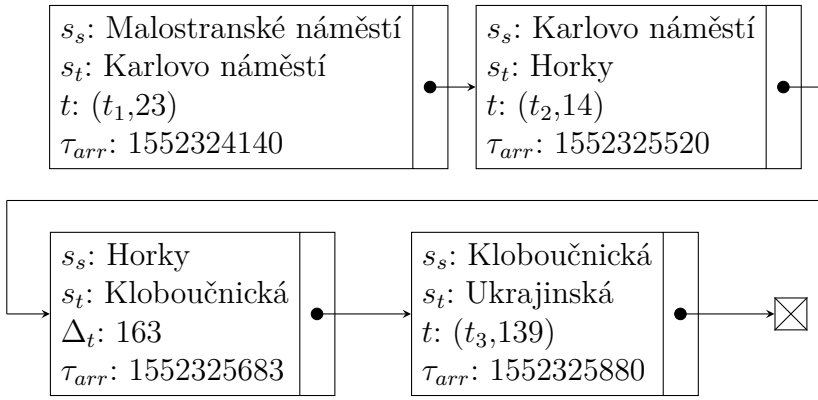
- Segment jízdy linky. Tyto segmenty se vytváří ve druhé fázi jednoho kola algoritmu. Obsahují navíc odkaz na jízdu linky $(t,r) \in \mathcal{T}$, ze které lze určit $\tau_{dep}(t,s_s)$ a $\tau_{arr}(t,s_t)$. Z informacích uložených v r lze dále zjistit různé informace o lince, jako například směr jízdy nebo její označení.

- Segment přestupu. Tyto segmenty se vytváří ve třetí fázi jednoho kola algoritmu. Obsahují navíc dobu trvání daného přestupu. Při inicializaci algoritmu ukládáme do hodnoty $\tau_0(s_s)$ *prázdný segment přestupu*, kde $s_s = s_t$ a doba trvání přestupu je 0 sekund. Příjezdový čas do s_s nastavíme na $\tau \in \Pi$.

Nyní je zřejmé, že pro segment lze vytvořit abstraktní třídu, od níž bude dědit třída pro segment jízdy linky a třída pro segment přestupu. Třída segmentu bude mít abstraktní metody pro získání jízdy linky, výchozí zastávky segmentu, cílové zastávky segmentu, času odjezdu z výchozí zastávky, času příjezdu do cílové zastávky, mezizastávek a doby trvání segmentu. V případě segmentu přestupu bude implementace metody pro získání jízdy linky prázdná, respektive bude vracet nedefinovanou hodnotu. Tímto způsobem snadno rozlišíme druhy segmentů. Implementace metody pro získání mezizastávek bude vracet prázdné pole.

Struktura pro spojení zastřešuje spojový seznam segmentů. V naší implementaci obsahuje konstruktor, který přijímá instance potomků třídy abstraktního segmentu, přičemž v konstruktoru dochází k otočení spojového seznamu a odstranění prázdných segmentů přestupu. Je vhodné, aby tato struktura obsahovala pomocné metody, které nám usnadní získat různé informace ze segmentů, jako například celkovou doba trvání spojení nebo počet přestupů.

Instanci třídy pro spojení ukazujeme na příkladě, viz Obrázek 3.2. Cílová zastávka prvního segmentu je shodná s výchozí zastávkou druhého segmentu, tudíž zde není nutný žádný segment přestupu. Hodnoty τ_{arr} jsou ve formátu Unixového času. Názvy zastávek jsou reálně používány pouze v klientských aplikacích, algoritmus počítá s identifikátory zastávek, což jsou přirozená čísla. My jsme zde pro jednoduchost použili názvy.



Obrázek 3.2: Příklad reprezentace spojení *Malostranské náměstí - Ukrajinská* v paměti.

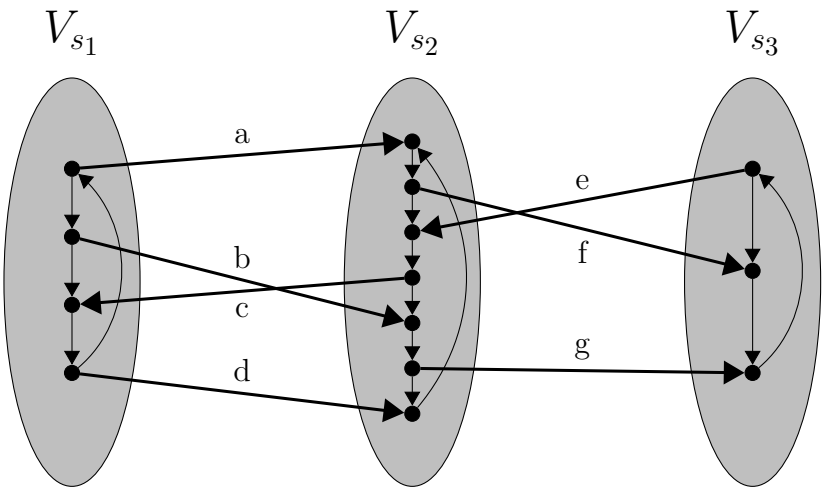
3.3 Jiné možné přístupy

V sekci 3.2 jsme detailně rozebrali RAPTOR algoritmus. Nyní si ukážeme další možnosti řešení problému nejdřívějšího příjezdu a pokusíme se vysvětlit, proč je naše řešení nejlepší možné. Ukážeme si pouze zjednodušené principy, pokud by se čtenář chtěl dozvědět o dané problematice více, je možné si přečíst odborný článek [5], z něhož jsme čerpali informace pro tuto sekci.

Většina řešení využívá namodelování jízdního řádu grafem, na který se poté spustí Dijkstrův algoritmus, ten vyřeší problém nejdřívějšího příjezdu. Předpokládáme, že Dijkstrův algoritmus [3, str. 146] je čtenáři dobře znám i z jiných oblastí teoretické informatiky, proto si zde uvedeme pouze dva nejvíce používané modely pro vytvoření grafu a problémem nejkratší cesty v grafu se nebudeme zabývat.

3.3.1 Time-expanded model

Časově expandovaný graf [5, str. 6] $G = (V,E)$ je orientovaný graf, kde každá hrana v grafu je ohodnocená nějakým přirozeným číslem $n \in \mathbb{N}$. Množinu vrcholů tvoří všechny časové události v jízdním řádu, množinu hran tvoří všechna elementární spojení a pro všechny zastávky $s \in \mathcal{S}$ navíc zavedeme hrany mezi každou časovou událostí, která se váže k zastávce s.



Obrázek 3.3: Vizualizace časově expandovaného grafu pro zastávky $s_1, s_2, s_3 \in \mathcal{S}$. Hrany mezi jednotlivými zastávkami reprezentují elementární spojení.

Uvažme jednu jízdu linky $(t,r) \in \mathcal{T}$, pak množina vrcholů grafu pro danou jízdu (t,r) je dána všemi definovanými časy příjezdu a odjezdu v dané jízdě, tedy množinou $V_{(t,r)} = \bigcup_{s \in r} \{\tau_{arr}(t,s), \tau_{dep}(t,s)\}$. Pro graf celé dopravní sítě platí $V = \bigcup_{(t,r) \in \mathcal{T}} V_{(t,r)}$. Vrcholy budeme sdružovat do skupin V_s podle zastávky $s \in \mathcal{S}$, které daná časová událost patří.

Nyní si popíšeme množinu hran E, hrany patřící do této množiny lze rozdělit na dva druhy:

- Mějme jízdu linky $(t,r) \in \mathcal{T}$, kde trasu linky tvoří posloupnost zastávek $s_1, \ldots, s_{|r|}$. Pak elementární spojení pro jízdu (t,r) budou tvořit čtveřice $(s_i, s_{i+1}, \tau_{dep}(t,s_i), \tau_{arr}(t,s_{i+1}))$ pro $i \in \{1, \ldots, |r| - 1\}$. Množinu všech elementárních spojení jízdy (t,r) budeme značit $\mathcal{E}_{(t,r)}$. Pro tento druh hran bude platit, že $(\tau_1, \tau_2) \in E \iff \exists (t,r) \in \mathcal{T}\ \exists s_1, s_2 \in r : (s_1, s_2, \tau_1, \tau_2) \in \mathcal{E}_{(t,r)}$.
- Mějme $V_s \subset V$ pro zastávku $s \in \mathcal{S}$, tedy všechny vrcholy grafu asociované s danou zastávkou s. Pro jednoduchost předpokládejme, že množina $V_s = \{\tau_1, \ldots, \tau_{|V_s|}\}$ je uspořádaná. Pro takovou množinu V_s bude množina hran E_s rovna $E_s = \{(\tau_k, \tau_{k+1}) : 1 \leq k \leq |V_s| - 1\} \cup \{(\tau_{|V_s|}, \tau_1)\}$. Pro celý graf dostáváme množinu tohoto druhu hran danou předpisem $E = \bigcup_{s \in \mathcal{S}} E_s$.

Cenu hrany tvoří v obou případech rozdíl hodnot její vrcholů, tedy pro hranu $(\tau_1, \tau_2) \in E$ platí, že její cena je rovna číslu $\tau_2 - \tau_1$. V grafu tedy pro každou zastávku vznikne určitá množina vrcholů, kde jednotlivé vrcholy budou tvořit

cyklus s uspořádáním podle jednotlivých časových událostí. Mezi jednotlivými uzly dvou zastávek povede hrana, pokud existuje elementární spojení mezi nimi. Naše verze nezahrnuje přestupy mezi zastávkami, o jejich zahrnutí hovoří několik odstavců v referenčním článku [5].

Je zřejmé, že tento přístup nebude příliš efektivní, neboť velikost množiny $|V|$ bude, v průměrném případě, v řádech statisíců až malých milionů vrcholů, zatímco hledaná nejkratší cesta bude obsahovat malé desítky vrcholů. Z každého vrcholu povede minimálně jedna a nanejvýše dvě hrany, což plyne z konstrukce hran. S odvoláním na časovou složitost $\mathcal{O}(|E| + |V| \log |V|)$ Dijkstrova algoritmu [3, str. 149] a zmíněná fakta je zřejmé, že toto řešení nemůže být rychlejší než naše řešení ze sekce 3.2, které pracuje v lineární časové složitosti s mnohem méně údaji.

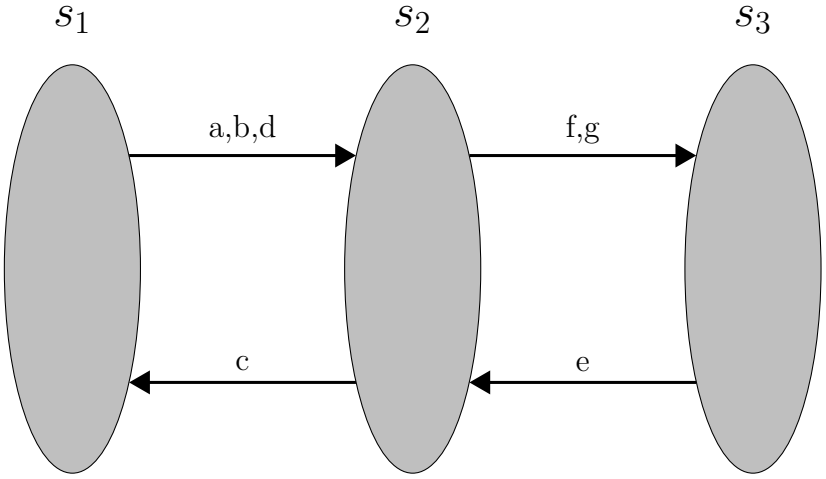
Na závěr této sekce si uveďme tvrzení, které potvrdí korektnost tohoto přístupu.

Tvrzení 3. *Nechť s_s,$s_t \in \mathcal{S}$ je výchozí, respektive cílová zastávka hledaného spojení a $\tau \in \Pi$ čas odjezdu. Dále nechť $v_1 = \min\{v \in V_{s_1} : v \geq \tau\}$. Pak problém nejdřívějšího příjezdu ze zastávky s_s do zastávky s_t s časem odjezdu nejdříve v τ lze převést na problém nejkratší cesty v časově expandovaném grafu z vrcholu v_1 do libovolného vrcholu $v_2 \in V_{s_2}$.*

Důkaz. Viz referenční článek [5, str. 7]. □

3.3.2 Time-dependent model

Časově závislý graf [5, str. 8] $G = (V,E)$ je orientovaný graf. Na rozdíl od časově expandovaného grafu má mnohem méně vrcholů, neboť $V = \mathcal{S}$. Označme $\mathcal{E}_{s_1,s_2}$ množinu všech elementárních spojení mezi zastávkami $s_1, s_2 \in \mathcal{S}$. Pak pro množinu hran E platí, že $(s_1,s_2) \in E \iff \mathcal{E}_{s_1,s_2} \neq \emptyset$, tedy hrana mezi vrcholy s_1 a s_2 vede právě tehdy, pokud existuje nějaké elementární spojení mezi zastávkami s_1 a s_2. V Obrázku 3.3 jsme modelovali jízdní řád časově expandovaným grafem, v Obrázku 3.4 nalezneme model téhož jízdního řádu pomocí časově závislého grafu.



Obrázek 3.4: Vizualizace časově závislého grafu pro zastávky $s_1, s_2, s_3 \in \mathcal{S}$. Označení hran znázorňuje všechna elementární spojení, které můžeme pro dané dvě zastávky v daném směru použít.

Cena hrany závisí na čase $\tau \in \Pi$, v kterém se ji algoritmus pokusí využít pro vyřešení problému nejdřívějšího příjezdu. Zde můžeme využít naši funkci pro nalezení nejdřívější jízdy linky z Definice 4. Nechť $(s_1, s_2) \in E$ a $\tau_{s_1} \in \Pi$ je čas příjezdu do zastávky s_1. Označme $(t,r) = \min\{et(r, s_1, \tau_{s_1}) : r \in \mathcal{R} \wedge s_1 \in r \wedge s_2 \in r \wedge (\exists \tau_1, \tau_2 \in \Pi : (s_1,s_2,\tau_1,\tau_2) \in \mathcal{E}_{s_1,s_2})\}$. Pak $\tau_{s_2} = \tau_{arr}(t,s_2)$ je nejdřívější čas příjezdu do s_2 využitím elementárního spojení $(s_1,s_2,\tau_{dep}(t,s_1),\tau_{arr}(t,s_2)) \in \mathcal{E}_{s_1,s_2}$ a platí, že cena hrany (s_1,s_2) je rovna rozdílu $\tau_{s_2} - \tau_{s_1}$.

Bližší popis, jak lze upravit Dijkstrův algoritmus pro tento model, je k dispozici v referenčním článku [5, str. 9], společně s tvrzením dokazující korektnost tohoto přístupu a s technikou, jak do grafu zakomponovat přestupy mezi zastávkami.

Pro vyhledávání v jízdních řádech se obecně preferuje time-dependent model nad time-expanded modelem, zejména z důvodu velikosti jednotlivých grafů. Time-dependent model se efektivitou blíží k našemu řešení ze sekce 3.2, i přes to je naše řešení dvakrát až třikrát rychlejší co do času provedení požadavku. [4]

4. Architektura systému

V této kapitole popíšeme architekturu jednotlivých aplikací, jak spolu komunikují a pokusíme se najít řešení pro cíle, které jsme si stanovili v sekci 1.1.

4.1 Komponenty systému

Systém se skládá ze serverové, počítačové a mobilní aplikace. Klientské aplikace posílají své požadavky na vzdálený server, kde se vyřizují, a odpovědi na ně se odesílají zpět. Tento přístup jsme zvolili, neboť samotné vyhledávání má vysoké nároky na výkon procesoru a pokud by uživatelův stroj nebyl dostatečně výkonný, vyhledávání by mohlo být neefektivní. Dalším důvodem je vyšší spotřeba operační paměti, která by mohla být pro slabší mobilní zařízení neakceptovatelná. Nicméně jsme ponechali možnost uživatelům, jejichž stroj je dostatečně výkonný pro samotné vyhledávání a kteří nemají k dispozici server, jejž by mohli provozovat. Systém totiž podporuje běh v online a offline módu:

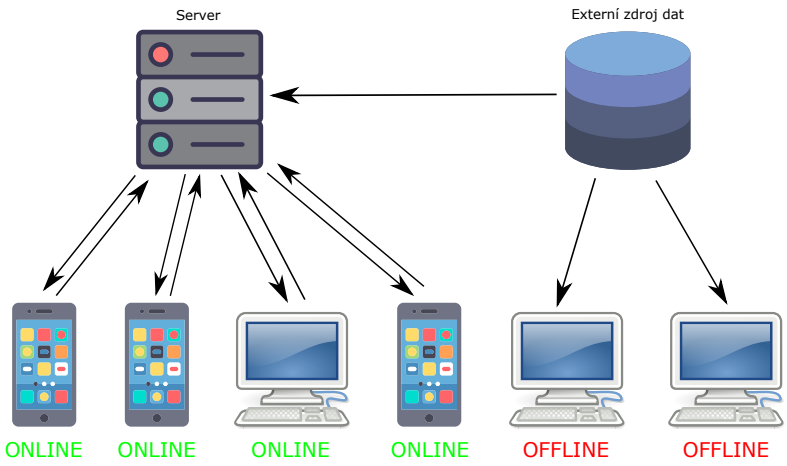
- V online módu může fungovat počítačová i mobilní aplikace. Obě aplikace vyžadují běh serverové aplikace. Serverová aplikace může běžet kdekoliv v síti, klientské aplikace se k ní připojují pomocí IP adresy. Veškeré vyhledávání pak probíhá v serverové aplikaci. Data jsou uložena na serveru, ten klientským aplikacím poskytuje základní data, která klientské aplikace potřebují pro svůj základní běh. Mezi tato data se řadí údaje o zastávkách a linkách, které jsou následně využité v mapě, „našeptávačích“ a při kontrolách korektnosti požadavku.

- V offline módu může fungovat pouze počítačová aplikace. Počítačová aplikace v tomto módu dokáže stáhnout data z uvedeného externího zdroje a zpracovat je. Po spuštění si aplikace data načte do operační paměti a vyhledávání v jízdních řádech se bude uskutečňovat nad těmito daty v rámci téhož procesu, tedy není zde nutnost mít spuštěnou serverovou aplikaci.

Motivace pro zahrnutí offline módu byl případ, kdy uživatel jede na dovolenou či pracovní cestu do zahraničí, kde nemá žádný nebo velmi omezený přístup k internetu. V takovém případě si může před odjezdem nastavit aplikaci pro cílovou oblast a nechat ji stáhnout data. Doba, po jakou data vydrží aktuální, se liší v závislosti na poskytovateli. Typicky se jedná o 1-2 týdny, což je dostatečně dlouhá doba, aby mohl uživatel po dobu svého pobytu v zahraničí vyhledávat spojení.

Online mód je vhodný pro případ, kdy má uživatel trvalý nebo částečný přístup k internetu. Pokud uživatel předpokládá, že bude bez připojení k internetu jen krátkou dobu, je možné si přidat vybraná spojení, zastávky či linky mezi oblíbené položky. Tyto oblíbené položky se následně stáhnou do zařízení, při kvalitním připojení se budou automaticky aktualizovat a bude možné je vyhledávat i offline. Motivací pro zahrnutí této funkce byly případy, kdy uživatel chce vyhledávat spojení v metru či se vyskytuje v nějaké restauraci ve sklepě a chce si najít spojení domů.

Ve zbytku této kapitoly se nadále budeme zabývat pouze online módem. Offline mód si totiž můžeme zjednodušeně představit jako sloučení serverové aplikace, která je přístupná pouze z localhostu, a počítačové aplikace, která hledá



Obrázek 4.1: Základní schéma fungování celého systému pro jeden konkrétní zdroj dat.

server na localhostu. K offline módu se později vrátíme v sekci 5.2.1, kde se budeme zabývat implementací back-endu klientských aplikací detailněji.

4.2 Zpracování dat

V sekci 2.1 jsme si představili nejčastěji používané zdroje dat. Uvedli jsme, že chceme implementovat podporu pro GTFS formát [1] a že jako zdroj dat chceme používat zejména otevřená data PID [2]. V této sekci se na data podíváme detailněji a ukážeme si všechny kroky, od stažení dat až po použití dat algoritmem.

4.2.1 Požadavky na data

Pro účely našich aplikací vytvoříme dva nové formáty, s hlavními cíli minimalizace místa na disku a maximalizace výkonu jádra. Důvod tohoto přístupu bude zřejmý v sekci 4.2.3, kde si rovněž popíšeme výhody i nevýhody našeho řešení.

Tyto formáty označíme jako *Transit Feed Data (TFD)* a *Transit Feed Basic (TFB)*, kde formát TFD bude využívat jádro systému k vyhledávání a formát TFB budou využívat klientské aplikace. Formát TFB je podmnožinou formátu TFD, neboť obsahuje pouze soubor se zastávkami, stanicemi a informacemi o linkách. Využíváme jej v mapě, „našeptávačích“ a při kontrolách korektnosti požadavku.

Požadavky na data jsou dána fungováním algoritmu ze sekce 3.2. Přesnou specifikaci obou formátů lze nalézt v Příloze A.1.2. V Obrázku A.1 je k nahlédnutí zjednodušené UML schéma obou formátů. V následujících odstavcích si uvedeme několik problémů, které bylo třeba při navrhování datových formátů řešit.

Zastávky a stanice

V celé kapitole 3 jsme hovořili pouze o zastávkách a o stanicích nepadla žádná zmínka. Důvod pro zavedení stanic je ten, že při vyhledávání spojení uživatel typicky nechce vyhledávat spojení jen z jedné zastávky, ale ze všech zastávek v celé stanici. Stanici proto definujeme jako soubor zastávek se stejným názvem. Každá

zastávka musí mít přiřazenou nějakou stanici, pokud existuje jen jedna zastávka s daným názvem, pak bude stanice obsahovat pouze tuto jednu zastávku. Při vyhledávání pak hledáme spojení ze všech zastávek výchozí stanice do libovolné zastávky cílové stanice.

Přestupy mezi zastávkami

Dále řešíme problém s přestupy mezi zastávkami a jak vypočítat čas přestupu. Přestupy mezi zastávkami jsou důležité zejména pro zastávky spadající pod jednu stanici, z důvodu přestupů mezi jednotlivými dopravními prostředky.

U zastávek známe jen jejich polohu, tedy nejjednodušší způsob, jak určit vzdálenost mezi nimi, je na základě letecké vzdálenosti. Na určení vzdálenosti mezi dvěma body na povrchu koule lze užít Haversine formuli.[1] Tento vzorec využívá ve velké míře goniometrické funkce, které jsou velmi náročné na procesorový čas.

Proto jsme zvolili určitý kompromis. Přestupy budou typicky na malé vzdálenosti, proto můžeme povrch Země aproximovat pomocí roviny. Na základě všech zastávek v seznamu určíme „průměrnou polohu zastávek“, což bude zprůměrovaná zeměpisná šířka a zeměpisná délka všech zastávek. V dalším kroku určíme „délku“ jednoho stupně zeměpisné šířky a zeměpisné délky. Délka jednoho stupně zeměpisné šířky je všude na světě stejná, jedná se o 111 kilometrů. Při výpočtu délky jednoho stupně zeměpisné délky vyjdeme z faktu, že 111,321 kilometrů je délka jednoho stupně zeměpisné délky na rovníku. Nyní stačí vypočítat cosinus průměrné zeměpisné délky a vynásobit ho touto hodnotou. Právě jsme převedli problém vzdálenosti na povrchu koule na problém vzdálenosti v rovině, tudíž můžeme k určení vzdálenosti mezi dvěma zastávkami využít Pythagorovu větu.

Spočtenou vzdálenost následně vydělíme průměrnou rychlostí chůze, načež dostaneme čas přestupu. Průměrná rychlost chůze je nastavena na 0,9 metrů za sekundu, nicméně v serverové aplikaci lze tato hodnota měnit z konfiguračního souboru, jak se dozvíme v sekci 5.1. Celkové trvání přestupu lze dále přizpůsobovat přímo z klientských aplikací, kde se doba trvání přestupů dá škálovat pomocí koeficientu.

Zmíněný přístup má několik nevýhod, zejména v rámci podzemní dopravy. Tento problém se snažíme vyřešit pomocí různých koeficientů, kterými násobíme dobu trvání přestupu a které jsou rovněž modifikovatelné v konfiguračním souboru.

