KU LEUVEN

FACULTEIT INDUSTRIËLE
INGENIEURSWETENSCHAPPEN

CAMPUS DE NAYER SINT-KATELIJNE-WAVER

Publiceren van updates over openbaarvervoersdata

Linked Connections

Dylan VAN ASSCHE

Promotor(en): Ann Philips

Co-promotor(en): dr. ing. Pieter Colpaert

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen:

Industriële Wetenschappen Electronica-ICT

Academiejaar 2018 - 2019

KU LEUVEN

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN

CAMPUS DE NAYER SINT-KATELIJNE-WAVER

Publiceren van updates over openbaarvervoersdata

Linked Connections

Dylan VAN ASSCHE

Promotor(en): Ann Philips

Co-promotor(en): dr. ing. Pieter Colpaert

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen:

Industriële Wetenschappen Electronica-ICT

Academiejaar 2018 - 2019

Acknowledgements

After months of hard work, I finally finished my thesis by writing this section. However, this wasn't possible without the help and support of other people!

First, I would like to thank Ann Philips from Campus De Nayer, KU Leuven. She was always there for me when I had a question. She allowed me to work on my thesis in all freedom, while steering me in the right direction. She always gave me useful feedback on my thesis and how I could improve it even further.

Secondly, I am also grateful for the experts in the field of Linked Connections: Julian Andres Rojas Melendez and Pieter Colpaert from IDLab Ghent. Their door was always open if I had a problem or when I needed some advice about my thesis. They also gave me some amazing insights about my thesis topic.

Thirdly, I would like to acknowledge Luc Sterckx as the second reader of this thesis. He provided some valuable comments on how I could bring my thesis text to the next level.

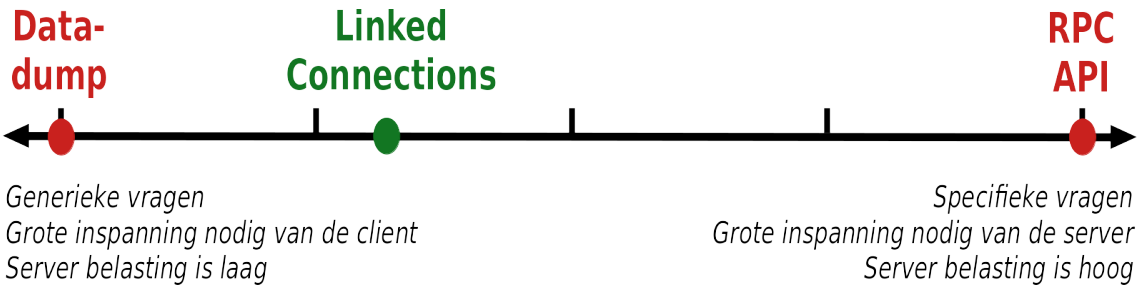
Finally, I want to express my gratitude to my mother Maggi Smets and my grand parents Maria Van Assche and Georges Van Assche for the encouragement through the years of my study. I wouldn't have accomplished all of this without their continuous support.

Thank you all!
Van Assche Dylan

Nederlandstalige samenvatting

Inleiding

In een wereld waar we steeds vast zitten in het verkeer, zijn we altijd op zoek naar alternatieven voor de wagen. Openbaar vervoer is één van zo'n alternatieven. Het openbaar vervoer is de laatste jaren aan een opmars bezig om het woon-werk verkeer beter te organiseren.
Om het openbaar vervoer makkelijker te kunnen gebruiken, zijn er reeds verschillende diensten beschikbaar. Zulke diensten voorzien ook real time informatie over je reis zoals afschaffingen, vertragingen, ...
Veel van deze diensten zijn gebaseerd op RPC API's (Remote Procedure Call Application Program Interface) of datadumps. Beiden hebben hun voordelen en nadelen.
Bij een datadump verricht de client het meeste werk, terwijl dit net omgekeerd is bij een RPC API. Een RPC API kan dus erg duur zijn voor de publiceerder van de data. Als pendelaar wensen we snel een alternatief voor onze reis, daarom zijn datadumps geen oplossing. Het zou minuten duren vooraleer we een alternatief kunnen berekenen.
Met een datadump kan je wel elke vraag beantwoorden, terwijl een RPC API slechts 1 vraag per keer kan beantwoorden. Elke nieuwe vraag vereist een bezoek aan de server. We krijgen sneller een alternatief van de server, maar RPC API's zijn moeilijker schaalbaar. De server infrastructuur moet steeds worden aangepast aan het aantal clients.
Linked Connections is een compromis tussen beide oplossingen (figuur 1). Het Linked Connections formaat is goedkoop te publiceren en te verwerken door clients. De data zijn opgedeeld in pagina's, enkel de data die nodig is, moet worden opgehaald bij de server. Elke pagina bevat een lijst van connecties gesorteerd op vertrektijd. Deze sortering laat ons toe om een gedeelte van de data op te halen. Bovendien, is dit meteen de juiste volgorde voor het CSA (Connection Scan Algorithm) routering algoritme. Linked Connections publiceert dezelfde pagina's voor alle clients. Op die manier kunnen we de data eenvoudig cachen.



Figuur 1: Linked Connections probeert een compromis te vinden tussen datadumps en RPC API's. Enkel de nodige data zijn opgehaald bij de server [1].

In deze thesis willen we de volgende onderzoeksvragen met bijhorende hypotheses beantwoorden:

1. **Hoe kunnen we een cache voor routeplanners up-to-date houden op een kost efficiënte manier voor zowel Open Data publiceerders als gebruikers?**
   In plaats van alle data opnieuw op te halen, halen we enkel de wijzigingen op. Hierdoor zouden we moeten kunnen besparen op bandbreedte en CPU verbruik.

2. **Hoe kunnen we real time Open Data updates efficiënt gebruiken in route planner algoritmes?**
   Het hergebruiken van de vorige berekeningen, zou het publiceren van real time Open Data efficiënter moeten maken.

Literatuurstudie

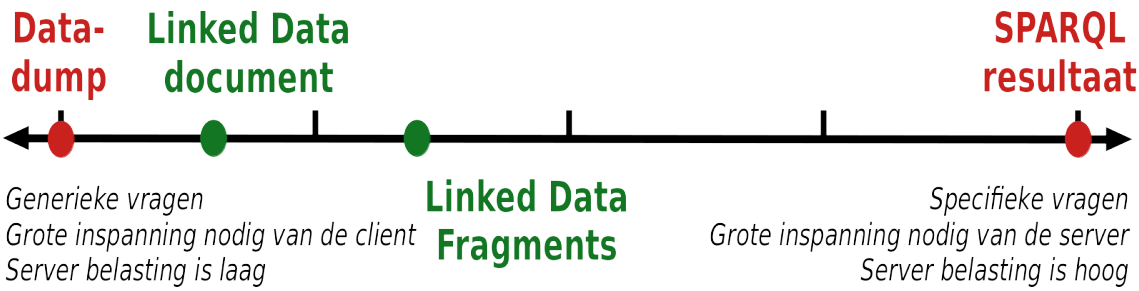
Ongeveer 30 jaar geleden heeft Tim Berners-Lee [2] het World Wide Web (WWW) uitgevonden. Het WWW heeft als doel om informatie te kunnen uitwisselen met elkaar op een eenvoudige manier. Na 30 jaar, maken we nog steeds gebruik van HTML (HyperText Markup Language) en het HTTP (HyperText Transport Protocol) om te communiceren op het Web. Er zijn verschillende diensten ontwikkeld op het Web voor allerlei toepassingen.

De NMBS (Nationale Maatschappij der Belgische Spoorwegen) heeft een routeplanner ontwikkeld. Hiermee kunnen reizigers hun reis met de trein plannen. Maar de data waren niet vrij toegankelijk, externe ontwikkelaars konden geen gebruik maken van de data.

In 2008 werd het iRail project in het leven geroepen door Yeri Tiete en Pieter Colpaert. De Open Data gemeenschap wou de voordelen aantonen van Open Data voor de NMBS. In 2015, heeft de NMBS dan de stap gezet naar Open Data.

Concepten

Het Open Data principe is gedefinieerd door de Open Definitie [3, 4]:
"Open data is data that can be freely used, re-used and redistributed by anyone - subject only, at most, to the requirement to attribute and sharealike."



Figuur 2: Het Linked Data Fragments framework probeert een compromis te vinden tussen datadumps en SPARQL resources.

Deze bestaat uit 3 belangrijke pijlers: de data moet in een open formaat beschikbaar zijn, we mogen deze data hergebruiken en integreren met andere data en iedereen mag de data gebruiken.

We passen ook de concepten van het Semantic Web toe. Het Semantic Web wil een WWW zijn dat toegankelijk is voor mensen en machines. Het huidige WWW bestaat enkel uit natuurlijke talen zoals het Nederlands. Zulke talen zijn moeilijk te begrijpen door machines. Indien we structuur aanbrengen met behulp van RDF (Resource Description Framework) bijvoorbeeld, kunnen machines niet alleen het Web lezen maar ook begrijpen.

Daarna kunnen we verschillende bronnen met elkaar gaan linken (gedistribueerde databank) met behulp van URI's (Uniform Resource Identifier). Elk onderwerp krijgt zijn unieke URI. Hieruit ontstaan zogenaamde *triples*. Een triple bestaat uit een onderwerp, een predikaat en een object. Het onderwerp is het item dat beschreven wordt, het predikaat duidt de relatie aan tussen het onderwerp en het object. We noemen dit ook wel Linked Open Data (LOD).

Maar Linked Open Data moet aan een aantal voorwaarden voldoen (5 stars deployment schema for Open Data [14, 15]):

- **Een URI is de indentiteitskaart van het Web**: Elk object heeft zijn unieke URI.
- **Gebruik HTTP URI's**: Elke URI moet leiden naar data.
- **Nuttige informatie**: De data achter een URI moet nuttige informatie bevatten over het object.
- **Link met andere URI's**: Link nuttige URI's met de data, een client kan hiermee meer informatie opvragen indien nodig.

We moeten hierbij de afweging maken tussen datadumps en een SPARQL (SPARQL Protocol and RDF Query Language) resource. Bij een datadump moet de client het meeste werk verrichten, terwijl bij een SPARQL resource de server al het werk verricht (figuur 2).

Het Linked Data Fragments framework probeert hierbij een compromis te vinden tussen beide oplossingen. Hierdoor blijven de kosten voor de publiceerder van de data laag, terwijl de client geen datadumps moet verwerken. Dit framework splits de Linked Open Data op in kleinere fragmenten. Tussen fragmenten kan de client navigeren doormiddel van HTTP URI's. Indien de client meer resultaten wenst, kan de client steeds het volgende fragment opvragen. Deze fragmentatie is ideaal voor real time data omdat elk fragment apart gecachet kan worden. Indien een gedeelte van de real time data is veranderd, moet alleen dit fragment bijgewerkt worden.

Open Data in de openbaarvervoerssector

In de openbaarvervoerssector zijn er verschillende oplossingen om openbaarvervoersdata te publiceren. We kunnen deze opdelen in 2 grote categorieën: datadumps en RPC API's, zoals besproken in de inleiding.

Voor datadumps bestaat er het GTFS(-RT) formaat[1] dat door vele openbaarvervoersmaatschappijen in gebruik is. De NMBS bijvoorbeeld maakt gebruikt van GTFS en GTFS-RT om hun data te publiceren. Daarnaast zijn er nog andere formaten zoals NeTEX (Network Timetable Exchange), TransModel en SIRI (Service Interface for Real time Information).

Een andere manier om openbaarvervoersdata aan te bieden is door middel van een RPC API. Zo publiceert iRail de NMBS data voor al zijn clients. De client stelt de vraag aan de iRail server en deze probeert de vraag zo goed mogelijk te beantwoorden. Maar doordat elke API zijn eigen formaat aanbiedt, zijn ze moeilijk te combineren. De server doet hierbij het zware werk, de client wacht enkel op het antwoord.

Linked Connections probeert een compromis te vinden tussen beiden. Linked Connections past de Linked Open Data principes toe waardoor we verschillende bronnen beter kunnen integreren. De client moet de routering zelf uitvoeren. Hierdoor is de server minder belast en kan de client elke vraag over de data beantwoorden. Dankzij de streaming aanpak, kan de client al reeds een deel van de resultaten tonen als de verbinding wegvalt.

Dataformaten

Er verschillende dataformaten in omloop op het Web. Er zijn ook hier 2 grote groepen: tekst geëncodeerde data en binair geëncodeerde data.

Binaire data zijn enkel leesbaar door machines. Maar Open Data moet leesbaar zijn door zowel mensen als machines. Daarom is er weinig ondersteuning vanuit het Semantic Web en Open Data voor deze formaten. Ze zijn wel efficiënter dan de tekstgeëncodeerde formaten.

[1] GTFS staat voor General Transit Feed Specification en RT staat voor Real Time.

Er zijn 2 tekst geëncodeerde formaten die veel toegepast worden op het WWW: JSON (JavaScript Object Notation) en XML (Extensible Markup Language). Beide formaten zijn ondersteund vanuit het Semantic Web. Ze zijn beter leesbaar door mensen en machines. Uit onze literatuurstudie blijkt dat JSON veel lichter is op vlak van parsing en grootte dan XML. Daarom is JSON beter geschikt om Open Data te publiceren dan XML.

Transportprotocollen

Er bestaan verschillende technieken om data over te brengen. Het grootste verschil, tussen deze technieken, is wie de data transporteert.

Zo kan de client op regelmatige tijdstippen de server contacteren (polling). De client contacteert de server om te kijken of er nieuwe data beschikbaar is. Hiervoor moet er steeds een verbinding worden gemaakt met de server en terug worden afgebroken.

In het ander geval, zal de server de data pushen tot bij de client. De server beslist zelf wanneer het nodig is om data te transporteren. Alleen bij nieuwe informatie, onderneemt de server actie.

Doordat we werken met Linked Open Data, maken we ook gebruik van de huidige webstandaarden. Er wordt veel gebruik gemaakt van HTTP polling om de server te contacteren. Als push technologie gebruikt men vaak WebSockets of Server-Sent-Events (SSE). We hebben gekozen om te werken met SSE aangezien dit gebruik maakt van de HTTP standaard. SSE is ontwikkeld om real time data te versturen tussen de server en zijn clients.

Kost efficiënte caching methodes

Naast het juiste formaat en transportprotocol kunnen we nog verder optimaliseren met caching. Elke pagina die reeds opgehaald werd bij de server komt terecht in de client's HTTP cache. Wanneer de pagina opnieuw nodig is, haalt men deze op uit de cache van de client. Indien nodig, valideert de client of de opgeslagen versie nog up-to-date is. Beide partijen besparen hierdoor op bandbreedte en resources.

De HTTP cache is aangestuurd door de `Cache-control` header[2]. Met de `Cache-control` header kunnen we 3 onderdelen van de HTTP cache aansturen:

- **Cachebaarheid**: In welke mate mag de data gecachet worden?
- **Vervalbaarheid**: Wanneer is de opgeslagen data niet meer geldig?
- **Revalidatie**: Moet de client verifiëren dat de data nog geldig is, vooraleer deze te gebruiken?

In het huidige ontwerp van Linked Connections zijn de statische en real time data samengevoegd. De pagina's moeten steeds opnieuw gevalideerd worden. Indien we beide delen opsplitsen, kunnen we de statische data veel langer cachen.

[2]Een alternatief is de `Expires` header, maar deze header biedt minder functionaliteit aan.

Validatie kan gebeuren op basis van 2 HTTP headers: de `ETag` header of de `Last-Modified` header. We verkiezen het ETag systeem omdat de `Last-Modified` header een beperkte precisie heeft (1 seconde). Indien de data veranderen binnen de seconde, dan zal de client dit niet merken met de `Last-Modified` header, maar wel met het ETag systeem. Dankzij validatie kan een client zijn cache bijwerken. Als de data waren vervallen, maar deze zijn nog steeds niet veranderd op de server, dan stuurt de server meteen een `HTTP 304: Not-Modified` antwoord.

Connection Scan Algorithm

We gebruiken het CSA om een reis te berekenen. De reden hiervoor is dat het CSA geen prioriteitswachtrij bevat zoals het Dijkstra algoritme. Het CSA gebruikt in plaats daarvan een lijst van voertuigen, gesorteerd op vertrektijd. Hierdoor is het CSA minder complex en sneller dan het Dijkstra algoritme.

