# Jízdní řády

Lukáš Riedel

Specifikace

## Obsah

Poznámka: Text psaný *kurzívou* je velmi hrubý popis částí softwaru, které budou implementovány až v rámci bakalářské práce. V ročníkovém projektu (respektive zápočtových programech na C++ a C#) tedy tyto části zahrnuté nebudou. Podtrhnutý text bude zcela volitelná část, která oficiálně nebude obsahem ani RP, ani BP.

# 1 Úvod do problému

Cílem projektu je navrhnout a implementovat sadu aplikací pro pohodlné vyhledávaní v libovolných jízdních řádech, přičemž koncový uživatel bude moci vyhledávat jízdní řády pomocí desktopové, webové nebo *mobilní* aplikace. Desktopová aplikace, která bude mít user-friendly grafické rozhraní a jejíž cílová platforma bude Win32, bude uživateli v offline režimu umožňovat vyhledávání spojení v rámci jeho zvolených jízdních řádech a získávat informace o odjezdech z daných zastávek. Po přechodu do online režimu navíc bude moci sledovat aktuální situaci v provozu, tzn. mimořádnosti v dopravě a informace o výlukách. Online režim bude nutný pro občasnou aktualizaci jízdních řádů, kterou bude aplikace provádět sama v pravidelných intervalech (jednou za několik dnů), popřípadě, pokud dojde k překročení tohoto intervalu, kdykoliv se cílový stroj připojí k internetu. Pokud si uživatel nebude přát mít uložené jízdní řády na svém stroji, bude zde možnost se pomocí aplikace připojit ke vzdálenému serveru, kde budou uložena data. V tomto případě bude desktopová aplikace sloužit pouze jako prostředník mezi uživatelem a serverem. Tímto způsobem bude fungovat i webová aplikace. *Mobilní aplikace se bude lišit pouze tím, že bude možnost přejít do offline módu tím způsobem, že si uživatel bude moci stáhnout jízdní řády pro jeho oblíbenou trasu, které budou dostupné na několik dní dopředu, a bude mít tedy možnost vyhledávat spojení offline.* Součástí bude webová prezentace produktu napsaná v HTML5 a CSS3, kde budou všechny důležité informace.

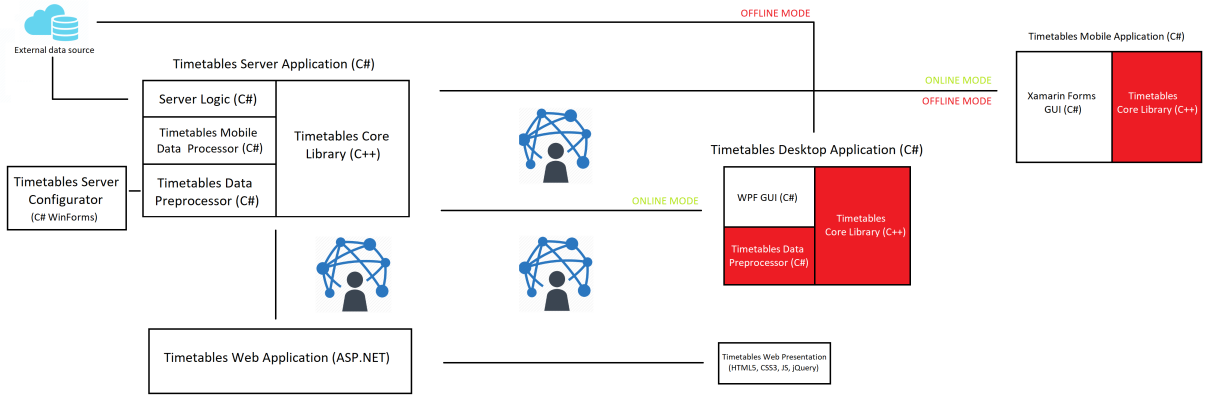
## 1.1 Dostupné zdroje

Jelikož je aplikace cílená pro Pražskou Integrovanou Dopravu, jsme nuceni jako hlavní zdroj dat použít informace o jízdních řádech v GTFS[1] formátu. Tento formát, navržený společností Google, je velmi variabilní. Cílem je optimalizovat řešení pro Prahu, nicméně aplikace také poskytne uživateli možnost jednoduše změnit cílové město - jízdní řády v tomtu formátu jsou totiž poskytovány i pro jiná velká (nejen) evropská města, například Londýn, Paříž nebo Řím. Algoritmů na hledání v jízdních řádech je celá řada, většina z nich používá nějakým způsobem modifikovaný Dijkstrův algoritmus. Toto řešení bude využívat RAPTOR [2] algoritmus, vyvinutý společností Microsoft, který nepotřebuje prioritní frontu a je dostatečně rychlý a efektivní. Algoritmus pracuje v kolech, kde v $k$-tém kole najde nejrychlejší spojení do všech zastávek, do nichž se můžeme dostat pomocí $k-1$ přestupů. Z toho plyne, že nehledáme spojení pouze do jedné cílové zastávky, nicméně do všech zastávek v grafu dopravní sítě. To ale nevadí, protože časová složitost zůstane stále velmi dobrá (lineární vůči trasám všech linek, každou procházíme pouze jednou) a spotřeba paměti nebude na dnešní poměry také nijak zásadní, neboť různá spojení rekonstruujeme pouze třemi pointery - odkaz na jízdu jedné konkrétní linky (tzv. trip), odkaz na nástupní stanici v daném tripu a odkaz na výstupní stanici v daném tripu. To znamená, že jedno spojení o $k$ přestupech spotřebuje (bez

overheadů všech použitých kontejnerů a tříd) pouze $24k$ bytů paměti na x64 platformě, $12k$ bytů paměti na x86 platformě. Ze software-inženýrského hlediska bude možnost jednoduše změnit vstupní (GTFS) formát, a sice přepsáním pouze jedné části aplikace, která se stará o parsování do jakéhosi mezi-formátu. Vše ostatní nebude třeba modifikovat.

# 2 Architektura aplikací

Cílová platforma desktopové aplikace bude Win32 kvůli podpoře WPF, konzolová serverová aplikace společně s konfigurátorem ve WinForms bude multiplatformní, tudíž server půjde pustit vzdáleně na jakékoliv distribuci Unixu mající nainstalovaný Mono framework. Pro funkčnost webové aplikace bude zapotřebí libovolný webový prohlížeč podporující ASP .NET. *Cílová platforma mobilní aplikace bude Android.* Hlavní jádro serverové, potažmo desktopové, aplikace bude psané v C++, vše ostatní v C#. Aplikace bude, co se knihoven týče, využívat funkce z STL (C++) a BCL (C#), pro interakci s uživatelem dále knihovnu boost na snadnou manipulaci s datem a časem z C++ a možná bude využita i knihovna Threading Building Block (TBB) od společnosti Intel, pro jednoduché zparalelizování routovacího algoritmu pomocí tasků (viz. níže).