Pokud využíváme metro a přestupujeme z linky na tutéž linku v opačném směru[2], dobu trvání přestupu vynásobíme hodnotou 0,5. Souřadnice zastávek bývají nastavené na konce nástupišť, úhlopříčná vzdálenost mezi dvěma konci nástupišť je příliš velká.

Pokud přestupujeme z jedné linky metra na jinou linku metra, trvání přestupu vynásobíme hodnotou 2. Dané zastávky jsou typicky kousek od sebe, protože linky se kříží mimoúrovňově, ale mnoho času při přestupu strávíme na eskalátorech. U

[1] `https://en.wikipedia.org/wiki/Haversine_formula`

[2] Tento problém jsme v rámci PID řešili zejména na počátku roku 2018, kdy docházelo k opravám ve stanici Muzeum a ve stanici Náměstí Míru se muselo přestupovat na vlak v opačném směru.

přestupů z podzemní dopravy na povrchovou a vice versa, z podobných důvodů jako v předchozím případě, násobíme tentokrát hodnotou 3.

V reálné situaci poskytují dopravní společnosti údaje o těchto přestupech ve formě separátního dokumentu. Struktura takového dokumentu se liší v závislosti na dopravní společnosti, z toho důvodu jsme toto řešení nezvolili, neboť bychom nesplnili jeden z cílů, který jsme si zadali. A sice podporovat libovolná data s jízdními řády.

Další nevýhodu si demonstrujeme na příkladě. Představme si řeku, kde nikde v okolí není most. Na obou březích řeky se vyskytuje zastávka, letecká vzdálenost mezi nimi je reálně pouze několik desítek metrů. Našim přístupem určíme minimální dobu přestupu, avšak reálná doba přestupu může být mnohonásobně větší.[3] Tento problém však v našich aplikacích neřešíme. Možným řešení by bylo využít například Google Maps API [6] a počítat přestupy mezi zastávkami tak, aby měla vzdálenost mezi nimi oporu ve fyzické mapě.

Dále je také nutné zajistit, aby nebyly vytvořeny přestupy mezi všemi zastávkami v dopravní síti. Velká většina přestupů by se nikdy reálně nevyužila, což by vedlo k plýtvání s místem na disku a následně i pamětí. Za tímto účelem budeme ukládat pouze přestupy, jejichž doba trvání nepřesahuje 600 sekund. Tato hodnota lze rovněž modifikovat v konfiguračním souboru.

Trasy linek

Do záznamu o lince nevkládáme její trasu, jak uvádíme v Definici 1. Je totiž nutné zohlednit jízdy linky, které začínají svou jízdu ve vozovně, respektive garáži, potažmo končí svou jízdu ve vozovně, respektive garáži. Rovněž existují linky, které se liší svou trasou i při běžném provozu.[4] Tudíž trasu linky musíme uvádět explicitně, neboť jedna linka může mít několik tras.

Pokud bychom měli předchozí odstavec uvést do kontextu definice jízdního řádu z Definice 1, pak je třeba říci, že *linka* z Definice 1 je totéž, co *trasa linky* ve formátu TFD.

Zavedení relativních časů

Relativní čas zde chápeme tak, že veškeré časy příjezdu a odjezdu jízd linek reprezentujeme pouze pomocí času v rámci dne, na datu nám zde nezáleží. Proto zde definujeme pojem služby, což bude soubor informací, díky kterým jsme schopni určit absolutní časy příjezdu a odjezdu.

Účel pro zavedení služeb je ušetření místa na disku. Jízda linky bude obsahovat údaje o čase, zatímco služba bude obsahovat údaje o datech. Tím pádem lze jízdu linky využít pro více dnů v týdnu. Kdybychom nezaváděli služby, pak bychom každou jízdu linky museli několikrát zduplikovat a pouze upravit datum, který by byl pro danou jízdu relevantní. Definice služeb kopíruje klasické jízdní řády, v těch se rozlišují tři základní služby, konkrétně služba pro všední dny, služba pro

[3] V Praze se jedná o případ autobusové zastávky Lahovice a vlakové zastávky Praha-Komořany. Náš přístup vypočítá, že přestup trvá 7 minut, avšak reálně trvá přestup více než půl hodiny.

[4] Příkladem může být pražská linka 22, kde každá druhá jízda je v trase Vypich - Nádraží Strašnice, respektive Bílá Hora - Nádraží Hostivař.

sobotu a služba pro neděli a státní svátky. Jízdní řád bývá pro každou službu jiný.

Každá služba má určené období platnosti. Dále obsahuje pro každý den v týdnu hodnotu indikující, zdali služba v daný den operuje. Služba rovněž může obsahovat seznam výjimečných událostí, které udávají přidání nebo odebrání platnosti služby pro daný datum.

Výjimečné události mohou přijít vhod, kdy je očekáváno vyšší využití hromadné dopravy, například při velkých kulturních událostech. Pokud by se taková událost konala například v jednu konkrétní neděli, kdy je obvykle provoz hromadné dopravy méně hustý, pak by dopravce přidal výjimečnou událost do služeb pro všední den pro tuto konkrétní neděli. To by mělo za následek, že v tuto konkrétní neděli by určité spoje jezdily se stejnou frekvencí jako ve všední den.

Při samotném vyhledávání implementujeme i možnost pro vyhledávání spojení, kdy je datum vyhledávání spojení mimo dané rozmezí platnosti služby. To by se mohlo stát, kdyby se data jízdních řádů nemohla aktualizovat, například kdyby byl externí zdroj dat nedostupný. Pokud se tak stane, uživateli se ve výsledcích zobrazí, že jízdní řád může být neaktuální.

Služby jsme zavedli kvůli otázce, jestli je relevantní využít danou jízdu linku pro vyhledávání spojení v zadaný datum. Kombinací období platnosti, hodnot pro dny v týdnu a seznamem výjimečných událostí jsme na tuto otázku schopni jednoznačně odpovědět.

Určení data expirace

Systém pro automatické aktualizace dat vyžaduje přesně určený okamžik, kdy má dojít k aktualizaci. Tento okamžik udává datum expirace. Datum expirace se určí na základě minimálního data expirace nějaké služby ze seznamu služeb, přičemž nepočítáme takové služby, které platí pouze po dobu jednoho dne.

4.2.2 Výchozí formát

Jak jsme zmínili v sekci 2.1, naše aplikace podporují GTFS formát [1]. Tento formát je velmi dobře zdokumentovaný. Obsahuje povinné a volitelné položky. Mezi povinné položky se, až na výjimky, řadí položky, které jsou zahrnuté v našich formátech TFD a TFB. Výjimky tvoří:

- Soubor s přestupy. Formát GTFS přestupy řadí mezi volitelné položky. Bylo by možné zpracovávat přestupy dodané GTFS formátem, kdyby daný soubor v dané datové sadě existoval. My jsme se rozhodli případnou existenci tohoto souboru ignorovat a počítat všechny přestupy způsobem popsaným v sekci 4.2.1, abychom zachovali konzistenci dat.

- Soubor se stanicemi. Formát GTFS definuje pouze zastávky, tudíž je potřeba stanice vytvořit.

- Soubor s trasami linek. Formát GTFS specifikuje pouze informace o linkách, jejich trasy jsou uvedeny až v samotných jízdách. My vyžadujeme strukturu, která sdružuje všechny jízdy se stejnou trasou, proto jsme zavedli záznam

pro trasu linky. Tento požadavek je dán fungováním algoritmu, jak je patrné z jeho popisu v sekci 3.2.

Mezi volitelné položky GTFS formátu se řadí například průjezdní body tras linek, na základě kterých lze v mapě zrekonstruovat přesnou trasu spojení. Dalším příkladem volitelné položky jsou informace o tarifních pásmech. My chceme podporovat zpracování libovolných dat v GTFS formátu, tudíž jsme podporu pro tyto volitelné položky neimplementovali. Pokud by se v budoucnu stalo, že se tyto položky v GTFS formátu stanou povinnými, pak implementujeme podporu i pro ně.

Jelikož GTFS formát obsahuje mnoho volitelných položek, pak formát souborů nemůže být pevně daný, jak je tomu u našich formátů (viz Příloha A.1.2). Avšak všechny soubory v GTFS formátu mají stejné uspořádání, jedná se o tabulky podobné formátu CSV. Tabulka lze rozdělit na řádky a sloupce, sloupce jsou oddělené středníkem a řádky jsou oddělené znakem konce řádky. Jeden řádek značí jeden záznam, jeden sloupec představuje určitou hodnotu (například identifikátor, název zastávky či čas odjezdu) v záznamu.

Na prvním řádku souboru jsou vždy uvedeny názvy sloupců, respektive názvy položek, které daný sloupec reprezentuje. Tyto názvy položek vychází ze specifikace formátu GTFS [1]. My, na základě tohoto řádku, vytvoříme slovník, kde každému sloupci přiřadíme jeho pořadové číslo a název položky, kterou reprezentuje. Při zpracovávání řádků následně k jednotlivým položkám přistupujeme přes tento slovník.

O bližší podobě GTFS formátu je možné se dočíst ve specifikaci formátu [1]. My ji považujeme za implementační detail GTFS parseru, a proto se jí nadále nebudeme zabývat.

4.2.3 Předzpracování dat

Předzpracováním dat nazýváme proces, který zpracuje data ze zadaného externího zdroje a který na výstupu vydá data ve formátech TFD a TFB. Konkrétně se jedná o posloupnost následujících kroků:

1. Stažení dat v GTFS formátu z uvedeného externího zdroje. Musí se jednat o archivovaná data ve formátu ZIP.

2. Načtení stažených do operační paměti preprocesoru.

3. Uložení načtených dat do TFD formátu, respektive TFB formátu.

4. Smazání stažených GTFS dat.

Krok předzpracování dat jsme do systému zahrnuli, neboť má řadu výhod:

- V sekci 4.2.2 jsme uvedli výčet položek, které GTFS formátu chybí a které náš systém vyžaduje. Pokud bychom dané položky vytvářeli v okamžik, kdy jádro systému načítá data do paměti, ztratili bychom tím mnoho času. Nejnáročnější operace je vytvoření přestupů, kde časová složitost roste kvadraticky vzhledem k počtu zastávek. Výpočet, kterým určujeme vzdálenost mezi dvěma zastávkami, je časově náročný, neboť ve výpočtu užíváme Pythagorovu větu. Všechny netriviální aritmetické floating-point operace jsou

obecně náročné na procesorový čas, v našem případě se jedná zejména o operaci odmocniny.

Vytváření těchto položek proto delegujeme do procesu předzpracování dat. Doba běhu preprocesoru, od stažení dat po vytvoření nového formátu, záleží na velikosti dat, řádově se jedná o desítky sekund. Načítání předzpracovaných dat jádrem systému poté trvá malé jednotky sekund. Pokud bychom neprováděli předzpracování dat a nechali jádro systému načíst data v GTFS formátu, mohlo by se jednat až o malé desítky sekund, neboť by se tyto položky musely vytvářet při každém spuštění aplikace znovu.

Z předchozího odstavce plyne, že první spuštění aplikace sice bude pomalejší, protože se musí čekat na předzpracování dat, nicméně poté každé další spuštění proběhne citelně rychleji. Preprocesor může být spouštěn i na pozadí, což činí aktualizaci dat snazší.

- Preprocesor můžeme nastavit tak, aby stahoval data z více externích zdrojů. Zpracovávání jednotlivých zdrojů může probíhat paralelně. Poté, co se data ze všech zdrojů paralelně načtou do paměti, tato data preprocesor sekvenčně sloučí. A to pouhým přeindexováním položek. Obecně se předpokládá, že pokud budeme vyžadovat data z více zdrojů, pak se bude jednat o přilehlé oblasti. Tím pádem necháme preprocesor dopočítat přestupy mezi zastávkami z jednotlivých oblastí, aby bylo možné vyhledávat spojení, která projíždí více oblastmi.

- Hlavním produktem předzpracování dat je sice TFD formát, nicméně klientské aplikace vyžadují i základní data, která poskytujeme ve formě TFB formátu. Data v TFB formátu jsou vedlejším produktem předzpracování dat. Pokud bychom vypustili krok předzpracování dat, pak by muselo základní data vytvářet jádro systému, což by opět mohlo zpomalit inicializaci.

- Jádro systému je optimalizované na maximální výkon, jak se dozvíme v sekci 4.3. Formát TFD je uzpůsoben tak, aby jej jádro dokázalo načíst do paměti co nejrychleji. Pokud by došlo ke změně GTFS formátu a my vypustili krok předzpracování dat, pak bychom museli přeprogramovat jádro systému tak, aby reflektovalo dané změny. Obecně platí, že upravovat optimalizovaný kód je náročné. Jelikož máme preprocesor a pevně daný TFD formát, který je zpracováván jádrem, tak při případných změnách GTFS formátu by bylo nutné přeprogramovat pouze parsery v preprocesoru, do jádra systému by nebylo nutné zasahovat.

- Do preprocesoru můžeme snadno přidat podporu pro jiný datový formát, například JDF formát ze sekce 2.1. Pro tento přístup máme připravené rozhraní a v Příloze A.4.1 jsme vytvořili návod, jak dodat preprocesoru parser pro nový formát. Zde pouze uvedeme, že naše podpora pro dodávání nových parserů je navržena tak, aby vývojář parseru nemusel znát specifikaci TFD formátu, ba dokonce aby nemusel ani vědět o jeho existenci.

- Používáním externích zdrojů vždy vzniká riziko, že požadovaná data nebudou mít správný formát. Díky preprocesoru můžeme validitu dat kontrolovat v procesu předzpracovávání dat a s případnými nesrovnalostmi patřičně

nakládat. V každém případě budou výsledkem data ve vždy validním TFD formátu, respektive TFB formátu. Jádro systému, optimalizované na výkon, poté při načítání TFD formátu nemusí provádět žádné kontroly validity dat, neboť se spoléhá na korektnost práce preprocesoru. Pokud by nějaká chyba v TFD formátu měla i přes to vzniknout, pak se bude pravděpodobně jednat o chybu fatální, což bude mít za následek vyhození výjimky.

Veškerý přínos předzpracovávání dat by se dal shrnout do několika vět. Jádro může být optimalizované na maximální výkon, neboť spoléhá na korektnost TFD formátu. Veškeré záležitosti s daty řeší preprocesor a pokud by mělo dojít k jakékoliv změně dat, bude nutné přeprogramovat pouze parsery v preprocesoru, zatímco jádro bude moci zůstat netknuté.

4.3 Jádro systému

Jádro systému musí být schopné následujících činností:

- Musí při startu aplikace načíst předzpracovaná data do operační paměti a po dobu běhu je uchovávat v paměti. Doba načítání dat závisí na jejich velikosti, jedná se řádově o jednotky sekund. Kdyby se měla data načítat znovu s každým požadavkem, čas na vykonání požadavku by mohl vzrůst o stovky až tisíce procent. Ze stejného důvodu nebylo zvoleno řešení s databází.

- Musí být schopné vykonat požadavek na nalezení spojení s využitím načtených dat. Požadavek musí obsahovat identifikátor výchozí a cílové stanice a nejdřívější čas odjezdu. Dále musí obsahovat buď počet spojení, nebo maximální čas příjezdu. Mezi volitelné položky se řadí maximální počet přestupů, koeficient rychlosti chůze nebo volba dopravních prostředků. Vyhledáváním spojení se zabývá celá kapitola 3.

- Musí být schopné vykonat požadavek na nalezení odjezdů s využitím načtených dat. Klientské aplikace rozlišují mezi požadavkem na vyhledání odjezdů ze stanice a požadavkem na vyhledání informací o lince. V jádru tyto požadavky splývají, požadavek na vyhledání informací o lince lze převést na požadavek na vyhledání odjezdů z výchozích stanic dané linky, přičemž se omezíme pouze na danou linku. Požadavek musí obsahovat nejdřívější čas odjezdu, dále buď počet odjezdů, nebo maximální čas odjezdu. Mezi další parametry požadavku se řadí identifikátor stanice, respektive identifikátor linky. Pro požadavek na vyhledání odjezdů ze stanice volitelně identifikátor linky, díky němuž se lze omezit na odjezd jedné konkrétní linky z dané stanice.

Životní cyklus jednoho požadavku lze rozdělit na tři části. První částí je zaznamenání daného požadavku a zpracování dat z jeho těla. Předpokládá se, že korektnost dat v požadavku je zkontrolována na straně klienta v uživatelském rozhraní. Pokud dojde k chybě při zpracování požadavku, například pokud je identifikátor stanice mimo rozsah kolekce, dojde k výjimce, kterou musí řešit nadřazené vrstvy. Druhou částí je samotné zpracování požadavku, v této části nemůže dojít k výjimce. Po zpracování požadavku se nalezené výsledky dočasně uloží. Třetí částí je požadavek na zobrazení výsledků. Požadavek na zobrazení

výsledků lze vykonávat opakovaně. Po ukončení spojení jsou nalezené výsledky v jádru zdestruovány.

V jádru je kladen požadavek na maximální výkon, za účelem co nejrychlejší exekuce požadavků. Snažíme se optimalizovat i dobu načítání dat do paměti, proto se při načítání dat nekontroluje jejich validita. Předpokládá se, že předzpracovaná data v TFD formátu jsou vždy validní. Pokud by mělo dojít k chybě ve formátu načítaných dat, vznikne výjimka, kterou musí řešit nadřazené vrstvy.

Pro zajištění maximálního výkonu je dále vhodné, aby se veškeré optimalizace jádra provedly již při jeho překladu. Není vhodné nechávat optimalizace až na runtime, jelikož za runtimu není možné dosáhnout tak masivních optimalizací jako při překladu. Pro tento účel je vhodné napsat jádro v nativním kódu a propojit ho s managovaným kódem pomocí interoperability, která je detailně popsaná v Příloze A.4.2.

4.4 Aktualizace

Rozlišujeme dva základní druhy aktualizací, aktualizaci na serveru a aktualizaci v klientovi.

Aktualizace na serveru

Tento druh aktualizace synchronizuje data uložená na serveru s daty z externích zdrojů. V sekci 4.2.1 jsme uvedli, že formát TFD obsahuje položku, jež udává datum expirace dat. Tato položka se využívá při tomto typu aktualizace. Aktualizaci na serveru definujeme jako posloupnost následujících kroků:

1. Předzpracování dat z externích zdrojů.

2. Načtení dat do paměti jádrem systému.

3. Přiřazení adresy nových dat do proměnné typu ukazatel. Data, na která proměnná ukazuje, využívá server k vyhledávání v jízdních řádech.

4. Zdestruování starých dat, pokud existují.

Aktualizaci na serveru je možné provést třemi způsoby:

- Start aplikace. Při startu aplikace dojde ke kontrole, jestli jsou data aktuální. Pokud jsou již po expiraci, proces aktualizace probíhá na popředí.

- Automatická aktualizace. Aktualizace se automaticky spustí vždy, jakmile dojde k expiraci dat. Proces automatické aktualizace probíhá na pozadí, během automatické aktualizace pracuje server nad neaktuálními daty.

- Vynucení aktualizace dat. Jak vynutit aktualizaci popíšeme v sekci 5.1, kde se budeme zabývat serverem detailněji.

Aktualizace v klientovi

Tento druh aktualizace synchronizuje data uložená v klientovi s daty uloženými na serveru. Klient vyžaduje data v TFB formátu, server je má načtená ve své paměti a poskytuje je klientům. Aktualizace v klientovi se provádí při spuštění klientské aplikace a při odeslání libovolného požadavku na server.

Data v TFB formátu mají přiřazenou verzi. Verzi přiřazuje datům preprocesor při vytváření dat. Naše implementace považuje verzi za posloupnost deseti náhodných znaků. Na základě porovnání verzí dat u klienta a dat na serveru se data synchronizují. Synchronizace probíhá v těchto krocích:

1. Klient pošle na server požadavek, v jehož těle je uvedena verze klientských dat.

2. Server požadavek zpracuje, porovná verzi dat z požadavku a verzi dat načtených v paměti.

3. Pokud se verze shodují, server pošle klientovi odpověď s prázdným tělem. Pokud se verze liší, do odpovědi se vloží nová data a jejich verze.

4. Klient přijme odpověď ze serveru. Pokud je odpověď neprázdná, zpracuje data z těla odpovědi tak, že je načte do paměti a rovněž je uloží na disk pro další spuštění aplikace. Stejným způsobem aktualizuje verzi dat.

Reindexace položek a mapování mezi verzemi

Na závěr této sekce zmiňme jeden neřešený problém s aktualizacemi, který by bylo vhodné v budoucnu vyřešit. Pro komunikaci mezi klientem a serverem používáme identifikátory dané TFB formátem. Tedy pro označení stanice nepoužíváme její název, nýbrž její identifikátor. Stejně tak neoznačujeme zastávku pomocí její GPS souřadnic, respektive linku pomocí jejího označení. Pokud by vlivem aktualizace dat došlo k reindexaci těchto identifikátorů, popřípadě jejich úplně změně, může to na klientské straně způsobit problémy. Například by uživatel přišel o své oblíbené položky, respektive v oblíbených položkách by se místo nich objevily položky, které odpovídají novým identifikátorům. Momentálně přenecháváme tento problém na zdroji dat, tedy předpokládáme, že s každou aktualizací dat nedojde k přeindexování, neboli že záznamy o zastávkách a linkách v externím zdroji dat se budou vyskytovat ve stejném pořadí jako v předešlé verzi, a že případné nové položky se budou přidávat na konce souborů.

Jedním z možných řešení by bylo přidat do předzpracování dat novou vrstvu, která by tvořila mapování mezi starými a novými identifikátory. Toto mapování bychom pak odesílali klientovi s každou novou verzí dat, načež by si klient opravil identifikátory, aby odpovídaly nové verzi. Jiným možným řešením by bylo nevytvářet mapování, ale pokoušet se v rámci předzpracování dat přiřazovat položkám v nové verzi dat identifikátory ze staré verze dat, případné nové záznamy vkládat na konce souborů. Obou přístupů lze docílit pomocí porovnávání názvů stanic, GPS souřadnic zastávek a označení linek ve staré a nové verzi.

4.5 Klientská knihovna

Počítačová a mobilní aplikace mají mnoho společného, za tímto účelem jsme vytvořili klientskou knihovnu sjednocující společné části. V této sekci popíšeme nejdůležitější problémy na straně klienta, které bylo třeba řešit.

4.5.1 Lokalizace aplikací

Jedním z cílů práce bylo umožnit lokalizovat aplikace do jazyka, který si uživatel zvolí. Spousta aplikací využívá pevně dané lokalizační soubory zabudované do zdrojů aplikace. Další možností je využít služeb Google Translation API[5], kde se do klíčové metody překladače uvede kód jazyka dle normy ISO639-1 a překládaný text v angličtině, výsledkem je text v zadaném jazyce.

Pevně zabudovanými lokalizacemi bychom uživatele značně omezili a strojový překlad nemusí být pro naše účely vhodný, proto jsme se rozhodli zvolit jinou možnost. Náš návrh umožňuje každému uživateli vyrobit si vlastní lokalizační soubor a dodat ho aplikaci „zvenčí“ prostřednictvím XML souboru. Klientské aplikace očekávají lokalizační soubory v připravené složce, následně se ze všech relevantních souborů v této složce vytvoří seznam, ze kterého lze vybrat požadovanou lokalizaci. Angličtina se považuje za výchozí jazyk, ta je již součástí distribuce klientské knihovny. Při vytváření vlastní lokalizace je vhodné se řídit strukturou již existujícího lokalizačního souboru, například lokalizačním souborem pro češtinu, který je součástí Přílohy A.5.

4.5.2 Metody zobrazování výsledků

V úvodu jsme si jako jeden z cílů stanovili, aby klientské aplikace poskytovaly uživateli přívětivé uživatelské rozhraní s přehledným zobrazováním výsledků. V této sekci si ukážeme univerzální přístup k řešení tohoto problému, který jsme zvolili.

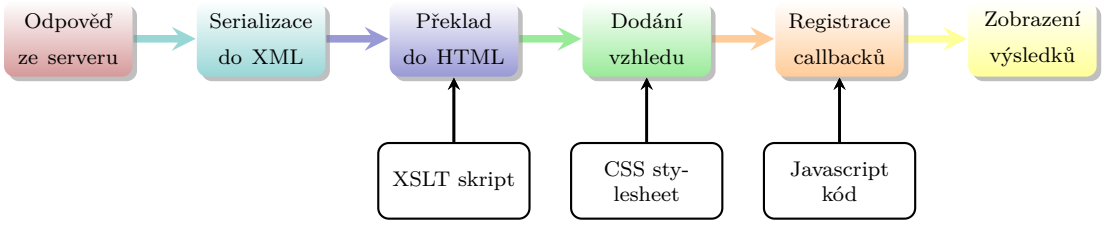
Spousty dnešních mobilních aplikací fungují jako webové aplikace. Vývojář do své aplikace implementuje pouze webový prohlížeč, ve kterém se následně vykonává veškerá akce. My jsme se v obou našich klientských aplikacích rozhodli pro podobný přístup, s tím rozdílem, že webový prohlížeč používáme pouze pro zobrazování výsledků.