Het CSA gebruikt enkele termen om een reis te beschrijven:

- **Connectie**: Het kleinste onderdeel van een *route*. Een connectie is gedefineerd tussen 2 *haltes* met een vertrek- en aankomsttijd.
- **Halte**: Een halte is een plaats waar reizigers kunnen vertrekken of aankomen.
- **Voertuig**: Een voertuig kan reizigers ophalen en afzetten bij een *halte*. Een voertuig rijdt over een *connectie*.
- **Route**: Een route bevat een aantal *connecties* die de reiziger toelaten om van de vertrekhalte tot aan de eindhalte te geraken.
- **Reis**: Een reis bevat een aantal *routes* die voldoen aan de vraag van reiziger.

Om het CSA uitbreidbaar te houden, gebruiken we de CSA profiel variant. Er zijn 2 profielen: het `TreinProfiel` en het `StationHalteProfiel`. Het `TreinProfiel` wordt bijgehouden in de T array. Het `StationHalteProfiel` in de S array. De details van de CSA profiel variant zijn beschreven in de CSA paper [5].

Het CSA is afhankelijk van de vorige toestand van de *T* en *S* arrays bij elke iteratie, zoals beschreven in de paper. Zodra we de connecties van een reis manipuleren, moeten we onze reis herberekenen. De volgende parameters van een connectie kunnen hierbij invloed hebben:

- De vertrek- en aankomsttijd.
- De vertrek- en aankomsthalte.
- De minimale overstaptijd tussen 2 voertuigen.

Zodra een voertuig in de reis vertraging oploopt, veranderen de vertrek- en aankomsttijden van een connectie. Als het voertuig vertraging heeft, dan is, bij een overstap, de overstaptijd tussen 2 voertuigen ook verschillend. De snelheid waarmee het CSA een nieuwe reismogelijkheid berekent is afhankelijk van het aantal connecties dat het verwerkt. Door dit aantal te verminderen, krijgen we sneller een nieuwe reismogelijkheid.

Implementatie

Om Linked Connections uit te breiden met real time informatie, moeten we dit ondersteunen op 3 plaatsen (figuur 3):

1. **Linked Connections Server**: De server moet de real time informatie aanbieden aan de clients.
2. **QRail bibliotheek**: De client bibliotheek moet overweg kunnen met deze nieuwe informatie.
3. **LCRail client**: De applicatie moet deze informatie kunnen visualiseren.

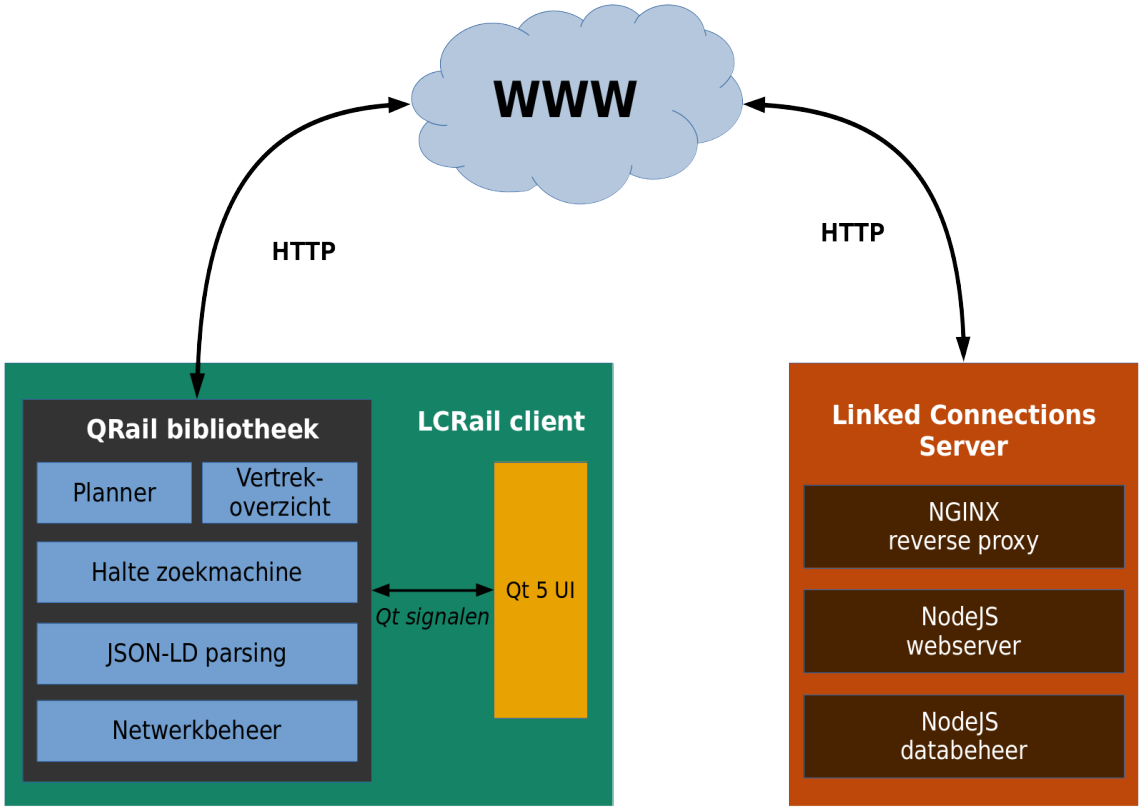
Linked Connections Server

In deze thesis hebben we het real time resource aan de server toegevoegd. Dit resource publiceert de evenementen die gebeuren op het netwerk van een openbaarvervoersmaatschappij. Indien een voertuig vertraging heeft of is afgelast, publiceert dit resource meteen deze informatie.

Hiervoor werd de `gtfsrt2lc` bibliotheek aangepast. Deze bibliotheek converteert de GTFS-RT data naar Linked Connections. We hebben ondersteuning toegevoegd voor afgelaste voertuigen. Daarnaast hebben we ook de parser aangepast omdat deze geen rekening hield met een eigenschap van de GTFS-RT specificatie[3].

Om de evenementen te publiceren volgens de Linked Open Data principes, maken we gebruik van de SOSA (Sensor, Observation, Sample and Actuator) ontologie. Deze ontologie is een gestandariseerde specificatie voor het publiceren van observaties en evenementen. Door het veelvuldig gebruik van deze ontologie is deze ideaal voor onze toepassing.

[3]De NMBS maakt onder andere gebruik van deze eigenschap. Ze publiceren enkel de haltes waar de informatie effectief wijzigt (update haltes). De haltes tussen beide update haltes, moet de client zelf aanvullen.



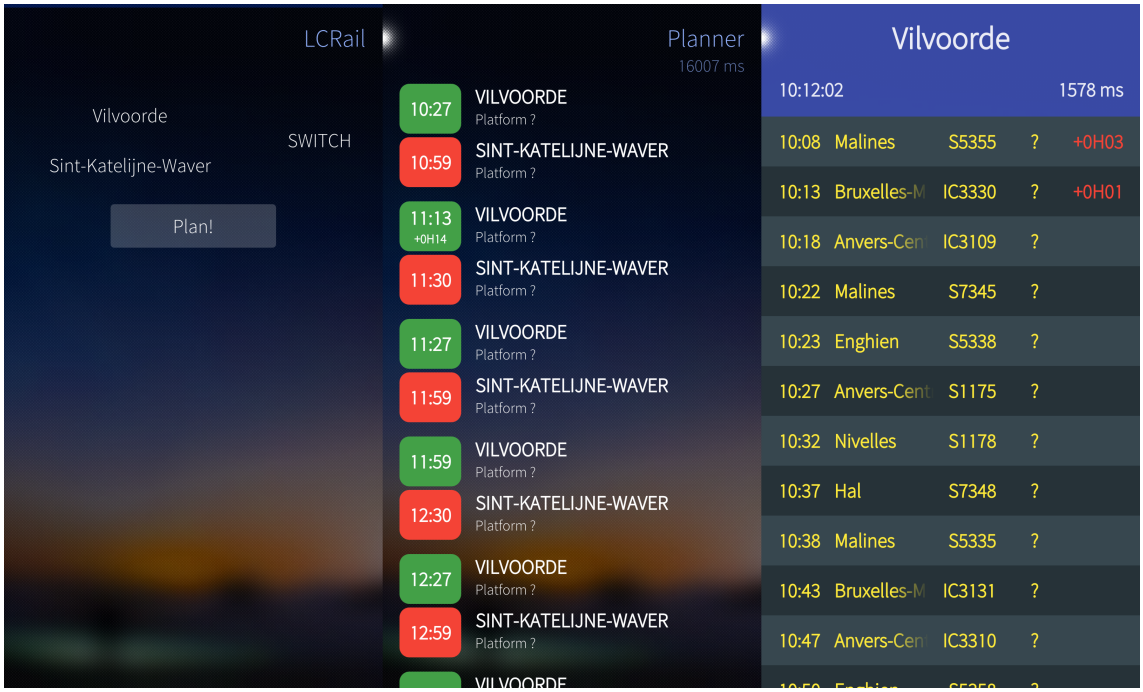
Figuur 3: De architectuur van onze Linked Connections omgeving. De LCRail client gebruikt de QRail bibliotheek om de vraag van de gebruiker te beantwoorden via Qt signalen. De QRail bibliotheek haalt de nodige data op, berekent een vertrekoverzicht of reis en geeft deze informatie terug aan LCRail. De Linked Connections Server publiceert alle Linked Connections pagina's op het WWW.

Om onze testen te kunnen uitvoeren, implementeren we deze resource met HTTP polling (polling) en SSE (push).

De QRail bibliotheek en de LCRail client

Daarnaast hebben we ook een eigen bibliotheek geschreven (QRail). Deze bibliotheek laat ons toe om elk aspect van de Linked Connections dataverwerking te manipuleren. QRail implementeert een aangepaste variant van de CSA profiel variant. Deze versie is gebaseerd op het werk van Bert Marcelis [6]. We hebben ook het Earliest Arrival CSA voorop geplaatst. Op deze manier kunnen we vooraf onbereikbare connecties uit onze data filteren.

Naast een routeplanner, bevat QRail ook een eigen geschreven vertrekoverzicht algoritme. Hiermee kunnen de vertrekkende voertuigen van een halte worden weergegeven. Dit algoritme is sneller dan het CSA omdat het enkel connecties moet filteren op basis van een halte. Om informatie te kunnen opvragen over een halte, hebben we ook een SQLite databank toegevoegd met alle NMBS haltes.



Figuur 4: Screenshots van de LCRail applicatie op een Sailfish OS mobiel toestel. Aan de linkerkant zien we de selectie van de haltes voor de routeplanner. In het midden een overzicht van de routes voor een bepaalde reis. Aan de rechterkant zien we het vertrekoverzicht gevisualiseerd voor het station Vilvoorde.

Deze bibliotheek is geschreven in Qt C++. Helaas is er geen ondersteuning voor HTTP polling of SSE in Qt. We hebben onze eigen interface geschreven die beiden ondersteunt, gebaseerd op de SSE interface van JavaScript.

De QRail bibliotheek gebruikt de real time resource om sneller nieuwe resultaten te berekenen. Voor het vertrekoverzicht zal elk evenement bekeken worden door de bibliotheek. Van zodra een evenement een connectie bevat uit het vertrekoverzicht, zal QRail deze bijwerken. Voor het CSA ligt dit iets anders. Door de afhankelijkheid van de T en S arrays, kunnen we de connecties niet zomaar vervangen. Om het CSA toch efficiënter te maken, hebben we het algoritme aangepast. Per Linked Connections pagina neemt het CSA een snapshot (foto) van beide arrays. Zodra er een connectie verandert die gebruikt is in de reis, herstellen we de juiste snapshot. We herberekenen enkel het deel van de reis dat onderhevig is aan de connectie wijziging.

Dit rollback systeem is enkel mogelijk als we steeds over een bijgewerkte pagina cache beschikken. Daarom gaat QRail elk evenement meteen in zijn pagina cache plaatsen. Als de algoritmes een pagina opvragen, krijgen ze meteen de laatste versie uit de cache. Hierdoor is geen validatie van de pagina's meer nodig.

De LCRail client is een Sailfish OS applicatie geschreven in Qt. Deze applicatie maakt een visualisatie van de QRail bibliotheek. Een voorbeeld van de LCRail client is te zien in figuur 4.

Resultaten

Testomgeving

De testomgeving bestaat uit 2 delen:

1. **Pushing of polling experiment**: Dit experiment draait op de Virtual Wall en een Digital Ocean droplet. Dankzij de Virtual Wall kunnen we realistische scenario's testen van server en client applicaties. Met dit experiment bekijken we wat de beste aanpak is om de data over te brengen. We testen het gebruik van HTTP polling (polling) en SSE (push). De Digital Ocean droplet voert de taak van de server uit in dit experiment.

2. **Verwerken van updates experiment**: Dit experiment draait op een Digital Ocean droplet als Linked Connections Server. De clients zijn 2 Sailfish OS mobiele toestellen. Op deze apparaten draait de QRail bibliotheek samen met de LCRail client. Met behulp van dit experiment bekijken we hoe efficiënt we de real time data kunnen updaten op de client. Daarnaast bekijken we ook hoe snel we de gebruiker kunnen informeren.

Testresultaten

We bespreken nu de resultaten van elk experiment individueel.

Pushing of polling experiment

SSE (push) levert ons een besparing op in CPU and RAM verbruik van de server. De server moet slechts eenmalig een connectie opzetten met elke client, elke aanvraag lezen, . . . Dit is niet het geval met HTTP polling.

Er treden in beide gevallen pieken op in het CPU verbruik. Deze ontstaan bij het opzoeken van de data en deze door te sturen naar de clients. Bij SSE zijn de pieken korter, kleiner en minder frequent (15 %) dan bij HTTP polling (35 %). Bij SSE kan men door eenmalig de data op te zoeken, alle clients bedienen. Bij HTTP polling, moet dit process herhaalt worden per client.

Met SSE (2,3 %) is het gemiddelde CPU verbruik bijna 4 keer lager dan met HTTP polling (8,9 %). Het gemiddelde RAM verbruik ligt 28 % lager. Deze resultaten zijn gemeten met 1500 actieve clients.

Verwerken van updates experiment

Dankzij onze aanpassingen aan het vertrekoverzicht en routering algoritmes, is het verbruik van CPU en netwerk resources gedaald. Afhankelijk van de client's hardware, kunnen we tot 10 maal sneller een vertrekoverzicht bijwerken. Voor het routering algoritme, kunnen we 2 tot 3 maal sneller een route bijwerken dan in de originele implementatie.

Daarnaast, gebruikt de client gemiddeld 50 tot 70 % minder CPU om het vertrekoverzicht te vernieuwen (in vergelijking met de originele versie). Bij het CSA, is het gemiddeld CPU verbruik met 34 tot 50 % gedaald ten opzichte van de originele implementatie.

De hoeveelheid verzonden data is gedaald met 90 % voor zowel HTTP polling als SSE. Voor de ontvangen data besparen we vooral bij het gebruik van SSE (15 % - 21 %). De reden hiervoor is dat HTTP polling ook data ontvangt als er geen nieuwe updates zijn.

Conclusie

Aan de hand van de literatuurstudie en de testresultaten gaan we onze onderzoeksvragen en hypotheses beantwoorden.

Hoe kunnen we een cache voor routeplanners up-to-date houden op een kost efficiënte manier voor zowel Open Data publiceerders als gebruikers?

We hebben verschillende oplossingen vergeleken in onze literatuurstudie en testen. Het grote verschil zit in het pushen of pollen van de data. Bij pushing zal de server de data zelf naar de client sturen. Bij polling zal de client de server regelmatig vragen of er nieuwe data beschikbaar is.

Uit onze resultaten kunnen we besluiten dat pushing de beste oplossing is voor Open Data publiceerders en gebruikers. Er zal enkel data overgedragen worden indien dat nodig is. Het resource verbruik van de client en de server liggen hiermee lager dan bij de traditionele aanpak, zoals voorspelt in de hypothese.

Hoe kunnen we real time Open Data updates efficiënt gebruiken in route planner algoritmes?

Real time data updates efficiënt verwerken, vereist extra logica. De rekentijd was dezelfde om een vertrekoverzicht of reis bij te werken als deze te genereren in de originele implementatie. Indien we enkel de wijzigingen verwerken met onze algoritmes, konden we de rekentijd, CPU en bandbreedte verbruik sterk doen dalen. Dit hadden we verwacht uit onze hypothese.

Als toekomstig werk kunnen we QRail bibliotheek nog verder optimaliseren door gebruik te maken van prefetching, smart pointers voor beter geheugenbeheer en snellere JSON parsers.