## 2.1 Timetables Data Preprocessor

Tato část aplikace bude napsaná v C#. Bude zahrnuta v serverové aplikaci a rovněž bude potřebná pro "offline" fungování desktopové aplikace. Ve finálních aplikacích bude představovat jediné vlákno. Spustí se při prvním spuštění aplikace na daném stroji a následně v pravidelných intervalech, kdy bude k dispozici připojení k internetu a kdy aplikace nebude moc využívaná (jednou za několik dní, nebo pokaždé chvíli před expirací dat) kvůli aktualizaci dat. Bude kontrolovat validitu dat a její činnost by se dala rozdělit na tři části:

1. Stahování GTFS dat z předem navoleného zdroje dat a jejich následnou dekomprimaci. Z toho plyne, že podmínkou pro zdroj dat je ten, aby data byla v zip formátu. Bude možnost navolit více zdrojů dat tak, aby aplikace mohla fungovat v rámci nějaké větší oblasti, pro kterou nejsou dostupné data v celkové podobě, ale po kouskách. Ve světovém kontextu by toto dávalo velmi dobrý smysl například pro východní pobřeží

USA, konkrétně oblast od NYC po Washington, kde si, velmi pravděpodobně, bude každé město udržovat data o jízdních řádech pouze pro sebe. Jinými slovy, bude možnost data libovolně namergovat. Mergování dobu běhu preprocessoru příliš neovlivní, neboť data můžeme zpracovávat paralelně. V kontextu serverové aplikace, respektive konfigurátoru, se předpokládá, že se budou moci navolit adresy ručně a pokud dojde ke změně během běhu serveru, dojde ke změně zdrojů dat po prvním restartu serveru. Pro desktopovou aplikaci se předpokládá, že URL ke zdroji dat nebude zadrátováno do kódu, ale bude možnost ho rovněž nastavit ručně. Předpoklad je takový, že URL ke zdroji dat bude v nějakém (pro to určeném) textovém souboru, tudíž nebude možnost měnit jízdní řády interaktivně přímo z aplikace. Tento princip dá pokročilejšímu uživateli svobodu zvolit si, v rámci jakého města bude aplikace pracovat, a neznalému uživateli nedá možnost aplikaci rozbít. Tato adresa se bude moci dát nastavit přímo z instalačního balíčku.

2. Jak již bylo řečeno, GTFS formát je velmi variabilní. Některá povinná data jsou pro náš účel zbytečná, stejně tak se najdou nepovinná data, která potřebujeme. Příkladem nepovinných dat jsou například časy přestupů mezi jednotlivými zastávkami (tzv. footpaths). Tato část aplikace bude obecný GTFS formát přizpůsobovat tak, aby se spojení mohla vyhledávat dostatečně rychle. To znamená, že pokud bude soubor obsahující footpaths chybět, dojde k jejich dopočítání na základě GPS souřadnic zastávek. Předpokládá se, že vzhledem k tomu, že pomocí GPS souřadnic můžeme určit pouze vzdušnou vzdálenost mezi dvěma souřadnicemi, tak bude zvolena dostatečně malá průměrná rychlost chůze s tím, že budeme uvažovat pouze footpaths s dobou chůze do 10 minut. Časy přestupů z podzemní dopravy na povrchovou (a vice versa) jsou momentálně zdvojnásobeny, vůči ostatním časům. Časy přestupů v rámci metra, v rámci jedné stanice, v rámci jedné linky, jsou násobeny koeficientem 0.5. A to proto, že GPS souřadnice zastávek jsou nastaveny na začátky nástupišť, úhlopříčná vzdálenost je příliš velká. Časy přestupů v rámci metra, v rámci jedné stanice, v rámci různých linek, jsou násobeny koeficientem 1.5. Dalším příkladem činnosti této části může být například to, že v GTFS formátu jsou (povinně) uvedeny pouze údaje o jednotlivých nástupištích. Uživatel typicky nechce vyhledávat odjezdy pouze z jednoho nástupiště, ale ze všech nástupišť v rámci jedné zastávky (stanice). Aplikace tedy projde všechny nástupiště a na základě přesné shody jmen vytvoří jednotlivé zastávky (stanice) sdružující všechny nástupiště, které pod ně spadají. Do třetice uvedeme fakt, že aplikace vhodně předpočítá různé indexy a uvede data do stavu, abychom k požadovaným datům po načtení do paměti mohli efektivně přistupovat a aby nedocházelo k žádným memory leakům. Stručně řečeno, tato část načte obecná GTFS data do paměti a výstupem aplikace budou data (opět soubory), která budou plně přizpůsobena pro činnost hlavní aplikace. Předpokládá se, že na průměrně výkonném stroji poběží tato aplikace (vlákno) nejvýše 15-20 sekund (+ čas potřebný na stažení zip archivu).

3. Smazání zip archivu a GTFS databáze. Tyto soubory již nebudeme potřebovat, budeme mít vytvořené vlastní.

Důvod pro zvolení C# je takový, že vzhledem k povaze aplikace není třeba takový výkon,

aby musela být psaná v C++. Navíc, jelikož bude hlavní část aplikace napsaná rovněž C#, dojde ke snazšímu provázání těchto dvou aplikací. V neposlední řadě můžeme zmínit, že dekomprimace souborů lze provádět přímo z .NET kódu, čímž zlepšíme přenositelnost kódu - totiž v C++ žádná funkce na dekomprimaci souborů není a práce se soubory je tam rovněž náročnější. Na code-readability a přenositelnosti zde záleží více, když není potřeba žádný extrémní výkon. Výhoda této aplikace je rovněž ta, že ušetříme čas načítání hlavní aplikace (předpoklad zrychlení je 500%), potažmo i něco málo z paměti. Popřípadě, kdybychom chtěli aplikaci uzpůsobit i jiným formátům, než je GTFS, stačilo by upravit pouze tuto aplikaci, respektive připsat jednu třídu (viz. níže), a do žádné jiné aplikace bychom vůbec nemuseli zasahovat.

Data se z GTFS formátu načtou do paměti v přirozené formě. Musíme tak činit sekvenčně, neboť na sobě jednotlivá data různě závisí. GTFS formát je velmi obecný, pro nás mají význam pouze tyto položky:

1. Calendar - Obsahuje údaje o jednotlivých services. Díky tomu víme, kdy daný trip operuje. Každá service je indexovaná svým ID. Dále obsahuje binární hodnoty od pondělí do neděle, kde 1 signalizuje, že daná service v daný den operuje a 0 analogicky pro opačný případ. Můžeme reprezentovat buď polem boolovských proměnných, nebo pomocí bitů například v charové proměnné. Jelikož je počet všech services poměrně malý (většinou max 100-200), obětujeme 1 kB paměti pro reprezentaci v poli, abychom dosáhli lepší code-readability. Rovněž zde nalezneme dvě data, která nám udávají, v jakém rozmezí jsou jízdní řády platné.