Webový prohlížeč jako prostředník

Pokud přijme klient odpověď ze serveru, jedná se o nějaký objekt. Takový objekt má jasně dané rozhraní a je možné ho serializovat do XML souboru. Struktura takového XML souboru je nám dobře známa (viz Příloha A.2.2), tudíž jsme schopni vytvořit XSLT skript, který z daného XML souboru vytvoří HTML kód. To má za následek vytvoření statické webové stránky s výsledky ze serveru. V dalším kroku se do daného HTML kódu vloží odkaz na CSS soubor, jenž dané webové stránce dodá příjemný vzhled. Ještě, než se stránka zobrazí uživateli, spustí klient v dané webové stránce javascriptový kód, který zaregistruje callbacky zpět do managovaného kódu. Tento krok je důležitý zejména ze dvou důvodů. Zajistí

[5] `https://cloud.google.com/translate/docs/`

se tím funkčnost tlačítek ve webové stránce, ať už tlačítka pro detail nalezeného spojení či zobrazení mapy. Dále je možné danou webovou stránku lokalizovat pomocí přístupu, který jsme popsali v sekci 4.5.1. Vizualizace všech akcí popsaných v tomto odstavci je k nahlédnutí v Obrázku 4.2. Přesný popis interoperability Javascriptu s managovaným kódem je k dispozici v Příloze A.4.3.



Obrázek 4.2: Popis jednotlivých kroků od přijetí odpovědi ze serveru až po zobrazení výsledků uživateli.

Za hlavní a jedinou nevýhodu tohoto přístupu považujeme časté volání managovaného kódu z Javascriptu. Jelikož se jedná pouze o zobrazování výsledků uživateli, které se děje jednou za čas, tento fakt nemá vliv na rychlost celého systému. Tudíž jsme se rozhodli tuto nevýhodu přejít, neboť výhod je celá řada:

- Všechny XSLT, CSS a Javascript soubory dodáváme aplikaci „zvenčí“. V počítačové aplikaci k těmto souborům přistupujeme pomocí souborového systému, v mobilní aplikaci jsou tyto soubory zahrnuté v balíčku s aplikací. Pokud by naše klientské aplikace dostal do rukou zkušený uživatel a nelíbil se mu způsob zobrazování výsledků, jaký jsme zvolili, může si dané XSLT a CSS soubory upravit dle své libosti. Pak se mu budou výsledky zobrazovat tak, jak si vysnil.

- Webový prohlížeč, který jsme použili v počítačové aplikaci, má podporu pro tisk aktuální webové stránky. Díky našemu přístupu můžeme snadno podporovat tisk výsledků, aniž bychom museli psát další kód.

- Mezi zobrazování výsledků řadíme i zobrazování informací o výlukách a mimořádných událostech v dopravě. Tyto informace čerpáme z RSS zdrojů. Naše řešení je uzpůsobeno pro zobrazení informací tak, jak je poskytuje PID [2]. Při změně datové sady bude potřeba změnit i zdroje těchto informací. Pak bude nutné upravit XSLT soubory pro výluky a mimořádnosti tak, aby reflektovaly dané RSS zdroje. Jiný zásah do aplikace nebude nutný.

- Oblíbené položky se ukládají do XML souborů, jak se dozvíme v sekci 4.5.3. Pro zobrazení oblíbeného spojení stačí načíst daný soubor do paměti a aplikovat na něj stejný postup.

- Díky CSS3 můžeme vytvořit responzivní layout pro zobrazování výsledků, aniž bychom museli hýbat s uspořádáním jednotlivých elementů.

- V budoucnosti by se mohlo stát, že server reimplementujeme pomocí Web API. Požadavky na server bychom zasílali pomocí HTTP GET metody, načež by nám server vrátil stránku s výsledky jako odpověď. Pokud by

daná odpověď byla v XML formátu, pak bychom pouze přeskočili krok serializace do XML a další zásah do zobrazování výsledků by nebyl nutný. V případě JSON formátu odpovědi bychom pouze JSON formát převedli na XML formát, popřípadě upravili způsob překladu do HTML kódu.

- Pokud bychom chtěli vytvořit, vedle počítačové a mobilní aplikace, i webovou aplikaci, mohli bychom použít stejný přístup a využít již existující skripty.

Podpora pro mapu

Mapu jsme se rozhodli do klientských aplikací zakomponovat pro dva účely:

- Umožnit uživateli zobrazit si mapu dopravní sítě, kde budou vyobrazené jednotlivé zastávky, přičemž po kliknutí na danou zastávku se uživateli ukáží nejdřívější odjezdy z dané zastávky. Toto může přijít vhod zejména v situaci, kdy se uživatel vyskytne v nějaké velké stanici, kterou nezná, v rámci které se nachází mnoho zastávek. S našimi aplikacemi může jednoduše zjistit, z jaké zastávky odjíždí jeho spoj, a vydat se na určité místo.

- Umožnit uživateli zobrazit si trasu nalezeného spojení, kde budou vyobrazené jednotlivé zastávky. Uživatel toto využije zejména při přestupech na jinou linku. Rovněž je zamýšleno, že po kliknutí na nějakou zastávku ve spojení se zobrazí nejdřívější odjezdy ze stanice, pod kterou spadá, přičemž budeme uvažovat pouze odjezdy po času příjezdu do dané zastávky v rámci nalezeného spojení. Tím uživateli poskytneme informace o spojích, na které může bezprostředně přestoupit.

Za těmito účely jsme zvolili Google Maps API [6]. Další přípustnou variantou jsou Mapy.cz[6], jež rovněž poskytují kvalitně zdokumentované API. My dali přednost službám od společnosti Google, neboť chceme, aby aplikace fungovaly pro celý svět a předpokládáme, že zvolená mapa bude uživatelům na druhé straně zeměkoule poskytovat kvalitnější a aktuálnější údaje. Mapa portálu Mapy.cz se pro Českou republiku aktualizuje jednou týdně a pro zbytek světa jednou měsíčně.[7] Zatímco u mapy portálu Google závisí interval aktualizace na osídlení a progresi daného regionu, kde hustě osídlené oblasti se aktualizují jednou týdně a méně osídlené oblasti několikrát do měsíce.[8] Dalším důvodem pro zvolení Google Maps je jejich snazší zakomponování do mobilní aplikace, neboť implementace Google Maps již existují v podobě pluginů.

4.5.3 Oblíbené položky

Mezi cíle práce jsme rovněž zahrnuli možnost zvolit si oblíbené položky, které budou automaticky synchronizovány se serverem a bude je možné vyhledávat i bez přístupu k internetu. Na konci sekce 4.1 jsme uvedli motivaci pro tuto službu. Nyní si popíšeme náš přístup k řešení problému a uvedeme jiné možné alternativy.

[6] `https://api.mapy.cz/`

[7] `https://napoveda.seznam.cz/cz/mapy/mapove-podklady/`

[8] `https://www.techjunkie.com/how-often-google-maps/`

Pokud chceme implementovat podporu pro automatickou synchronizaci, jsou dva možné přístupy, jak tak lze učinit:

- Implementovat logiku pro synchronizaci na serveru. Klient by odeslal na server požadavek na přidání spojení mezi oblíbené položky, server by požadavek zaznamenal, odeslal klientovi požadovaná data a poznamenal si k danému klientovi uloženou oblíbenou položku a termín další automatické synchronizace. Tato automatické synchronizace by šla implementovat:

  - Pomocí push událostí nebo jiné podobné služby. Server by se pokusil klientovi odeslat aktualizované oblíbené položky vždy, jakmile by nastal termín další automatické synchronizace, bez nutnosti vznesení požadavku ze strany klienta.
  - Jakmile by klient odeslal libovolný požadavek na server. S každým takovým požadavkem by se zkontrolovalo, zdali je nutné aktualizovat oblíbené položky v klientovi, a pokud ano, položky by se aktualizovaly.

  Pokud by klient danou oblíbenou položku odstranil, na server by se odeslal požadavek na odstranění oblíbené položky, což by mělo za následek odstranění patřičného záznamu na straně serveru.

- Implementovat logiku pro synchronizaci v klientovi. Termíny expirace oblíbených položek by byly uložené na straně klienta, v případě expirace by se odeslal požadavek na server, který by klientovi odeslal nová data. Odstranění oblíbené položky by bylo možné v tomto případě řešit pouhým odstraněním souboru na disku.

My jsme se rozhodli implementovat logiku pro synchronizaci v klientovi, protože:

- Zachováme tím jednoduchý model serveru. Kdybychom implementovali logiku pro synchronizaci na serveru, vznikla by tím další režie, která by měla za následek celkové zpomalení běhu serveru.

- Logika je implementována v klientovi, tudíž sám klient se může rozhodnout, kdy bude položky synchronizovat. Toto může přijít vhod zejména v situacích, kdy chceme data synchronizovat jen za určitých podmínek, například pouze při Wi-Fi připojení.

Dále jsme řešili problém, jakým způsobem budou oblíbené položky v klientovi ukládané. Při řešení tohoto problému jsme uvažovali tyto varianty:

- Implementovat do jádra systému vrstvu, která z dat vybere jen to nutné, aby se dalo v jakoukoliv denní dobu vyhledat dané spojení, popřípadě odjezdy. Tato část dat by se následně poslala klientovi, ten by si je uložil na disk a vyhledávání by probíhalo u klienta offline.

- Vyhledat všechna spojení, popřípadě odjezdy, pro dané časové rozmezí, například jeden den. Tyto výsledky by se následně poslaly klientovi, který by si je uložil a pomocí vhodného dotazovacího jazyku z nich sestavoval odpověď na uživatelův požadavek.

My jsme se rozhodli pro druhou variantu, neboť nevyžaduje žádný zásah do jádra systému. Další výhodou může být jednoduchost zobrazování nalezených výsledků, neboť můžeme použít postup popsaný v sekci 4.5.2. Nevýhodou může být vyšší četnost aktualizací.

Odpověď se uloží jako celek do XML souboru, v závislosti na počtu nalezených výsledků se může jednat o stovky kilobyte až malé jednotky megabyte. Za účelem snížení velikostí těchto souborů by bylo do budoucna vhodné ukládat tyto soubory v JSON formátu, jelikož soubory v JSON formátu jsou typicky menší než soubory v XML formátu.

Datum a čas další synchronizace udává položka zhotovení požadavku. To je také jediný důvod, proč odpovědi ze serveru tuto položku obsahují (viz struktura XML dokumentů v Příloze A.2.2). K času zhotovení požadavku se přičte časový úsek, jak dlouho mají oblíbené položky být aktuální, například jeden den. Tím získáme datum a čas expirace oblíbené položky.

5. Implementace

V kapitole 3 a 4 jsme nalezli řešení pro cíle ze sekce 1.1. V této kapitole se podíváme detailněji na samotnou implementaci.

5.1 Serverová aplikace

Serverová aplikace vyžaduje pro svůj běh verzi platformy .NET alespoň 4.6.1. Aplikaci lze spustit pouze pod operačním systémem Windows, neboť jádro aplikace je psáno v nativním kódu.

Při návrhu serveru bylo našim cílem zachovat co nejjednodušší model serveru a pokusit se delegovat veškerou režii do klientů.

5.1.1 Moduly serveru

Mezi moduly serveru řadíme již zmíněné nativní jádro systému, které obstarává vyhledávání v jízdních řádech. Dále zde nalezneme preprocesor dat, který zpracovává data z uvedených externích zdrojů. V rámci serveru jsme dále museli řešit způsob přenosu dat po síti, způsob zajištění automatických aktualizací či způsob vyřizování požadavků a jak spolu budou jednotlivé moduly komunikovat. O způsobech řešení zmíněných problémů budou pojednávat následující odstavce.

5.1.2 Služby serveru

Server poskytuje tři základní služby, přičemž každá služba využívá svůj vlastní port:

- Služba pro vyhledávání spojení. Server pro tuto službu naslouchá na portu 27000.
- Služba pro vyhledávání odjezdů. Server pro tuto službu naslouchá na portu 27001.
- Služba pro poskytování dat v TFB formátu klientům. Server pro tuto službu naslouchá na portu 27002.

Komunikaci mezi klientem a serverem zajišťuje transportní protokol TCP z rodiny protokolů TCP/IP. Alternativou k protokolu TCP je protokol UDP, nicméně ten je nespolehlivý a tudíž pro náš účel nevhodný. Po síti přenášíme binárně serializované objekty, podporu pro binární serializaci obstarává platforma .NET.

Server jsme se rozhodli rozdělit na tři služby, aby bylo možné určitou část serveru vypnout. O konfiguraci serveru se budeme bavit v sekci 5.1.3, zde pouze zmíníme, že každé službě lze změnit port, na němž bude server naslouchat. Port je z definice přirozené číslo, my jsme zavedli možnost přiřadit portu i zápornou hodnotu. Pokud se portu dané služby přiřadí libovolné záporné číslo, nejedná se o chybu. V takovém případě se server pro danou službu nespustí a server nebude moci vyřizovat klientovy požadavky, které obstarává daná služba.

Největší požadavky na parametry procesoru klade služba pro vyhledávání spojení. Zbylé dvě služby lze efektivně provozovat i na méně výkonném stroji. Jelikož jsme schopni určité služby vypnout, tak je i možné rozdělit běh serveru na více strojů. Službu pro vyhledávání spojení bychom mohli spustit na velmi výkonném stroji, zatímco zbylé dvě služby na stroji s průměrnou výbavou. Následně bychom na lokální síti vhodně nastavili port forwarding, aby byl server „zvenčí“ stále přístupný pouze pod jednou IP adresou. Tímto přístupem bychom mohli docílit zrychlení vyřizování požadavků, kdyby jich na server chodilo velké množství.

Každá služba má přiřazené jedno vlákno a je implementována jako nekonečná smyčka, ve které server naslouchá na daném portu a pasivně čeká na požadavky od klientů. Jakmile je nějaký požadavek zaznamenán, předá se k řešení na threadpool a na výsledek požadavku se nečeká, odpověď se automaticky odešle zpět klientovi. Tímto způsobem řešení jsme splnili jeden z cílů, kde jsme si předsevzali, že server bude vykonávat požadavky asynchronně.

5.1.3 Konfigurace serveru

Server lze konfigurovat z konfiguračního souboru, ten musí být ve složce se spustitelným souborem serveru a musí mít název `settings.xml`. Pokud pří spuštění serveru není soubor nalezen, aplikace se uživatele zeptá na URL zdroje dat. Dále se nastavení serveru přiřadí výchozí hodnoty a server je spuštěn.

Konfigurační soubor má strukturu XML souboru (viz Příloha A.2.1). Konfigurační soubor lze rozdělit na tři části:

- Část potřebná pro běh serveru. Tato část obsahuje čísla portů služeb tak, jak bylo popsáno v sekci 5.1.2.

- Část potřebná pro předzpracování dat. Tato část obsahuje koeficienty přestupů, rychlost přestupů a maximální dobu přestupů tak, jak bylo popsáno v sekci 4.2.1.

- Část potřebná pro stažení dat. Tato část obsahuje URL adresy ke všem zdrojům dat, které má server zpracovat.

5.1.4 Logování akcí

Za účelem monitorování činnosti serveru bylo implementováno logování akcí. Veškerá činnost serveru se vypisuje na konzoli a do textového souboru `.log`, který se nachází ve složce se serverem. Podpora pro logování je implementována pomocí eventů, tudíž lze přidat i jiný způsob logování pouhým zaregistrováním příslušného callbacku.

5.1.5 Běh serveru

V této sekci popíšeme všechny akce, které server při svém běhu vykonává.

Spuštění serveru

Posloupnost kroků po spuštění je následující:

1. Načtení konfiguračního souboru, existuje-li.

2. Iniciální aktualizace dat. Server zkontroluje, zdali jsou data aktuální. Pokud ano, pokračuje dalším krokem. Pokud ne, data aktualizuje na popředí. Kroky aktualizace byly popsány v sekci 4.4. Data ve formátu TFD se ukládají do složky `data`, zatímco data ve formátu TFB se ukládají do složky `basic`.

3. Spuštění serveru jako takového, ten se postará o spuštění jednotlivých služeb a o spuštění jádra, které načte data do paměti.

4. Start vlákna, které obstarává automatickou aktualizaci. Vlákno načte do paměti datum expirace dat a uspí se, dokud tento okamžik nenastane. Poté aktualizuje data na pozadí a tento proces opakuje.

5. Server naslouchá na portech, které byly uvedeny v konfiguračním souboru.

Pokud byly všechny kroky úspěšné, server by měl být ve stavu, jak je k vidění na Obrázku 5.1. Pokud nastane chyba při inicializaci, server danou chybu zaloguje a ukončí svou činnost. Součástí každé chyby bývá text výjimky vyvolané z .NET, který je lokalizován do jazyka operačního systému.

```
26.03.2019 10:13:09 - Preprocessor: Trying to download data from URL http://data.pid.cz/PID_GTFS.zip.
26.03.2019 10:13:16 - Preprocessor: Data from URL http://data.pid.cz/PID_GTFS.zip downloaded successfully.
26.03.2019 10:13:16 - Preprocessor: Trying to parse data downloaded from http://data.pid.cz/PID_GTFS.zip.
26.03.2019 10:13:39 - Preprocessor: The data downloaded from http://data.pid.cz/PID_GTFS.zip parsed successfully.
26.03.2019 10:13:39 - Preprocessor: Trying to merge the data.
26.03.2019 10:13:39 - Preprocessor: Data merged successfully.
26.03.2019 10:13:39 - Preprocessor: Trying to create new data feed.
26.03.2019 10:13:41 - Preprocessor: Data feed created successfully.
26.03.2019 10:13:43 - The server has started.
26.03.2019 10:13:43 - Auto update: Another auto update is scheduled to be at 29.03.2019 0:00:00.
26.03.2019 10:13:43 - Server RouterServer listening at 0.0.0.0:27000.
26.03.2019 10:13:43 - Server DepartureBoardServer listening at 0.0.0.0:27001.
26.03.2019 10:13:43 - Server BasicDataFeedServer listening at 0.0.0.0:27002.
```

Obrázek 5.1: Úspěšný start serveru.

Ovládání serveru

Aplikace poskytuje základní příkazy, jimiž lze ovládat server. Jedná se o tyto příkazy:

- Příkaz `HELP` zobrazí nápovědu obsahující tento seznam příkazů.
- Příkaz `ABORT` vypne server.
- Příkaz `FORCEUPDATE` vynutí aktualizaci dat.
- Příkaz `RESTART` restartuje server.

Přijetí požadavku

Každému požadavku na vyhledání spojení, respektive odjezdů předchází požadavek na aktualizaci klientských dat, jak bylo popsáno v sekci 4.4.

Pokud zadá uživatel v klientské aplikaci požadavek na vyhledání, bez újmy na obecnosti, spojení, z pohledu serveru se jedná o následující proces:

1. Přijetí požadavku na aktualizaci klientských dat.
2. Zpracování požadavku, porovnání verzí klientských dat, zhotovení odpovědi.
3. Odeslání odpovědi.
4. Přijetí požadavku na vyhledání spojení.
5. Zpracování požadavku, vyhledání spojení, zhotovení odpovědi.
6. Odeslání odpovědi.

Příklad úspěšného vyřízení požadavku je k nahlédnutí v Obrázku 5.2. Pokud dojde k chybě při zpracování požadavku, server danou chybu zaloguje a ve vyřizování požadavku dále nepokračuje. Příklad nedokončeného požadavku je k nahlédnutí v Obrázku 5.3. Součástí každého požadavku je zalogování základních dat z jeho těla, aby bylo možné, v případě chyby při vyřizování, dohledat příčinu.

```
26.03.2019 10:47:48 - Connection request from 127.0.0.1.
26.03.2019 10:47:48 - Received data feed request from 127.0.0.1. Data: Version: 5ESKHYRNVW.
26.03.2019 10:47:48 - Data feed response to 127.0.0.1 was successfully send.
26.03.2019 10:47:48 - Connection request from 127.0.0.1.
26.03.2019 10:47:48 - Received router request from 127.0.0.1. Data: Source station ID: 1690. Target station ID: 2035. Max tran
sfers: 5. Count: 5.
26.03.2019 10:47:48 - Router response to 127.0.0.1 was successfully sent.
```

Obrázek 5.2: Úspěšně vyřízený požadavek na vyhledání spojení.

```
26.03.2019 10:49:11 - Connection request from 127.0.0.1.
26.03.2019 10:49:11 - Received data feed request from 127.0.0.1. Data: Version: 5ESKHYRNVW.
26.03.2019 10:49:11 - Data feed response to 127.0.0.1 was successfully send.
26.03.2019 10:49:11 - Connection request from 127.0.0.1.
26.03.2019 10:49:11 - Received router request from 127.0.0.1. Data: Source station ID: 624. Target station ID: 415. Max transf
ers: 100. Count: -1.
26.03.2019 10:49:28 - Router request from 127.0.0.1 could not be processed. Exception: Nelze zapsat data do přenosového připoj
ení: Stávající připojení bylo vynuceně ukončeno vzdáleným hostitelem.
```

Obrázek 5.3: Nedokončený požadavek na vyhledání spojení z důvodu ukončení klientské aplikace před přijetím odpovědi.

Ukončení běhu serveru

Aplikaci lze bezpečně vypnout použitím libovolného uživatelského rozhraní, lze použít systémové rozhraní nebo příkaz `ABORT`.

5.2 Klientské aplikace

Počítačová aplikace vyžaduje pro svůj běh verzi platformy .NET alespoň 4.6.1. Aplikaci lze spustit pouze pod operačním systémem Windows, protože obsahuje, kvůli podpoře offline módu, i jádro aplikace, které je psáno v nativním kódu.

Mobilní aplikace je určena pro Android verze 8.1 (API Level 27 Oreo), minimální podporovaná verze je Android verze 5.0 (API Level 21 Lollipop). Mobilní aplikace vyžaduje pro svůj běh přístup k poloze zařízení a souborovému systému, bez těchto oprávnění nebude aplikace fungovat korektně. O možnostech rozšíření mobilní aplikace pro běh na jiné platformě pojednává Příloha A.4.4.

5.2.1 Back-end klientských aplikací

Obě klientské aplikace využívají služeb klientské knihovny, která byla popsána v sekci 4.5. Způsoby vyřizování požadavků, komunikace se serverem či načítání externích zdrojů jsou pro obě aplikace totožné, aplikace se až na výjimky liší pouze svým uživatelským rozhraním.

V této sekci se zaměříme na back-end klientských aplikací, zatímco na front-end aplikací se zaměříme v sekci 5.2.2 pro počítačovou aplikací, respektive v sekci 5.2.3 pro mobilní aplikaci.

Za složku se zdroji aplikace budeme pro počítačovou aplikaci označovat složku, kde se vyskytuje spustitelný soubor aplikace, a pro mobilní aplikaci složku `assets` v balíčku s aplikací.

Za složku s daty aplikace budeme pro počítačovou aplikaci označovat složku, kde se vyskytuje spustitelný soubor aplikace, a pro mobilní aplikaci složku v umístění s absolutní cestou `/storage/emulated/0/timetables` v souborovém systému zařízení.

Konfigurace aplikací

Aplikace se primárně konfigurují pomocí konfiguračního souboru. Soubor musí být ve složce s daty aplikace a musí mít název `settings.xml`. Konfigurační soubor má strukturu XML souboru (viz Příloha A.2.1). Alternativně je možné upravovat konfigurační soubor pomocí konfigurátoru, o kterém pojednává Příloha A.3.

Pokud počítačová aplikace nemůže načíst konfigurační soubor, tedy neexistuje nebo je vadný, pak se při startu aplikace spustí přímočarý dialog, který vytvoří konfigurační soubor na základě zadaných hodnot.

V mobilní aplikaci složka s daty po instalaci neexistuje, ta se vytvoří až při prvním spuštění aplikace. V tento okamžik se rovněž do složky s daty zkopíruje předpřipravený konfigurační soubor ze složky se zdroji aplikace.