Abstract

Publishing real time public transport data can be a heavy task for Open Data publishers (RPC APIs) and consumers (data dumps). With Linked Connections we try to find a compromise between both worlds. Since Linked Connections is not optimised for real time data yet, we investigate how the Linked Connections server can publish real time data in a cost-efficient way. At the same time, we examine how we can optimise the algorithms to use Linked Connections on the clients.

We propose a list of changes to these algorithms to reduce the processing time and computing resources. By using a separate real time resource on the Linked Connections server, only the updates to the data must be transferred. We want to reduce further the necessary resources for the client and the server for Linked Connections.

We achieve this goal by modifying the Connection Scan Algorithm (CSA) and liveboard algorithms on the client. We reduced the processing time of a journey with a factor of 2 - 3 using the CSA. A liveboard could be updated 10 times faster than in the original implementation. The amount of network bandwidth is reduced by 15 % for the CSA and by 21 % for the liveboard algorithm.

Based on these results, we conclude that a publish-subscribe approach (Server-Sent-Events) is more efficient than a polling approach (HTTP polling) for real time Open Data. Using a real time resource, the efficiency of the client's algorithms is significantly improved.

Keywords: Linked Open Data, World Wide Web, route planning, public transport, real time data

Abstract

Publiceren van real time openbaarvervoersdata kan een zware opdracht zijn voor Open Data publiceerders (RPC API's) en gebruikers (datadumps). Met Linked Connections proberen we een compromis te vinden tussen beide oplossingen. Omdat Linked Connections nog niet geoptimaliseerd is voor real time data, onderzoeken we hoe we Linked Connections kunnen publiceren op een kost efficiënte manier. Tegelijkertijd, bekijken we hoe we de algoritmes kunnen optimaliseren om Linked Connections te gebruiken op de clients.

We hebben een aantal aanpassingen voorgesteld om deze algoritmes te versnellen en minder resources te laten verbruiken. Door gebruik te maken van een aparte real time resource op de Linked Connections server, moeten enkel de wijzigingen aan de data worden overgebracht. We willen het resource verbruik van zowel de client als de server verder reduceren voor Linked Connections.

Om dit doel te bereiken, hebben we het Connection Scan Algorithm (CSA) en vertrekoverzicht algoritme aangepast op de client. We hebben hiermee de verwerkingstijd van een reis door het CSA, gereduceerd met een factor van 2 - 3. Een vertrekoverzicht kon 10 maal sneller bijgewerkt worden dan in de originele implementatie. De hoeveelheid bandbreedte is gereduceerd met 15 % voor het CSA en met 21 % voor het vertrekoverzicht algoritme.

Gebaseerd op deze resultaten, kunnen we besluiten dat een publiceer-abonneer oplossing (Server-Sent-Events) efficiënter is dan een polling oplossing (HTTP polling) voor real time Open Data. Het gebruik van een real time resource, verbetert sterk de efficiëntie van de algoritmes op de client.

Keywords: Linked Open Data, World Wide Web, routeplanning, openbaar vervoer, real time data

Contents

Acknowledgements **v**

Nederlandstalige samenvatting **vii**

Abstract **xix**

Contents **xxv**

Figures **xxx**

Tables **xxxi**

Abbreviations **xxxiii**

1 Introduction **1**

1.1 Problems 2

1.2 Research questions and hypotheses 3

2 Literature study **5**

2.1 Background concepts 6

2.1.1 Open Data 6

2.1.2 Semantic Web 6

2.1.3 Open Data in public transport 9

2.2 An overview of technologies used to publish and consume real time Open Data . . . 16

2.2.1 Common data formats 16

2.2.2 Transfer protocols 22

2.2.3 Suitable technologies for real time Open Data publishing and consuming . . . 28

2.3 Keeping the client's cache up to date in a cost-efficient way 31

2.3.1 Cacheability 32

2.3.2 Expiration 32

2.3.3 Revalidation and reloading 33

2.4 Connection Scan Algorithm . . . 34
2.4.1 Terminology . . . 34
2.4.2 The Connection Scan Algorithm profile variant . . . 35
2.4.3 Influence of changes on the CSA profile variant . . . 36
2.5 Conclusion . . . 37

3 Implementation **39**
3.1 Specifications . . . 39
3.2 Architecture overview . . . 40
3.3 Linked Connections Server . . . 42
3.3.1 Transforming GTFS-RT into Linked Connections . . . 42
3.3.2 Cancellations support . . . 46
3.3.3 Publication of events . . . 47
3.3.4 Real time resource . . . 50
3.4 QRail library . . . 50
3.4.1 SSE and HTTP polling in Qt . . . 52
3.4.2 Real time data caching using updates . . . 53
3.4.3 CSA snapshot rollback . . . 55
3.4.4 Liveboard regeneration . . . 58
3.4.5 Page cache on the client . . . 59
3.4.6 Updates processing on the client . . . 59
3.5 LCRail client . . . 62
3.5.1 Features . . . 66
3.5.2 Automatic UI updates . . . 67
3.6 Conclusion . . . 67

4 Results **69**
4.1 Test environment . . . 69
4.1.1 Pushing or polling . . . 70
4.1.2 Processing of updates . . . 72
4.2 Test results . . . 74
4.2.1 Pushing or polling . . . 74
4.2.2 Processing of updates . . . 77
4.3 Discussion . . . 84
4.3.1 Pushing or polling . . . 84
4.3.2 Processing of updates . . . 85

4.4 Conclusion 87

5 Conclusion **89**

A Code overview **99**

A.1 Server side 99

A.2 Client side 100

A.3 General flow 101

A.3.1 Server side 101

A.3.2 Client side 103

List of Figures

1 Linked Connections probeert een compromis te vinden tussen datadumps en RPC API's. Enkel de nodige data zijn opgehaald bij de server [1]. viii

2 Het Linked Data Fragments framework probeert een compromis te vinden tussen datadumps en SPARQL resources. ix

3 De architectuur van onze Linked Connections omgeving. De LCRail client gebruikt de QRail bibliotheek om de vraag van de gebruiker te beantwoorden via Qt signalen. De QRail bibliotheek haalt de nodige data op, berekent een vertrekoverzicht of reis en geeft deze informatie terug aan LCRail. De Linked Connections Server publiceert alle Linked Connections pagina's op het WWW. xiv

4 Screenshots van de LCRail applicatie op een Sailfish OS mobiel toestel. Aan de linkerkant zien we de selectie van de haltes voor de routeplanner. In het midden een overzicht van de routes voor een bepaalde reis. Aan de rechterkant zien we het vertrekoverzicht gevisualiseerd voor het station Vilvoorde. xv

1.1 Linked Connections tries to find a compromise between data dumps, which require a big effort from the client, and RPC APIs, which require a big effort from the server. Linked Connections requires the client to implement the routing algorithm itself, but only the necessary data for the routing must be streamed to client in small data fragments [1]. 2

2.1 The Linked Data Fragments framework tries to find a compromise between LOD dumps and SPARQL queries. 9

2.2 The structure of a GTFS[4], the data is split in multiple text files and distributed in a single ZIP file. 10

2.3 Current existing implementations of the TransModel specification, each one implements only a part of the TransModel specification. 11

2.4 The architecture of an RPC API. The client asks a specific question and the server replies with a specific answer [6]. 14

2.5 RDF data can be represented in a RDF graph [7]. 19

2.6 HTTP polling mechanism, only when the resource is updated, new data is returned to the client. All other requests return exactly the same data. 23

2.7 HTTP long polling mechanism, only when the resource is updated, new data is returned to the client. The connection stays open until an update is returned to the client. 24
2.8 HTTP chunked streaming, the resource is streamed to the client. When the resource is updated, only the update is streamed to the client. 25
2.9 HTTP long polling mechanism, only when the resource is updated, new data is returned to the client. The connection stays open until an update is returned to the client. 26
2.10 MQTT broker and client, the broker only pushes data to the client about the subscribed topics. 27
2.11 Retrieving updates using polling or streaming [8]. Streaming requires less interaction between the client and the server to retrieve the same amount of data than polling. This technique allows us to significantly reduce the amount of allocated resources for the client and the server to interact with each other. 29
2.12 HTTP caching negotiation between the client and the server [9]. 31

3.1 The architecture overview of our Linked Connections setup. The LCRail client interacts with the QRail library to answer the question of the user using Qt signals. The QRail library retrieves all the data and executes the necessary algorithms for the LCRail client. The Linked Connections Server publishes all the Linked Connections pages to the WWW. 41
3.2 The difference between the arrival and departure delays for GTFS-RT and Linked Connections. GTFS-RT specifies both at a stop, while Linked Connections defines both for a connection. 44
3.3 The GTFS-RT update B has a different arrival delay than the GTFS-RT update A. This will introduce a conflict: Connection B will normally have the arrival delay of the update A, but it has to be the arrival delay of update B. 45
3.4 Caching real time data using updates, only the changes are pushed to the client to reduce the amount of bandwidth and other resources. If the connection is lost with the server, the client revalidates its pages and subscribe again to the real time resource. 54
3.5 The CSA profile variant calculates the routes in the opposite direction of the user's journey. 57
3.6 Page cache update algorithm. Pages with a different departure delay can move to another page. Otherwise, the connections are not sorted anymore by departure time. In all other cases, the connection can just be replaced in its current page. 61
3.7 Screenshots of the LCRail application on a Sailfish OS smartphone. On the left, we see the selection of the stops for the route planner. In the middle, the results of the route planner and on the right the liveboard visualisation. 62

3.8 The architecture of the LCRail client. The main function initialise QRail and the QML engine. The QAbstractListModel class allows us to create interactive QML lists with C++ data. Qt signals are used to communicate between each part of the client. . . . 66
3.9 Stream processing of the liveboard data in LCRail. 68

4.1 The Virtual Wall at IDLab Ghent. The Virtual Wall I has 206 nodes and the Virtual Wall II has 161 nodes at their disposal. Each node can be configured over SSH with the jFed tool. 71
4.2 Our experiment in the Experimenter Toolkit UI of the jFed tool. We allocated 6 nodes with each 250 clients on the Virtual Wall. 71
4.3 The CPU and memory usage of the server when 1500 clients are actively polling the server every 30 seconds (see spikes). The data on the server is refreshed every 30 seconds. The benchmark ran for 30 minutes. 75
4.4 The CPU and memory usage of the server when 1500 clients are connected using SSE. The server pushes the new data every 30 seconds to the 1500 clients (spikes). The benchmark ran for 30 minutes. 76
4.5 Amount of sent data of the liveboard for the 3 approaches on a single device in time. The benchmark ran for 30 minutes. 77
4.6 Amount of received data of the liveboard for the 3 approaches on a single device in time. The benchmark ran for 30 minutes. 78
4.7 The mean CPU usage of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 78
4.8 The mean RAM usage of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 79
4.9 The sum of the amount of sent data of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 79
4.10 The sum of the amount of received data of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 80
4.11 The mean refresh time of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 80
4.12 Amount of sent data of the CSA for the 3 approaches on a single device. The benchmark ran for 30 minutes. 81
4.13 Amount of received data of the CSA for the 3 approaches on a single device. The benchmark ran for 30 minutes. 81
4.14 The mean CPU usage of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 82
4.15 The mean RAM usage of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 82

4.16 The sum of the amount of sent data of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 83
4.17 The sum of the amount of received data of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 83
4.18 The mean refresh time of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes. 84

A.1 Processing of the GTFS-RT stream in the `gtfsrt2lc` library. 102
A.2 Real time resource publishing in the Linked Connections Server. 103
A.3 Stop search in the LCRail client. 104
A.4 Processing of updates, retrieved from the real time resource. 105
A.5 Retrieving a liveboard for a given stop. 106
A.6 Planning a journey for a given departure, arrival stop and departure time. 107
A.7 Updating a watched liveboard. 108
A.8 Updating a watched journey. 109

List of Tables

2.1 An overview of the existing public transport data specifications that are commonly used by transport agencies. 15
2.2 An overview of the different data formats that are commonly used on the Web. 21
2.3 An overview of the different protocols that are commonly used on the Web. 28

4.1 An overview of the hardware for the pushing or polling experiment. 70
4.2 An overview of the hardware for the processing of updates experiment. 73

List of Abbreviations

API	Application Program Interface
CEN	European Committee for Standardization
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
CSA	Connection Scan Algorithm
CSV	Comma-Separated Values
ECMA	European Computer Manufacturers Association
gRPC	gRPC Remote Procedure Calls
GTFS	General Transit Feed Specification
GTFS-RT	General Transit Feed Specification Real Time
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
IoT	Internet of Things
IMEC	Interuniversitair Micro-Elektronica Centrum
JSON	JavaScript Object Notation
JSON-LD	JSON for Linking Data
LC	Linked Connections
LDF	Linked Data Fragments
LOD	Linked Open Data
NeTEx	Network Timetable Exchange
NMBS	Nationale Maatschappij der Belgische Spoorwegen
MQTT	MQ Telemetry Transport
ODF	Open Document Format
ProtoBuff	Protocol Buffers
QML	Qt Modeling Language
RAM	Random Access Memory
RDF	Resource Description Framework
RPC	Remote Procedure Call
SFOS	Sailfish OS
SIRI	Service Interface for Real time Information
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language

SSE	Server-Sent Events
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
UI	User Interface
URI	Uniform Resource Identifier
WS	WebSockets
WWW	World Wide Web
XML	Extensible Markup Language

1 Introduction

In a world where we are always stuck in traffic, we are continuously looking for alternatives. Public transport is one of those alternatives and became an essential part of the city in the last decade to efficiently organise transportation for the commuter traffic. To use public transport in an easy way, several services on the World Wide Web arose to help commuters with schedules, delays, cancellations and many more aspects of public transport. All of these services today are based on data dumps or specific RPC APIs (Remote Procedure Call Application Program Interface) of public transport agencies to operate. Both approaches are the opposite of each other: data dumps requires a lot of effort from the client while an RPC API requires the server to do most of the work.

If a client uses a data dump for example to plan a route for a commuter, the client must read the enormous file, create an abstraction of the data and perform routing on top of the abstraction. This whole process can take several minutes to calculate the possible routes for a commuter's journey. In case the data has been changed, the complete data dump is downloaded again and the whole process is executed from scratch on the new data. This approach is unacceptable since we want quickly an overview of all possible routes for a journey, otherwise the commuter might miss his or her connection.

In case of an RPC API, the client will just ask the server a specific question and the server will reply with a specific list of possible routes for a journey. If another question needs to be answered, a new request must be made to the server. Only questions that are supported by the server can be answered, in comparison to data dumps where every question can be answered by the client. Since a server has a lot more processing power than a simple phone for example, the routing algorithm can be executed much quicker than on a phone. However, scaling this solution costs a lot of money since the server infrastructure must be able to keep up with all the requests from the clients.

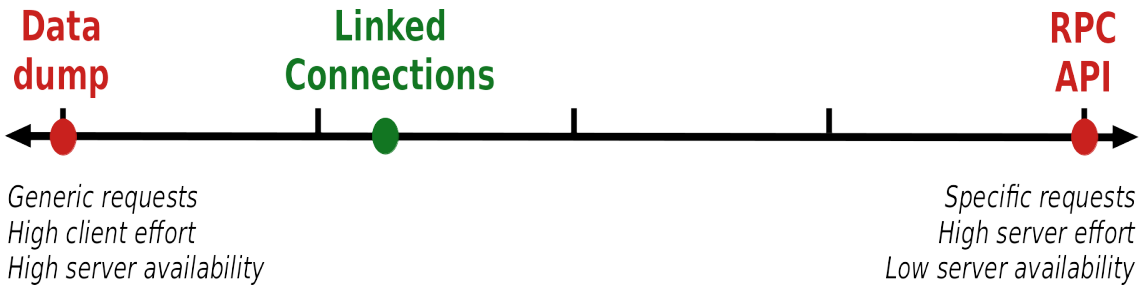


Figure 1.1: Linked Connections tries to find a compromise between data dumps, which require a big effort from the client, and RPC APIs, which require a big effort from the server. Linked Connections requires the client to implement the routing algorithm itself, but only the necessary data for the routing must be streamed to client in small data fragments [1].

We will discuss both approaches further and why Linked Connections can be a solution for publishing and consuming real time Open Data of public transport in chapter 2.