2. Calendar Dates - Jeden záznam v tomto souboru nám říká něco o případných mimořádnostech v provozu dané service. Vždycky musí obsahovat ID dané service, datum mimořádnosti a typ mimořádnosti. Typ bude v našem případě pouze přidána/odebrána v daný datum, proto to můžeme reprezentovat boolovskou proměnnou. Při čtení tohoto souboru nevytváříme žádnou novou strukturu, pouze rozšiřujeme strukturu vytvořenou v předchozím bodě o mimořádnosti.

3. Routes - Obsahuje nejstručnější informace o provozu jednotlivých linek, jako je krátký název dané línky (například A, 22, 135...), dlouhý název linky (například Depo Hostivař - Nemocnice Motol), barvu linky v HEX formátu a typ linky, přičemž my si vystačíme s osmi typy (Tram = 0, Subway = 1, Rail = 2, Bus = 3, Ship = 4, CableCar = 5, Gondola = 6, Funicular = 7). Každá linka je rovněž indexována nějakým stringovým identifikátorem, proto je pro linky nejvhodnější strukturou unordered map/Dictionary, zatímco pro services to může být vector/List.

4. Stops - Obsahuje údaje o jednotlivých nástupištích, indexace pomocí stringů. Pro nás hrají roli pouze název zastávky a GPS souřadnice, konkrétně zeměpisná šířka a délka. Opět by bylo vhodné uchovávat si informace v unordered mapě/Dictionary. Při čtení dat je nutné dbát položku o typu lokace, kde uvažujeme pouze lokace typu 0, což je právě nástupiště. Zbylé dvě možnosti jsou stanice (ty si vytvoříme dle svých kritérií sami) a vchody/východy do/ze stanic.

5. Trips - Trip je trasa linky s časy odjezdu. Typicky máme daleko více tripů než linek. Například odjezd metra A ze stanice Depo Hostivař v 7:00:00 je jiný trip než odjezd

téže linky ze stejné stanice v čas 7:10:00. Tripy se indexují přirozenými čísly, tudíž nám jakožto struktura stačí vector/List. Dále zde máme odkazy na jednotlivé linky v podobně jejich identifikátorů, to stejné platí pro services. U tripů nám stačí si uchovávat jen headsign, což je název cílové stanice ("to, co je na přijíždějící tramvaji/metru/autobusu"). Trasu určíme z následujícího bodu.

6. Stop Times - Toto je nejrozsáhlejší položka všech dat a obsahuje údaje o odjezdech. Každá položka obsahuje identifikátor tripu, identifikátor zastávky, o kolikátou zastávku v řadě se jedná a časy příjezdu a odjezdu z dané zastávky. Ke každé zastávce přidáme ukazatel na každý stop time, který je asociován k dané zastávce (snadné vyhledávání departure board) a to stejné učiníme pro každý trip, jemuž daný stop time přísluší. Tak jsme schopni zrekonstruovat trasu tripu.

Struktura pro GPS souřadnice bude navíc obsahovat statickou metodu pro počítání doby chůze mezi dvěma body. Defaultní průměrná rychlost je nastavena na 0,8 m/s a metoda využívá Haversine formuli.
Pro správné fungování RAPTOR algoritmu ještě potřebujeme vytvořit trasy pro tripy, které sdílí stejné zastávky. Údaje o linkách přímo použít nemůžeme, protože například ačkoliv linka 22 v trase Vypich - Nádraží Strašnice sdílí stejné zastávky, tak tatáž linka v trase Bílá Hora - Nádraží Hostivař je v podstatě linka úplně jiná. Toto uděláme právě v rámci preprocessingu. Položku routes si označíme jako routes info a vytvoříme skutečnou položku routes, kde jedna route bude obsahovat posloupnost zastávek in-order, dále odkaz na route info a odkazy na všechny tripy, seřazené podle času odjezdu. To si přetřídíme již v rámci preprocessingu dat.
Pro algoritmus také potřebujeme vědět, jaké routes prochází jednotlivými stops. Budeme procházet všechny route a u každé zastávky do ni přidáme odkaz na danou route. Struktura pro tento přístup může být klidně vector/List. Toto si ale nemusíme předpočítávat, protože by to bylo stejně efektivní, jako kdybychom to počítali přímo při inicializaci dat z C++. Proto to budeme dělat pokaždé při spuštění hlavní aplikace.
Rovněž musíme dopočítat footpaths, tedy dobu přestupu mezi jednotlivými zastávkami. Původní plán byl takový, že GTFS data, která tuto volitelnou položku obsahují, ji budou také využívat. Nakonec vyšlo najevo, že bude lepší vše předpočítat ručně, aby to bylo konzistentní a měli jsme časy mezi všemi zastávkami, které potřebujeme. Jediná nevýhoda tohoto přístupu je, že čas běhu se společně s rostoucími zastávkami kvadraticky zvyšuje. Haversine formule potřebuje ke své činnosti goniometrické funkce, proto je tato část algoritmu skutečně velmi časově náročná. Údaje o footpaths budeme uchovávat u každého nástupiště (stop) ve formě mapy/SortedDictionary a to tak, že klíčem bude intová hodnota, za jak dlouho jsme schopni při chůzi průměrnou rychlostí dojít do zastávky, která se schovává pod tímto klíčem (v sekundách). Pro ušetření paměti (a konec konců i času) budeme do paměti ukládat pouze hodnoty nepřevyšující 600 sekund. Při mergování dat budeme uvažovat hodnoty nepřevyšující 900 sekund, neboť přestup mezi dvěma oblastmi může zabrat více času.