Výchozí hodnota IP adresy serveru je, v našem případě, nastavena na hodnotu `undefined`. Pokud se tato hodnota vyskytuje v konfiguračním souboru aplikace, pak se aplikace při svém startu zeptá uživatele na IP adresu serveru a zadanou adresu zapíše do konfiguračního souboru pro další spuštění aplikace.

Pokud bychom distribuovali aplikaci například pomocí služby Google Play, pak bychom pravděpodobně očekávali, že někde v síti existuje server, ke kterému

se aplikace bude připojovat. Pak by bylo nutné ve výchozím konfiguračním souboru nahradit hodnotu `undefined` IP adresou daného serveru, aby se při prvním spuštění aplikace nezobrazil dialog pro zadání IP adresy.

V dalších sekcích se dozvíme, že uživatelské rozhraní klientských aplikací obsahuje položku s nastavením, odkud lze měnit uživatelské nastavení aplikace. Uživatelské nastavení má rovněž oporu v konfiguračním souboru. Konfigurační soubor s nastavením se vždy při startu klientské aplikace načte do paměti a při jakékoliv změně uživatelského nastavení se hodnoty v paměti uloží do konfiguračního souboru na disk, aby byly k dispozici i při dalších spuštěních aplikace.

Při implementaci jsme se rozhodli, že položky konfiguračního souboru spojené s fungováním aplikace[1] nebudeme do rozhraní s uživatelským nastavením zahrnovat. Tyto položky jsou modifikovatelné pouze z konfiguračního souboru. Pokud by bylo možné modifikovat tyto položky přímo z aplikace, pak by mohl nezkušený uživatel nedopatřením celou aplikaci rozbít. Zkušený uživatel dokáže upravovat tyto položky z konfiguračního souboru, o kterém nezkušený uživatel nemusí vůbec vědět.

Implementace klientské knihovny

Klientská knihovna obsahuje následující položky:

- Podporu pro lokalizaci aplikací. V sekci 4.5.1 jsme uvedli, že aplikace očekávají lokalizační soubory v určité složce. Tato složka se nachází ve složce se zdroji aplikace a nese název `loc`. Struktura lokalizačního XML souboru kopíruje rozhraní třídy pro lokalizaci, načtení dané lokalizace lze provést pouhou deserializací XML dokumentu do instance třídy pro lokalizaci. Pokud při deserializaci nastane jakákoliv chyba, například lokalizační soubor nemá správný formát, pak se automaticky nastaví angličtina jakožto výchozí jazyk.

- Podporu pro zobrazování výsledků. Tuto podporu obstarává metoda, mezi jejíž parametry se řadí objekt určený k serializaci, cesta ke XSLT skriptu, cesta k CSS stylesheetu a cesta k javascriptovému kódu. Metoda vrátí vytvořený HTML kód, který se předá uživatelskému rozhraní k zobrazení výsledků.

  Relativní cesty k daným skriptům jsou uvedeny v kódu aplikace. Obecně platí, že všechny skripty se vyskytují ve složce se zdroji aplikace, přičemž jejich další umístění dále závisí na jejich typu:

  - Skripty XSLT musí být v podsložce `xslt`.
  - Stylesheety CSS musí být v podsložce `css`.
  - Soubory s javascriptovým kódem musí být v podsložce `js`.

- Podporu pro oblíbené položky. Tato podpora zahrnuje ukládání nalezených výsledků, načítání uložených výsledků, kontrolu aktuálnosti uložených výsledků a filtrování výsledků na základě uživatelova požadavku.

[1] Řadíme sem zejména adresu serveru, čísla portů a URL adresy RSS zdrojů.

Kontrola aktuálnosti spočívá v porovnání data vytvoření oblíbené položky s předpokládaným datem expirace. Současné nastavení aplikací počítá s ukládáním oblíbených položek na 24 hodin dopředu, přičemž 6 hodin před expirací položek se bude aplikace pokoušet danou oblíbenou položku aktualizovat. To znamená, že aplikace nebude usilovat o aktualizaci oblíbené položky prvních 18 hodin od jejího vytvoření, poté se ji pokusí aktualizovat hned, jakmile to bude možné. Tyto dvě konstanty jsou momentálně součástí kódu aplikace, nicméně je zde připravené rozhraní pro snadné zakomponování těchto konstant do konfiguračních souborů aplikací.

Oblíbené položky se ukládají a načítají pomocí jisté abstrakce. Všechny oblíbené položky se ukládají do složky `cached`, která se nachází ve složce s daty aplikace. Název jednotlivých položek se liší v závislosti na jejich typu:

- Spojení se ukládají do souborů s názvem `jo-{ID výchozí stanice}-{ID cílové stanice}.fav`.
- Stanice se ukládají do souborů s názvem `si-{ID stanice}.fav`.
- Linky se ukládají do souborů s názvem `li-{ID linky}.fav`.

Pokud chceme danou oblíbenou položku načíst do paměti, pak se dané abstrakci předá pouze její typ a příslušné identifikátory. Abstrakce zná formáty souborů, tedy porovná očekávaný název souboru naší oblíbené položky s názvy souborů ve složce `cached`. Pokud je příslušný soubor nalezen, pak do paměti načte danou oblíbenou položku. A to tak, že daný soubor deserializuje do instance připravené třídy. Dané soubory svou strukturou kopírují odpovědi ze serveru (viz Příloha A.2.2).

Pokud chceme danou oblíbenou položku odstranit, lze tak učinit pouhým smazáním příslušného souboru. Nezavádíme žádný seznam oblíbených položek, informace o oblíbených položkách čerpáme vždy z výstupu pomocných metod souborového systému, díky nimž jsme schopni získat seznam souborů v dané složce, respektive seznam všech oblíbených položek.

- Podporu pro mapu. V mobilní aplikaci implementujeme mapu pomocí pluginu[2], v počítačové aplikace používáme Google Maps Javascript API [6]. Pro počítačovou aplikaci máme připravené rozhraní pro tvorbu javascriptového kódu, který do mapy přidá relevantní prvky, například úsečku nebo označení zastávky.

- Podporu pro komunikaci se serverem. Při popisu serverové aplikace v sekci 5.1 jsme uvedli, že pro komunikaci využíváme TCP protokol. Klientské aplikace musí nejprve pomocí techniky three-way handshake navázat spojení se serverem, až poté lze zapisovat do síťového proudu. Pokud je three-way handshake neúspěšný, předá se zpráva uživatelskému rozhraní, které informuje uživatele o nedostupnosti serveru. V našich aplikacích máme časový limit na provedení three-way handshake nastavený na 5 sekund.

- Podporu pro načtení základních dat. Klientské aplikace očekávají data ve formátu TFB ve složce `basic`, která se nachází ve složce s daty aplikace.

[2] `https://github.com/amay077/Xamarin.Forms.GoogleMaps`

Klientská knihovna obsahuje třídy, do jejichž instancí lze tento formát naparsovat.

Požadavky

Rozlišujeme čtyři druhy požadavků:

- Požadavek na aktualizaci základních dat. Tento požadavek je zaslán na server vždy při startu aplikace.
- Požadavek na vyhledání spojení. Tomuto požadavku vždy předchází požadavek na aktualizaci základních dat.
- Požadavek na vyhledání odjezdů ze stanice. Tomuto požadavku vždy předchází požadavek na aktualizaci základních dat.
- Požadavek na vyhledání informací o lince. Tomuto požadavku vždy předchází požadavek na aktualizaci základních dat.

Požadavek na přidání položky mezi oblíbené lze převést na nějaký z již zmíněných požadavků, dle typu oblíbené položky. Jedná se o vyhledávání, kde nejdřívější čas odjezdu je daný okamžik a maximální čas příjezdu je daný okamžik plus konstanta 24 hodin.

Nyní si uveďme modelový příklad přidání, bez újmy na obecnosti, spojení mezi oblíbené položky:

1. Uživatel přidá spojení mezi své oblíbené položky.
2. Na disku se vytvoří XML soubor reprezentující dané oblíbené spojení, prozatím je prázdný.
3. Klient aktualizuje dané oblíbené spojení na pozadí.

Aktualizace oblíbeného spojení je následující proces:

1. Na server se pošle požadavek na vyhledání daného spojení s nejdřívějším časem odjezdu v daný okamžik, maximálním časem příjezdu posunutým o 24 hodin a nekonečně mnoha přestupy.
2. Server zhotoví odpověď na požadavek, pošle ho klientovi, ten ho přijme, serializuje ho do XML a uloží do příslušného XML souboru.

Pokud chceme vyhledat spojení, pak je posloupnost kroků následující:

1. Uživatel zadá v uživatelském rozhraní požadavek na vyhledání spojení.
2. Klient zkontroluje, jestli je spojení mezi oblíbenými položkami. Pokud ne, pošle uživatelův požadavek na server. V takovém případě se pokračuje krokem číslo 5.

3. Spojení je mezi oblíbenými položkami. Klient zkontroluje, jestli je oblíbené spojení aktuální.

   Pokud ano, deserializuje dané oblíbené spojení z XML souboru do paměti.

   Pokud ne, pošle uživatelův požadavek na server a pokračuje krokem číslo 5. Rovněž na pozadí aktualizuje dané oblíbené spojení.

4. Pomocí dotazovacího jazyku se vyfiltrují výsledky, které vyhovují uživatelovu požadavku. Vždy se bude jednat o „přeskočení“ všech položek, které nevyhovují nejdřívějšímu času odjezdu, dále o omezení na hledaný počet.

5. Výsledky se zobrazí uživateli.

Online a offline mód

Motivaci pro zavedení obou módů jsme již popsali v sekci 4.1. Nyní si ve výčtu uvedeme odlišnosti mezi online a offline módem z pohledu počítačové aplikace:

- Konfigurační soubor. Mezi módy lze přepínat pouze z konfiguračního souboru pomocí položky `OfflineMode`.

  Pokud je zapnutý offline mód, pak musí konfigurační soubor obsahovat adresu zdroje dat.

  Pokud je zapnutý online mód, pak musí konfigurační soubor obsahovat adresu serveru a porty pro jednotlivé služby, na nichž server naslouchá.

- Moduly počítačové aplikace. Počítačová aplikace musí obsahovat jádro systému a preprocesor dat.

  Pokud je zapnutý offline mód, při startu aplikace se data aktualizují na popředí a jádro načítá data do paměti.

  Pokud je zapnutý online mód, jádro systému i preprocesor dat zůstávají nečinnými.

- Vyřizování požadavků. Před odesláním libovolného požadavku se kontroluje, jaký mód je nastavený.

  Pokud je zapnutý offline mód, požadavek se automaticky deleguje do jádra systému.

  Pokud je zapnutý online mód, požadavek projde procesem popsaným v předchozích odstavcích a případně se odešle na server.

  Zvláštním požadavkem je požadavek na aktualizaci základních dat. V případě offline módu tento požadavek nevykonává žádnou akci. Základní data v klientu, který běží v offline módu, jsou vždy aktuální, neboť si je i sám aktualizuje.

- Oblíbené položky. Oblíbené položky ztrácí v případě offline módu smysl, neboť se vždy vyhledává offline.

  Pokud je zapnutý offline mód, oblíbené položky slouží pouze jako zkratka pro vyhledávání daných položek. Soubor reprezentující danou oblíbenou položku je vždy prázdný.

Pokud je zapnutý online mód, vše funguje dle popisu z předchozích odstavců.

Služby určování polohy

Aplikace využívají služby určování polohy tak, jak je poskytuje daný operační systém. Tyto služby jsou využívány v mapě, která se po zobrazení přiblíží na uživatelovu aktuální polohu. Dále při zadávání požadavků, kde lze použít aktuální polohu k nalezení nejbližší zastávky v okolí.

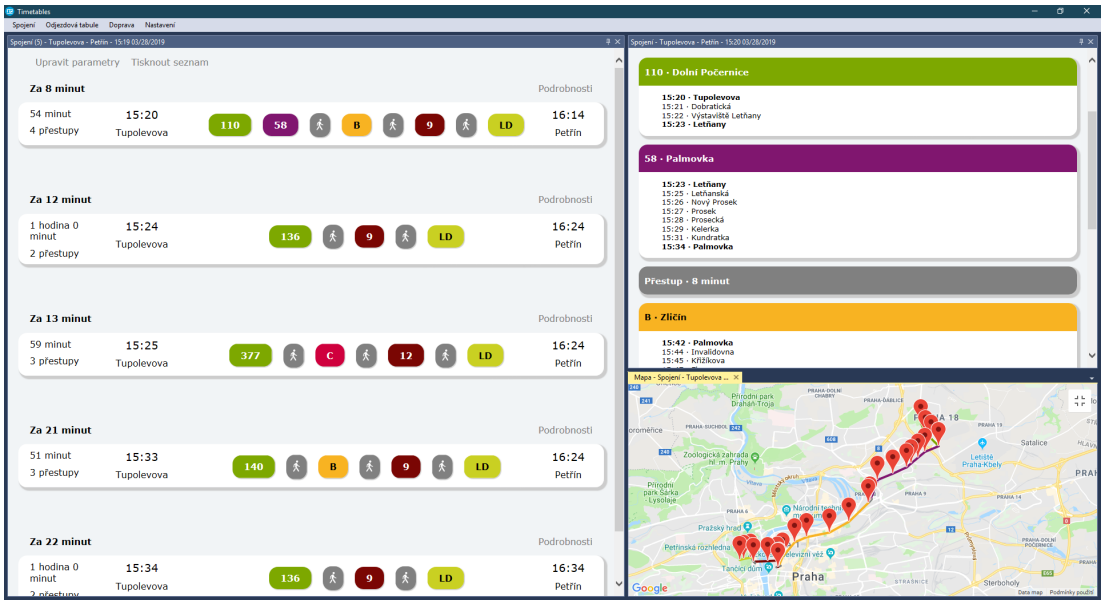
5.2.2 Front-end počítačové aplikace

V této sekci na příkladech popíšeme prostředí uživatelského rozhraní počítačové aplikace a jak jednotlivé části uživatelského rozhraní komunikují s backendem aplikace.

Prostředí aplikace

Uživatelské rozhraní aplikace je navrženo dle vzoru MDI. Tento návrh umožňuje uživateli vytvořit v rámci aplikace více oken. To může přijít vhod zejména při porovnávání nalezených spojení, jelikož každá uživatelova akce (například zobrazení detailů spojení či mapy) má za následek vytvoření nového okna (viz Obrázek 5.4).

Samotné prostředí aplikace je inspirováno vývojovým prostředím Visual Studio.[3] Je použit volně dostupný plugin[4], který oknům v aplikaci dodá požadované vlastnosti. Zobrazování výsledků je inspirováno vzhledem vyhledávače Mapy.cz.[5]



Obrázek 5.4: Prostředí počítačové aplikace.

[3] https://visualstudio.microsoft.com/
[4] http://dockpanelsuite.com/
[5] https://www.seznam.cz/jizdnirady/

Barevné motivy

Naše uživatelské rozhraní podporuje tři barevné motivy, lze je měnit v nastavení. Jedná se o modrý, světlý a tmavý motiv. V našich příkladech budeme používat vždy modrý motiv. Současná implementace podporuje pouze změnu motivu prostředí, barevný motiv zobrazování výsledků je vždy stejný. Do budoucna by bylo vhodné barevné motivy přepracovat, aby i zobrazování výsledků bylo rovněž v daném barevném motivu. Toho by šlo docílit přidáním CSS stylesheetů a přepínáním mezi nimi na základě zvoleného barevného motivu.

Současná implementace nastavuje barevný motiv tak, že se rekurzivně prochází všechny ovládací prvky v okně a nastavuje se jim barva popředí, respektive barva pozadí v závislosti na daném barevném motivu. Zde vzniká problém, jelikož existují prvky, u kterých není možné změnit jejich barvy[6] a neexistuje ani žádná přímočará cesta, jak změnu barev umožnit. Pravděpodobně by bylo nutné předefinovat vykreslovací funkce daných prvků za užití funkcí operačního systému, což by vedlo ke ztrátě přenositelnosti kódu. My jsme zvolili kompromis, kde měníme pouze barvu pozadí daného okna, dále barvu textu v daném okně a barvu všech tlačítek v daném okně, abychom zachovali konzistenci barev. Proto v současné implementaci může tmavý motiv vypadat nepřirozeně. V budoucnu by bylo vhodné reimplementovat uživatelské rozhraní pomocí nějakého modernějšího frameworku[7], který má podporu pro tvorbu moderních uživatelských rozhraní.

Spuštění aplikace

Po spuštění aplikace se nejprve načte nastavení z konfiguračního souboru. Poté se zobrazí načítací lišta, která symbolizuje průběh načítání dat v závislosti na módu:

- V případě online módu aplikace aktualizuje základní data na popředí, je-li to nutné. Tento proces typicky proběhne typicky během jedné sekundy, ovšem závisí na stabilitě sítě.
- V případě offline módu aplikace aktualizuje data na popředí, je-li to nutné. Tento proces zahrnuje předzpracování dat a načtení dat do paměti jádrem. Doba načítání závisí na připojení k internetu a velikosti zpracovávaných dat. Proces typicky proběhne během několika desítek sekund.

Dále se zkontroluje, jestli je nutné aktualizovat oblíbené položky, a pokud ano, tak se aktualizují na pozadí. Po tomto kroku se zobrazí uživatelské rozhraní a aplikace je připravena vyřizovat požadavky.

Nastavení aplikace

Po kliknutí na položku *Nastavení* v hlavním menu aplikace se zobrazí okno s nastavením aplikace. Okno s nastavením obsahuje:

- Seznam barevných motivů. Na výběr je modrý, světlý a tmavý motiv. Po změně motivu je nutné aplikaci restartovat, uživatel bude na tuto skutečnost upozorněn.

[6] Jedná se o prvek sloužící k výběru data a času, prvek reprezentující seznam, prvek reprezentující „našeptávač“ a prvek reprezentující hlavní menu aplikace.

[7] Například Windows Presentation Foundation (WPF), my používáme Windows Forms.

- Seznam lokalizací. V současné verzi je implementována podpora pro český a anglický jazyk. Nové lokalizační soubory lze přidávat na základě informací uvedených v sekci 5.2.1. Po změně jazyka je nutné aplikaci restartovat, uživatel bude na tuto skutečnost upozorněn.

- Tlačítko pro aktualizaci dat. V případě online módu se aktualizují základní data aplikace. V případě offline módu se aktualizují data z externího zdroje. Tento proces zahrnuje i předzpracování dat.

- Tlačítko pro aktualizaci oblíbených položek. Tato položka vynutí aktualizaci oblíbených položek bez ohledu na jejich stav. V případě offline módu se tato položka v nastavení nezobrazí.

Okno pro zadávání požadavků na vyhledání spojení

Okno lze zobrazit z hlavního menu aplikace, je k dohledání pod položkami *Spojení*, dále *Najít spojení*. Okno obsahuje tyto položky:

- Okénko pro zadání výchozí stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.

  Vedle tohoto okénka se nachází obrázek, který reprezentuje aktuální polohu uživatele na základě systémového určování polohy. Po kliknutí na tento obrázek se do okénka pro zadání výchozí stanice vyplní nejbližší stanice v okolí.

- Okénko pro zadání cílové stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.

- Okénko pro zadání data a času nejdřívějšího času odjezdu. Výchozí hodnota je aktuální systémový čas.

- Okénko pro zadání maximálního počtu přestupů. Výchozí hodnota je číslo pět.

- Okénko pro zadání počtu hledaných spojení. Výchozí hodnota je číslo pět.

- Seznam možností, dle kterých lze nalezené výsledky třídit. Jedná se o třídění na straně klienta, tento údaj není obsahem požadavku, který se odesílá na server.

- Tlačítka, pomocí nichž lze zvolit rychlost chůze. K dispozici jsou následující možnosti:

  – Pomalá rychlost chůze. Časy přestupů se násobí koeficientem 1,5.
  – Střední rychlost chůze. Časy přestupů se násobí koeficientem 1.
  – Rychlá rychlost chůze. Časy přestupů se násobí koeficientem 0,5.

  Výchozí hodnota je střední rychlost chůze.

- Tlačítka, pomocí nichž lze vybrat specifické dopravní prostředky. Patří sem metro, tramvaj, autobus, vlak, loď a lanovka. Jako výchozí hodnota jsou navoleny všechny dopravní prostředky.

Dále je zde tlačítko na odeslání požadavku, které požadavek na základě zadaných hodnot vytvoří a předá k řešení back-endu aplikace. Po kliknutí na toto tlačítko se tlačítko vypne a čeká se na odpověď z back-endu aplikace. Je-li požadavek úspěšně vyřízen, toto okno se skryje a zobrazí se okno s výsledky. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se požadavek neprovádí a uživatel bude vyzván k opravě údajů.

Okno pro zadávání požadavků na vyhledání odjezdů ze stanice

Okno lze zobrazit z hlavního menu aplikace, je k dohledání pod položkami *Odjezdová tabule*, dále *Najít odjezdy ze stanice*. Okno obsahuje tyto položky:

- Okénko pro zadání stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.

  Vedle tohoto okénka se nachází obrázek, který reprezentuje aktuální polohu uživatele na základě systémového určování polohy. Po kliknutí na tento obrázek se do okénka pro zadání stanice vyplní nejbližší stanice v okolí.

- Okénko pro zadání data a času nejdřívějšího času odjezdu. Výchozí hodnota je aktuální systémový čas.

- Seznam linek projíždějících zadanou stanicí. Tento seznam se zpřístupní v okamžik, kdy je v okénku pro zadání stanice validní název stanice.

  Pokud uživatel nezvolí žádnou hodnotu, vyhledají se odjezdy všech linek z dané stanice. Pokud uživatel nějakou hodnotu zvolí, vyhledají se odjezdy dané linky z dané stanice.

- Okénko pro zadání počtu hledaných odjezdů. Výchozí hodnota je číslo pět.

Dále je zde tlačítko na odeslání požadavku, které požadavek na základě zadaných hodnot vytvoří a předá k řešení back-endu aplikace. Po kliknutí na toto tlačítko se tlačítko vypne a čeká se na odpověď z back-endu aplikace. Je-li požadavek úspěšně vyřízen, toto okno se skryje a zobrazí se okno s výsledky. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se požadavek neprovádí a uživatel bude vyzván k opravě údajů.

Okno pro zadávání požadavků na vyhledání informací o lince

Okno lze zobrazit z hlavního menu aplikace, je k dohledání pod položkami *Odjezdová tabule*, dále *Najít informace o lince*. Okno obsahuje tyto položky:

- Okénko pro zadání označení linky. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny linky, jejichž označení začíná na daný řetězec.

- Okénko pro zadání data a času nejdřívějšího času odjezdu. Výchozí hodnota je aktuální systémový čas.

- Okénko pro zadání počtu hledaných odjezdů. Výchozí hodnota je číslo pět.

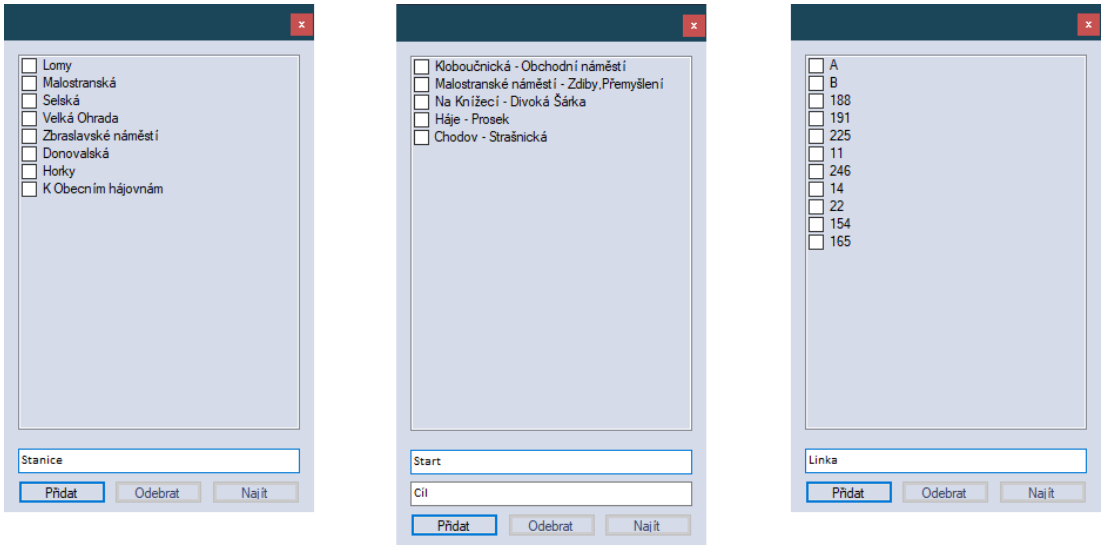
Dále je zde tlačítko na odeslání požadavku, které požadavek na základě zadaných hodnot vytvoří a předá k řešení back-endu aplikace. Po kliknutí na toto tlačítko se tlačítko vypne a odpověď z back-endu aplikace. Je-li požadavek úspěšně vyřízen, toto okno se skryje a zobrazí se okno s výsledky. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná linka neexistuje, pak se požadavek neprovádí a uživatel bude vyzván k opravě údajů.