1.1 Problems

At the IDLab research group of the University of Ghent, they looked for a compromise (Linked Connections) between those approaches by creating a format which is easy to publish for the server and relative easy to process for the client. Linked Connections publishes a list of all departures for all vehicle in fragments. Each fragment covers a certain time range and can be downloaded individually by the clients. This way, the server can serve the same list to all its clients (cacheable). The clients can create their own route planning algorithm on top of Linked Connections. Last but not least, they must only download the part of the data that they need to perform the calculation of the possible routes for a commuter's journey. The clients have to do more work than with an RPC API, but they get the flexibility of data dumps: every question can be answered, which is not possible with an RPC API.

Every time the data is updated, only the changed fragment must be downloaded again. To check if the data needs to be refreshed, the server is polled every couple of seconds by the client. If every client starts to poll the server, the server load and the necessary bandwidth might increase again.

In this thesis we will try to find a better approach to publish the real time data of Linked Connections in more cost-efficient way for both publishers and consumers by using existing standard technologies to reduce the necessary resources of both parties, to the minimum. We will try to accomplish this goal by implementing and comparing hypothesized cost-efficient ways to update the cache of route planners and create a routing algorithm that is able to consume these approaches in an efficient way. If one of these approaches is a clear winner, it will be implemented in the Linked Connections framework.

1.2 Research questions and hypotheses

1. **How can we keep a cache for route planners up to date in a cost-efficient way for both Open Data publishers and consumers?**
   We suspect that we can improve the cache of route planners in a more cost-efficient way by using updates. Instead of fetching the complete data set again, we only fetch the changes to the data set. This way, we should be able to reduce the network bandwidth and computing resources for Open Data publishers and consumers.

2. **How can we consume real time Open Data updates in route planner algorithms in an efficient way?**
   Instead of recalculating a complete new route if the data has been updated, we only update the affected data. Since not all the data is changed at once, the routing algorithms must be able to reuse their previous calculations. With this approach, the results must be faster updated than in the current approach. The user of the data should be faster informed about any changes in his or her trip.

2

Literature study

Approximately 30 years ago, Tim Berners-Lee [2] invented the World Wide Web as a way to share information with each other in an easy way. A lot of technologies from that time are long superseded by newer ones, but not the Web! The architecture of the Web has not been changed a lot since it was invented. We still use HTML (HyperText Markup Language) as a mark up language for web pages and the HTTP protocol to interact with others on the Web. During the last decade, several services arose on the Web like search engines, social media and other tools to make it easier for everyone to use the Web and communicate with each other. Today, you can edit your documents on the Web, read the newspaper online, react on each other comments, ...

Another example of a service on the Web is finding the optimal route for your public transport journey. The NMBS (Nationale Maatschappij der Belgische Spoorwegen) has build a routing tool on their website and in their mobile applications to calculate a commuter's journey on the Belgian rail network. However, external developers and researchers could not get access to the data behind the routing tools or experiment with it.

In 2008, iRail[1] was founded by Yeri Tiete and Pieter Colpaert [10]. In this project, the Open Data community proved the NMBS that opening up their data was the right thing to do if they want to be more transparent, innovative and improve their services in general [11]. Finally in 2015, the NMBS announced their first steps towards Open Data. Thanks to this policy, developers can create awesome applications and tools upon these data. For example: routing planning applications like *BeRail*[2], *HyperRail*[3], but also a delay tracker for your train: *SNCBAlerts*[4] and many other tools.

[1] https://irail.be

[2] https://openrepos.net/content/minitreintje/berail

[3] https://hyperrail.be/

[4] https://github.com/SNCBAlerts

In this chapter, an overview of the necessary background concepts are given before we investigate which technologies are already available to publish real time Open Data on the World Wide Web (WWW). Last but not least, we will investigate how to consume these real time data in a cost-efficient way.

2.1 Background concepts

2.1.1 Open Data

Open Data is defined by the Open Definition [3, 4]:
"Open data is data that can be freely used, re-used and redistributed by anyone - subject only, at most, to the requirement to attribute and sharealike."

This definition summarizes 3 important pillars of the Open Data principles:

1. **Access to the data**: The data must be available in an open format as whole.
2. **Re-using**: The user of the data must be allowed to mix it with other data or redistribute it.
3. **No discrimination**: Everyone can use the data without any discrimination against groups or people.

Thanks to this strict definition of Open Data, we can achieve "*interoperability*". Without interoperability, other people may use the data, but not combine them with other data.

For example: HERE maps forbids developers to use their search engine if they do not use HERE maps tiles in their application[5].

2.1.2 Semantic Web

The Semantic Web aims to be the next generation of the World Wide Web [12]. The Semantic Web wants to create an Internet that is accessible for both humans and machines. The Web of today contains almost only pages written in a natural language like English or Dutch. This results in the fact that automated tools can not understand what is written on these pages.

For example, a search engine can not understand what the user wants to search. It just crawls the Web and matches the keywords given by the user with the crawled pages. However, the search engine itself will not understand the information in these pages.

[5] https://developer.here.com/faqs

2.1.2.1 Structured data

To create smarter tools, we have to adjust the way we publish our data. Searching on the Web is almost the same as executing an SQL (Structured Query Language) statement on a database: we ask the database for all the rows that are matching with our query. This is exactly the same as the goal of the Semantic Web: creating a huge decentralised database on the Web.

To achieve this goal, the data should be described (structured data) using a description language like RDF (Resource Description Framework). Once described, machines can read the data and understand it, just like humans. Structured data is described in *triples*. A triple consists of: a subject, a predicate and an object. The subject is the entity that is being described, the predicate shows the relation of the subject with the other entity (object).

Every subject requires an unique ID to be able to uniquely identify the object we are describing and everyone on the Web should be able to create them. Fortunately, the Web already has such mechanism: the URI (Uniform Resource Identifier).

We will illustrate this using an example: `http://irail.be/vehicle/IC123`. The owner of the `.be` domain can create unique subdomains like `irail.be`. Only the owner of `irail.be` can create resource URIs of the vehicles he or she wants to describe: `http://irail.be/vehicle/IC123`. The URI is only used to uniquely identify the resource, the URI might lead to a non-existing page on the Web! Although, this is not ideal of course. It is useful to make the URI dereferenceable. This way, we can always retrieve more information about the item, described by the URI.

2.1.2.2 Linked Open Data

By combining the power of Open Data and the idea behind structured data, we can create Open Data that is understandable by machines and humans. In order to reach a better level of interoperability, we should be able to link different data sets to each other. By doing this, the Web becomes a distributed database of all kinds of Open Data, which we call Linked Open Data (LOD) [13]. Thanks to LOD, users do not have to download the complete data dump anymore about an object on the Web, but only the part they really need to reach their own specific goals.

To create usable Linked Open Data, several principles like the 5 stars deployment scheme for Open Data for example [14, 15], must be honored:

1. **URIs are the identifiers of the Web**: Every object that is described on the Semantic Web requires an unique URI.
2. **Use HTTP URIs**: People can easily look up the data behind the HTTP URI with standard tools.
3. **Provide useful information behind HTTP URIs**: When the HTTP URI is followed by a client, return some useful information about the object.
4. **Include other URIs in the data**: Link to other URIs in your data, a client can discover then even more information if needed about the object.

If those principles are followed correctly, a machine follows an URI in the LOD, it will find another page with even more URIs describing the followed resource. It can follow those URIs too, to get even more information about the object. However, this approach requires a lot of work from the client or the server to discover all these links. The workload increases exponentially, every link that has been discovered contains another list of links to other information on the Web.

Today's Web offers 3 common ways to access Linked Open Data:

1. **SPARQL query**: The server retrieves all the triples that match with the SPARQL (SPARQL Protocol and RDF Query Language) query from the client.
2. **Linked Data document**: A specific item in the data is described on this page using triples. The client can download the page and follow the URIs if needed.
3. **Data dump**: All the triples in the data set are retrieved at once, the client has to crawl the data completely by itself.

All of these approaches require a lot of work from one party, it should be possible to find a compromise between them to query efficiently data resources as a client and minimize the resource usage of the server. At IDLab in Ghent, they created the Linked Data Fragments (LDF) framework [16, 17] to achieve this goal (figure 2.1). Using the LDF framework, several fragmentations can be developed, for example: the Triple Pattern Fragment.

The Linked Data Fragments framework splits a LOD dump into several smaller pieces (fragments) where each fragment contains a part of the data set, some control information to navigate to other fragments (hypermedia) and metadata. With this approach, SPARQL queries can be solved much more efficiently by the server and clients can still query data resources in an efficient way. With the LDF framework, the server can already serve a part of the results of the SPARQL query to the client when a single fragment is ready. If the client wants more information, it can fetch the next fragment if needed. Such fragmentation is ideal for real time data since each fragment can be cached in an easy way. Updating a fragment only requires to fetch it again using the URI of the fragment (section 2.3).

Figure 2.1: The Linked Data Fragments framework tries to find a compromise between LOD dumps and SPARQL queries.

2.1.3 Open Data in public transport

As mentioned before, Open Data in the public transport sector is important for innovation and the development of tools to help the commuter in his or her public transport journey. The next sections will discuss the available specifications and approaches to publish public transport data as Open Data, the Linked Connections standard and why the Linked Connections is necessary to improve the experience of the commuter.

2.1.3.1 Data dumps

Several specifications already exist today to publish (real time) public transport data. Public transport agencies package their data for a certain time range in one big data dump and distribute the data along the clients. As soon as something has been changed in the data, the whole dump must be downloaded again. Using a data dump for routing purposes requires a big effort from the client.

GTFS and GTFS-RT The General Transit Feed Specification (GTFS) [18, 19] defines an open standard to exchange public transport data like schedule, geographic and fare information. The public transport agency publish a GTFS feed which can be consumed in an interoperable way by applications and other clients. GTFS is already implemented by more than 1350 public transport agencies and consumed by hundreds of applications. GTFS was developed by Google and it was first used in Google Maps to provide public transport information for their users. An overview of the structure of different types of GTFS files can be seen in figure 2.2, while an example of a GTFS feed can be found on the website of iRail organisation[6].

[6] https://gtfs.irail.be/nmbs/gtfs/

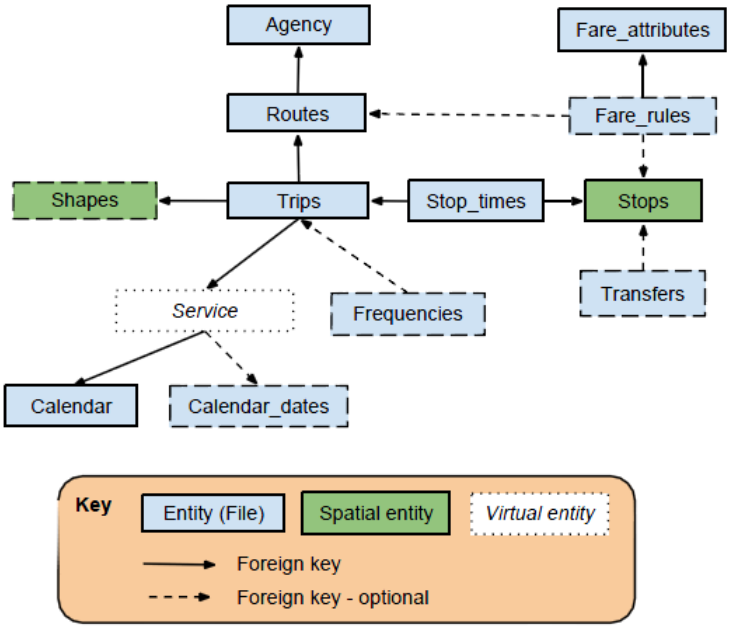


Figure 2.2: The structure of a GTFS[7], the data is split in multiple text files and distributed in a single ZIP file.

GTFS uses text oriented Comma-Separated Values (CSV) files (distributed in ZIP files) in order to be readable by humans and machines in an easy way. Each file covers several hours or days of information which is a lot when downloading and processing these files on low power clients. GTFS only provides the necessary information to communicate with commuters about the public transport service and not the operational details of it. In order to provide real time data to the commuter, GTFS Real Time (GTFS-RT) was developed [20]. Using GTFS-RT, the agency can provide real time data about delays, cancellations, location of the vehicles, service alerts, ... GTFS-RT uses Protocol Buffers as data format for a fast and efficient transmission, more information about this data format can be found in section 2.2.1.4.

The GTFS and GTFS-RT specifications cover the following aspects of public transport data:

- Network topology
- Timing information
- Vehicle scheduling

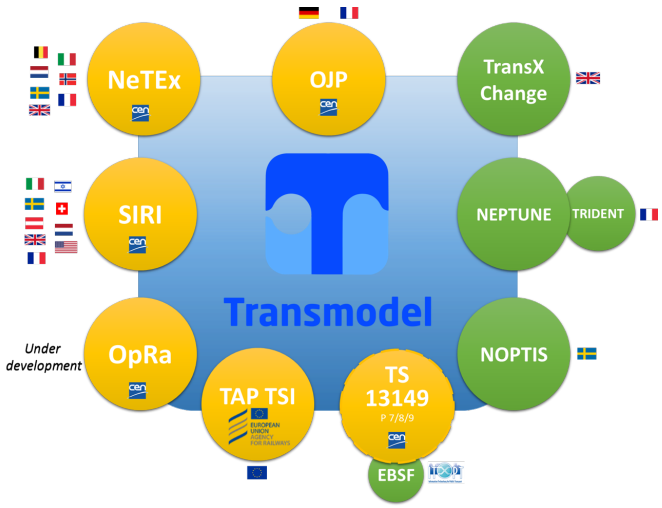


Figure 2.3: Current existing implementations of the TransModel specification, each one implements only a part of the TransModel specification.

TransModel TransModel is a European CEN (European Committee for Standardisation) standard [21] to provide a European Reference Data Model for public transport data. TransModel is developed at a conceptual level to support the development of public transport applications, their interaction with other applications and the integration of new applications in existing systems. TransModel provides only a reference standard for public transport agencies, they are not required to implement TransModel completely, but it can be useful as a framework to design architectures, databases and data exchange interfaces. Several standards (figure 2.3) are created on top of TransModel like the Network Timetable Exchange standard (NeTEx) and the Service Interface for Real time Information (SIRI) standard. It is used by the Norwegian public transport company Entur[8].

[8] https://developer.entur.org/content/real-time-information

The TransModel specification covers the following aspects of public transport data:

- Network topology
- Timing information
- Vehicle scheduling
- Fare information
- Operations monitoring and control
- Fare management
- Management information and statistics
- Driver management
- Driver scheduling
- Rostering
- Driving personnel disposition

Network Timetable Exchange Network Timetable Exchange (NeTEx) is a European CEN standard [22] to exchange public transport schedules and other information. NeTEx implements a subset of the TransModel specification, since NeTEx users are particularly interested in exchanging public transport data and not into data about the scheduling of the drivers for example.
The NeTEx specification implements the following subset of the TransModel specification:

- Network topology
- Timing information
- Vehicle scheduling
- Fare information

This is sufficient for developers to create applications to help the commuters with their journey and inform the commuter about the accessibility of a stop for example. NeTEx uses XML as its data format to package a complete data set as a single document which can be validated and managed using existing XML tools. NeTEx provides much more information than GTFS (for example: fare information). NeTEx can be converted into GTFS without any problem. However, it is not possible to convert GTFS to NeTEx since NeTEx describes the data in much more detail than GTFS. GTFS does not contain all the required data to generate a NeTEx data set. Like GTFS, a NeTEx data dump provides several hours or days of information which can be a lot for low power clients.

[8] https://www.transitwiki.org/TransitWiki/images/f/ff/GTFS_data_model_diagram.PNG

Service Interface for Real time Information The Service Interface for Real time Information (SIRI) standard [23, 24] is also based on the TransModel specification, but in comparison to NeTEx, SIRI is optimised for real time data exchange. SIRI was initially developed by a collaboration between France, Germany, Scandinavia and the United Kingdom. SIRI is currently used by several public transport agencies like Transport For London in the United Kingdom. As seen in figure 2.3, SIRI is also a CEN standard like NeTEx and TransModel.

The SIRI specification implements the following subset of the TransModel specification and 2 additional services:

- Network topology in real time
- Timing information in real time
- Vehicle scheduling in real time
- Real time status of available facilities at stops
- Operations monitoring (general and incidents information)

2.1.3.2 RPC APIs

Another way to publish public transport information is through route planning RPC APIs. A traditional RPC API, like the iRail API[9] or the one from the Deutsche Bahn[10] , provides a list of endpoints where the client can ask a specific question. The server starts then a procedure to answer this question and returns the answer to the client. The server can access databases or execute heavy calculations while the client waits for an answer. The answer is very specific for the asked question. If the client wants to know the answer of another question, a new request must be made to the server. In figure 2.4, the architecture of an RPC API is visualised.