Po naparsování GTFS formátu do paměti aplikace dojde ke vhodnému výpisu dat do nových souborů, aby se daly použít k výpočtům. To bude probíhat v podstatě tak, že původní soubory okleštíme na nutné minimum (viz. předchozí text), popřípadě vytvoříme soubor s

footpaths a rozhodně vytvoříme soubor se stanicemi (stations). Při výstupu nástupišť (stop) tedy bude navíc každý záznam obsahovat intový identifikátor, kterým budou indexovaný stanice v nově vzniklém souboru. Dále soubor s routes přeměníme na routes info, soubor routes vytvoříme podle výše zmíněných pravidel a do každého tripu navíc přidáme identifikátor na danou route. Každá route navíc bude obsahovat ID na příslušné route info. Dále na první řádku každého nového souboru, pro nějž budeme později vytvářet v paměti kontejner, připíšeme počet záznamů, abychom si na danou velikost mohli kontejner před načítáním dat přizpůsobit a ušetřit tak čas i počet alokací. Nově vytvoříme soubor s jediným záznamem, v němž bude na první řádce napsáno, kdy jízdní řády expirují (konec platnosti první service v datech). Variabilitu GTFS při parsování ošetříme tím způsobem, že využijeme první řádky souboru, kde je pokaždé napsáno, jaká data daný sloupec obsahuje. Již na prvním řádku tedy ověříme, jestli daný soubor obsahuje vše, co potřebujeme. Následně, protože nikde není psáno, v jakém pořadí mají sloupce vystupovat, vytvoříme Dictionary s klíčem string a hodnotou int , kde string je název daného sloupce a int je index, tedy o kolikátý sloupec se jedná. K datům budeme přistupovat přes tento slovník. Vytvoříme interface IDataFeed, který bude obsahovat seznam všech souborů, které musí výsledná data obsahovat (viz. níže) a metodu CreateDataFeed, která tyto soubory vytvoří. Pro každý soubor vytvoříme abstraktní třídu, která bude definovat datové položky v daném souboru a způsob, jakým se mají vypisovat. Parsování GTFS souborů vyřešíme tak, že pro každý nový soubor vytvoříme třídu GtfsXXX, kde XXX je název abstraktní třídy, od níž se bude dědit. Takže abstraktní třídy budou obsahovat vše potřebné pro výpis nového formátu, zatímco zděděné třídy budou obsahovat vše potřebné pro parsování GTFS formátu. Děláme to takto proto, abychom umožnili snadné rozšíření o parsování nějakého nového formátu do formátu potřebného pro práci knihovny pro jízdní řády. Celou práci aplikace bude obsluhovat jedna generická funkce, kde se jako typ funkce zadá formát, z kterého se mají data parsovat. Respektive, tento formát bude reprezentován třídou, v našem případě GtfsDataFeed, která bude implementovat rozhraní IDataFeed. Konkrétně by vše popsané (zjednodušeně, bez paralelismu) mohlo vypadat třeba takto:

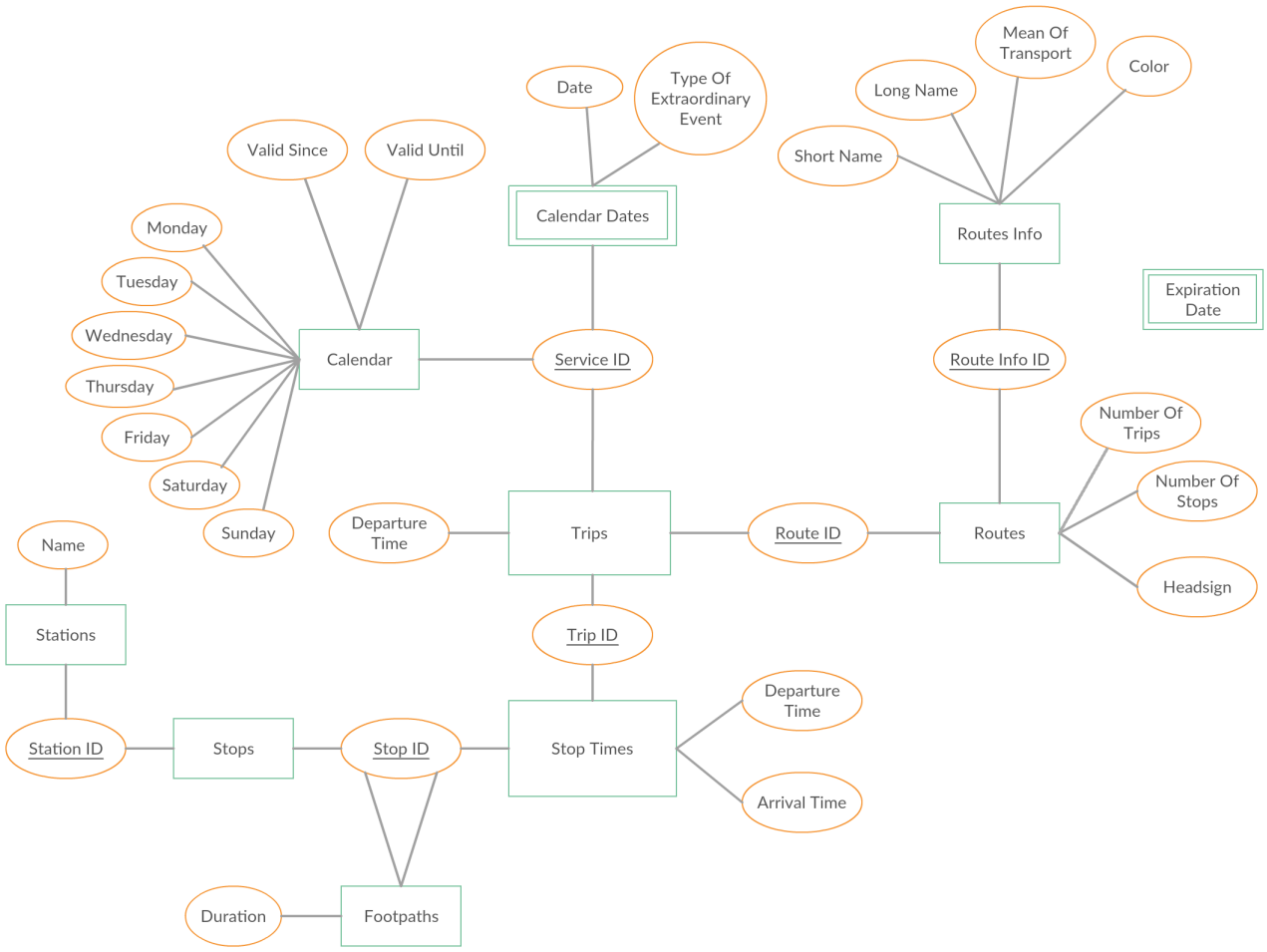
```
public interface IDataFeed {
    Calendar Calendar { get; }
    CalendarDates CalendarDates { get; }
    RoutesInfo RoutesInfo { get; }
    Stops Stops { get; }
    Stations Stations { get; }
    Footpaths Footpaths { get; }
    Trips Trips { get; }
    StopTimes StopTimes { get; }
    Routes Routes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

public static class DataFeed {
  public static event DataProcessingEventHandler DataProcessing;
  public static void GetAndTransformDataFeed<T>(params string[] urls) where T : IDataFeed
    {
    List<IDataFeed> dataList = new List<IDataFeed>();
```

```
21
22      for (int i = 0; i < urls.Length; i++)
23      {
24        try
25        {
26          Downloader.GetDataFeed($"{ i }_temp_data/", urls[i]);
27          dataList.Add((T)Activator.CreateInstance(typeof(T), (string)$"{ i }_temp_data/"));
28          DataProcessing?.Invoke($"Everything OK.");
29        }
30
31        catch (Exception ex)
32        {
33          DataProcessing?.Invoke($"ERROR. Showing details...");
34        }
35      }
36
37      IDataFeed mergedData = MergeMultipleDataFeeds(dataList);
38
39      mergedData.CreateDataFeed("data/");
40
41      mergedData.CreateBasicData("basic/");
42
43      for (int i = 0; i < urls.Length; i++)
44        Downloader.DeleteTrash($"{ i }_temp_data/");
45      }
46  }
```