Okna pro oblíbené položky

Existují tři druhy oblíbených položek, každý druh má své vlastní okno:

- Oblíbená spojení. Toto okno je k dohledání pod položkami *Spojení*, dále *Oblíbená spojení*.
- Oblíbené stanice. Toto okno je k dohledání pod položkami *Odjezdová tabule*, dále *Oblíbené stanice*.
- Oblíbené linky. Toto okno je k dohledání pod položkami *Odjezdová tabule*, dále *Oblíbené linky*.

Zde popíšeme, bez újmy na obecnosti, manipulaci s oknem reprezentující oblíbená spojení. Ostatní dvě okna mají stejnou funkcionalitu (viz Obrázek 5.5).



Obrázek 5.5: Levé okno reprezentuje oblíbené stanice. Okno uprostřed reprezentuje oblíbená spojení. Okno vpravo reprezentuje oblíbené linky.

Okno obsahuje zaškrtávací seznam oblíbených položek, dále následující ovládací prvky:

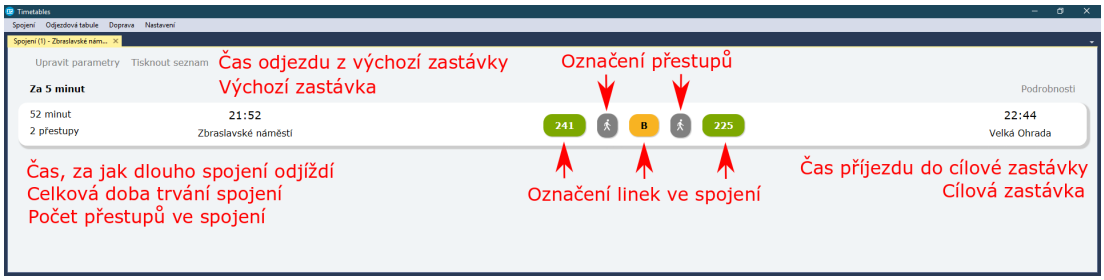
- Okénko pro zadání výchozí stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.

- Okénko pro zadání cílové stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.
- Tlačítko na přidání zadaného spojení. Pokud jsou údaje o stanicích nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se dané spojení mezi oblíbené položky nepřidá a uživatel bude vyzván k opravě údajů.
- Tlačítko pro odebrání vybraných spojení. Toto tlačítko je neaktivní, bude aktivní po zaškrtnutí nějakého oblíbeného spojení. Po stisknutí tlačítka se smažou všechna vybraná oblíbená spojení.
- Tlačítko pro nalezení vybraných spojení. Toto tlačítko je neaktivní, bude aktivní po zaškrtnutí nějakého oblíbeného spojení. Po stisknutí tlačítka se otevřou předvyplněná okna pro nalezení zaškrtnutých oblíbených spojení.

Oblíbené položky se podmíněně aktualizují při startu aplikace a při vyhledávání daných položek. Aktualizují se vždy, klikne-li uživatel na tlačítko pro aktualizaci oblíbených položek v nastavení aplikace a po přidání položky mezi oblíbené.

Okno pro zobrazení nalezených spojení

Toto okno se zobrazí, pokud je požadavek na vyhledání spojení úspěšný.



Obrázek 5.6: Popis položek u nalezeného spojení.

Okno obsahuje následující ovládací prvky:

- Tlačítko na změnu parametrů spojení. Po stisknutí tohoto tlačítka se dané okno zavře a znovu se zobrazí okno pro zadávání požadavků.

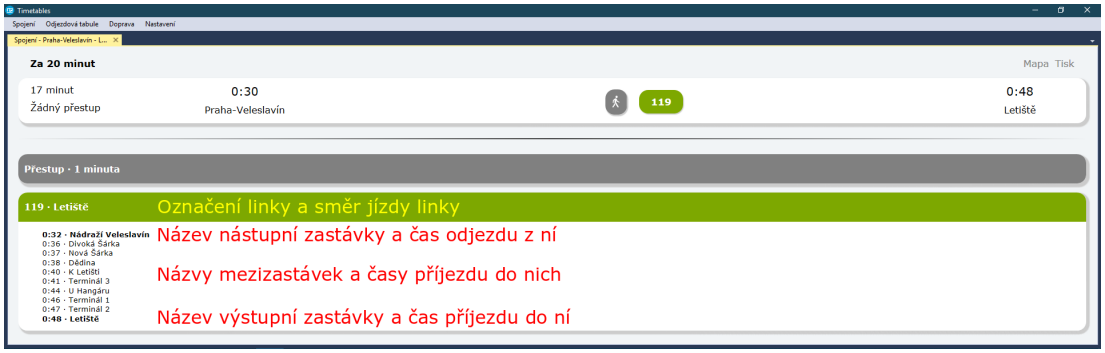
  Jedná se o instanci okna, které předcházelo zobrazení výsledků. Tedy každé okno pro zobrazení nalezených spojení obsahuje ukazatel na okno, ve kterém jsme zadávali požadavek. To má za následek, že všechny parametry budou předvyplněné.
- Tlačítko pro tisk seznamu spojení. Po stisknutí tohoto tlačítka se zobrazí dialog tisku. Seznam se vytiskne v režimu pro nízká rozlišení.
- Pro každé nalezené spojení je zde tlačítko, po jehož stisknutí se zobrazí podrobnosti daného spojení.

Dále obsahuje základní informace o nalezených spojeních (viz Obrázek 5.6).

Po kliknutí na tlačítko pro zobrazení podrobností spojení se otevře nové okno s daným spojením. Toto okno obsahuje:

- Tytéž základní informace o spojení.
- Tlačítko pro zobrazení mapy s trasou spojení. Tato trasa je aproximována úsečkami. Rovněž se zobrazí stanice v trase spojení. Po kliknutí na stanici se zobrazí pět nejdřívějších odjezdů z dané stanice, kde jako čas nejdřívějšího odjezdu se nastaví čas příjezdu do dané stanice v rámci daného spojení.
- Tlačítko pro tisk. Po stisknutí tohoto tlačítka se zobrazí dialog tisku. Spojení se vytiskne v režimu pro nízká rozlišení.

Dále obsahuje detail daného spojení. Sem řadíme detailní informace o jednotlivých spojích a přestupech (viz Obrázek 5.7).

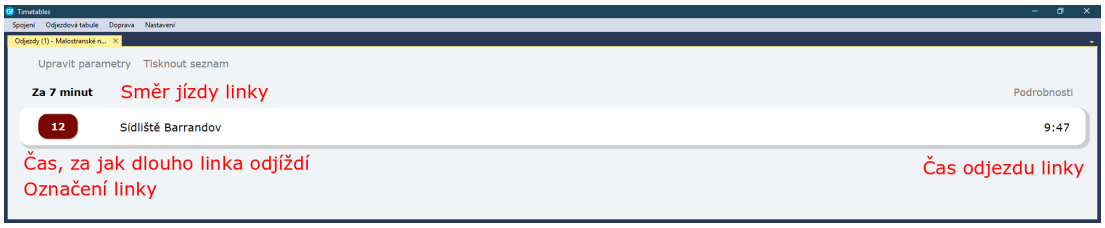


Obrázek 5.7: Popis položek v detailu nalezeného spojení.

Okno pro zobrazení nalezených odjezdů

Toto okno se zobrazí, pokud je požadavek na vyhledání odjezdů ze stanice či informací o lince úspěšný. Okno obsahuje stejné ovládací prvky jako okno pro zobrazení nalezených spojení, jejich funkcionalita je rovněž stejná.

Okno obsahuje základní informace o nalezených odjezdech (viz Obrázek 5.8). Způsob zobrazování informací o lince je stejný jako způsob zobrazování odjezdů ze stanice, zobrazují se odjezdy dané linky z její výchozích stanic.



Obrázek 5.8: Popis položek u nalezeného odjezdu.

Po kliknutí na tlačítko pro zobrazení podrobnosti odjezdu se otevře nové okno s daným odjezdem. Toto okno má stejný charakter jako okno pro detail spojení.

Okna pro zobrazení informací z dopravy

Řadíme sem dvě okna:

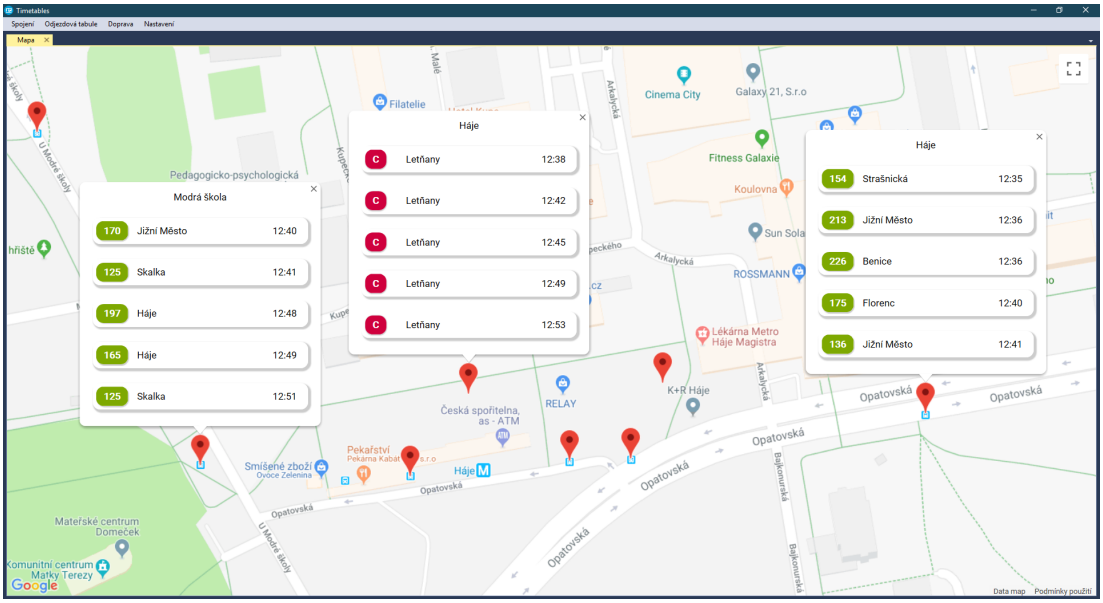
- Okno pro zobrazení mimořádných událostí v dopravě. Toto okno je k dohledání pod položkami *Doprava*, dále *Mimořádné události*. Po otevření okna se zobrazí informace o mimořádných událostech v dopravě tak, jak je poskytuje dopravce.

- Okno pro zobrazení výluk. Toto okno je k dohledání pod položkami *Doprava*, dále *Výluky*. Po otevření okna se zobrazí informace o výlukách tak, jak je poskytuje dopravce.

Tato okna vyžadují přístup k internetu, neboť se informace stahují z RSS zdrojů uvedených v konfiguračním souboru. Okna jsou stejného typu, jejich obsah se liší pouze zdrojem dat a použitými skripty.

Okno s mapou

Toto okno je k dohledání pod položkami *Odjezdová tabule*, dále *Zobrazit mapu*. Po otevření okna se zobrazí mapa, kde jsou vyobrazeny všechny zastávky (nikoliv stanice) v dopravní síti. Po kliknutí na libovolnou zastávku se zobrazí pět nejdřívějších odjezdů z dané zastávky (viz Obrázek 5.9).



Obrázek 5.9: Mapa dopravní sítě.

5.2.3 Front-end mobilní aplikace

V této sekci na příkladech popíšeme prostředí uživatelského rozhraní mobilní aplikace a jak jednotlivé části uživatelského rozhraní komunikují s back-endem aplikace.

Prostředí aplikace

Uživatelské rozhraní aplikace obsahuje jednu hlavní aktivitu, ve které se nachází hlavní menu a oblast pro zobrazování dat. Zobrazování dat má charakter navigace, tzn. například po zobrazení detailů spojení se lze stisknutím tlačítka zpět vrátit na seznam všech nalezených spojení. Pokud je historie navigace prázdná a uživatel stiskne tlačítko zpět, pak se aplikace ukončí. Historie navigace se smaže vždy, jakmile dojde k navigaci z hlavního menu aplikace.

Spuštění aplikace

Po spuštění aplikace se zkontroluje, jestli má aplikace potřebná oprávnění. Pokud ne, požádá o ně uživatele. Mezi oprávnění, o která je nutné explicitně žádat, řadíme:

- Přístup k poloze zařízení. Poloha telefonu je nutná pro správné fungování mapy a pro hledání nejbližších stanic v okolí.

- Přístup k souborovému systému zařízení. To je nutné pro aktualizaci základních dat aplikace, ukládání oblíbených položek a nastavení aplikace.

Pokud jsou všechna oprávnění aplikaci udělena, aplikace může zahájit svou činnost.

Aplikace nejprve načte nastavení, dále požádá operační systém o informace o aktuální poloze zařízení. Je vhodné tento krok zahrnout do procesu spuštění aplikace, nastaví se tím „přibližná“ poloha zařízení, což má za následek, že každé další volání funkce pro získání polohy proběhne citelně rychleji. Dále se načtou základní data z úložiště, popřípadě proběhne pokus o aktualizaci. Poslední krok je aktualizace oblíbených položek. Aplikace nejprve zkontroluje, jestli je nutné oblíbené položky aktualizovat. Pokud ano a jsou splněny podmínky[8] pro aktualizaci, pak je aktualizuje na pozadí. Dále zaregistruje callback, který zařídí, že jakmile se zařízení připojí k libovolné Wi-Fi síti, pak se oblíbené položky aktualizují vždy a bez ohledu na to, je-li to nutné.

Nastavení aplikace

Po kliknutí na položku *Nastavení* v hlavním menu aplikace se zobrazí rozhraní pro nastavení aplikace. Nastavení aplikace obsahuje:

- Seznam lokalizací. V současné verzi je implementována podpora pro český a anglický jazyk. Nové lokalizační soubory lze přidávat na základě informací uvedených v sekci 4.5.1. Po změně jazyka je nutné aplikaci restartovat, uživatel bude na tuto skutečnost upozorněn.

- Modifikátor rychlosti přestupů. Jedná se o posuvník, jehož dolní mez je hodnota 0,5 a horní mez je hodnota 1,5. Na základě navolené hodnoty se při vyhledávání spojení budou časy přestupů násobit tímto koeficientem.

[8] Zařízení je připojené k Wi-Fi síti nebo je povolena možnost aktualizovat oblíbené položky i prostřednictvím mobilních dat, tato možnost bude zmíněna v následujících odstavcích.

- Omezení na specifické dopravní prostředky. Zvolené dopravní prostředky se poté budou využívat při vyhledávání spojení. Je implementována podpora pro metro, tramvaj, autobus, vlak, lanovku a loď.
- Možnost aktualizovat oblíbené položky při mobilních datech. Oblíbené položky se jinak aktualizují pouze při Wi-Fi připojení, je-li zvolena tato možnost, pak se budou aktualizovat i bez nutnosti připojení k Wi-Fi síti.

Zadávání požadavků na vyhledání spojení

Po kliknutí na položku *Najít spojení* v hlavním menu aplikace se zobrazí rozhraní pro zadání požadavku na vyhledání spojení. Toto rozhraní obsahuje:

- Okénko pro zadání výchozí stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.

  Vedle tohoto okénka se nachází obrázek, který reprezentuje aktuální polohu uživatele na základě systémového určování polohy. Po kliknutí na tento obrázek se do okénka pro zadání výchozí stanice vyplní nejbližší stanice v okolí.
- Okénko pro zadání cílové stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.
- Okénko pro zadání data a času nejdřívějšího času odjezdu. Výchozí hodnota je aktuální systémový čas.
- Posuvník pro výběr maximálního počtu přestupů. Výchozí hodnota je číslo pět.
- Posuvník pro výběr počtu hledaných spojení. Výchozí hodnota je číslo pět.

Dále je zde tlačítko na odeslání požadavku, které požadavek na základě zadaných hodnot vytvoří a předá k řešení back-endu aplikace. Po kliknutí na toto tlačítko se tlačítko vypne a čeká se na odpověď z back-endu aplikace. Je-li požadavek úspěšně vyřízen, zobrazí se výsledky. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se požadavek neprovádí a uživatel bude vyzván k opravě údajů.

Je zde také tlačítko pro přidání spojení mezi oblíbené položky. Pokud uživatel klikne na toto tlačítko, pak se mezi oblíbená spojení přidá spojení se zadanou výchozí a cílovou zastávkou. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se přidání neprovede a uživatel bude vyzván k opravě údajů.

Oblíbená spojení jsou rovněž součástí rozhraní. Rozhraní pro oblíbené položky bude popsáno v následujících odstavcích.

Zadávání požadavků na vyhledání odjezdů ze stanice

Po kliknutí na položku *Najít odjezdy ze stanice* v hlavním menu aplikace se zobrazí rozhraní pro zadání požadavku na vyhledání odjezdů ze stanice. Toto rozhraní obsahuje:

- Okénko pro zadání stanice. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny stanice, jejichž název začíná na daný řetězec.

  Vedle tohoto okénka se nachází obrázek, který reprezentuje aktuální polohu uživatele na základě systémového určování polohy. Po kliknutí na tento obrázek se do okénka pro zadání stanice vyplní nejbližší stanice v okolí.

- Okénko pro zadání data a času nejdřívějšího času odjezdu. Výchozí hodnota je aktuální systémový čas.

- Seznam linek projíždějících zadanou stanicí. Tento seznam se zpřístupní v okamžik, kdy je v okénku pro zadání stanice validní název stanice. Seznam lze zobrazit kliknutím na daný ovládací prvek.

  Pokud uživatel nezvolí žádnou hodnotu, vyhledají se odjezdy všech linek z dané stanice. Pokud uživatel nějakou hodnotu zvolí, vyhledají se odjezdy dané linky z dané stanice.

- Posuvník pro zadání počtu hledaných odjezdů. Výchozí hodnota je číslo pět.

Dále je zde tlačítko na odeslání požadavku, které požadavek na základě zadaných hodnot vytvoří a předá k řešení back-endu aplikace. Po kliknutí na toto tlačítko se tlačítko vypne a čeká se na odpověď z back-endu aplikace. Je-li požadavek úspěšně vyřízen, zobrazí se výsledky. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se požadavek neprovádí a uživatel bude vyzván k opravě údajů.

Je zde také tlačítko pro přidání stanice mezi oblíbené položky. Pokud uživatel klikne na toto tlačítko, pak se mezi oblíbené stanice přidá uvedená stanice. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná stanice neexistuje, pak se přidání neprovede a uživatel bude vyzván k opravě údajů.

Oblíbené stanice jsou rovněž součástí rozhraní. Rozhraní pro oblíbené položky bude popsáno v následujících odstavcích.

Zadávání požadavků na vyhledání informací o lince

Po kliknutí na položku *Najít informace o lince* v hlavním menu aplikace se zobrazí rozhraní pro zadání požadavku na vyhledání informací o lince. Toto rozhraní obsahuje:

- Okénko pro zadání označení linky. Jakmile uživatel začne do okénka psát, zobrazí se „našeptávač“, který uživateli nabídne všechny linky, jejichž označení začíná na daný řetězec.

- Okénko pro zadání data a času nejdřívějšího času odjezdu. Výchozí hodnota je aktuální systémový čas.

- Posuvník pro zadání počtu hledaných odjezdů. Výchozí hodnota je číslo pět.

Dále je zde tlačítko na odeslání požadavku, které požadavek na základě zadaných hodnot vytvoří a předá k řešení back-endu aplikace. Po kliknutí na toto tlačítko se tlačítko vypne a čeká se na odpověď z back-endu aplikace. Je-li požadavek úspěšně vyřízen, zobrazí se výsledky. Pokud jsou údaje nesprávně vyplněny,

například pokud zadaná linka neexistuje, pak se požadavek neprovádí a uživatel bude vyzván k opravě údajů.

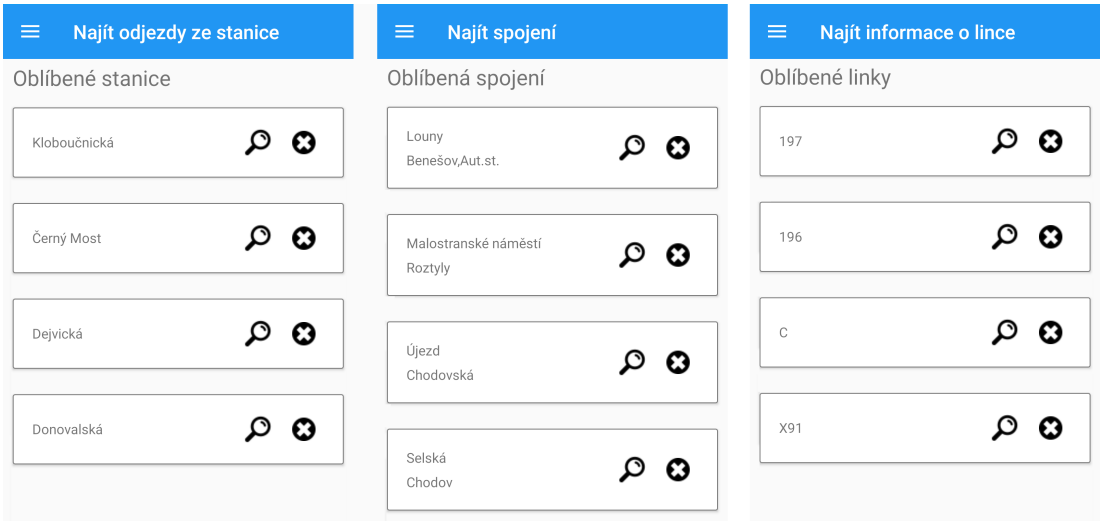
Je zde také tlačítko pro přidání linky mezi oblíbené položky. Pokud uživatel klikne na toto tlačítko, pak se mezi oblíbené linky přidá uvedená linka. Pokud jsou údaje nesprávně vyplněny, například pokud zadaná linka neexistuje, pak se přidání neprovede a uživatel bude vyzván k opravě údajů.

Oblíbené linky jsou rovněž součástí rozhraní. Rozhraní pro oblíbené položky bude popsáno v následujících odstavcích.

Oblíbené položky

Rozhraní pro oblíbená spojení, oblíbené stanice i oblíbené linky je stejné. Zde popíšeme, bez újmy na obecnosti, rozhraní pro oblíbená spojení.

Oblíbená spojení jsou součástí rozhraní pro zadávání požadavku na vyhledání spojení. Každé oblíbené spojení obsahuje ikonku lupy a ikonku křížku (viz Obrázek 5.10). Po stisknutí ikonky lupy se předvyplní výchozí stanice oblíbeného spojení do okénka pro zadání výchozí stanice, respektive cílová stanice oblíbeného spojení do okénka pro zadání cílové stanice. Dané oblíbené spojení lze vyhledat stisknutím tlačítka *Najít*, po jehož stisknutí se back-end aplikace postará o zobrazení oblíbeného spojení. Ikonka křížku reprezentuje smazání daného oblíbeného spojení.



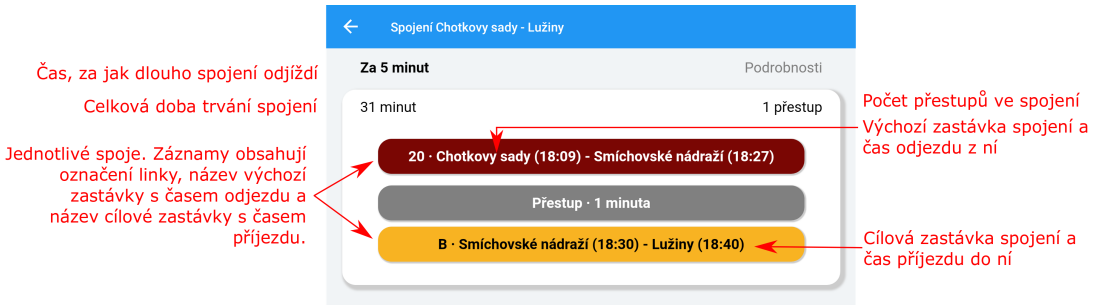
Obrázek 5.10: Vlevo jsou vyobrazené oblíbené stanice. Uprostřed jsou vyobrazená oblíbená spojení. Vpravo jsou vyobrazené oblíbené linky.