An RPC API requires a continue Internet connection between the client and the server and a big infrastructure is needed to keep up with amount of requests from clients (the iRail API serves more than 900 000 requests each day to its clients[11]). If the client wants to route between multiple public transport agencies, the client has to combine these APIs together which is a difficult task since each API has its own representation of the data. The specifications we discussed above avoid this issue by defining a standard which has to be followed to implement the specification. An RPC API requires less processing power of low power clients since the question is already answered by the server, the answer must only be parsed by the client and shown to the commuter.

[9] https://api.irail.be

[10] https://developer.deutschebahn.com/store/apis/info?name=Fahrplan-Free&version=v1&provider=DBOpenData

[11] https://hello.irail.be/2018/03/16/one-million-daily-requests-where-do-they-come-from-and-how-well-cope-with-them/

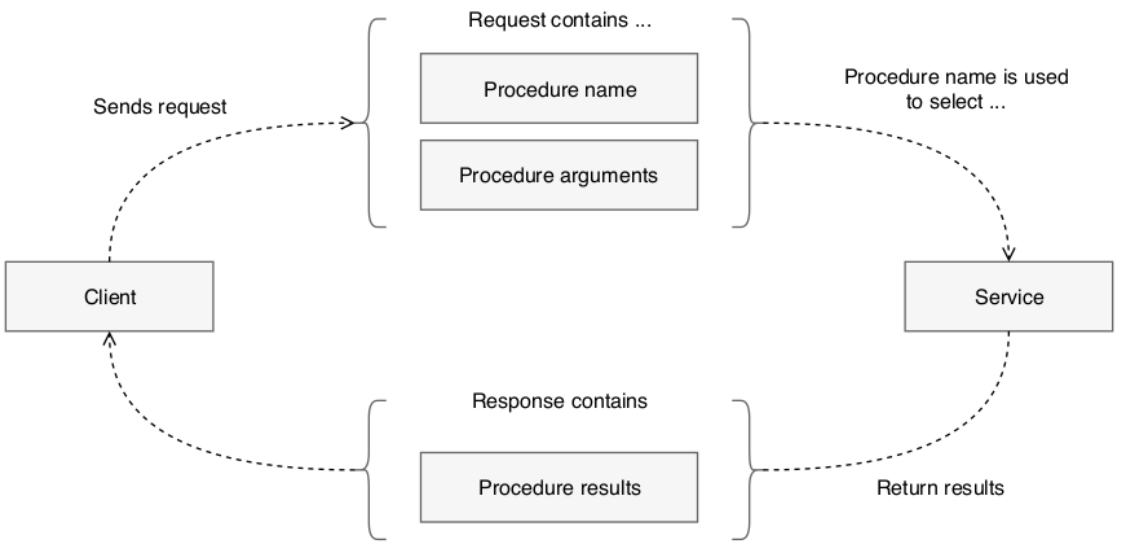


Figure 2.4: The architecture of an RPC API. The client asks a specific question and the server replies with a specific answer [6].

2.1.3.3 Linked Connections

Linked Connections tries to publish Linked Open Data of public transport data in a smarter way (as mentioned in chapter 1) than the previous approaches which are described above. By listing all connections by departure time in separated pages, clients can download the pages they need to calculate the possible routes of a commuter's journey.

When another page has to be retrieved, the client can follow automatically the URIs on the page to the next or previous page. Linked Connections is a compromise between an RPC API and data dumps by publishing the public transport data as pages with connections fragments (figure 1.1). It adopts the Linked Open Data and Linked Data Fragments principles (section 2.1.2.2).

Using the Linked Open Data principles, we can combine in an easy way multiple Linked Connections resources when calculating possible routes for a commuter's journey and create an intermodal route planner which can be extended with additional public transport agencies over time.

Since the data is directly available, the client can select a routing algorithm that fits best for the commuter's needs. It is possible to extend an existing algorithm with additional features like wheelchair accessibility [25] of platforms in an easy way.

Goals

1. **Make data dumps more accessible to clients**: Clients do not have to download the complete data set to start routing. They only need to fetch the part they really need while still be able to access the raw transport data.
2. **Reduce the publishing costs of data**: Since only a part of the data is transferred to the client, the amount of bandwidth and other resources are reduced for both parties.
3. **Publish the public transport data as Open Data**: No restrictions are applied to access the data, as explained in section 2.1.1.

With Linked Connections, the commuter can access a personalised router planner, suitable to his or her needs. The commuter can even calculate a journey when the network coverage is poor, thanks to the streaming approach of Linked Connections. However, implementing a routing application using Linked Connections requires much more effort than implementing a client for an RPC API since the routing has to be done on the client side.

Table 2.1 An overview of the existing public transport data specifications that are commonly used by transport agencies.

	Raw data	**Server's effort**	**Client's effort**	**Updating data effort**	**Every questions can be answered**
Data dumps	Yes +	Little +	Big -	Big -	Yes +
Routing APIs	No -	Big -	Little +	Little +	No -
Linked Connections	Yes +	Moderate +/-	Moderate +/-	Moderate +/-	Yes +

2.1.3.4 Routing algorithms

As explained before, the client have to do the routing calculation itself. We can rely on several mathematical algorithms to solve this problem. Most algorithms that are used today for routing calculation are variants on the Dijkstra algorithm. However, it uses a slow priority queue. We will use the Connection Scan Algorithm (CSA). The priority queue is replaced in the CSA by a list of vehicles, sorted by departure time. Thanks to this approach, the CSA is faster than a Dijkstra based algorithm [5].

As explained in [6], the CSA is the most suitable routing algorithm for Linked Connections. The main reasons are:

1. It can use the streaming approach of Linked Connections.
2. It is compatible with the Open World Assumption of the Semantic Web.
3. CSA is faster than a Dijkstra based algorithm.

In section 2.4, we will discuss the CSA further.

2.2 An overview of technologies used to publish and consume real time Open Data

A lot of data resources are already available today on the World Wide Web[12], all of these resources use various formats to represent the data to the consumers. To compare different technologies used to publish and consume real time Open Data we need to evaluate first the most common data formats and the necessary transfer protocols to deliver the Open Data from the publisher to the consumer.

In this section, we will first give an overview of some common data formats which are heavily used on the World Wide Web to exchange data. Afterwards, we will compare different widely used protocols on the Web and which combination is the most suitable for real time Open Data.

2.2.1 Common data formats

We will compare several data formats [26] with each other that are usable for publishing and consuming Open Data on the Web. The focus will be on the following parameters to compare different data formats with each other:

- **Human and machine friendly**: Is the format easy to read for humans and machines?
- **Semantic Web**: Is the format supported by the Semantic Web standards [14]?
- **Efficiency**: How much overhead causes the format when transferring the data from the publisher to the consumer?
- **Standardised**: Is the format standardised and easy to use for the consumer and publisher?

[12] The European Union already provides more than 12 500 Open Data sets on the Web through their European Union open data portal (https://data.europa.eu/euodp/en/data).

```
{
    "fruit": ["apple", "pineapple", "cherry"],
    "healthy": true,
    "eachDayAtLeast": 2,
    "taste": "delicious"
}
```

Snippet 2.1: JSON snippet

```
{
    "@context": "https://json-ld.org/contexts/person.jsonld",
    "@id": "http://dbpedia.org/resource/John_Lennon",
    "name": "John Lennon",
    "born": "1940-10-09",
    "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Snippet 2.2: JSON-LD snippet

2.2.1.1 JSON

JSON (JavaScript Object Notation) is a data format that is lightweight on one hand, while on the other hand it is easy to write and read for both humans and machines [27]. JSON was inspired by the object literals of JavaScript and is written in plain text. A JSON example can be seen in snippet 2.1. Since it is so easy to use, it is widely adopted by the Web and programming languages. You can find for almost any programming language a JSON parser and each web browser provides native support through its JavaScript engine for JSON. JSON is standardised in the ECMA-404 specification[13].

Due to the fact that JSON is written in plain text, the efficiency is almost always lower than a byte oriented data format like MessagePack (section 2.2.1.3) or Protocol Buffers (section 2.2.1.4), but it performs better than XML when it comes to processing, size and CPU resource usage (JSON uses 50 % less memory than XML for example). Thanks to this text oriented approach, humans can still read JSON data without any problem [28].

```
<?xml version="1.0"?>
<person>
    <name>Dylan Van Assche</name>
    <occupation>Student Industrial Engineering: Electronics-ICT</occupation>
    <website>https://www.dylanvanassche.be</website>
</person>
```

Snippet 2.3: XML snippet

In the world of the Semantic Web, the World Wide Web Consortium (W3C) community extended the original JSON specification with semantic properties to easily generate Linked Open Data and share it on the Web. They called the new standard JSON-LD (JSON for Linking Data) [29]. JSON-LD is already adopted by search engines like Google, Bing, ... to provide a richer description of a link in their search results[14]. If we compare both (snippets 2.1 and 2.2) with each other you will see that the syntax stays the same, but JSON-LD added two special fields to the data:

- `@context`: Provides the meta data of the data, the context defines how the data should be interpreted.
- `@id`: Uniquely identifies the data using an unique URI (Uniform Resource Identifier).

JSON data can be read and written by humans and machines. However, only humans can understand what the JSON data really means. By extending JSON with a context, machines can understand the JSON data too.

2.2.1.2 XML

A counterpart of JSON is XML (Extensible Markup Language). XML is a markup language that is readable by humans and machines like JSON [12]. It is widely used on the Web to provide web feeds, in the design world to create vector graphics (Scalable Vector Graphics) or even for creating rich documents using the OpenDocument (ODF) standard. Thanks to its wide adaption and standardisation [30] by W3C, a lot of tools and APIs (Application Program Interfaces) are available to work with XML data. An XML sample can be seen in snippet 2.3.

One of the advantages XML has over JSON is the possibility to directly add the meta data to the data. However, XML can not use arrays which might result in a lot of duplicate meta data. As mentioned earlier in section 2.2.1.1, XML is slower to process and bigger than JSON in size. Because of these disadvantages, XML is less suitable as a data format for real time Open Data when it comes to cost-efficiency and low-power devices. In 2004, the World Wide Web Consortium (W3C) published an extension to XML: the Resource Description Framework. Using RDF, we can add context to XML data for the Semantic Web, as described in chapter 1. An example of RDF data can be found in snippet 2.4 and figure 2.5.

[13] https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[14] https://schema.org/

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns="http://www.example.org/~joe/contact.rdf#">
    <foaf:Person rdf:about= "http://www.example.org/~joe/contact.rdf#joesmith">
        <foaf:mbox rdf:resource="mailto:joe.smith@example.org"/>
        <foaf:homepage rdf:resource="http://www.example.org/~joe/"/>
        <foaf:family_name>Smith</foaf:family_name>
        <foaf:givenname>Joe</foaf:givenname>
        </foaf:Person>
</rdf:RDF>
```

Snippet 2.4: RDF snippet [7]

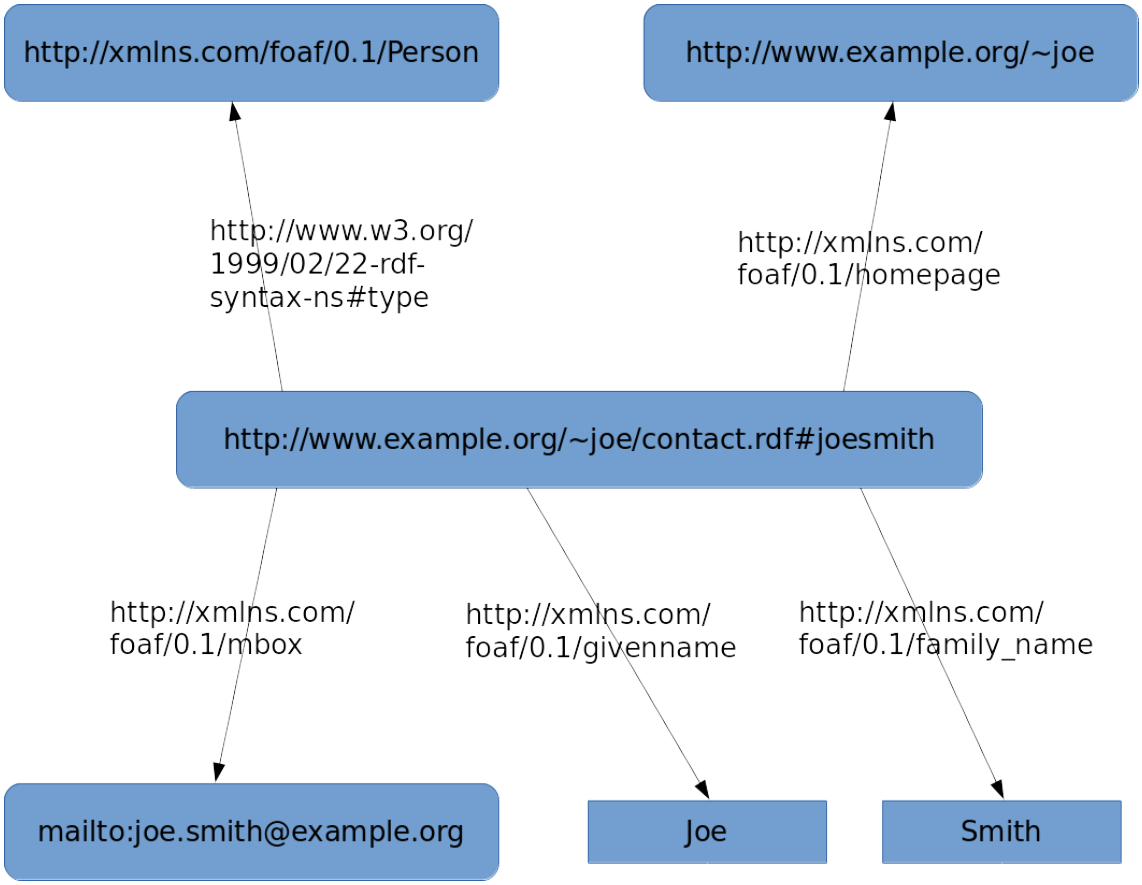


Figure 2.5: RDF data can be represented in a RDF graph [7].

```
DF 00 00 00 04 A5 66 72 75 69 74 DD 00 00 00 03 A5 61 70 70 6C 65 A9 70
69 6E 65 61 70 70 6C 65 A6 63 68 65 72 72 79 A7 68 65 61 6C 74 68 79 C3
AE 65 61 63 68 44 61 79 41 74 4C 65 61 73 74 02 A5 74 61 73 74 65 A9 64
65 6C 69 63 69 6F 75 73
```

Snippet 2.5: MessagePack snippet in hexadecimal format

```
message Person {
  required string name = "Dylan Van Assche";
  required int32 id = 2;
  required string occupation = "Student";
  optional string website = "https://www.dylanvanassche.be"
}
```

Snippet 2.6: Protocol Buffers schema snippet

2.2.1.3 MessagePack

MessagePack [31] provides an efficient standardised binary serialisation format[15]. The goal is to be smaller than JSON while providing the same level of interchangeability.

MessagePack is lighter than JSON due to its binary serialisation of the data, but it is almost unreadable for humans while machines are able to read it without any problem. Another downside of MessagePack is the requirement of a separate parser while JSON support is directly available in every modern web browser. An example of MessagePack data can be seen in snippet 2.5, the snippet contains the same data as the previous snippet in JSON (snippet 2.1).

Unfortunately, there is no standard available for Linked Open Data in MessagePack. The reason for this is that Linked Open Data should be readable and understandable by both parties, to comply with the 5-stars deployment schema for Linked Open Data [14].

2.2.1.4 Protocol Buffers

Protocol Buffers are a data interchange format developed by Google [32]. They use a schema to define how the data looks like (snippet 2.6). Protocol Buffers automatically encodes the data in the most efficient way as binary data. One of the advantages of using a schema is that Protocol Buffers are directly structured data and that is backwards compatibility with the previous versions.

Protocol Buffers provide an excellent and cost-efficient way to exchange data between publishers and consumers [33]. However, if the schema of the data is not available, reading the data is almost impossible for a machine. As a result of the binary encoding of the data, Protocol Buffers are definitely not human friendly, as explained in section 2.2.1.3.