Následuje seznam souborů, které bude nový formát obsahovat s názvy sloupců (ty ovšem nikde v souborech napsané nejsou, jedná se o "implementační tajemství"). Pro připomenutí platí, že na první řádce každého souboru je vždy počet záznamů, které následují. Záznamy mají vždy fixní velikost (fixní počet sloupců). Je to tak pro ušetření počtu alokací při načítání do vectorů a také proto, abychom mohli načíst všechna data pomocí for cyklu. Záznamy od sebe nejsou odděleny newliny, vše je napsané v jedné řádce. Jednotlivé sloupce jsou od sebe odděleny středníkem. Jednotlivé soubory budou mít příponu *.tfd (transit feed data).

Calendar - ServiceID, Monday ... Sunday, ValidSince, ValidUntil
CalendarDates - ServiceID, Date, TypeOfExtraordinaryEvent
RoutesInfo - RouteInfoID, ShortName, LongName, MeanOfTransport, Color
Stops - StopID, ParentStationID
Stations - StationID, Name
Footpaths - Duration, FirstStop, SecondStop
Trips - TripID, ServiceID, RouteID, DepartureTime (relative time since midnight, in seconds)
StopTimes - TripID, StopID, ArrivalTime, DepartureTime (both seconds since departure from the first stop in the trip)
Routes - RouteID, RouteInfoID, NumberOfStops, NumberOfTrips, Headsign
ExpirationDate - ValidUntil




Pro jakoukoliv aplikaci s GUI, tato část bude doplněna o preprocessing základních dat. Bude generovat pouze údaje o zastávkách, respektive stanicích, kde se budou tentokrát držet i GPS souřadnice (viz. dále, mobilní aplikace). A u každé zastávky budou vypsány linky, které jimi projíždí. Toto je nutné pro základní funkčnost aplikace, abychom mohli v rámci aplikace vybrat, jaké spojení vlastně chceme vyhledat, než dojde k odeslání požadavku na server a aby fungoval našeptávač pro stanice (viz. dále). Tato data budou mít následující formát. Platí stejná pravidla jako výše. Jednotlivé soubory budou mít příponu *.tfb (transit feed basic).

RoutesInfo - RouteInfoID, ShortName, MeanOfTransport, Color
Stops - StopID, ParentStationID, Latitude, Longitude, List Of All Throughoging Routes (separated by a backtick)
Stations - StationID, Name
ExpirationDate - ValidUntil

Během sepisováním této specifikace došlo k microbenchmarkům na datech pro Prahu u obou přístupů, a sice načítání dat pomocí knihovny z meziformátu a pak přímo z GTFS formátu. Samotný GTFS formát zabírá na disku 105 MB, zatímco meziformát pouze 35 MB. Po načtení GTFS formátu do paměti (v x86 režimu) aplikace využívá 160 MB, zatímco po

načtení meziformátu pouze 100 MB - je to dáno tím, že můžeme využívat efektivnější datové struktury (kvůli předpřipravené indexaci můžeme využít vector místo mapy, nějaké struktury můžeme úplně vypustit, nemusíme si například pamatovat GPS souřadnice zastávek...). Ještě uvedeme, že čas inicializace dat z GTFS formátu je 12 sekund, zatímco čas inicializace z meziformátu pouze 2-3 sekundy - to je proto, že si spoustu věcí předpočítáme již do meziformátu, ušetříme počet alokací díky rezervaci místa ve vectorech atd. Zbývá podotknout, že čas přípravy meziformátu je společně se stahováním GTFS dat zhruba půl minuta.

Ještě je zde problém s časem v GTFS formátu, a sice že můžou obsahovat i data převyšující čas 23:59:59. V GTFS formátu je čas 25:15:00 a 1:15:00 ekvivalentní, a oba to jsou validní časy. Je to tak proto, že v GTFS se o datech (téměř) vůbec nemluví, pouze o času a dnech v týdnu. Podle dne v týdnu určíme, kdy daný trip operuje. Nicméně může existovat trip, který vyjíždí v operační den ve 23:50:00 a do cílové stanice dojíždí další den v 0:10:00. Kdyby onen trip v ten další den neoperoval, byl by zde problém, protože by nikdy nedojel do cílové stanice. Proto je toto řešeno tak, že trip odjíždí v operační den ve 23:50:00 a dojíždí v "ten samý"operační den ve 24:10:00, což je v reálné situace samozřejmě nedefinovaný čas. Při operacích s časem v C# používáme klasickou a přirozenou strukturu DateTime, zatímco při práci v C++ si vytvoříme kopii DateTime z .NET, která bude obsahovat podobné funkce a čas se bude reprezentovat jako Unix timestamp, tedy v proměnné time_t si budeme uchovávat počet sekund od 1.1.1970 0:00. Časy reprezentujeme relativně. Konkrétně, čas odjezdu tripu (tzn. odjezd z první zastávky) je počet sekund od půlnoci daného dne. Takže například odjezd ve 12:00:00 bude mít hodnotu 43200 sekund. V rámci tripu, odjezd z jednotlivých zastávek určujeme relativně vůči odjezdu tripu. Takže například, pro stejný trip, který odjíždí ve 12:00:00 a ze třetí zastávky v pořadí odjíždí v 12:10:00, bude čas odjezdu u této zastávky nastavený na 600 sekund. Děláme to takto, protože chceme, aby knihovna byla v budoucnu použitelná i pro nějakou větší oblast, například celou republiku nebo Evropu. Jinak bychom mohli časy uchovávat absolutně, nicméně pak by délka jízdy jednoho tripu nesměla převýšit 24 hodin, což typicky v MHD nenastane. Toto je ale čistě programátorský problém, uživateli se vše bude zobrazovat v "normálních"časech.

## 2.2 Timetables Core Library

Tato část aplikace bude napsaná v C++ a bude plně optimalizovaná na výkon. Důvod pro zvolení C++ je tedy zřejmý. Aplikace bude zahrnovat dva základní druhy požadavků. Ještě podotkneme, že aplikace je schopna načíst i neaktuální jízdní řády a vyhledávat v nich - nicméně uživatel bude upozorněn na skutečnost, že nalezené spojení využívá neaktuální jízdní řády, a že by měl zvážit jejich aktualizaci.