Oblíbené položky se aktualizují pouze tehdy, je-li zařízení připojeno k Wi-Fi síti. Tuto podmínku lze rozšířit úpravou uživatelského nastavení, kde uživatel může zvolit, že se oblíbené položky mohou aktualizovat i prostřednictvím mobilních dat. Při startu aplikace a při vyhledávání daných oblíbených položek se oblíbené položky aktualizují podmíněně.

Je-li zařízení připojeno k Wi-Fi síti, pak se oblíbené položky aktualizují vždy při spuštění aplikace, tedy bez ohledu na to, je-li to nutné. Dále se aktualizují vždy, když dojde ke změně Wi-Fi sítě.

Zobrazení nalezených spojení

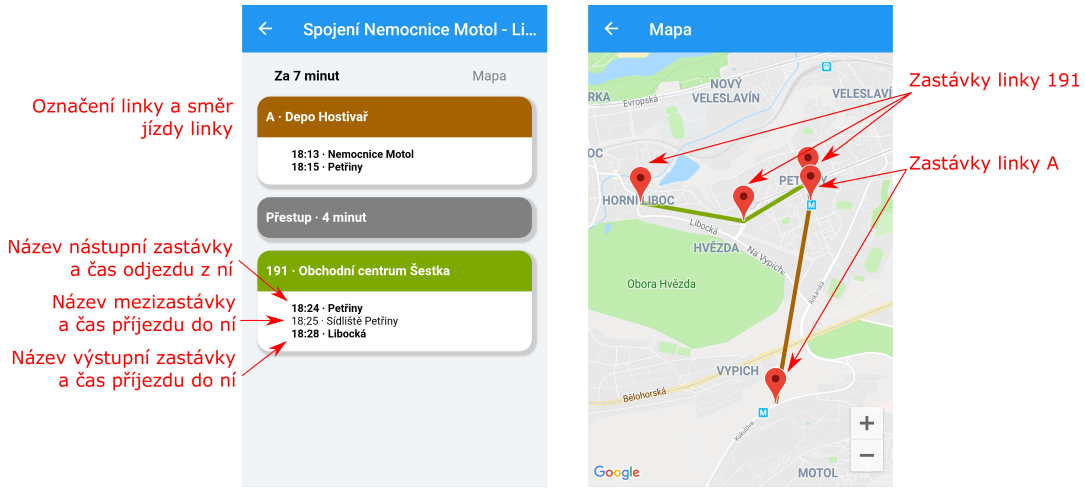
Rozhraní pro zobrazení nalezených spojení se zobrazí, je-li požadavek na vyhledání spojení úspěšný. Toto rozhraní obsahuje základní informace o nalezených spojeních (viz Obrázek 5.11). Pro každé nalezené spojení je zde tlačítko, po jehož stisknutí se zobrazí podrobnosti daného spojení.



Obrázek 5.11: Popis položek u nalezeného spojení. Snímek obrazovky byl pořízen v širokém zobrazení.

Po kliknutí na tlačítko pro zobrazení podrobností spojení se zobrazí detaily daného spojení. Sem řadíme detailní informace o jednotlivých spojích a přestupech (viz Obrázek 5.12).

Dále je zde tlačítko pro zobrazení mapy s trasou spojení. Tato trasa je aproximována úsečkami (viz Obrázek 5.12). Rovněž se zobrazí stanice v trase spojení. Po dvojkliku na stanici se zobrazí pět nejdřívějších odjezdů z dané stanice, kde jako čas nejdřívějšího odjezdu se nastavní čas příjezdu do dané stanice v rámci daného spojení.

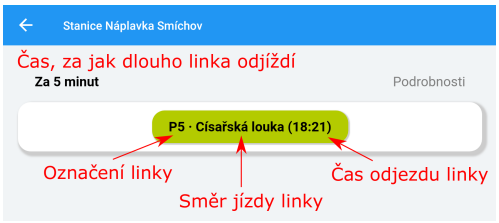


Obrázek 5.12: Na obrázku vlevo je popis položek v detailu nalezeného spojení. Na obrázku vpravo je totéž spojení zobrazené na mapě.

Zobrazení nalezených odjezdů

Rozhraní pro zobrazení nalezených odjezdů se zobrazí, je-li požadavek na vyhledání odjezdů ze stanice či informací o lince úspěšný. Toto rozhraní obsahuje

základní informace o nalezených odjezdech (viz Obrázek 5.13). Způsob zobrazování informací o lince je stejný jako způsob zobrazování odjezdů ze stanice, zobrazují se odjezdy dané linky z její výchozích stanic.



Obrázek 5.13: Popis položek u nalezeného odjezdu. Snímek obrazovky byl pořízen v širokém zobrazení.

Po kliknutí na tlačítko pro zobrazení podrobností odjezdu se zobrazí detaily daného odjezdu. Rozhraní pro zobrazení detailů odjezdu má stejný charakter jako rozhraní pro zobrazení detailů spojení.

Zobrazení informací z dopravy

Řadíme sem dvě položky:

- Mimořádné události v dopravě. Tato položka je k dohledání v menu aplikace pod položkou *Mimořádné události*. Po zvolení dané položky se zobrazí informace o mimořádných událostech v dopravě tak, jak je poskytuje dopravce.
- Výluky. Tato položka je k dohledání v menu aplikace pod položkou *Výluky*. Po zvolení dané položky se zobrazí informace o výlukách tak, jak je poskytuje dopravce.

Tyto položky vyžadují přístup k internetu, neboť se informace stahují z RSS zdrojů uvedených v konfiguračním souboru. Položky jsou stejného typu, jejich obsah se liší pouze zdrojem dat a použitými skripty.

Zobrazení mapy

Mapa je k dohledání v menu aplikace pod položkou *Zobrazit mapu*. Na mapě jsou vyobrazeny všechny zastávky (nikoliv stanice) v dopravní síti. Po kliknutí na libovolnou zastávku se zobrazí název dané zastávky. Po opětovném kliknutí na danou zastávku se zobrazí pět nejdřívějších odjezdů z dané zastávky, využije se rozhraní pro zobrazení nalezených odjezdů.

6. Experimentální srovnání řešení

V této kapitole budeme porovnávat dobu běhu implementace Algoritmu 1 ze sekce 3.2 na různých procesorech s různými podmínkami.

6.1 Spuštění experimentu

Program pro náš experiment je přeložen překladačem Microsoft Visual C++ 2017 (64 bit) s maximálními optimalizacemi. Jízdní řád, nad kterým bude algoritmus spouštěn, je dostupný v otevřených datech PID [2]. Pro běh experimentu je nutné data z daného zdroje stáhnout, poté nechat zpracovat preprocesorem a následně zkopírovat do složky se spustitelným souborem experimentu. Prvních dvou kroků je možné docílit například spuštěním serveru, který data zpracuje automaticky.

Při experimentu budeme sledovat průměrnou dobu vyhledání jednoho spojení v různých denních dobách. Denní doby byly zvoleny následovně:

- Noc. Pro náš experiment jsme zvolili čas 2:00. V noci bývá doprava nejméně hustá.
- Ranní špička. Pro náš experiment jsme zvolili čas 8:00. Během ranní špičky bývá doprava nejhustší.
- Poledne. Pro náš experiment jsme zvolili čas 13:00. Během poledne je doprava hustší než v noci, ale ne tak hustá jako během dopravních špiček.

Při každém požadavku budeme vyhledávat 10 spojení, přičemž každý experiment spustíme stokrát. Z jednotlivých výsledků následně spočítáme aritmetický průměr, to všechno abychom eliminovali případné odchylky vlivem činnosti operačního systému. Rovněž provedeme experiment, kde budeme sledovat dobu potřebnou pro uložení oblíbeného spojení.

Pro výše popsaný postup jsme zvolili dvě spojení, která jsou dostatečně různorodá, abychom je dokázali porovnat a byli schopni učinit nějaký závěr.

- Spojení *Pod Jezerkou - Malostranské náměstí*. Toto spojení využívá pouze jeden přestup, buď ve stanici *Novoměstská radnice* (z linky 14 na linku 22) nebo ve stanici *I.P.Pavlova* (z linky 11 na linku 22). Očekává se nízký počet zpracovaných linek a zastávek, stejně tak celkový čas.
- Spojení *Horčičkova - Letiště*. Toto spojení projíždí celou mapou Prahy, přičemž existuje několik možných variant, které se mezi sebou liší pouze minimálně. Během testování jsme toto spojení považovali za jedno z nejsložitějších, tudíž zde očekáváme jeden z největších celkových časů, kterého můžeme během vyhledávání spojení v Praze dosáhnout.

6.2 Prostředí experimentu

Experiment proběhl na počítači s operačním systémem Windows 10 Pro a operační pamětí DDR3 2x4 GB 1600 MHz. Pro porovnání vlivu procesoru na algoritmus jsme experiment prováděli na třech různých procesorech, a to Intel Core

i7-2600, Intel Core i5-2400 a Intel Pentium G620. Všechny procesory mají socket LGA1155 (rok výroby 2011), tudíž ostatní komponenty počítače mohly zůstat stejné po dobu všech experimentů. Porovnání jednotlivých procesorů nalezneme v Tabulce 6.1, která kopíruje výstup programu CPU-Z.[1]

Zpracovávání linek algoritmem může způsobovat výpadky cache, neboť k linkám se přistupuje náhodně a frekventované linky mohou obsahovat data o velikosti až malých stovek kilobyte. U procesorů i7-2600 a i5-2400, nataktovaných na stejnou frekvenci[2], budeme pozorovat zejména vliv velikosti třetího levelu procesorové cache na dobu běhu algoritmu. Očekávaný rozdíl je minimální, nicméně porovnání by mohlo být zajímavé.

Procesor Pentium G620 má vůči dvěma svým konkurentům o 800 MHz nižší maximální frekvenci, v tomto případě budeme sledovat zejména pokles výkonu v závislosti na frekvenci procesoru, nicméně svou roli zde sehraje i velikost třetího levelu cache. Počet fyzických jader je v tomto experimentu irelevantní, neboť naše implementace využívá jednovláknovou verzi. Vyšší počet jader může v naší implementaci přijít vhod při větším počtu konkurentních požadavků na vyhledání spojení, což není předmětem tohoto experimentu.

	i7-2600	i5-2400	Pentium G620
Frekvence	3,4 GHz (3,8 GHz)	3,1 GHz (3,4 GHz)	2,6 GHz
Počet jader	4 (8)	4 (4)	2 (2)
L1-D cache	4x 32 kB	4x 32 kB	2x 32 kB
L2 cache	4x 256 kB	4x 256 kB	2x 256 kB
L3 cache	8 MB	6 MB	3 MB

Tabulka 6.1: Porovnání procesorů v experimentu.

6.3 Výsledky experimentu

V experimentu jsme sledovali tyto hodnoty:

- Celkový čas potřebný na uložení spojení mezi oblíbené položky.
- Průměrný čas potřebný pro nalezení jednoho spojení.
- Celkový počet označených zastávek při vyhledávání jednoho spojení.
- Celkový počet procházených linek při vyhledávání jednoho spojení.
- Celkový počet volání funkce pro nalezení nejdřívější jízdy linky při vyhledávání jednoho spojení.

[1] `https://www.cpuid.com/softwares/cpu-z.html`

[2] Při experimentu na i7-2600 jsme zakázali turbo režim, pak jsou maximální frekvence totožné.

- Průměrný počet kol při vyhledávání jednoho spojení (počet přestupů nastaven na nekonečno).

Výsledky experimentu lze nalézt v Tabulce 6.2 a 6.3.

	Noc			Ráno			Poledne		
	i7	i5	P	i7	i5	P	i7	i5	P
Uložení ob. spojení (s)	7,1	7,3	10,2	7,1	7,3	10,2	7,1	7,3	10,2
Jedno spojení (ms)	39	43	60	40	43	59	41	43	61
Počet ozn. zastávek	3893			3964			4041		
Počet zprac. linek	5603			5655			5734		
Počet volání funkce *et*	43617			43465			44150		
Počet kol	5,4			5,4			5,2		

Tabulka 6.2: Výsledky experimentu spojení *Pod Jezerkou - Malostranské náměstí*.

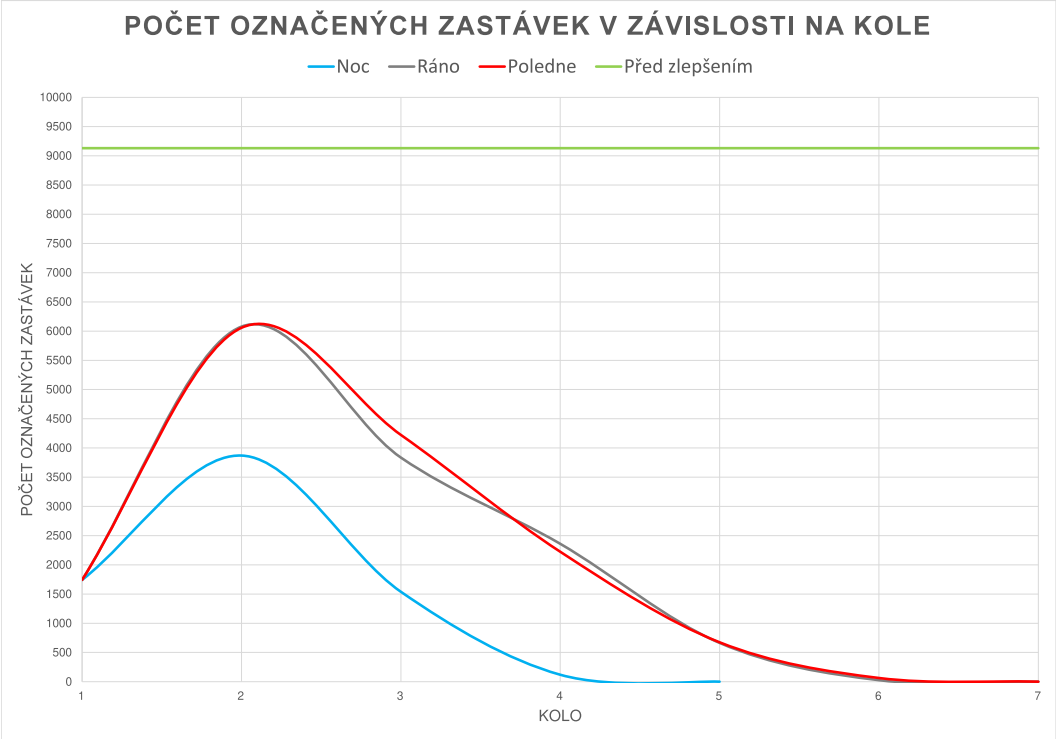
	Noc			Ráno			Poledne		
	i7	i5	P	i7	i5	P	i7	i5	P
Uložení ob. spojení (s)	44,1	46,0	61,3	44,1	46,0	61,3	44,1	46,0	61,3
Jedno spojení (ms)	76	78	110	113	118	160	119	120	163
Počet ozn. zastávek	9739			15331			15250		
Počet zprac. linek	9243			12827			12387		
Počet volání funkce *et*	85257			128204			125578		
Počet kol	6,0			7,3			7,5		

Tabulka 6.3: Výsledky experimentu spojení *Horčičkova - Letiště*.

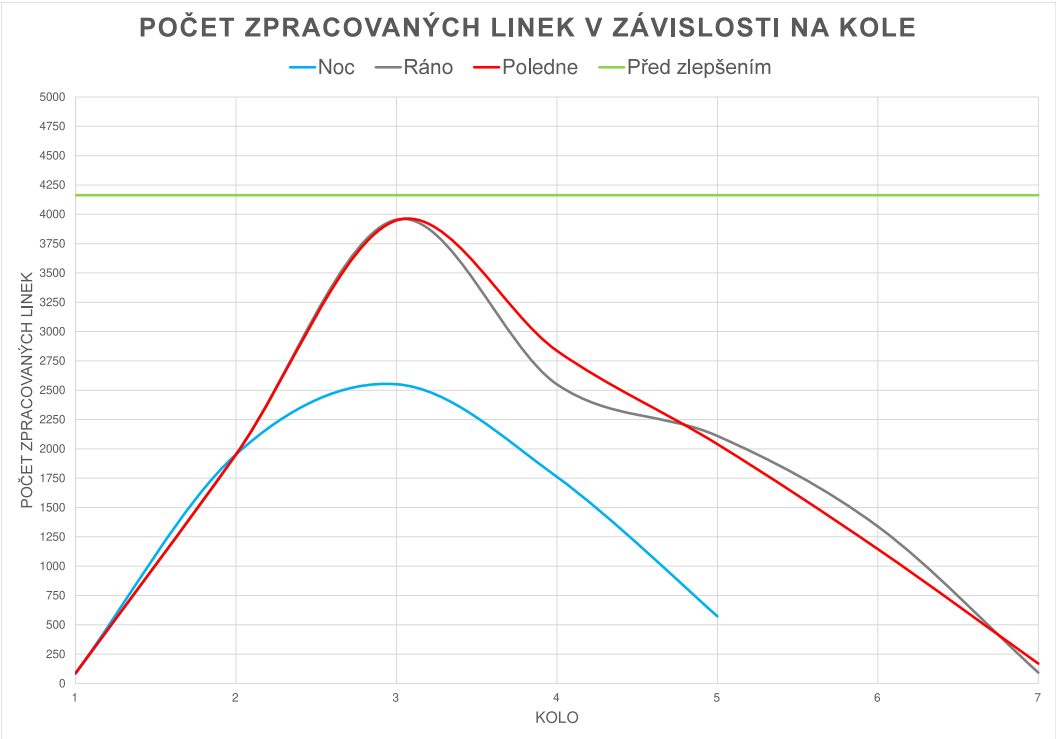
Experiment jsme prováděli nad jízdním řádem s 4163 linkami, 73768 jízdami linek a 9131 zastávkami. Z výsledků můžeme usoudit, že vzhledem k počtu navštívených zastávek a zpracovaných linek jsme, díky zlepšením ze sekce 3.2.2, dosáhli přibližně 75% zlepšení v počtu zpracovávaných zastávek a 55% zlepšení v počtu procházených linek.[3]

Pro výsledky z Tabulky 6.3 jsme vytvořili dva grafy (viz Obrázek 6.1 a Obrázek 6.2), které vizualizují dané zlepšení. Porovnáváme zde počty označených zastávek a zpracovaných linek před zlepšením a po zlepšení v jednotlivých kolech.

[3]V kole k by se jinak zpracovalo $|\mathcal{S}|$ zastávek a $|\mathcal{R}|$ linek. Tato čísla vynásobíme průměrným počtem kol na spojení, zlepšení následně spočítáme porovnáním těchto hodnot s výsledky experimentu.



Obrázek 6.1: Vizualizace zlepšení při zpracovávání zastávek.

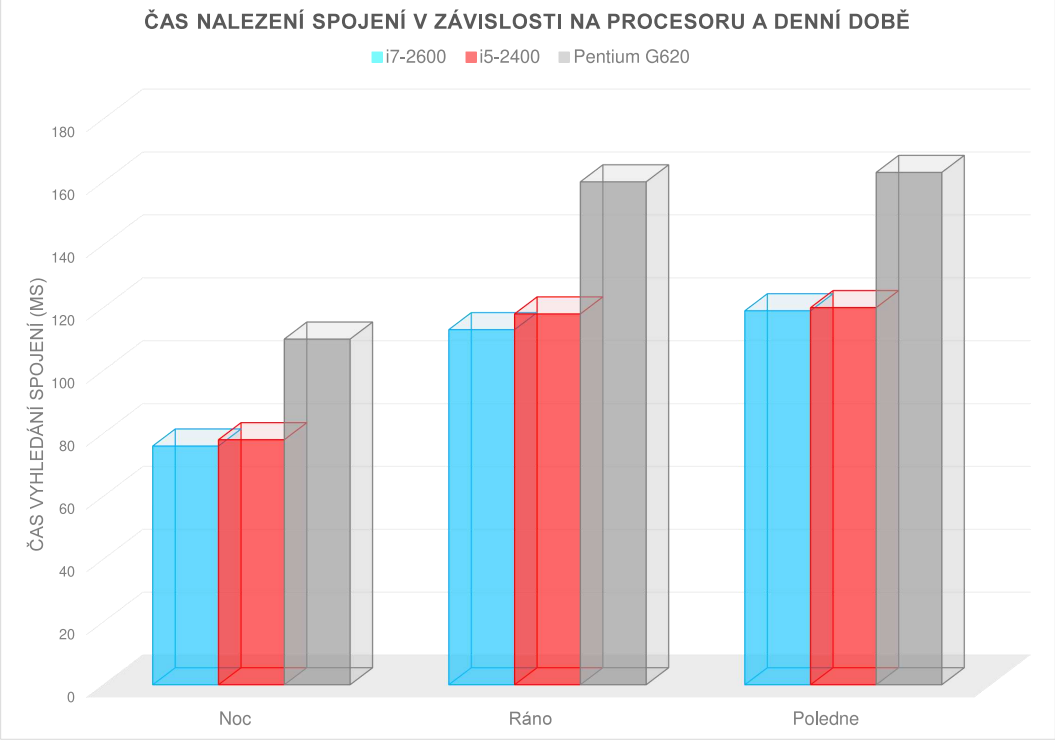


Obrázek 6.2: Vizualizace zlepšení při zpracovávání linek.

Při porovnání výsledků z procesorů i7-2600 a i5-2400 je vidět, že 2 MB rozdíl ve třetí úrovni procesorové cache je sice minimální, ale lze ho zaregistrovat. Jelikož se při zpracovávání linek k linkám přistupuje náhodně, tak ani neexistuje přímočarý způsob, jak výpadkům cache zabránit.

Výkon procesoru Pentium G620, v porovnání s jeho dvěma konkurenty, je citelně menší. Tento pokles výkonu přičítáme zejména menší maximální frekvenci. Velikost cache zde také hraje roli, nicméně oproti předchozímu porovnání méně znatelnou, neboť při nižší frekvenci „čeká“ procesor na data z paměti mnohem „kratší“ dobu.

Pro lepší vizualizaci vlivu výkonu procesoru na dobu běhu algoritmu jsme pro výsledky z Tabulky 6.3 vytvořili graf (viz Obrázek 6.3). Tento graf porovnává časy potřebné pro vyhledání jednoho spojení v závislosti na procesoru a denní době.



Obrázek 6.3: Vizualizace vlivu procesoru a denní doby na čas potřebný pro vyhledání spojení.

Zhodnocení výsledků

Našim cílem bylo implementovat algoritmus tak, aby vyhledávání spojení proběhlo v co nejkratším čase a na uživatelském rozhraní se neprojevilo žádné zpoždění. Článek [7], zabývající se zpožděními v uživatelském rozhraní, definuje tři „magické“ konstanty pro lidský mozek:

- Hranice 0,2 sekundy. Pokud akce v uživatelském rozhraní proběhne do této doby, pak má uživatel pocit, že se tak stalo instantně a neregistruje žádné zpoždění.

- Hranice 1,0 sekundy. Pokud akce v uživatelském rozhraní proběhne do této doby, pak uživatel sice zaznamená zpoždění, ale stále je schopen udržet pozornost a je ochoten tolerovat zpoždění.
- Hranice 10 sekund. Tato hodnota je hranicí, kdy uživatel ztrácí o prováděnou akci zájem.

My, s dobou potřebnou pro vykonání akce pro vyhledání jednoho spojení, splňujeme požadavky první kategorie, což je skvělý výsledek. Typicky bude uživatel vyhledávat více spojení, například pět, pak náš algoritmus splňuje požadavek při nejhorším na druhou kategorii, což je také dobrý výsledek.

Pokud by se vyhledávání uskutečňovalo nad větší oblastí, pak by pravděpodobně bylo nutné algoritmus dále optimalizovat. Například by bylo zajímavé algoritmus, na základě návodu uvedeného v sekci 3.2.3, zparalelizovat a sledovat čas běhu na větším počtu procesorových jader.