[15] https://github.com/msgpack/msgpack/blob/master/spec.md

```
struct Person {
  1: string name = "Dylan Van Assche",
  2: i32 id = 2,
  3: string occupation = "Student",
  4: string website = "https://www.dylanvanassche.be"
}
```

Snippet 2.7: Apache Thrift Definition snippet

2.2.1.5 Apache Thrift

Apache Thrift [34] is an alternative for Protocols Buffers. It was originally developed by Facebook and open sourced in 2007. It is now further developed under the Apache Software Foundation. The philosophy behind Apache Thrift can be summarised in 4 pillars:

1. **Simplicity**: No unnecessary dependencies and a simple code base.
2. **Transparency**: Common programming language concepts (idioms) are used.
3. **Consistency**: No language-specific features in the core libraries.
4. **Performance**: Performance is more important than elegance.

Apache Thrift encodes the data into binary data with a Thrift Definition file (snippet 2.7). This is almost the same as the schema file of Protocol Buffers (section 2.2.1.4). The data types and services interfaces are described in this file. Without the Thrift file, the data is unreadable for a machine. Since this data is binary encoded, a human cannot read the data either. In comparison to Protocol Buffers, Apache Thrift has RDF support built in.

Table 2.2 An overview of the different data formats that are commonly used on the Web.

	Size	**Parsing speed**	**Widely used and standarised**	**Semantic Web support**	**Readable by humans and machines**	**Meta data integrated with the data**
JSON	Small +	Fast +	Yes +	Yes +	Yes +	No -
XML	Big -	Slow -	Yes +	Yes +	Yes +	Yes +
MessagePack	Small +	Fast +	Yes +	No -	No -	No -
Protocol Buffers	Small +	Fast +	Yes +	No -	No -	Yes +
Apache Thrift	Small +	Fast +	Yes +	Yes +	No -	Yes +

2.2.2 Transfer protocols

In the previous section, we looked at different data formats that are usable to deliver real time Open Data from the publisher to the consumer. In this section, we will compare different transfer protocols that are in use to handle the transport of real time data between the publisher and the consumer and how the consumer can use these protocols to keep its cache up to date.

The most used protocol today on the World Wide Web is HTTP (HyperText Transfer Protocol). It was originally developed at CERN (European Organization for Nuclear Research) in 1991 by Tim Berners-Lee [2] together with the URI (Uniform Resource Identifier) and HTML specifications. To transfer changes to the data on the server to the client, several techniques were developed in the past upon the HTTP standard, which are going to be discussed below. Besides HTTP, other transport protocols are used as well.

2.2.2.1 HTTP polling

One of the oldest and most used methods to keep the client's cache up to date is by polling the server at a certain interval. Between 2 requests, the cache returns the latest retrieved version of the data to the client. If the data has been changed on the server in meantime, the cache returns stale (out-of-date) data to the client. A typical HTTP polling mechanism (figure 2.6) operates as follows:

1. The client sends a HTTP request to the server.
2. The server replies immediately with the requested data.
3. The client processes the data.
4. The client waits a certain time and repeats the process.

An example of HTTP polling can be found in the Sailfish OS (SFOS) application "BeRail"[16]. BeRail polls the iRail API from time to time to keep the user's train journey up to date.

2.2.2.2 HTTP long polling

In section 2.2.2.1, we looked at HTTP polling. HTTP long polling is an extension of this mechanism. The main difference is that the server keeps the connection open until the server has a new version of the data for the client. As soon as the server has an update available for the client, it sends the data back and closes the connection. This is only possible when both parties set the `Connection` header to `keep-alive` (this is only necessary for HTTP/1.0 since HTTP/1.1 has persistent connections by default) and avoid a connection time out on the server and client. Otherwise, the connection is dropped after a certain time by the client or server to regain the allocated resources to the connection in case of a transfer error [35].

[16] https://www.github.com/iRail/harbour-berail

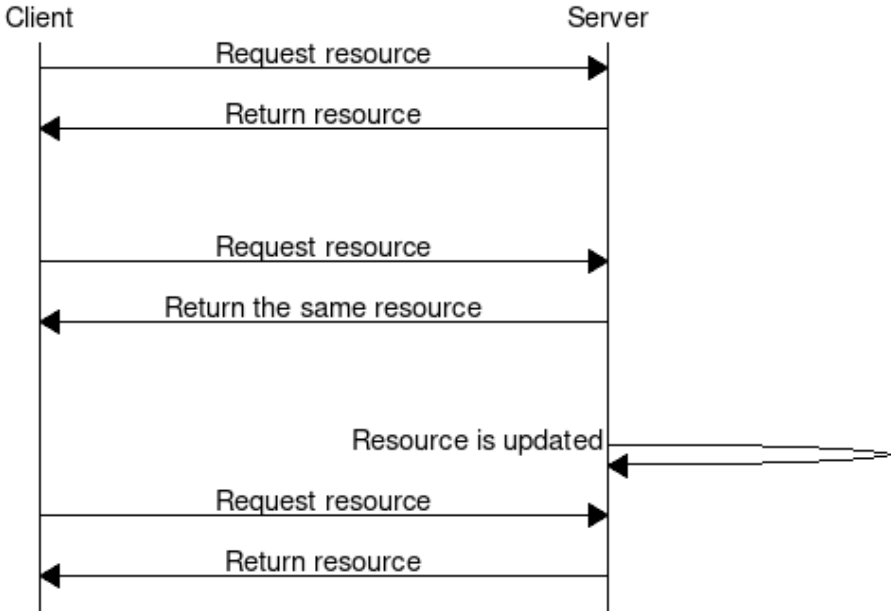


Figure 2.6: HTTP polling mechanism, only when the resource is updated, new data is returned to the client. All other requests return exactly the same data.

HTTP long polling (figure 2.7) always performs the same steps over and over again:

1. The client sends a HTTP request to the server.
2. The server keeps the connection open without sending any data back to the client.
3. Both parties wait without closing the connection.
4. As soon as the server has new data ready for the client, it sends the data back to the client and closes the connection.
5. The client processes the data and sends directly a new request to the server to repeat the process.

The main advantages over standard HTTP polling (section 2.2.2.1) are:

- **Less resources:** The server and client only have to transfer data when new data is available.
- **No stale cache:** As soon as new data is available, the client will receive it.

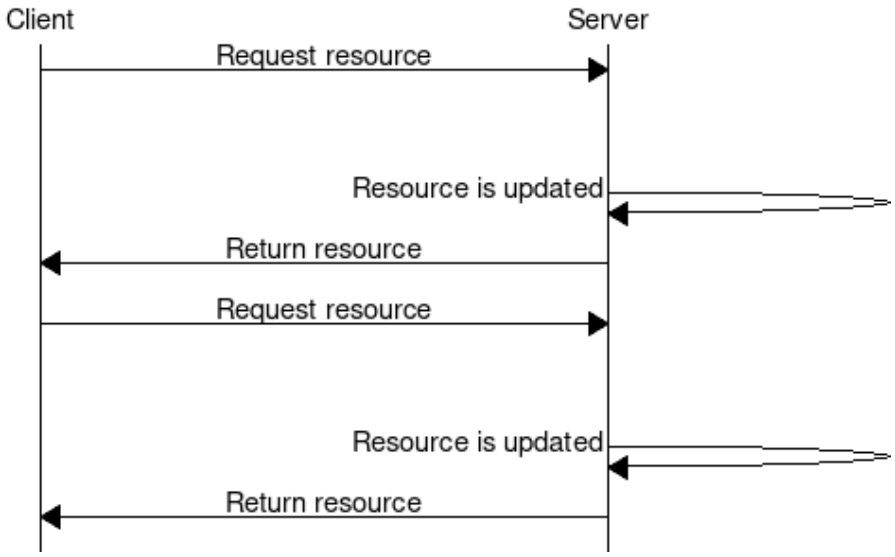


Figure 2.7: HTTP long polling mechanism, only when the resource is updated, new data is returned to the client. The connection stays open until an update is returned to the client.

2.2.2.3 HTTP chunked streaming

The HTTP/1.1 specification introduced another way to deliver real time data to the client with HTTP chunked streaming. Using the HTTP header `Transfer-Encoding: chunked`, the client knows that the HTTP reply is chunked by the server. The client will read the stream and assemble it again when the whole reply has been received. This is only possible due to the fact that HTTP/1.1 connections are persistent by default, as mentioned in section 2.2.2.2. However, each client can maintain a limited amount of HTTP/1.1 simultaneous connections to a server depending on the client. Most web browsers limit the amount to 6-8 connections for each server [36].

When compressing (section 2.2.3.2) is applied to the data, the data is first compressed before it is chunked. This way, the client only has to decompress the whole reply instead of every chunk, which leads to less CPU usage when processing HTTP chunked data.

HTTP chunked streaming (figure 2.8) interaction between a client and a server is as followed:

1. Client sends a HTTP request to the server.
2. Server keeps the connection open and sends back a first chunk with the `Transfer-Encoding: chunked` header.
3. Server keeps sending chunks while the client is processing the chunks.
4. When the data is transferred, the server can close the connection.
5. Client assembles the chunks and process the reply.

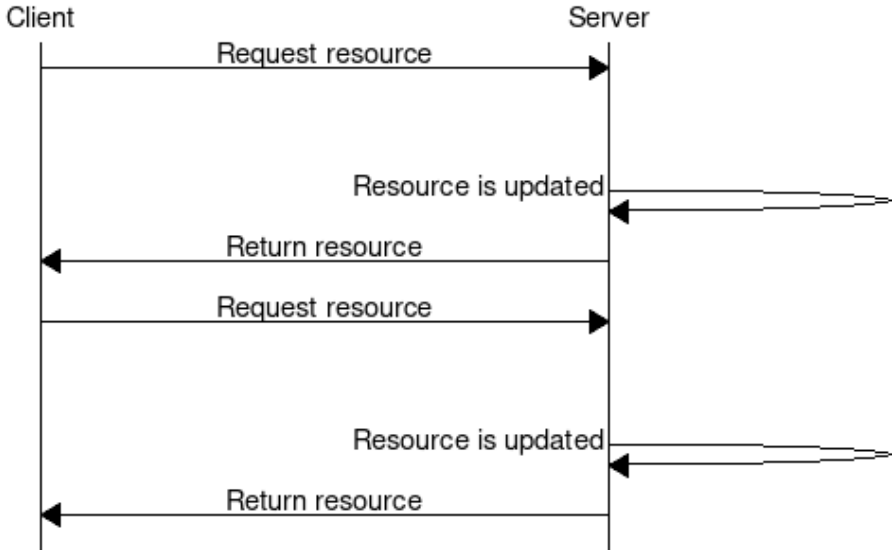


Figure 2.8: HTTP chunked streaming, the resource is streamed to the client. When the resource is updated, only the update is streamed to the client.

A real life implementation of HTTP chunked streaming can be found in the Twitter real time tweets API[17]. The API allows developers to stream Tweets in real time into their applications.

2.2.2.4 Server-Sent Events

With the introduction of HTTP/2 in 2015, a new standard arose to keep the client's cache up to date without any interaction from the client itself, by allowing the server to push data to the client over HTTP/2 without a request from the client.

With Server-Sent-Events (SSE), the client can save resources and only process data when it is really available as with HTTP/1.1 chunked streaming (section 2.2.2.3), while the server does not have to maintain a continue open connection to the client if a network proxy server is available to take over the connection management. This is one of the main advantages of Server-Sent Events over HTTP/1.1 chunked streaming to lose a connection and automatically re-establish it when network coverage is available again. When the connection is re-established, the client will receive automatically all the missed events from the server without any interaction on supported networks. It is even possible to deliver the messages using a network push service to further minimise the battery and resource usage of the client [37]. Most browsers support SSE already, but not all of them.

[17] https://developer.twitter.com/en/docs/tweets/filter-realtime/guides/connecting.html

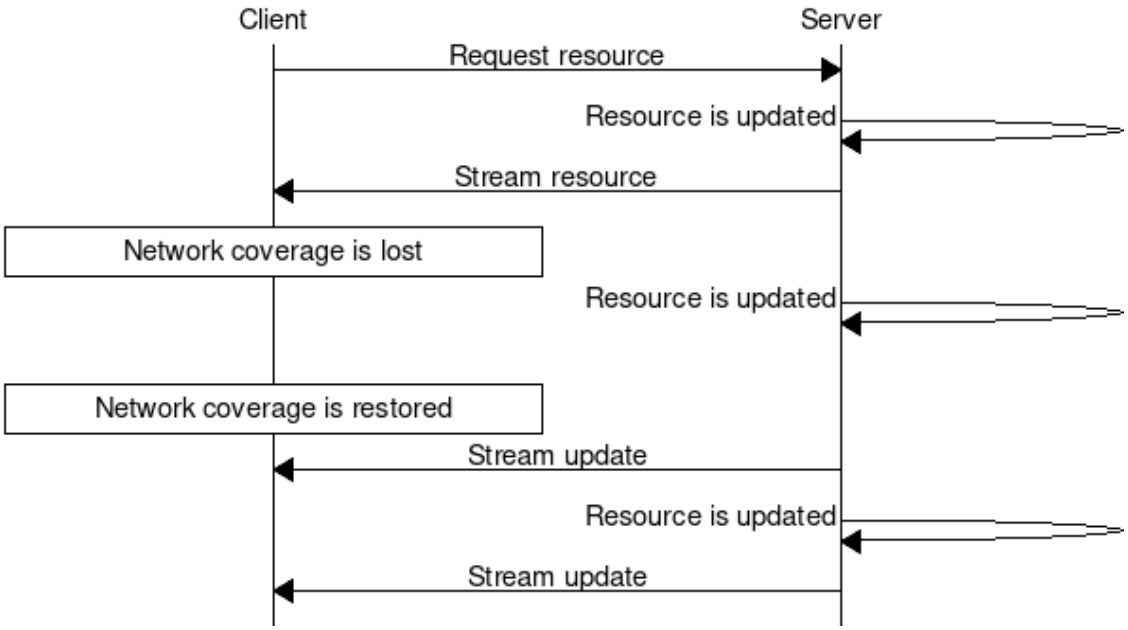


Figure 2.9: HTTP long polling mechanism, only when the resource is updated, new data is returned to the client. The connection stays open until an update is returned to the client.

SSE operates (figure 2.9) as followed:

1. The client connects to the SSE resource of the server.
2. When the server has new data available, it pushes the data to the SSE resource.
3. The client receives an event with the data and process the data.

2.2.2.5 gRPC

gRPC is a Remote Procedure Call (RPC) framework based on HTTP/2 and Google's Protocol Buffers [38]. Thanks to Protocol Buffers, gRPC is able to run very efficiently and makes it easy to scale applications because of its architecture. This data format was originally developed at Google to provide a seamless and transparent way of communication between clients and servers. It is now available as open source for anyone.

gRPC is capable of delivering real time data updates thanks to the bi-directional streaming support. However, gRPC uses Protocol Buffers as it is data format which violates the principles of Linked Open Data, as explained before in section 2.2.1.4.

2.2.2.6 WebSockets

WebSockets (WS) do not use HTTP to transport the data but it uses a direct TCP (Transmission Control Protocol) connection to communicate bi-directional between the server and the client [39].

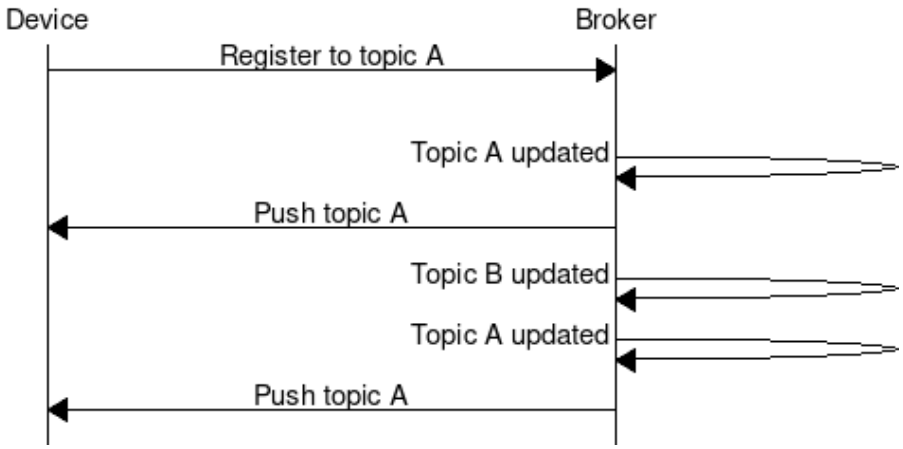


Figure 2.10: MQTT broker and client, the broker only pushes data to the client about the subscribed topics.

When using WebSockets, we need to implement our own custom protocol on top of WebSockets to handle errors and other features that are normally automatically handled by the HTTP protocol. Most modern web browsers support WebSockets, but some might support only a limited amount of WebSockets features [40].