1. Zobrazení prvních n odjezdů z dané zastávky v daný datum a čas. K tomu asi není co víc dodávat, najde se zastávka odpovídající danému jménu, která bude obsahovat reference na všechny nástupiště, které pod ni spadají. Každé nástupiště v sobě obsahuje údaje o všech odjezdech všech linek za jeden den. Při procházení odjezdů je nutné dbát, jestli je odjezd v daný datum relevantní, neboť o víkendy, státní svátky atd. platí jiné jízdní řády než ve všední dny. I na toto se pamatuje, totiž každý trip obsahuje položku,

kdy daný trip operuje (tzv. services). Najde se prvních n odjezdů splňující kritéria a vrátí se ve formě kontejneru obsahující nějakou developer-friendly datovou strukturu, jejíž obsah půjde snadno zobrazit z GUI.

2. Nalezení n nejrychlejších spojení mezi dvěma zastávkami v daný datum a čas, přičemž bude možnost navolit maximální počet přestupů a zvolit si průjezdné body. K tomu se bude využívat RAPTOR [2] algoritmus, o němž již byla řeč. Slušelo by se podotknout, že z testování společností Microsoft vyplynulo, že při vhodné implementaci a vhodných strukturách dokáže tento algoritmus nalézt libovolné spojení na grafu Londýna do 8 milisekund. Dijkstrův algoritmus je dle různých benchmarků až stokrát pomalejší. Výstup bude podobný jako v posledním případě. Algoritmus bude pracovat paralelně v rámci procházení tras, protože to je nejpomalejší a zároveň jediná možná zparalelizovatelná část algoritmu. Algoritmus bude pracovat paralelně jen tehdy, když to bude dávat smysl - například když počet tras k projití bude větší než 100. Bude to realizováno pomocí tasků z TBB knihovny, přičemž všechny trasy k projití se rovnoměrně rozdělí do n částí, kde n je počet fyzických jader procesoru (popř. i virtuálních, podporuje-li procesor HyperThreading). Toto číslo jednoduše zjistíme pomocí C++ 11. Každá část pak bude náležet jednomu vláknu, které bude sekvenčně procházet trasy, které mu přísluší.

Předpoklad je takový, že tato část bude s hlavní částí aplikace (myšleno s tou částí s GUI) provázána pomocí interoperability C++/CLI se C#. Tato část bude sloužit pouze k počítání výsledků, veškerá logika bude napsaná v C#. Tudíž z C# se budou inicializovat C++ objekty a volat C++ funkce. Budeme mít dvě třídy, konkrétně router (vyhledávání spojení) a departure_board (zobrazování odjezdů ze zastávky). Každá třída se dá rozdělit na tři části, proč tomu tak je, se dozvíme dále. Bez újmy na obecnosti, uveďme příklad pro departure_board.

1. Inicializace požadavku. Na žádost uživatele se vytvoří instance třídy departure_board, kde se v rámci konstruktoru nalezne (podle jména) stanice, kterou uživatel zadal. Dále se inicializuje čas odjezdu a počet odjezdů, kolik chce uživatel zobrazit. Konstruktor může vyhodit výjimku, když se například nenalezne požadovaná stanice.

2. Provedení požadavku. Po zavolání noexcept metody obtain_departure_board() se na základě dat v objektu naleznou požadované odjezdy.

3. Zobrazení požadavku. Po zavolání metody show_departure_board() se formou vhodně zvolené datové struktury předají data vypočítané v předchozím kroku.

Deklarace třídy departure_board tedy může vypadat například takto.

```
1 class departure_board {
2 private:
3   std::vector<Timetables::Structures::departure> found_departures_;
4   const Timetables::Structures::station& station_;
5   const Timetables::Structures::date_time& earliest_departure_;
6   const std::size_t count_;
7 public:
8   departure_board(const Timetables::Structures::data_feed& feed,
9   const std::wstring& station_name, const Timetables::Structures::date_time&
```

```
10    earliest_departure, const size_t count) : earliest_departure_(earliest_departure),
11    count_(count), station(feed.stations().find(station_name)) {}
12
13    void obtain_departure_board();
14
15    inline const std::vector<Timetables::Structures::departure>& show_departure_board()
16    { return found_departures_; }
17  };
```

Pro třídu router to bude dosti podobné, nicméně tam bude více privátních datových položek, z důvodu větší komplexnosti algoritmu.

## 2.3 Timetables Server Application

Aplikace bude napsaná v C# pro snadnou komunikaci po síti prostřednictvím .NET Remoting a kvůli lepší přenositelnosti. Bude se jednat o konzolovou aplikaci bez uživatelského rozhraní a měla by být multiplatformní. Z konzole půjdou zadávat různé základní příkazy, typu restart serveru, vynucená aktualizace jízdních řádů apod. Na konzoli se budou pouze zobrazovat chybové hlášky, popřípadě když bude docházet k aktualizacím dat, spouštění nových vláken a tak dále. Bude obsahovat preprocessing dat a hlavní knihovnu v C++, s kterou bude propojena pomocí C++/CLI technologie. Jediný účel této aplikace bude tedy ten, že bude zpracovávat požadavky ze sítě na vyhledání spojení/zobrazení odjezdů ze zastávky, počítat výsledky a zpět posílat odpovědi na požadavky, případě chybové hlášky. Bude to realizováno pomocí dvou, respektive čtyř front. Bez újmy na obecnosti uveďme příklad pro departure_board.

1. Přijde požadavek na zobrazení N odjezdů ze zastávky X v čas Y. Požadavek se zařadí do vstupní fronty. Až na něj přijde řada, dojde k inicializaci objektu. Pokud se inicializace nepodaří, přesune se do výstupní fronty a bude v sobě držet chybový stav.

2. Inicializace byla úspěšná. Vytvoří se nové vlákno, které bude mít požadavek na starosti. To na něm zavolá metodu obtain_departure_board(). Až metoda skončí, objekt se přesune do výstupní fronty a vlákno bude terminováno.

3. Na požadavek přišla ve výstupní frontě řada. Po síti se odešle zpět ke koncovému uživateli, kde se v GUI (desktopové, webové, nebo *mobilní* aplikace) zobrazí výsledek.

Server bude nastavitelný pomocí XML souboru. Zde budou uvedeny odkazy ke zdrojům dat a případě další údaje potřebné pro síť (například číslo, na kterém server naslouchá). Dále například interval, v jakém se mají data aktualizovat apod.