7. Závěr

Prvotní myšlenka byla vytvořit jednoduchou konzolovou aplikaci, která bude schopna vyhledávat spojení a odjezdy ze stanice v jízdních řádech.

Postupem času se aplikace stala jádrem systému, dále byl vytvořen preprocesor dat. Integrace těchto dvou modulů a vytvoření klientské knihovny měly za následek vznik počítačové aplikace, která zpočátku byla schopna pracovat pouze v offline módu. Počítačová aplikace rozšířila původní konzolovou aplikaci o interaktivní uživatelské rozhraní, dále o možnost zobrazit mapu dopravní sítě a možnost zobrazit informace z dopravy.

Další etapou vývoje byl vznik serverové aplikace, což mělo za následek rozšíření funkcionality počítačové aplikace o online mód. V tomto období rovněž vznikla podpora pro oblíbené položky, počítačová a serverová aplikace již znaly svou finální podobu.

Nakonec byla vytvořena mobilní aplikace. Klientská knihovna již existovala pro počítačovou aplikaci, tudíž implementace mobilní aplikace sestávala pouze z tvorby uživatelského rozhraní.

Sjednocením zmíněných aplikací vznikl systém, který je obsahem této práce. Hlavním rysem celého systému je jeho snadná rozšiřitelnost a přizpůsobitelnost. Rozšiřitelností zde rozumíme možnost snadné implementace podpory pro nový formát či možnost implementace nové klientské aplikace bez nutnosti jakéhokoliv zásahu do systému. Přizpůsobitelnost zde chápeme jako možnost specifikovat vlastní zdroje dat, RSS zdroje pro informace z dopravy, možnost snadné úpravy způsobu zobrazování výsledků či rozšíření klientských aplikací o novou lokalizaci.

Do budoucna je zamýšleno zparalelizovat algoritmus, na základě popisu ze sekce 3.2.3, a snížit tak čas potřebný na vyhledání spojení. Dále by bylo vhodné vyřešit problém s aktualizacemi ze sekce 4.4.

Jednou z priorit je rozšířit server o podporu vyřizování požadavků pomocí Web API.[1] Pak by bylo možné implementovat i webovou aplikaci, bez nutnosti omezovat se na jazyk ASP.NET.[2]

Implementace podpory pro JDF formát je také předmětem budoucích prací. S touto podporou by bylo možné sjednotit zdroje dat pro dráhu, regionální linkové autobusy a hromadnou dopravu velkých měst, což by mělo za následek vznik vyhledávače pokrývající celou republiku.

[1] Zamýšlená implementace podpory pro Web API je zanalyzována v Příloze A.4.5.

[2] Zamýšlená implementace webové aplikace je zanalyzována v Příloze A.4.6.

Reference

[1] GTFS Static Overview. `https://developers.google.com/transit/gtfs/`.

[2] Otevřená data PID. `https://pid.cz/o-systemu/opendata/`.

[3] Martin Mareš and Tomáš Valla. *Průvodce labyrintem algoritmů*. CZ.NIC, z. s. p. o., 2017.

[4] Daniel Delling, Thomas Pajor, and Renato Werneck. Round-Based Public Transit Routing. *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, 2012.

[5] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 2007.

[6] Google Maps Platform Documentation. `https://developers.google.com/maps/documentation/`.

[7] Robert B. Miller. Response time in man-computer conversational transactions. *International Business Machines Corporation*, 1968.

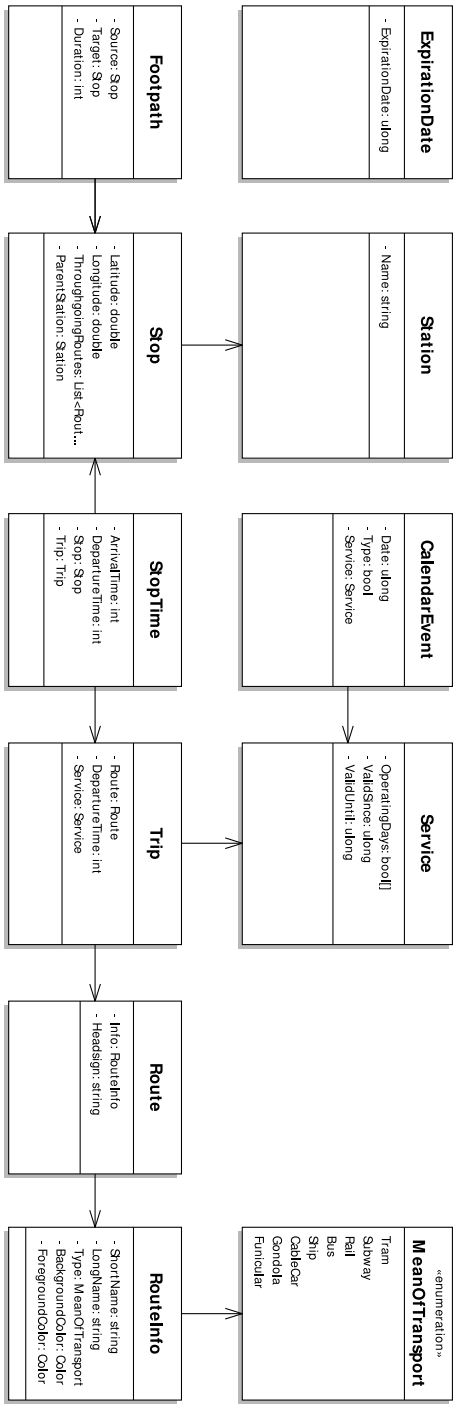
[8] Colin Robertson, Alma Jenks, and Mike Jones. Calling Native Functions from Managed Code. 2016.

[9] David Britch. Implementing a HybridWebView. 2019.

A. Přílohy

A.1 Datové formáty

A.1.1 Schéma datových formátů



Obrázek A.1: Sjednocení TFD a TFB formátu vizualizované zjednodušeným UML schématem.

A.1.2 Specifikace datových formátů

Všechny soubory mají následující formát, přičemž formát jednotlivých záznamů se liší v závislosti na položce:

```
{počet záznamů v souboru}
{záznam};...;{záznam};
```

Identifikátory jsou ve všech případech přirozená čísla a soubory jsou setříděné vzestupně dle klíčového identifikátoru pro daný soubor, existuje-li takový.

Formát TFD

Soubor „calendar.tfd“ obsahuje údaje o službách. Jeden záznam služby má následující formát:

```
{identifikátor služby};{pondělí};{úterý};{středa};{čtvrtek};
{pátek};{platí od (YYYYMMDD)};{platí do (YYYYMMDD)}
```

Soubor „calendar_dates.tfd“ obsahuje údaje o výjimečných situacích. Jeden záznam výjimečné situace má následující formát:

```
{identifikátor služby};{datum (YYYYMMDD)};{binární hodnota
typu události}
```

Soubor „footpaths.tfd“ obsahuje údaje o přestupech. Jeden záznam přestupu má následující formát:

```
{trvání v sekundách};{identifikátor výchozí zas.};{identifikátor
cílové zas.}
```

Soubor „routes.tfd“ obsahuje údaje o trasách linek. Jeden záznam trasy linky má následující formát:

```
{identifikátor trasy};{identifikátor linky};{počet zastávek v
trase};{počet jízd pro danou trasu};{směr}
```

Soubor „routes_info.tfd“ obsahuje údaje o linkách. Jeden záznam linky má následující formát:

```
{identifikátor linky};{označení linky};{dlouhý název linky};
{dopravní prostředek};{barva pozadí (RGB)};{barva popředí (RGB)}
```

Záznam `{dopravní prostředek}` může obsahovat následující číselné hodnoty:

```
Tram = 1, Subway = 2, Rail = 4 Bus = 8, Ship = 16,
CableCar = 32, Gondola = 64, Funicular = 128
```

Soubor „stations.tfd“ obsahuje údaje o stanicích. Jeden záznam stanice má následující formát:

```
{identifikátor stanice};{název stanice}
```

Soubor „stop_times.tfd“ je setříděný vzestupně dle času příjezdu a obsahuje údaje o zastavení jízd linek. Jeden záznam zastavení jízdy linky má následující formát:

```
{identifikátor jízdy linky};{identifikátor zastávky};
{čas příjezdu (sekundy od odjezdu jízdy linky)};{čas odjezdu
(sekundy od odjezdu jízdy linky)}
```

Soubor „stops.tfd“ obsahuje údaje o zastávkách. Jeden záznam zastávky má následující formát:

```
{identifikátor zastávky};{identifikátor stanice}
```

Soubor „trips.tfd“ je setříděný vzestupně dle času odjezdu a obsahuje údaje o jízdách linek. Jeden záznam jízdy linky má následující formát:

```
{identifikátor jízdy linky};{identifikátor služby};{identifikátor
trasy linky};{čas odjezdu (sekundy od půlnoci)}
```

Formát navíc obsahuje soubor „expires.tfd“, který obsahuje jediný řádek, kde je ve formátu `YYYYMMDD` uveden datum expirace daných dat.

Formát TFB

Soubor „routes_info.tfb“ obsahuje údaje o linkách. Jeden záznam linky má následující formát:

```
{identifikátor linky};{označení linky};{dopravní prostředek};
{barva pozadí (RGB)};{barva popředí (RGB)}
```

Záznam `{dopravní prostředek}` může obsahovat následující číselné hodnoty:

```
Tram = 1, Subway = 2, Rail = 4 Bus = 8, Ship = 16,
CableCar = 32, Gondola = 64, Funicular = 128
```

Soubor „stations.tfb“ obsahuje údaje o stanicích. Jeden záznam stanice má následující formát:

```
{identifikátor stanice};{název stanice}
```

Soubor „stops.tfb“ obsahuje údaje o zastávkách. Jeden záznam zastávky má následující formát:

```
{identifikátor zastávky};{identifikátor stanice};{zeměpisná
šířka};{zeměpisná délka};{projíždějící linky}
```

Záznam `{projíždějící linky}` má následující formát:

```
{identifikátor linky}‘...‘{identifikátor linky}‘
```

Neobsahuje-li zastávka žádnou linku, která jí projíždí, pak je záznam prázdný.

A.2 Struktury XML dokumentů

A.2.1 Konfigurační soubory

Serverová aplikace

```
<Timetables>
  <RouterPort>Celé číslo</RouterPort>
  <DepartureBoardPort>Celé číslo</DepartureBoardPort>
  <BasicDataPort>Celé číslo</BasicDataPort>
  <CoefficientUndergroundTransfersWithinSameLine>
    Desetinné číslo
  </CoefficientUndergroundTransfersWithinSameLine>
  <CoefficientUndergroundTransfersWithinDifferentLines>
    Desetinné číslo
  </CoefficientUndergroundTransfersWithinDifferentLines>
  <CoefficientUndergroundToSurfaceTransfer>
    Desetinné číslo
  </CoefficientUndergroundToSurfaceTransfer>
  <MaximalDurationOfTransfer>
    Celé číslo (sekundy)
  </MaximalDurationOfTransfer>
  <AverageWalkingSpeed>
    Desetinné číslo (metry za sekundu)
  </AverageWalkingSpeed>
  <DataFeeds>
    <DataFeed>
      <Name>Název datové sady, nepovinné</Name>
      <Link>URL adresa datové sady</Link>
    </DataFeed>
    ...
  </DataFeeds>
</Timetables>
```

Mobilní aplikace

```
<Timetables>
  <Language>Název lokalizace</Language>
  <ExtraEventsUri>Adresa URL k RSS s mimořádnostmi</ExtraEventsUri>
  <LockoutsUri>Adresa URL k RSS s výlukami</LockoutsUri>
  <ServerIp>Adresa serveru ve formátu IPv4 nebo "undefined"</ServerIp>
  <RouterPort>Celé číslo</RouterPort>
  <DepartureBoardPort>Celé číslo</DepartureBoardPort>
  <BasicDataPort>Celé číslo</BasicDataPort>
</Timetables>
```

Počítačová aplikace

```
<Timetables>
  <Language>Název lokalizace</Language>
  <OfflineMode>Hodnota "true" nebo "false"</OfflineMode>
  <ExtraEventsUri>Adresa URL k RSS s mimořádnostmi</ExtraEventsUri>
  <LockoutsUri>Adresa URL k RSS s výlukami</LockoutsUri>
  <FullDataUri>Adresa URL ke zdroji dat</FullDataUri>
  <ServerIp>Adresa serveru ve formátu IPv4</ServerIp>
  <RouterPort>Celé číslo</RouterPort>
  <DepartureBoardPort>Celé číslo</DepartureBoardPort>
  <BasicDataPort>Celé číslo</BasicDataPort>
</Timetables>
```

A.2.2 Odpovědi ze serveru

Struktura pro nalezená spojení

```
<?xml version="1.0" encoding="utf-16"?>
<RouterResponse
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <CreatedAt>Datum a čas vytvoření odpovědi dle ISO8601</CreatedAt>
  <Journeys>
    <Journey>
      <JourneySegments>
        <JourneySegment xsi:type="TripSegment">
          <SourceStopID>Identifikátor výchozí zastávky</SourceStopID>
          <TargetStopID>Identifikátor cílové zastávky</TargetStopID>
          <DepartureDateTime>Dle ISO8601</DepartureDateTime>
          <ArrivalDateTime>Dle ISO8601</ArrivalDateTime>
          <Headsign>Směr jízdy linky</Headsign>
          <LineLabel>Označení linky</LineLabel>
          <Outdated>Hodnota "true" nebo "false"</Outdated>
          <LineName>Název linky</LineName>
          <LineColor Hex="Formát #{0xR}{0xG}{0xB}">
            <R>Hodnoty 0-255</R>
            <G>Hodnoty 0-255</G>
            <B>Hodnoty 0-255</B>
          </LineColor>
          <LineTextColor Hex="Formát #{0xR}{0xG}{0xB}">
            <R>Hodnoty 0-255</R>
            <G>Hodnoty 0-255</G>
            <B>Hodnoty 0-255</B>
          </LineTextColor>
          <MeanOfTransport>
             Hodnota "Tram", "Subway", "Rail", "Bus", "Ship",
             "CableCar", "Gondola" nebo "Funicular"
          </MeanOfTransport>
          <IntermediateStops>
            <IntermediateStop>
              <Arrival>Příjezd do dané mezizastávky dle ISO8601</Arrival>
```

```
            <StopID>Identifikátor mezizastávky</StopID>
          </IntermediateStop>
          ...
        </IntermediateStops>
      </JourneySegment>
      <JourneySegment xsi:type="FootpathSegment">
        <SourceStopID>Identifikátor výchozí zastávky</SourceStopID>
        <TargetStopID>Identifikátor cílové zastávky</TargetStopID>
        <DepartureDateTime>Dle ISO8601</DepartureDateTime>
        <ArrivalDateTime>Dle ISO8601</ArrivalDateTime>
      </JourneySegment>
      ...
    </JourneySegments>
  </Journey>
  ...
 </Journeys>
</RouterResponse>
```

Struktura pro nalezené odjezdy

```
<?xml version="1.0" encoding="utf-16"?>
<DepartureBoardResponse
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <CreatedAt>Datum a čas vytvoření odpovědi dle ISO8601</CreatedAt>
 <Departures>
   <Departure>
     <StopID>Identifikátor zastávky</StopID>
     <Outdated>Hodnota "true" nebo "false"</Outdated>
     <Headsign>Směr jízdy linky</Headsign>
     <LineLabel>Označení linky</LineLabel>
     <LineName>Název linky</LineName>
     <LineColor Hex="Formát #{0xR}{0xG}{0xB}">
       <R>Hodnoty 0-255</R>
       <G>Hodnoty 0-255</G>
       <B>Hodnoty 0-255</B>
     </LineColor>
     <LineTextColor Hex="Formát #{0xR}{0xG}{0xB}">
       <R>Hodnoty 0-255</R>
       <G>Hodnoty 0-255</G>
       <B>Hodnoty 0-255</B>
     </LineTextColor>
     <MeanOfTransport>
        Hodnota "Tram", "Subway", "Rail", "Bus", "Ship",
        "CableCar", "Gondola" nebo "Funicular"
     </MeanOfTransport>
     <DepartureDateTime>Dle ISO8601</DepartureDateTime>
     <IntermediateStops>
       <IntermediateStop>
         <Arrival>Příjezd do dané mezizastávky dle ISO8601</Arrival>
```

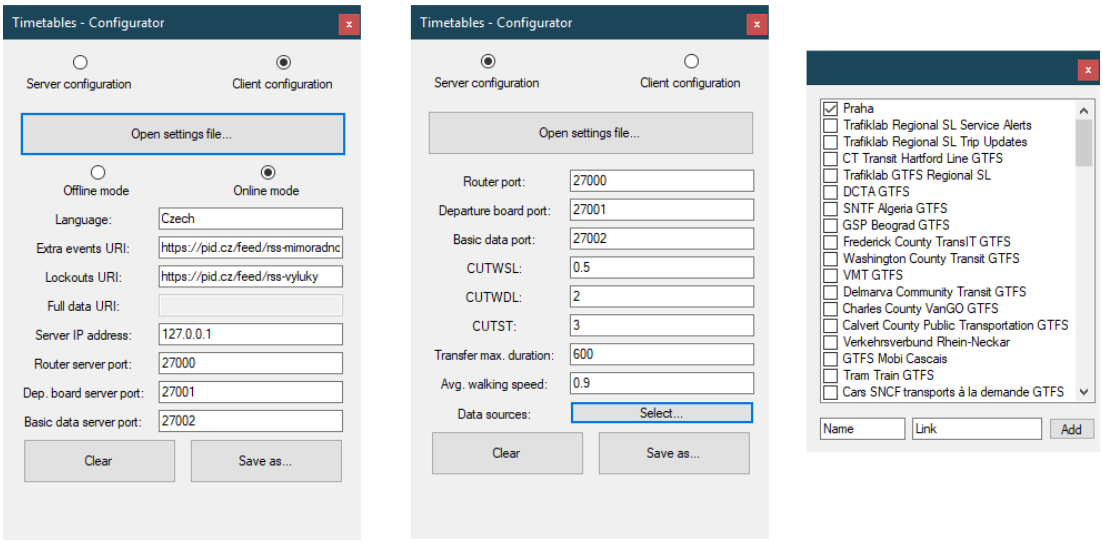
```
        <StopID>Identifikátor mezizastávky</StopID>
      </IntermediateStop>
      ...
    </IntermediateStops>
  </Departure>
  ...
 </Departures>
</DepartureBoardResponse>
```

A.3 Konfigurátor aplikací

Za účelem snadného vytváření a editace konfiguračních souborů byla vytvořena aplikace běžící nad platformou .NET. Tato aplikace umožňuje uživatelům načíst libovolný konfigurační soubor, editovat ho pomocí intuitivního uživatelského rozhraní a uložit ho v požadovaném formátu.

Pro jednoduchý výběr datových sad v rámci serverového konfiguračního souboru jsme implementovali podporu pro Transit Feed API.[1] Provozovatel tohoto API poskytuje JSON dokument, v němž jsou uvedeny desítky až stovky datových sad ve formátu GTFS. Je nutné mít na paměti, že formát těchto datových sad nemusí splňovat normu GTFS, validitu dat API nezaručuje. Tudíž se může stát, že konfigurátor uvede takovou datovou sadu, kterou naše aplikace nebudou umět zpracovat. V konfigurátoru je samozřejmě implementována možnost přidat do konfiguračního souboru vlastní datovou sadu, neboť se očekává, že tento postup konfigurace bude uživatelem preferovaný.

Uživatelské rozhraní konfigurátoru je vyobrazeno na Obrázku A.2.



Obrázek A.2: V okně vlevo je vidět příklad načteného konfiguračního souboru pro počítačovou aplikaci. V okně uprostřed je vidět příklad načteného konfiguračního souboru pro serverovou aplikaci. V okenním dialogu vpravo jsou vidět zdroje dat tak, jak je poskytuje Transit Feed API.

[1] `https://transitfeeds.com/api/`

A.4 Implementační detaily

A.4.1 Návod na implementaci parseru

Celé předzpracovávání dat obstarává následující metoda:

```
public static void GetAndTransformDataFeed<T>(params Uri[] urls)
       where T : IDataFeed
```

Jako typový parametr se jí předá třída, která obstará naparsování stažených dat. Jako parametry se metodě předají URL externích zdrojů dat, která chceme zpracovat. Z hlavičky metody je zřejmé, že typový parametr musí být třída implementující rozhraní `IDataFeed`. Dané rozhraní má následující podobu:

```
public interface IDataFeed {
    Calendar Calendar { get; }
    CalendarDates CalendarDates { get; }
    RoutesInfo RoutesInfo { get; }
    Stops Stops { get; }
    Stations Stations { get; }
    Trips Trips { get; }
    StopTimes StopTimes { get; }
    Routes Routes { get; }
    Footpaths Footpaths { get; }
    DateTime ExpirationDate { get; set; }
    void CreateDataFeed(string path);
    void CreateBasicData(string path);
}
```

Na začátek podotkněme, že implementaci metod pro vytvoření TFD, respektive TFB formátu musí dodávat až třídy implementující dané rozhraní, nicméně tělo těchto metod by mělo být vždy stejné. Správné řešení by bylo dodat těmto metodám defaultní implementaci, přičemž by byly metody v rozhraní virtuální a v případě nutnosti by bylo možné je předefinovat. Toto řešení bude možné až ve verzi jazyka C# 8.0, kde bude možné, mimo jiné, dodat metodám v rozhraní defaultní implementaci tak, jak je to možné například v jazyce Java. V době[2] sepisování tohoto textu a finálních úprav implementace ještě tato verze jazyka k dispozici nebyla.

Typy reprezentující položky v rozhraní jsou abstraktní třídy. Abstraktní třídy obsahují metody `Write(StreamWriter)` a `WriteBasic(StreamWriter)`, kde je to relevantní. Pomocí těchto metod se vytvoří data v TFD formátu, respektive v TFB formátu.

Pokud chceme vytvořit nový parser, je potřeba vytvořit třídu implementující rozhraní `IDataFeed`, pro GTFS parser se jedná o třídu `GtfsDataFeed`. Pokud vytvoříme instanci této třídy, pak se načtou GTFS data z umístění, které zadáme do konstruktoru třídy jako parametr.

Pro každou abstraktní položku je dále vhodné vytvořit potomka, který v konstruktoru naparsuje danou položku z formátu, pro který píšeme parser. V každé abstraktní položce je připravena instance datové struktury, typicky se jedná

[2] Q1/2019

červeno-černý strom nebo dynamické pole, do které lze přiřazovat instance jednotlivých záznamů. Tato kolekce je opatřena modifikátorem `protected`, položky v kolekci budou použity při volání metod pro zapsání nového formátu.

Uveďme příklad na GTFS parseru. Uvažme situaci, kdy chceme naparsovat data o zastávkách. Pak vytvoříme třídu `GtfsStops`, která bude dědit od abstraktní třídy `Stops`. Do jejího konstruktoru napíšeme parser, který soubor se zastávkami zpracuje tak, že pro každý řádek v souboru vytvoří záznam reprezentující zastávku a tento záznam přidá do připravené kolekce. V rámci konstruktoru třídy `GtfsDataFeed` vytvoříme instanci třídy `GtfsStops` a odkaz na tuto instanci přiřadíme do vlastnosti Stops. Takto postupujeme pro všechny položky.

Výpis položky Stops do TFD formátu následně zařídíme tak, že v těle metody `CreateDataFeed` zavoláme metodu `Write` na instanci Stops. Obdobně postupujeme pro TFB formát. Jak jsme zmínili v prvním odstavci, až bude možné uvést defaultní implementaci rozhraní, pak tento krok nebude nutný.

Do budoucna se počítá s úpravou klíčové metody tak, aby se typ externího zdroje specifikoval v parametru, tedy nikoliv v typovém parametru metody. Pak by bylo možné slučovat data z různých formátů. Za tímto účelem by bylo nutné rozšířit konfigurační soubory tak, aby každý datový zdroj rovněž obsahoval svůj výchozí formát. Metodě by se poté předávali jako parametry dvojice `(Uri, Type)`, kde bychom poté pomocí reflexe vytvářeli instance daného parseru.