WebSockets are an ideal protocol for instant messaging. The Slack API[18] heavily uses WebSockets to send and receive messages in their applications and bots.

2.2.2.7 MQTT

MQTT (MQ Telemetry Transport) [41] is a standardised connectivity protocol since 1991 that is designed to run in unreliable environments. MQTT is mostly used in the IoT (Internet of Things) world and it is not adopted by web browsers in contrast to WebSockets (section 2.2.2.6).

The design goals of MQTT are the following:

- **Simple and lightweight:** The user can use a low-bandwidth connection or a low-power device.
- **Reliable:** MQTT can run on high-latency or unreliable networks without any issue.
- **Subscribing:** MQTT works by subscribing to a resource instead of polling.

MQTT uses a broker (server) which filters all the messages that are transported over the MQTT network. Each device subscribes (registers) to a topic (resource) if it wants to receive messages about that topic. If the device wants to send a message about a certain topic, it publishes (sends) the message to the topic on the broker. The broker of the MQTT network decides which devices will receive the message and which not. Devices that did not subscribe to the topic are not waked up, and do not need to spend resources to messages that are not useful for them.

[18] https://api.slack.com/

Table 2.3 An overview of the different protocols that are commonly used on the Web.

	Widely used	Stale cache	Client's effort	Server's effort	Streaming support
HTTP polling	Yes +	Yes -	Moderate +/-	Moderate +/-	No -
HTTP long polling	No -	No +	Big -	Big -	No -
HTTP chunked streaming	Yes +	No +	Little +	Little +	Yes +
Server-Sent-Events	Yes +	No +	Little +	Little +	Yes +
gRPC	Yes +	Yes -	Little +	Little +	Yes +
WebSockets	Yes +	No +	Moderate +/-	Moderate +/-	Yes +
MQTT	Yes +	No +	Little +	Little +	No -

As we said before, MQTT is heavily used in the IoT world, but it is also used by instant messaging platforms like Facebook Messenger. Facebook adopted the MQTT specification[19] since they wanted a transport protocol that is lightweight, reliable and efficient when the user has a poor network coverage or uses a low-power device.

2.2.3 Suitable technologies for real time Open Data publishing and consuming

We discuss first which is the right technology for real time Open Data. Afterwards, we propose several improvements for the chosen technology.

2.2.3.1 Selecting the right technology for real time Open Data

In the previous sections, we looked at different data formats and transport protocols for real time Open Data. However, what is the most suitable technology for real time Open Data?

First of all, the data must be readable by humans and machines in an easy way as proposed by the 5-stars deployment schema for Open Data. To comply with this requirement, only text based data formats are considered for real time Open Data.

Secondly, the cost for hosting and distributing the data should be cost-efficient for both parties. We already compared XML and JSON with each other in section 2.2.1.2, JSON is a clear winner here when it comes to efficiency and costs. Less resources and bandwidth are needed to transport and process JSON than XML. In our implementation of Linked Connections we will use JSON-LD as data format to distribute the Linked Connections fragments because JSON-LD is the most suitable data format available today for Linked Open Data.

[19] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

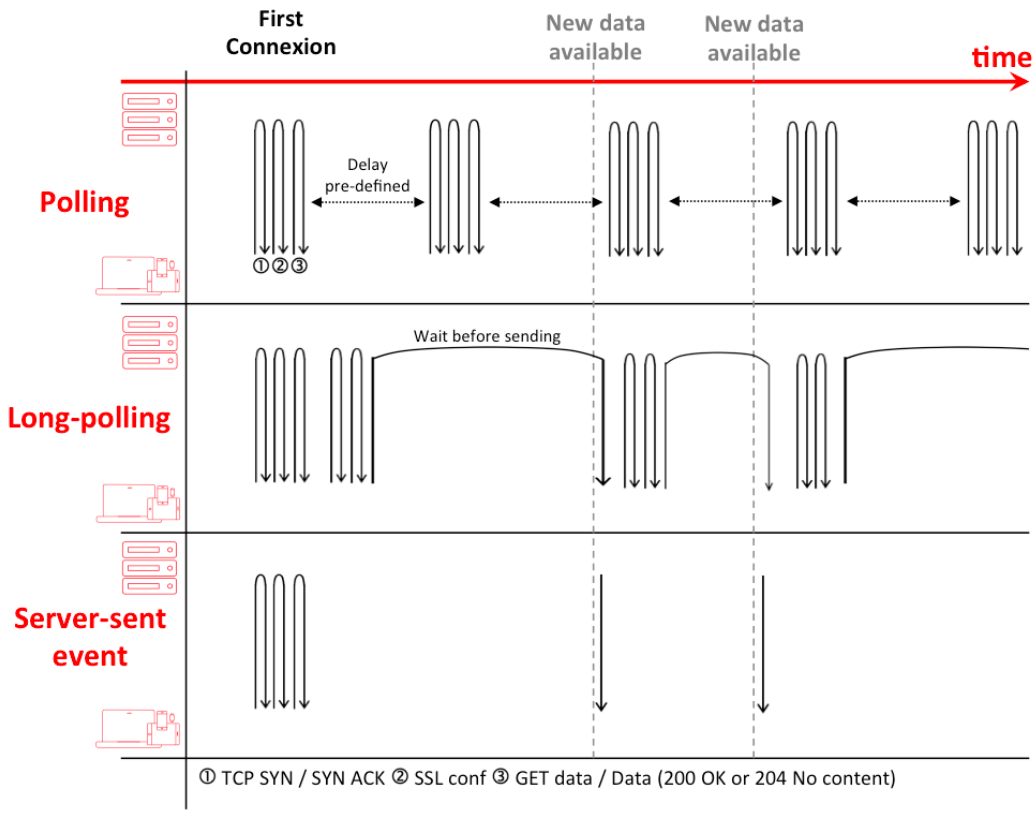


Figure 2.11: Retrieving updates using polling or streaming [8]. Streaming requires less interaction between the client and the server to retrieve the same amount of data than polling. This technique allows us to significantly reduce the amount of allocated resources for the client and the server to interact with each other.

Last but not least, the transport protocol is also an important factor. Real time Open Data should be available for anyone without any restrictions. By using a common protocol, like HTTP it is easy for any consumer to access the data. Non-HTTP protocols are less suitable for distributing Open Data since they are not directly supported by web browsers and other consumers. The current implementation of Linked Connections is based on HTTP polling. When the amount of real time Open Data increases, HTTP polling might be using too much resources of the server and increase the cost of distributing Linked Connections, this problem is illustrated in figure 2.11.

We believe that a push approach like SSE can be a solution for this problem. By subscribing each client to a resource – where we only publish delta updates of the data – we should be able to reduce the bandwidth and resource usage of delivering updates to the consumers.

2.2.3.2 Further improvements

The transport of the data can be further optimised by using several techniques like compressing, pagination, . . . We will focus on techniques that are commonly used with the HTTP protocol since that is the most suitable technology available today for real time Open Data publishing and consuming, as explained in section 2.2.3.1.

Compressing One of the most used methods to reduce the response size is by compressing the data using a compressing algorithm. When the client does a HTTP request to the server, it advertises which compression algorithms are supported by the client using the `Accept-Encoding` header. The server replies with the compressed data along side with the `Content-Encoding` header, the client knows which compressing algorithm should be used then for decompression. Some compression algorithms that are widely used on the Web are: `gzip` and `deflate`.

Compression leads automatically to an amount of overhead, it is only useful when the size of the data is big enough.

Pagination Pagination is a technique to split up a big response into several smaller ones. This is heavily used in case of Linked Connections where a huge data set of connections is divided into smaller fragments (pages), sorted by departure time. Each page contains 50 kB of connections[20]. The pages are indexed by the departure time of the first connection of each page.

Each fragment has a next and previous URI field, if the client wants more data, the client only has to follow the URI to the next or previous fragment. The URI of a Linked Connections page is as followed: `http://example.com/{agency}/connections?departureTime={departure time}`. By changing the departure time query in the URI, we can navigate through the Linked Connections pages.

HTTP/2 push The HTTP/2 specification [42] allows servers to push resources to the client together with the request of the client. We will explain this using a example of pagination (section 2.2.3.2):

- The client needs to fetch 10 different pages from the server.
- The client starts to fetch the first page and tells the server that it will need the next 9 pages too.
- The server will retrieve those 10 pages at the same time and send the first one back, while pushing the next 9 pages to the client using the HTTP/2 push feature.
- The client requests the next pages, but all those pages are already available. The client can get them directly from its local cache.

The result of this feature is that only 1 real HTTP request is executed by the client to get all the 10 pages. This reduces the HTTP overhead and allows the client to serve more pages from cache.

[20] This value can be modified in the settings of Linked Connections Server.

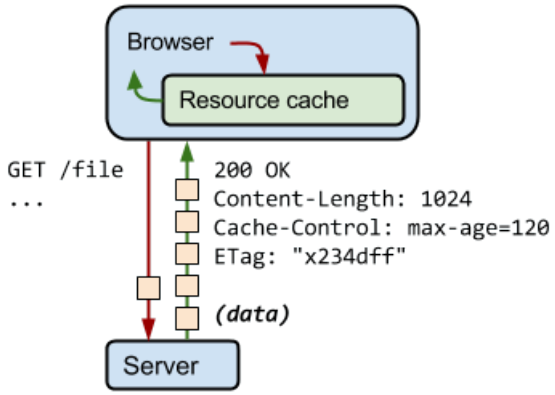


Figure 2.12: HTTP caching negotiation between the client and the server [9].

Caching In order to reduce the bandwidth and resource usage, *caching* can be applied [43]. Each resource that is received is stored in the cache – as soon as the same resource is requested again – it is loaded from cache (if the data is still valid). In chapter 2.3, we will explain more about caching and how it can be used to keep the client's cache up to date in a cost efficient way.

2.3 Keeping the client's cache up to date in a cost-efficient way

Resources are expensive, especially if the outcome is the same as before.

Every time a client requests a page on the Internet, a lot of data[21] needs to be transferred from the server to the client, even if the content is the same as before. To avoid these situations, the HTTP protocol introduced caching. Caching allows the client to determine if a certain resource in its cache is still valid and avoids unnecessary requests to the server. If we want to create a cost-efficient Linked Connections client, caching will be an important player to achieve this goal.

HTTP caching [43, 44, 45] is all based on the `Cache-control` header[22]. This header provides a way for the client and the server to communicate about the caching of a HTTP request. If a client needs to request a resource from the server, it will first check its cache for the resource. If the resource is still valid (fresh), the data is served from cache without performing a HTTP request to the server (if allowed by the server). If not, the data is stale and needs to be fetched from the server. In figure 2.12, the caching negotiation process is explained with a browser as client.

An alternative HTTP caching mechanism is available through the `Expires` header. We will not use this header since the same functionality can be achieved with the `Cache-control` header and when the `Cache-control` is provided by the server, the `Expires` header is ignored by the client.

[21] Because of the usage of JavaScript libraries, images and webfonts on each web page, most pages are already bigger than 2 megabyte according to Content Delivery Network providers (https://www.keycdn.com/support/the-growth-of-web-page-size).

[22] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

2.3.1 Cacheability

The cacheability of the data can be specified in `Cache-control` header using the following directives:

- `public`: The data may be cached in a public shared cache.
- `private`: The data may be cached, but in a private cache. This mode is mostly used when handling sensitive data.
- `no-cache`: The client must check first if a new version is available before serving the data from cache.
- `only-if-cached`: The server wants the client to retrieve the data once and serve it then from its cache.

Since we are working with Open Data, the cacheability can be set to `public` for static data. However, the real time data requires to set the `Cache-control` header to `no-cache` to make sure that the real time data is still up to date. The client will check this using a revalidation request (section 2.3.3).

2.3.2 Expiration

The `Cache-control` header also allow us to specify for how long the data is still fresh using several directives:

- `max-age=<seconds>`: The maximum amount of time before a resource becomes stale relative to the time of the request.
- `max-stale[=<seconds>]`: The client accepts a stale response from the server. Optionally, the amount of time can be supplied.
- `min-fresh=<seconds>`: The client wants a response from the server that is still valid for a certain amount of time.

With real time Open Data, the data can change at any point in time. We can not guarantee that the resource is not going to be stale in a specified amount of time. By splitting the real time Open Data into static data and real time data, we can easily cache the static data for a long period of time.

We will illustrate this using an example: information about the trip of a vehicle is almost constant in time, it is cacheable for at least a whole day while the delay information about the vehicle's trip is changing continuously in time and can not be cached.

2.3.3 Revalidation and reloading

The way a client revalidates and reloads a resource can be set in `Cache-control` header with these directives:

- `must-revalidate`: The client has to verify the status (revalidate) of a resource before using it. Expired resources should be removed from the cache.
- `immutable`: The response will not change over time, the client needs to wait until the resource expires before revalidating the resource.

We already separated the real time and static data from each other in the previous paragraph, here we can apply the same idea. The static data is not changing over time until it expires. We can apply the `immutable` directive to this data, while the real time data needs the `must-revalidate` directive since it may change continuously in time.

When a resource needs to be revalidated, the client can do this in several ways using the HTTP protocol: `ETag`[23] and `Last-Modified`[24] headers provide several mechanisms to accomplish this.

2.3.3.1 ETag

ETags are hashed versions of the data, the server sends this tag together with the data in the `ETag` header. If a client wants to revalidate a resource, the client sends the HTTP request to the server with the `If-Match: <ETag>`[25] header. When the server receives the request, it will look up the data and check the ETag. If both matches, the server sends directly a `HTTP 304: Not Modified` response with an empty body.

This technique reduces the amount of bandwidth and the response time of the server if the data has not been changed.

2.3.3.2 Last-Modified

The `Last-Modified: <timestamp in GMT>` header, which the server sends together with the data, indicates when the requested data were last modified on the server. When the client sends its revalidation request, it will add the `If-Modified-Since: <timestamp in GMT>`[26] header. The same principles apply as with the ETags, the last modification time is checked and a `HTTP 304: Not Modified` response is returned when both timestamps are the same.

Since this method its accuracy is limited to a second, it is a fallback mechanism for the ETag system and has the same benefits as the ETag system: bandwidth usage and the response time of the server are reduced.

[23] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag
[24] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Last-Modified
[25] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Match
[26] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Modified-Since

2.4 Connection Scan Algorithm

In the previous chapter, we looked at several ways to deliver real time data to the client in order to keep its cache up to date. However, a client must also be able to consume these real time data in an efficient way. In the case of routing algorithms, we would like to avoid to completely recalculate all the routes of a commuter's journey when the data has been changed. The reason for this requirement is straightforward: commuters use most of time a mobile device to plan their journeys. A mobile device has a limited amount of resources (CPU power, battery, ...) at its disposal.

As explained before in section 2.1.3.4, we use the Connection Scan Algorithm (CSA) [5] to perform the route calculation for a commuter's journey. We will concentrate on the CSA profile variant since this variant allows us to easily extend the algorithm with additional fields for consuming updates in real time.

2.4.1 Terminology

Before we can explain how the CSA profile variant operates, we have to define several terms for the routing algorithm. CSA uses the following terms to describe a route:

1. **Connection**: A connection is the smallest part of a *route* and is defined between two *stops* with an departure time and arrival time.
2. **Stop**: A stop is a place where commuters can departure from or arrival to.
3. **Vehicle**: A vehicle can pick up and drop off commuters at a *stop* and drives on a certain *connection*.
4. **Route**: A route contains a list of *connections* (legs) that allows the commuter to go from his departure *stop* to his destination *stop*.
5. **Journey**: A journey has a list of possible *routes* for the commuter to get from the departure *stop* to the arrival *stop* in a given time range.