## 2.4 Timetables Server Configurator

Tato minimalistická aplikace napsaná v C# bude využívat technologii WinForms, které bude zahrnovat jediné okénko, pomocí kterého se bude konfigurovat server. Budou zde konfigurovatelné všechny položky z předchozí sekce. Jedná se o separátní aplikaci, protože při vzdáleném spouštění serveru se víc hodí mít pouze konzolovou aplikaci - tato bude obsahovat miniaturní GUI. Toto bude jen takový doplněk, XML soubor se dá samozřejmě přepisovat

"ručně". Aplikace bude využívat Transit Feed API[3], což je portál, kde je k nalezení více než 500 datových sad pro jízdní řády pro různé oblasti světa. Bude využíváno tím způsobem, že v konfigurátoru bude, mimo možnosti zadat URL k datovým sadám ručně, také možnost zobrazit si a vybrat jakoukoliv z datových sad na tomto portálu. To bude řešeno zvláštním okénkem, které bude implementováno nějakým list boxem. V tomto okénku budou názvy jednotlivých datových sad, přičemž v názvu datové sady se ve většině případů nachází název oblasti, kterou tato datová sada pokrývá. Po výběru oblasti se do konfigurátoru uloží URL adresa, na níž se datová sada nachází. Komunikace s Transit Feed API probíhá pomocí HTTP protokolu prostřednictvím GET/POST požadavků, odpovědí na tento HTTP požadavek je JSON dokument. Za tímto účelem bude pravděpodobně využita externí knihovna pro .NET s názvem Newtonsoft.Json, pro snadné parsování příchozích dat. Jediná nevýhoda je, že ačkoliv jsou data referencována na tomto portálu, tak to negarantuje jejich well-formedness.

## 2.5 Timetables Desktop Application

Aplikace bude zcela jistě naprogramována v C# a bude využívat technologie WPF. Aplikace bude uživateli poskytovat příjemné grafické rozhraní, kde jak pro zobrazování odjezdů, tak pro vyhledávání spojení, bude k dispozici "našeptávač"pro nalezení zastávky. Bude k dispozici hezky formátovaný RSS feed pro mimořádnosti v dopravě a informace o výlukách, který bude k dispozici při připojení k internetu. V nastavení bude možnost zvolit si barevný motiv (předpokládají se základní barvy) a do budoucna zde bude i možnost lokalizace aplikace, tzn. přepínání mezi anglickým a českým jazykem. Aplikace bude podporovat dva módy:

1. Offline mód. Tento mód nebude vyloženě offline, protože bude muset docházet k pravidelným aktualizacím jízdních řádů. Je to myšleno tak, že požadavky od uživatele se budou vyřizovat v rámci aplikace. V tomto módu bude spouštěn preprocessor dat, který bude stahovat data z externího zdroje (viz. výše) a přizpůsobovat data pro knihovnu v C++. To znamená, že v tomto módu bude mít aplikace v paměti načtená data pro jízdní řády a bude volat C++ funkce pro zpracování požadavků.

2. Online mód. V tomto módu bude aplikace spotřebovávat o mnoho méně paměti. Při běhu v tomto módu bude totiž aplikace obsahovat pouze údaje o stanicích a o tom, jaké linky jimi projíždějí. Jakékoliv uživatelovy požadavky se budou po síti posílat na server, kde bude docházet k jejich zpracování. Z toho tedy plyne, že aplikace bude nutně vyžadovat připojení k internetu a běžící server, kdesi v síti. V nastavení bude třeba zadat adresu k tomuto serveru. Předpoklad je takový, že pro předávání dat po síti budeme využívat službu .NET Remoting. Tedy v tomto módu se v žádném případě nebude pouštět preprocessor, ani volat funkce z C++ knihovny.

## 2.6 Timetables Web Application

Aplikace bude naprogramována v ASP .NET a bude určena k běhu na nějakém vzdáleném serveru, ne nutně na stejném serveru, kde bude běžet serverová aplikace pro jízdní řády. Bude se jednat v podstatě o aplikaci z předchozího bloku, která pracuje v online módu.

## 2.7 Timetables Mobile Data Processor

*Toto bude rozšíření serverové aplikace. Aplikace bude v C# a v rámci hlavní aplikace se bude jednat o jediné vlákno. Vstupem bude požadavek na vytvoření dat pro jednu trasu. Aplikace spustí vyhledávání spojení, typicky by se měla vyhledat spojení ve všech dnech v týdnu, ve všech denních dobách a i nějaké alternativy. Na základě výsledků tato aplikace vytvoří data pro danou trasu - na základě vygrepování všeho potřebného z hlavních dat, aby se na těchto datech mohl routovací algoritmus spouštět a vždycky (v každý den, každou denní dobu) mezi dvěma zadanými zastávkami nalezl několik optimálních spojení. Tato data si ve formátu ze sekce 2.1 uloží v zip formátu a do mobilní aplikace pošle odkaz, ze kterého si je mobilní aplikace stáhne. Tato forma cachování byla zvolena proto, aby se, v případě, že bude více uživatelů chtít údaje o stejné trase, nemusela data počítat neustále znova a ušetřili jsme tak čas. Podobně to bude fungovat pro zobrazení odjezdů ze zastávek. Data budou rovněž obsahovat timestamp, do kdy jsou validní.*

## 2.8 Timetables Mobile Application

*Aplikace napsaná v C# a Xamarin.Forms pro cílovou platformu Android (s možností snadného rozšíření na iOS) bude velmi podobná jako ta desktopová s tím rozdílem, že nebude obsahovat offline mód tak, jak byl popisován v desktopové variantě. To znamená, že veškeré požadavky na zobrazení informací o zastávkách nebo spojeních se budou řešit online a to tím způsobem, že prostřednictvím služby .NET Remoting bude posílat data do serverové aplikace, kde se budou vyhledávat spojení a po síti se vracet zpět do aplikace mobilní. Mobilní aplikace bude v její základní verzi pouze takový zobrazovač, prohlížeč... Bude také obsahovat našeptávač, RSS feed i nastavení - stejně jako desktopová aplikace. Při vyhledávání zastávek se budou zobrazovat nejbližší zastávky, což se určí na základě GPS souřadnic mobilního telefonu. Bude možnost vyhledat spojení z/do bodu "moje poloha", přičemž se při zvolení této možnosti vyhledá nejbližší zastávka a z ní se bude hledat spojení. Přibude zde ještě možnost zobrazit si mapu, kde bude vyznačena uživatelova aktuální poloha a ve formě připínáčků budou na mapě vyznačeny všechny zastávky v dané oblasti. Po kliknutí na nějaký takový připínáček se odešle požadavek na server na zobrazení odjezdů z dané zastávky v aktuální čas, a ty se na základě přijatých dat nad připínáčkem ve zkrácené formě (tj. čas odjezdu, označení línky a headsign tripu) zobrazí. Tato přednost aplikace bude velmi výhodná ve chvíli, kdy se uživatel nachází na nějaké větší stanici a oblast stanice nezná. Pomocí mapy jednoduše zjistí, z jakého nástupiště a v kolik hodin odjíždí linka, na kterou potřebuje nastoupit.*
*Bude zde i možnost uložit si oblíbené trasy, které se budou moci vyhledávat offline. To stejné bude platit pro odjezdy z jednotlivých zastávek. Bude to realizováno tak, že se po síti pošle požadavek do serverové aplikace a aplikace z předchozí sekce ho zpracuje. Nalezne nejvhodnější spojení (respektive odjezdy z dané zastávky) i nějaké alternativy, z dat vygrepuje nutné údaje o odjezdech z potřebných zastávek apod. a tato aplikace si je stáhne. Výhodou tohoto offline přístupu je fakt, že kompletní data jsou poměrně hutná, pro Prahu mají zhruba 40 MB. Pokud se vygrepuje pouze to potřebné pro jednu trasu, budeme se pohybovat ve velikostech stovek kB, spíše i méně. I tato data budou mít nějaký datum expirace, po expiraci dat již nebude oblíbená trasa (resp. odjezdy ze zastávek) k dispozici a bude nutné připojení k internetu, aby se data zaktualizovala. Po odznačení oblíbené trasy (resp. zastávky) dojde ke*