A.4.2 Volání nativního kódu z managovaného kódu

Předpokládejme, že máme nativní knihovnu reprezentující jádro systému a chceme ji propojit s aplikací, která je psána nad .NET platformou. V takovém případě máme několik možností:

- Využít služby Platform Invoke. Tento přístup vyžaduje nativní knihovnu zkompilovanou do dynamicky linkované knihovny, v managovaném kódu se poté uvedou pouze hlavičky funkcí v DLL knihovně a do atributů se k funkcím přidá umístění dané knihovny. Je nutné zde řešit marshalování nativních objektů do managovaných objektů.
- Využít služby nějaké knihovny poskytující Platform Invoke, velmi oblíbená je knihovna CppSharp.[3]
- Využít služeb jazyka C++/CLI, který kombinuje dva typové systémy a poskytuje managovaný kód. V tomto případě C++/CLI projekt slouží pouze pro vytvoření wrapperů mezi nativním a managovaným kódem.

My jsme se rozhodli pro třetí možnost, neboť C++/CLI plně vyhovuje našim potřebám a není zde nutné žádné explicitní marshalování, respektive jazyk má pro marshalování zabudovanou podporu. Toto implicitní marshalování se označuje jako IJW[4] interoperabilita, případně C++ interoperabilita. Tento přístup je vývojáři preferovaný, jelikož je zde vyšší typová bezpečnost, je jednodušší na implementaci a poskytuje vyšší výkon než služby Platform Invoke.[5] Na druhou

[3] `https://github.com/mono/CppSharp`

[4] It Just Works

[5] `https://docs.microsoft.com/cpp/dotnet/using-cpp-interop-implicit-pinvoke`

stranu je tento přístup možný pouze za předpokladu, že máme k dispozici zdrojové kódy nativní knihovny. Více informací k dané problematice je k dispozici v článku [8], z nějž jsme čerpali informace.

Pro každou nativní třídu, k níž chceme přistupovat z managovaného kódu, vytvoříme *wrapper*. V našem případě se jedná o třídu, která ve svém konstruktoru načte data s jízdními řády do paměti a pomocí níž k nim lze přistupovat, a třídy jednotlivých požadavků, tedy třída pro požadavek na spojení a třída pro požadavek na odjezdy.

Wrapper je třída, ke které následně můžeme přistupovat z managovaného kódu. Typicky obsahuje, jakožto privátní datovou položku, ukazatel na instanci svého nativního ekvivalentu. Tato instance se vytvoří v konstruktoru wrapperu, odkud můžeme předat nativnímu konstruktoru parametry. Dále by měl mít wrapper finalizer, který uvolní místo v paměti využité nativní instancí objektu, a destruktor, který zavolá finalizer. Samotné wrappery dále kopírují rozhraní svých nativních ekvivalentů. V těle metod wrapperu se typicky pouze zavolá odpovídající metoda na instanci nativního typu a pokud má návratovou hodnotu, pak proběhne konverze z nativních typů do typů managovaných.

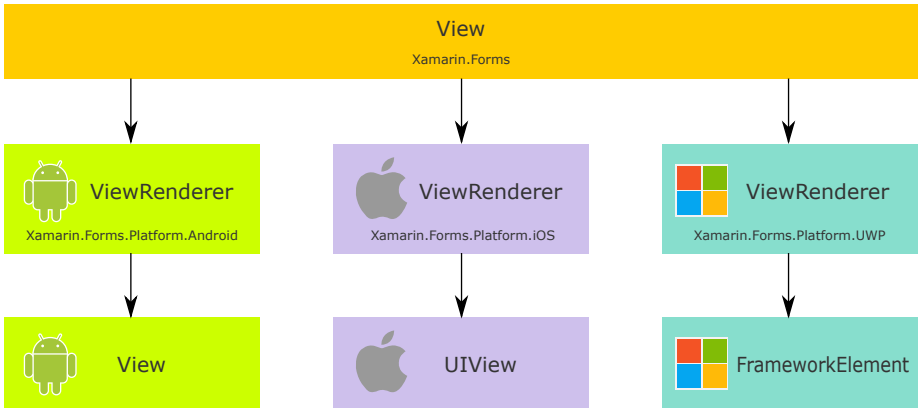
A.4.3 Volání managovaného kódu z Javascriptu

Většina implementací webových prohlížečů má pro interoperabilitu mezi kódem běžícím nad .NET a Javascriptem podporu. V .NET se očekává, že se do určité vlastnosti instance prohlížeče přiřadí *objekt pro skriptování*, což je instance třídy, jež obsahuje metody, které chceme volat z Javascriptu. Pokud chceme z Javascriptu zavolat nějakou metodu objektu pro skriptování, nejdříve se hledá shoda na základě názvu metody a počtu parametrů. Pokud není žádná metoda s daným názvem nalezena, vznikne v Javascriptu z prototypu objektu `Error` nový objekt a v exekuci kódu se do nejbližší `catch` klauzule, odchytávající objekt daného prototypu, nepokračuje. V opačném případě, je-li metoda nalezena, se z parametrů v Javascriptu vytvoří v .NET odpovídající primitivní typy a metoda se zavolá. Kromě primitivních typů je zde podpora i pro řetězce. Pokud má metoda návratovou hodnotu, je provedena konverze tentokrát opačným směrem a výsledek se předá zpět do Javascriptu. Kdyby nastal problém při zpracovávání parametrů, popřípadě kdyby došlo k výjimce v .NET, je scénář stejný, jako kdyby daná metoda neexistovala.

V počítačové aplikaci, psané nad Windows Forms, vše funguje tak, jak jsme si až doposud popsali. V mobilní aplikaci, psané nad Xamarin Forms, se situace krapet liší, neboť podporu pro náš záměr mají pouze nativní prohlížeče pro jednotlivé platformy (Android, iOS a UWP), zatímco třída webového prohlížeče od Xamarin Forms tuto přímou podporu nemá a neexistuje ani žádný přímočarý plugin, který by ji implementoval.

Nyní zjednodušeně popíšeme základní konstrukci webového prohlížeče v Xamarin Forms, podporující objekt pro skriptování, tak, jak ji popisuje článek publikovaný společností Microsoft [9], kde nalezneme i implementační detaily. Pokud chceme dosáhnout multiplatformnosti mobilní aplikace, je tento přístup nezbytný.

V Xamarin Forms jsou všechny třídy, jejichž instance lze zobrazit uživateli



Obrázek A.3: Znázornění vztahů mezi multiplatformní `View` třídou a jejími nativními ekvivalenty.

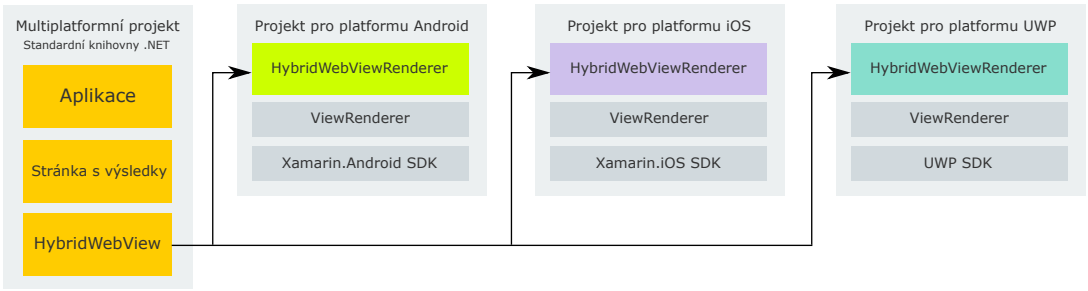
v uživatelském rozhraní, potomkem třídy `View`. Každá taková třída má pro danou platformu přiřazený nějaký *renderer*, což je třída, která danému ovládacímu prvku přiřadí jeho nativní ekvivalent. Uveďme si příklad pro platformu Android. Vznikne-li požadavek na zobrazení nějakého potomka třídy `View`, vznikne instance odpovídajícího rendereru pro daný ovládací prvek. Z tohoto rendereru dále vznikne nativní `View` objekt, který se uživateli zobrazí na obrazovce (viz Obrázek A.3). Zjednodušeně řečeno, pokud bychom chtěli zobrazit na obrazovce instanci multiplatformního webového prohlížeče, renderer zajistí, aby se nám zobrazil prohlížeč tak, jak je implementovaný v systému Android. Renderer může provádět i jiné činnosti, například související nastavení. Instance rendererů vznikají automaticky, vývojář k nim nikdy nepřistupuje přímo.

Již jsme zmínili, že podporu objektu pro skriptování mají pouze nativní prohlížeče, zatímco multiplatformní nikoliv. Za tímto účelem jsme se rozhodli rozšířit již existující multiplatformní prohlížeč. Jestliže chceme volat managovaný kód z Javascriptu, je potřeba, aby rozšířený prohlížeč uchovával odkazy na metody, které chceme volat z Javascriptu. Mimo odkazy by si dále měl pamatovat jejich názvy, abychom k nim mohli přistupovat. Při instanciaci našeho rozšířeného prohlížeče je potřeba, abychom zaregistrovali metody, jež chceme volat z Javascriptu.

V tento okamžik máme k dispozici prohlížeč s podporou, která se podobá objektu pro skriptování. Nicméně není možné ho zobrazit uživateli, neboť neexistuje renderer, který by našemu prohlížeči přiřazoval nativní objekt dané platformy. Proto je třeba ho pro každou z platforem vytvořit. Naše mobilní aplikace je určena pouze pro Android platformu, tudíž zde popíšeme princip tvorby rendereru pro Android. Základní schéma je k nahlédnutí v Obrázku A.4.

Každý vlastní renderer v Xamarin Forms musí dědit od připravené třídy `ViewRenderer<T1,T2>`, kde `T1` je třída, pro niž vytváříme renderer, a `T2` je třída, která reprezentuje ekvivalentní nativní třídu na dané platformě. Náš renderer zajistí, že se vytvoří instance nativního webového prohlížeče s daty našeho rozšířeného multiplatformního prohlížeče. Dále se nativnímu prohlížeči přiřadí instance třídy, kterou budeme nazývat *most do Javascriptu*. Jedná se o ekvivalent objektu pro skriptování.

Most do Javascriptu je, v případě platformy Android, třída, která musí dědit od základní třídy z objektové hierarchie Javy. Mimo odkaz na instanci našeho



Obrázek A.4: Znázornění vztahů mezi rozšířeným prohlížečem, který představuje třída `HybridWebView`, a jednotlivými vlastními renderery na všech platformách.

rendereru bude obsahovat metodu, která se bude volat z Javascriptu. Užitím této metody se budou nepřímo volat všechny zaregistrované metody v našem rozšířeném prohlížeči, neboť z rendereru lze získat danou instanci našeho rozšířeného prohlížeče.

Bylo by možné implementovat i přímé volání našich metod, nicméně pak by všechny naše metody musely být zahrnuty ve třídě mostu do Javascriptu, což by mohlo vést k duplikaci kódu, neboť most do Javascriptu je pro každou z platforem unikátní. Rovněž by bylo možné do našeho rozšířeného multiplatformního prohlížeče implementovat přímou podporu objektu pro skriptování, kde by se jen do určité vlastnosti přiřadila instance dané třídy s našimi metodami, bez nutnosti registrace callbacků. Pak bychom museli k přístupu k metodám využít reflexi, kde bychom v objektu hledali metody dle jejich názvu, což by ale vedlo k dalšímu zpomalení při volání metod.

A.4.4 Návod na rozšíření mobilní aplikace pro běh na jiné platformě

V současné verzi řešení je implementována podpora pro platformu Android. Mobilní aplikace lze snadno rozšířit i pro běh na jiné platformě, například iOS či UWP. Pro každou platformu je potřeba vytvořit nový projekt a odkázat se v něm na projekt multiplatformní.

Multiplatformní projekt obsahuje třídu `PlatformDependentSettings`, která má `public` modifikátor přístupnosti. Obsahuje položky, jejichž funkcionalita se liší v závislosti na platformě. Při startu aplikace je nutné tyto položky z projektu pro danou platformu nastavit, protože se jedná o volání funkcí, které nejsou dostupné z běžných knihoven .NET. Poskytuje je totiž API dané platformy, které je z multiplatformního projektu nedostupné. My i přes to API funkce dané platformy z multiplatformního projektu voláme, a to užitím delegátů. Jedná se o tyto delegáty:

- Delegát funkce, která vrátí seznam všech lokalizačních souborů v určené složce.

- Delegát funkce, která pro danou instanci třídy `FileInfo` vrátí proud dat pro čtení z daného souboru. Toto používáme pro čtení souborů ze složky se zdroji aplikace.

- Delegát funkce, která daný textový řetězec zobrazí na obrazovce ve formě informativní zprávy.

- Delegát funkce pro zobrazení dialogu. Mezi parametry této funkce se řadí textový řetězec, který bude zobrazen v dialogu, a delegát funkce, která přijímá jako parametr textový řetězec. Daná funkce bude zavolána po potvrzení dialogu, kde hodnota parametru bude zadaná hodnota v dialogu. Toto využíváme při nastavování IP adresy serveru.

Je také potřeba přiřadit položce `BasePath` hodnotu, která reprezentuje umístění v souborovém systému, kde se nachází složka s daty aplikace.

Dále je nutné napsat vlastní renderer pro rozšířený multiplatformní prohlížeč, který volá managovaný kód z Javascriptu. Tvorbu vlastního rendereru jsme popsali v sekci A.4.3, dále je vhodné řídit se článkem, který se danou problematikou zabývá [9].

A.4.5 Návrh podpory pro Web API

Pokud chceme implementovat podporu pro Web API, máme dvě možnosti:

- Rozšířit již existující serverovou aplikaci o tuto podporu. Vzhledem k existujícím klientským aplikacím a API, které server poskytuje, by nebylo vhodné aplikaci reimplementovat, muselo by se jednat o rozšíření již existujícího API.

- Nezasahovat do již existující serverové aplikace a vytvořit aplikaci novou, která bude poskytovat Web API. Taková aplikace by přijímala požadavky od klientů prostřednictvím protokolu HTTP, delegovala je již existujícímu serveru, z již existujícího serveru přijala odpověď na požadavek a ve formě XML či JSON dokumentu by požadavek odeslala klientovi. Jednalo by se o jakéhosi prostředníka mezi klientskou aplikací a již existujícím serverem.

Naše implementace počítá s druhou variantou, protože nechceme zasahovat do fungujícího systému, pouze ho rozšířit. Nová aplikace bude z pohledu již existujícího serveru pouze dalším klientem. Zároveň bude tato aplikace serverem pro aplikace, které budou chtít využívat služeb Web API. Dalším důvodem může být fakt, že nová aplikace může běžet kdekoliv v síti.

Při implementaci prostředníka bude nutné využít služeb jazyka ASP.NET, aby se prostředník dokázal připojit k serveru. Pro prostředníka tedy vytvoříme nový konfigurační soubor, jehož obsahem bude IP adresa serveru a koncové body jednotlivých služeb.

Implementaci prostředníka je možno pojmout jako implementaci další klientské aplikace, která své služby poskytuje dále. Za tímto účelem můžeme využít části již existující klientské knihovny.

Požadavky se prostředníkovi budou odesílat užitím metody GET, do parametrů vložíme parametry daného požadavku. Jinou možností je metoda POST, ta se využívá zejména v případě, kdy prostřednictvím HTTP protokolu posíláme citlivá data, což není náš případ.

Prostředníka budeme implementovat pomocí MVC vzoru, kde:

- Controller implementujeme pro všechny služby tím způsobem, že necháme controller zpracovat parametry specifikované v URL. Na základě těchto parametrů se vytvoří instance struktur, které klientské aplikace využívají pro komunikaci se serverem.

- Model bude tvořit odeslání požadavku na server a přijetí odpovědi ze serveru.

- View zobrazí serializovanou odpověď uživateli. V rámci ASP.NET budeme předávat objekt přijatý ze serveru, na základě hlavičky HTTP požadavku se dále daná odpověď serializuje do požadovaného formátu.[6]

Problém nastává v případě požadavku na aktualizaci základních dat. Předpokládá se, že Web API bude využívat zejména webová aplikace. Implementace webové aplikace se může lišit, mohlo by existovat dokonce více webových aplikací, zejména kdybychom prostředníka pro Web API zpřístupnili veřejnosti. V takovém případě by každý vývojář své webové aplikace musel implementovat podporu pro formát TFB, což by mohlo stát spoustu práce.

Proto nebudeme pro prostředníka implementovat podporu pro požadavek na aktualizaci základních dat. Místo toho implementujeme podporu pro požadavky, které by mohly vývojáři webové aplikace přijít vhod. Na základě těchto požadavků se budou v prostředníkovi vyhledávat v základních datech požadované položky. Mezi tyto požadavky řadíme:

- Požadavek na výpis všech stanic, jejichž jméno začíná na daný řetězec. Odpověď budou tvořit identifikátory a názvy daných stanic.

- Požadavek na výpis všech linek, jejichž identifikátor začíná na daný řetězec. Odpověď budou tvořit identifikátory a označení daných linek.

- Požadavek na výpis všech zastávek. Odpověď budou tvořit identifikátory, názvy a GPS souřadnice zastávek.

- Požadavek na výpis všech linek projíždějících danou stanicí. Odpovědi budou tvořit identifikátory a označení daných linek.

Je zřejmé, že pomocí těchto požadavků lze sestrojit úplná základní data tak, jak je poskytuje server ve formátu TFB. Tím pádem si může každá webová aplikace sestrojit a spravovat základní data sama, prostřednictvím těchto požadavků. Za tímto účelem by dále bylo vhodné přidat požadavek na získání verze základních dat v prostředníkovi, aby webová aplikace mohla data snadno aktualizovat. Tento přístup je ale nedoporučený, z důvodu již zmíněné pracné implementace. Doporučený postup je získávat potřebné údaje pomocí služeb prostředníka on-demand.

Požadavky na kontrolu existence stanice, respektive linky jsou implicitní součástí výše zmíněných požadavků. Pokud stanice, respektive linka neexistuje, pak je získaný seznam prázdný.

[6] V hlavičce lze specifikovat parametr `Accept`, hodnota tohoto parametru bude nejčastěji `application/xml` nebo `application/json`.

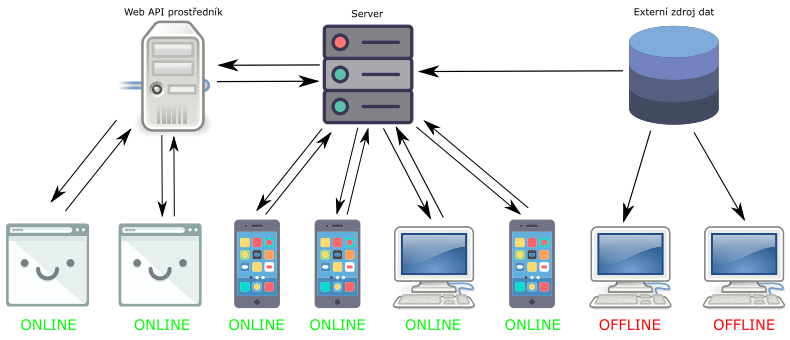
A.4.6 Návrh implementace webové aplikace

Existují dva druhy webových aplikací:

- Aplikace, které ke svému běhu potřebují server. Zdrojové kódy se vyskytují na webovém serveru, klient na ně zasílá požadavky, na serveru se generují statické webové stránky a ty se odesílají zpět klientovi. Nejčastěji se jedná o aplikace psané v jazycích PHP, ASP.NET či Node.js. Výhoda těchto aplikací je zejména ta, že jsou zdrojové kódy a implementační tajemství[7] odstíněna od uživatele.
- Aplikace, jejichž logika je implementována v klientovi. Typicky se jedná o aplikace, které ke svému běhu nepotřebují databázi. Jsou psané převážně v jazyku Javascript, pro stahování dat ze vzdálených API se využívá technologie AJAX. Výhoda těchto aplikací je ta, že ke svému běhu nepotřebují server, a tak mohou fungovat i offline.

Naše implementace počítá spíše s druhou variantou, za předpokladu, že existuje prostředník mezi webovou aplikací a serverem, který poskytuje Web API. Aplikaci by si pak každý uživatel mohl stáhnout a hostovat ve svém prohlížeči, nebylo by tedy nutné aplikaci hostovat na vzdáleném webovém serveru.

S prostředníkem by komunikovala prostřednictvím Fetch API, ten by jí posílal odpovědi na požadavky. Jelikož by aplikace byla psána v Javascriptu, tak by bylo vhodné, aby aplikace přijímala odpovědi ve formě JSON dokumentů a HTML elementy tvořila prostřednictvím Javascriptu. Jinou variantou by mohly být odpovědi ve formě XML dokumentů, pak bychom mohli využít již existující XSLT skripty.



Obrázek A.5: Schéma fungování celého systému pro jeden konkrétní zdroj dat.

A.4.7 Hlášení o chybách

Všechny aplikace mají implementovanou podporu pro ohlašování chyb vývojářům. Pokud dojde k neošetřené výjimce, automaticky se pošle vývojářům na e-mail hlášení, v jehož obsahu bude text výjimky, stacktrace a obsah konfiguračního souboru. E-mailová adresa, na kterou se mají hlášení odesílat, lze pro každou aplikaci, na úrovni zdrojových kódů, měnit pomocí vlastností. Dané vlastnosti se vyskytují vždy v souboru s entry pointem dané aplikace.

[7] Například přístupové údaje k databázi.

A.5 Obsah elektronické přílohy

Instalátory aplikací

Instalátory pro počítačovou a serverovou aplikaci distribuujeme ve formě MSI instalátoru. Po nainstalování jsou aplikace připravené k běhu. Instalátor zajistí stažení a nainstalování potřebných běhových prostředí.[8] Rovněž se vytvoří zástupce na ploše.

Instalátor pro mobilní aplikace distribuujeme ve formě APK archivu. Po nainstalování aplikace je aplikace připravena k běhu, nicméně je nutné jí dodat potřebná oprávnění.

Společně s aplikacemi dodáváme výchozí konfigurační soubory, které je možné libovolně měnit. Dané konfigurační soubory jsou součástí instalačních balíčků.

Zdrojové kódy

Součástí elektronické přílohy je Visual Studio řešení, které obsahuje zdrojové kódy. Kód je rozdělen do několika projektů:

- Core Library. Jedná se o implementaci jádra systému užitím jazyka C++.
- Client Library. Jedná se o implementaci klientské knihovny užitím jazyka C#.
- Managed Structures. Jedná se o managované ekvivalenty struktur z jádra systému.
- Managed Wrappers. Jedná se o most mezi nativním a managovaným kódem. Využíváme technologii C++/CLI.
- Data Preprocessor. Jedná se o implementaci předzpracování dat užitím jazyka C#.
- Configurator. Jedná se o spustitelnou aplikaci konfigurátoru. Je zde využit framework Windows Forms.
- Server Application. Jedná se o spustitelnou konzolovou aplikaci serveru.
- Desktop Application. Jedná se o spustitelnou počítačovou aplikaci. Je zde využit framework Windows Forms.
- Mobile Application. Jedná se o multiplatformní mobilní aplikaci. Je zde využit framework Xamarin Forms.
- Mobile Application Android. Jedná se o rozšíření multiplatformní mobilní aplikace tak, aby byla spustitelná na platformě Android. Je zde využit framework Xamarin Forms.
- Webbrowser Stuff. Jedná se o kolekci skriptů.
- Web Api Provider. Jedná se o jednoduchého zprostředkovatele Web API, příprava pro implementaci webové aplikace.

[8] Framework .NET 4.6.1 a běhové prostředí VC++ 2014.

- Benchmarks. Jedná se o program z kapitoly 6.

Všechny externí knihovny jsou spravované pomocí systému NuGet, který dané knihovny automaticky stáhne při kompilaci.

Po úspěšné kompilaci se automaticky nakopírují všechny potřebné externí soubory tam, kde je aplikace očekává.[9]

Řešení je rovněž ve správě verzí systému Git. Ve vzdálené repository se vždy nachází aktuální verze řešení. Tedy užitím příkazu `git pull` ve složce s řešením lze vždy získat aktuální verzi zdrojových kódů.

Ukázková data

Součástí elektronické přílohy jsou i ukázková data. Aplikace by měly pří prvním spuštění stáhnout data z přednastaveného zdroje, avšak může se stát, že v době čtení tohoto textu bude již daný externí zdroj nedostupný. V takovém případě je nutné extrahovat obsah archivu `sampledata.zip` do složky se spustitelným souborem serveru, respektive spustitelným souborem počítačové aplikace běžící v offline módu. Data jsou již předzpracována a datum expirace dat je nastaven na hodnotu nekonečno, tudíž je aplikace nebude nikdy aktualizovat.

[9]Konkrétně XSLT, CSS a JS skripty, dále výchozí konfigurační soubory.