*smazání dat, ta se již nadále nebudou aktualizovat.*
*Samotné vyhledávání bude vyřešeno opět pomocí Timetables Core Library, propojenou s touto aplikací pomocí C++/CLI. Aplikace on-demand načte data z disku (pro jednu trasu, pro jednu zastávku), která jsou relevantní pro daný offline požadavek uživatele. Doba načítání nebude příliš velká, protože se bude jednat o velmi malá data. Offline požadavky se co do doby odezvy vyrovnají online požadavkům, neboť ačkoliv zde bude overhead způsobený načítáním dat, tak algoritmy budou pracovat mnohem rychleji než v případě serveru, protože budou pracovat na mnohem menších datech. Tímto přístupem rovněž ušetříme spoustu paměti, což je v dnešní době stále něco, na čemž v případě mobilní platformy záleží.*

### 2.9 Timetables Web Presentation

Bude se jednat o webovou prezentaci produktu, která bude napsaná v HTML5, CSS3, Javascriptu a jQuery. Bude obsahovat uživatelskou dokumentaci produktu, informace pro vývojáře, odkazy na stažení, hlavní rysy aplikací, jejich přínos a rovněž bude obsahovat rozhraní pro vyhledávání v jízdních řádech, které bude poskytovat aplikace ze sekce 2.6.

## 3 Ne(vy)řešené problémy

1. Co s nevalidními daty, dále nezpracovávat nebo backtrackovat? Dále nezpracovávat znamená vyhodit výjimku, dále se o tento zdroj dat nestarat a pokračovat dále. Backtrackovat znamená odstranit z dat veškeré záznamy, které jsou nevalidní.

2. Co s dobami přestupů? Počítat doby přestupů na základě GPS souřadnic nástupišť není zrovna vhodné, například u přestupů z metra na povrchovou dopravu. Časy přestupů z podzemní dopravy na povrchovou (a vice versa) jsou momentálně zdvojnásobeny, vůči ostatním časům. Časy přestupů v rámci metra, v rámci jedné stanice, v rámci jedné linky, jsou násobeny koeficientem 0.5. A to proto, že GPS souřadnice zastávek jsou nastaveny na začátky nástupišť, úhlopříčná vzdálenost je příliš velká. Časy přestupů v rámci metra, v rámci jedné stanice, v rámci různých linek, jsou násobeny koeficientem 1.5.

3. Co s daty, ve kterých je zahrnuta nějaká větší oblast (například celá Evropa)? Data obsahující 100.000 zastávek by měly odhadem využívat 2 GB paměti. Server pokrývající tuto oblast by musel mít 8 GB paměti, aby mohl uspokojivě vyřizovat všechny požadavky. Zde přichází do úvahy využití nějaké databáze, nicméně to by drasticky zpomalilo algoritmy. Je potřeba najít kompromis, jestli záleží víc na rychlosti nebo spotřebě paměti.

*Aktuální micro-benchmarky a porovnání (21.3.2018). Následující test proveden na zmergovaných datech pro Prahu, Vídeň a Flixbus Europe. V době běhu preprocessoru je zahrnut i čas strávený stahováním jednotlivých *.zip archivů. Preprocessing je vícevláknový. Benchmarky provedeny na Intel Core i7-2600 @C 3,8GHz*

| *Název testu* | *Meziformát *.tfd* | *Gtfs formát *.zip* |
|---|---|---|
| *Velikost dat na disku* | *135 MB* | *797 MB* |
| *Doba běhu preprocessoru* | *55 s* | — |
| *Paměť jednorázově spotřebovaná preprocessorem (x86)* | *415 MB* | — |
| *Doba inicializace dat knihovnou* | *5 s* | *+/- 40 s* |
| *Paměť spotřebovaná knihovnou (x86)* | *150 MB* | *??? (+ ≈ 25%)* |
| *Paměť spotřebovaná knihovnou (x64)* | *210 MB* | *??? (+ ≈ 25%)* |
| *Průměrná doba vyhledávání spojení "uvnitř"grafu* | *40 ms* | *===* |
| *Průměrná doba vyhledávání spojení "vně"grafu* | *120 ms* | *===* |

*Z micro-behchmarků plyne, že pro uspokojivý běh serveru je nutné mít k dispozici +/- 6x větší paměť, než je velikost meziformátu na disku. Server s pamětí 16 GB by měl být dostatečný pro data pokrývající celou Evropu.*

*Aktuální micro-benchmarky a porovnání (31.3.2018). Následující test proveden na datech pro Prahu. Maximální optimalizace vyhledávacího algoritmu, bez paralelizace. Kód zkoumán na úrovni strojového kódu pomocí disassembleru, předělán interface pro třídu date_time, přeuspořádany datové položky jednotlivých tříd, aby třídy byly více cache-friendly. Menší počet alokací.*

| *Název testu* | *Profiler 21.3.2018* | *Profiler 31.3.2018* |
|---|---|---|
| *Doba inicializace dat knihovnou* | *3 s* | *2 s* |
| *Paměť spotřebovaná knihovnou (x64)* | *110 MB* | *105 MB* |
| *Průměrná doba vyhledávání spojení "uvnitř"grafu* | *40 ms* | *15 ms* |
| *Průměrná doba vyhledávání spojení "vně"grafu* | *120 ms* | *50 ms* |

# Reference


*[1] GTFS Format Reference*
`https://developers.google.com/transit/gtfs/reference/`

*[2] RAPTOR: Round-Based Public Transit Routing Algorithm*
`https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/raptor_alenex.pdf`

*[3] Transit Feed API*
`https://transitfeeds.com/api/`