In snippet 2.8, we show how a *connection* is translated into the Linked Connections specification [1, 46, 47]. Each connection has an unique URI (`@id`) since Linked Connections is Linked Open Data. This allows us to uniquely identify each connection by using the URI. Using the `@type` field we know that the data is a connection. The departure *stop*, departure time, arrival *stop* and arrival time are defined for this connection, the delays (in seconds) are already included in the timestamps as specified by the Linked Connections specification. The *vehicle* associated with this connection is given by the `gtfs:route` field, while the specific trip of the vehicle is available using the `gtfs:trip` field. The `direction` field allows us to show the commuter in which direction the vehicle is going.

```
{
      "@id": "http://irail.be/connections/8813045/20181018/P8600",
      "@type": "Connection",
      "departureStop": "http://irail.be/stations/NMBS/008813045",
      "arrivalStop": "http://irail.be/stations/NMBS/008812005",
      "departureTime": "2018-10-18T13:50:00.000Z",
      "arrivalTime": "2018-10-18T13:52:00.000Z",
      "departureDelay": 180,
      "arrivalDelay": 120,
      "direction": "Namur",
      "gtfs:trip": "http://irail.be/vehicle/P8600/20181018",
      "gtfs:route": "http://irail.be/vehicle/P8600"
}
```

Snippet 2.8: A connection defined according to the Linked Connections specification as JSON-LD

```
class TrainProfile {
    // The arrival time at the final destination
    QDateTime arrivalTime;

    // The number of transfers until the destination
    // when hopping on to this vehicle
    qint16 transfers;

    // The arrival connection for the next transfer or arrival
    QRail::Fragments::Fragment *arrivalConnection;
}
```

Snippet 2.9: The structure of a TrainProfile in Qt C++

The profile variant of the CSA is based on "*profiles*". There are 2 different profiles in use by the algorithm: `TrainProfiles` and `StationStopProfiles`. The `TrainProfiles` (snippet 2.9) are kept in the T array and the `StationStopProfiles` (snippet 2.10) in the S array. The CSA profile variant is described in detail in the CSA paper [5].

2.4.2 The Connection Scan Algorithm profile variant

The profile variant of the CSA scans each connection from the arrival stop to the departure stop. When a possible route is found, we can extract the legs of the route and add it to the journey of the commuter. When all connections are scanned in a specific time range ($[time_{\text{departure}}, time_{\text{arrival}}]$), the algorithm stops and the journey will contain a list of possible routes for the commuter.

```
class StationStopProfile {
    // The departure time in this stop
    QDateTime departureTime;

    // The arrival time at the final destination
    QDateTime arrivalTime;

    // The departure connection in this stop
    QRail::Fragments::Fragment *departureConnection;

    // The arrival connection for the next transfer or arrival
    QRail::Fragments::Fragment *arrivalConnection;

    // The number of transfers between standing
    // in this station and the destination
    qint16 transfers;
}
```

Snippet 2.10: The structure of a StationStopProfile in Qt C++

In the CSA paper [5], the necessary steps for each connection are explained. All these connections are sorted by departure time in descending order as required by the algorithm. The QRail library implements the CSA profile variant to calculate the possible routes of a journey.

The source code of the implementation in the QRail library can be found on Github[27].

2.4.3 Influence of changes on the CSA profile variant

Due to the nature of the CSA profile variant, we are depending on the previous state of the T and S array. The T and S array must be filled by going over the connections sorted by departure time in descending order. However, if we change any parameters what will be the influence on the algorithm?

The profile variant of the CSA depends on the following parameters:

- Departure and arrival time
- Departure and arrival stop
- Transfer time

[27] https://www.github.com/DylanVanAssche/QRail/blob/master/src/engines/router/routerplanner.cpp

If we change one of these parameters, almost every route of the journey needs to be recalculated. This happens a lot when you are using real time data on the client. As soon as a vehicle has a delay for example, the transfer time is affected. The commuter might miss his transfer and has to take another route to arrive at his or her destination. In case there is no other route directly available, the client needs to run the algorithm again with different departure and arrival times, if the commuter is already at an intermediate station when he or she missed transfer for example, the departure station also changes.

All those possible changes could lead to a recalculation of all the routes. This process can take some time, depending on how much connections have to be processed by the client[28]. If we are able to reduce the amount of connections that need to be processed when a change to the data is made, the CSA will be able to provide faster an alternative solution for the commuter and consume less resources in the process.

2.5 Conclusion

In this chapter, we first looked at the current technologies to publish and consume real time Open Data to select the right technology that is cost-efficient and well supported by most clients for the Web. It became clear that JSON and JSON-LD as data format are suitable for real time Open Data publishing thanks to their easy to read (for humans and machines) notation, fast parsing in comparison to other data formats and low bandwidth usage.

Afterwards, we focused on the transport protocol to transport these data between the server and the client. After comparing several widely used transport protocols on the Web, we can conclude that HTTP and SSE are the best transport protocols available today to publish real time Open Data on the Web since they are a widely adopted standard on the Web which goes hand in hand with the 5 stars deployment scheme for Linked Open Data.

In order to optimise the transport of the data even further using the HTTP protocol, we looked at several aspects of HTTP caching like: cacheability of the data, expiration and revalidation and how it is able to reduce the necessary resources to transport the data for all parties.

We explained first how the CSA profile variant operates. The CSA profile variant creates a list of profiles and scans them to find the fastest way to get the commuter between 2 stops. However, if the data was changed in the mean time, the algorithm must recalculate all the possible routes of the commuter's journey.

In order to consume these updates in a cost-efficient way, we discussed the influence of each parameter on the algorithm: the departure and arrival time, the departure and arrival stop and the transfer time. If one of these parameters are changed, the algorithm must recreate the journey of the commuter from scratch.

[28] An average mid-range smartphone can recalculate all the routes of a journey in approximately 15 seconds.

3 Implementation

In this chapter we describe our implementation and approaches to publish real time Open Data. First, we give an overview of the architecture. Secondly, we will discuss our modifications to the Linked Connections Server. Afterwards, we describe our own written libraries and clients (QRail and LCRail). Finally, we describe the modifications made to our libraries for processing real time updates.

3.1 Specifications

In this thesis, we want to inform the user about his or her public transport journey in a better way.

In case a vehicle is delayed or cancelled, we want to provide immediately an alternative journey in an cost-efficient way.

The user must also be able to view his or her journey through an UI. To improve the user experience, a notification must be generated when the journey has been updated.

3.2 Architecture overview

In figure 3.1, we show our architecture which we use in this thesis. Our architecture contains 3 main parts:

1. Linked Connections Server
2. QRail library
3. LCRail client

The Linked Connections pages are served on the WWW by the Linked Connections Server. The server runs a NodeJS web server behind a NGINX reverse proxy[1]. The Linked Connections Server contains 2 separate processes, with each their own task. The `web-server` process provides all the necessary HTTP communication and caching. The data processing part is handled by the `datasets` process block. Our Events Manager (section 3.3.3) and the `gtfsrt2lc` library (section 3.3.1) are contained in the `datasets` process. Our real time resource can be found in the `web-server` process (section 3.3.4).

The QRail library (section 3.4) is responsible for all the interaction with the Linked Connection Server over the WWW. The QRail library handles the HTTP communication. The library transforms the JSON-LD Linked Connections pages into objects which are used by the planner and liveboard algorithms. This process is handled asynchronously in the QRail library using Qt signals.

The same mechanism is also used by the LCRail client (section 3.5) to interface with the library. This approach keeps the UI (User Interface) of the client responsive at all times. The LCRail client provides interaction with the user through a Qt 5 UI.

[1] This setup is commonly used in a production environment. The reverse proxy server NGINX handles the HTTPS and domain configuration, network connections, caching and compression. Our application runs on the NodeJS instance. NGINX exposes the application on the privileged HTTP port (80).

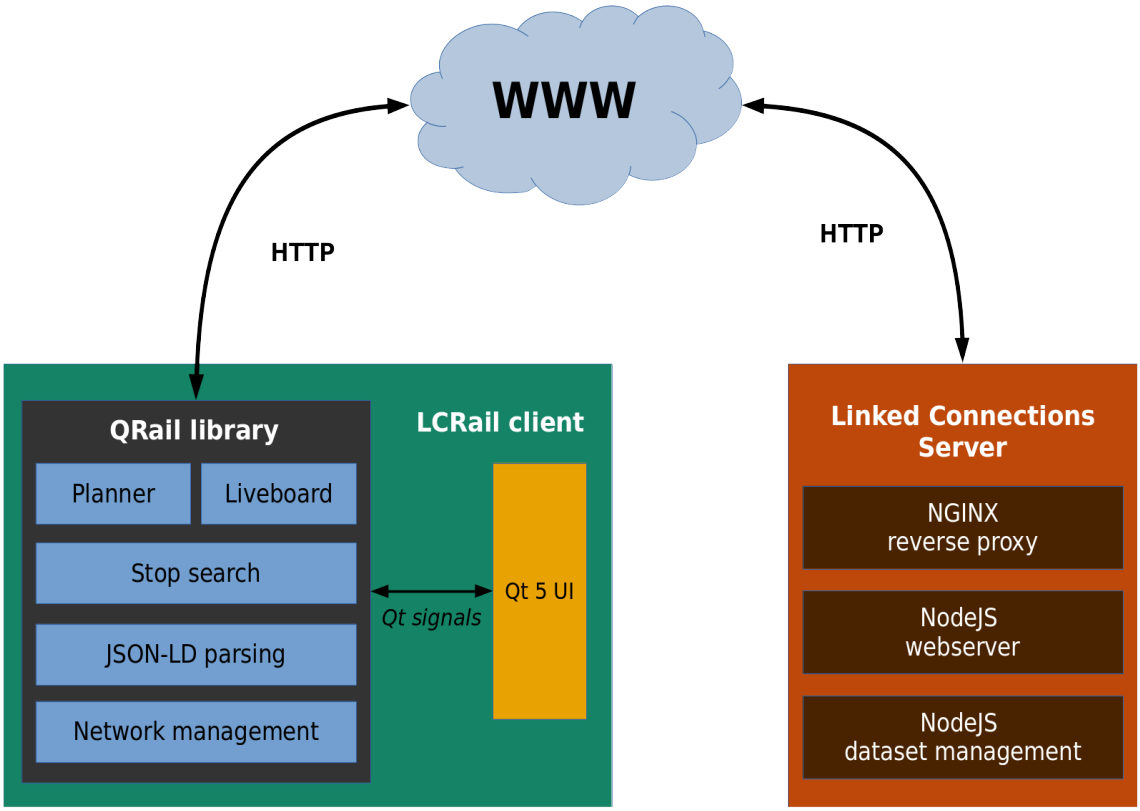


Figure 3.1: The architecture overview of our Linked Connections setup. The LCRail client interacts with the QRail library to answer the question of the user using Qt signals. The QRail library retrieves all the data and executes the necessary algorithms for the LCRail client. The Linked Connections Server publishes all the Linked Connections pages to the WWW.

3.3 Linked Connections Server

The Linked Connections Server is responsible for publishing the Linked Connections pages on the WWW. The Linked Connections Server can be divided in 3 main parts:

- **Catalog**: All the published data can be discovered using the `/catalog` resource. The client receives a list of links to all the data sets the server is publishing. This part of the server is not related to the scope of this thesis.
- **Connections**: Each client can retrieve the list of Linked Connections of each public transport agency from the `/{agency}/connections?departureTime={ISO timestamp}` resource. By specifying the `departureTime` parameter, we can navigate through the Linked Connections pages. The `departureTime` parameter specifies the departure time of the connections in a Linked Connections Page. Since the server publishes the same list of Linked Connections for each client, the data can be easily cached on the server.
- **Memento history querying**: Clients can also query the server for historical Linked Connections data using the `Accept-Date` header. This part of the server is not related to the scope of this thesis.

In this thesis, we focus on publishing real time Open Data in a cost-efficient way. To achieve this goal, we modified the Linked Connections Server by adding a real time resource (`/agency/events`) to the server. The motivation for this modification is based on the fact that only a subset of the content of Linked Connections pages, changes when a vehicle is cancelled or delayed. Therefore, instead of re-downloading the complete pages upon data updates, clients access directly the specific updates of each page through this resource.

3.3.1 Transforming GTFS-RT into Linked Connections

All these real time data is extracted from the NMBS GTFS-RT data stream (section 2.1.3.1). To parse this stream, we use the `gtfsrt2lc` NodeJS library[2]. This library transforms the gRPC GTFS-RT stream into Linked Connections (JSON-LD). However, this library was not adapted yet to the new publication method of the NMBS. The GTFS-RT specification mentions that the agency is not required to send the complete updated trip of the vehicle in their GTFS-RT stream. They only have to send all the stops where the real time information actually was changed. The client must apply this information himself for all intermediary stops between these update stops.

The current algorithm did not have access to the complete trip of each vehicle. This information is stored in the `stop_times.txt` GTFS file. Since querying such a file is time consuming[3], we created an index of this file. Afterwards, we wrote an algorithm (snippet 3.1) which can keep track of the last update stop. This algorithm can apply the real time information to all stops after the update stop. When a new update stop has been encountered, the whole process is repeated.

[2] https://github.com/linkedconnections/gtfsrt2lc

[3] For example, the NMBS `stop_times.txt` file size is around 20 MB.

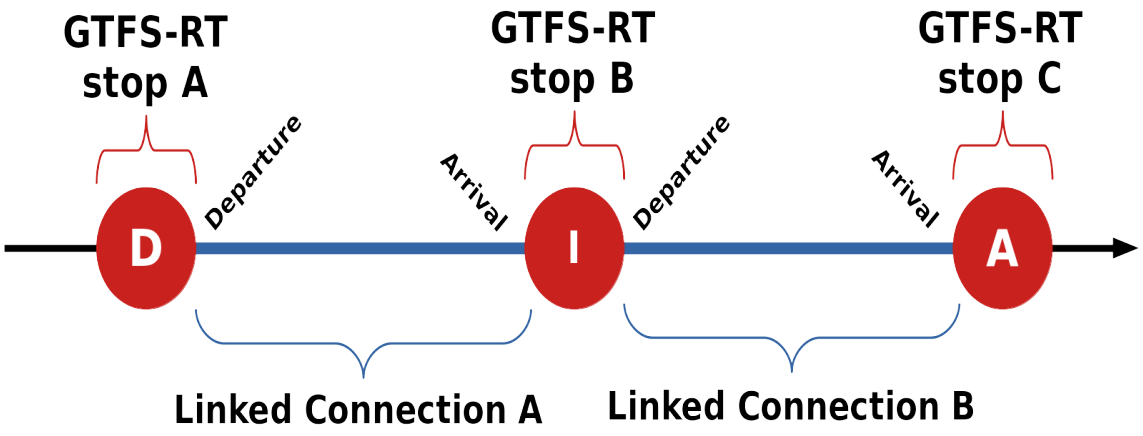
```
// Process every new update by updating the delays
if(stop['stop_id'] === update['stop_id']) {
        // We are in the middle of a trip
        if(update['departure'] && update['arrival']) {
        departureDelay = update['departure']['delay'];
        arrivalDelay = update['arrival']['delay'];
        }

        // The start of the updates only contains the departure delay
        if(update['departure'] && !update['arrival']) {
                departureDelay = update['departure']['delay'];
                arrivalDelay = 0;
        }

        // The end of the updates only contains the arrival delay
        if(!update['departure'] && update['arrival']) {
                departureDelay = 0;
                arrivalDelay = update['arrival']['delay'];
        }

        // Update the index counter
        update_index++;

        // Retrieve the update for the next iteration and conflict avoidance
        update = stop_times_updates[update_index];
}
```

Snippet 3.1: Converting the GTFS-RT delays to Linked Connections delays.

The algorithm must check if the stop is an arrival, departure or intermediate stop when it is calculating the delay of the vehicle. If we're at the departure stop (start of the vehicle's trip) for example, the vehicle can never have an arrival delay. At an intermediate stop, the vehicle will have both type of delays: the delay when arriving at the stop and the delay when departing from the stop. At the arrival stop of the vehicle (the final stop of the vehicle), the vehicle can only have an arrival delay. This is visualised in the figure 3.2.

```
// At the arrival stop, we do not have to look into the future
if(update_index < stop_times_updates.length) {
        // Check if our next stop, will be the next the stop update
        if(stop_times[j + 1]['stop_id'] === update['stop_id']) {
                arrivalDelay = update['arrival']['delay'];
        }
}
```

Snippet 3.2: A conflict is created when a new stop update will be found in the next iteration. The update variable is already set to the next update. If our next stop is the same as the next update stop, we resolve the conflict.



D	Departure stop
I	Intermediate stop
A	Arrival stop

Figure 3.2: The difference between the arrival and departure delays for GTFS-RT and Linked Connections. GTFS-RT specifies both at a stop, while Linked Connections defines both for a connection.

The calculated delays are applied on each stop of the trip until a new stop update has been found. A stop update is detected by comparing the `stop_id` of the next intermediate stop and the next update stop. If they are the same, the next iteration will have a new set of delays.

In case we are approaching a new update stop, the delays will change in the next iteration. This creates a conflict: the arrival delay of the previous stop update is used for the connection which arrives in the next update stop. This is wrong since the next update stop has a different arrival delay than the current stop update. We can summarise this to: the last connection – before a stop update – must have the arrival delay of the stop update, which we will encounter in the next iteration (figure 3.3).

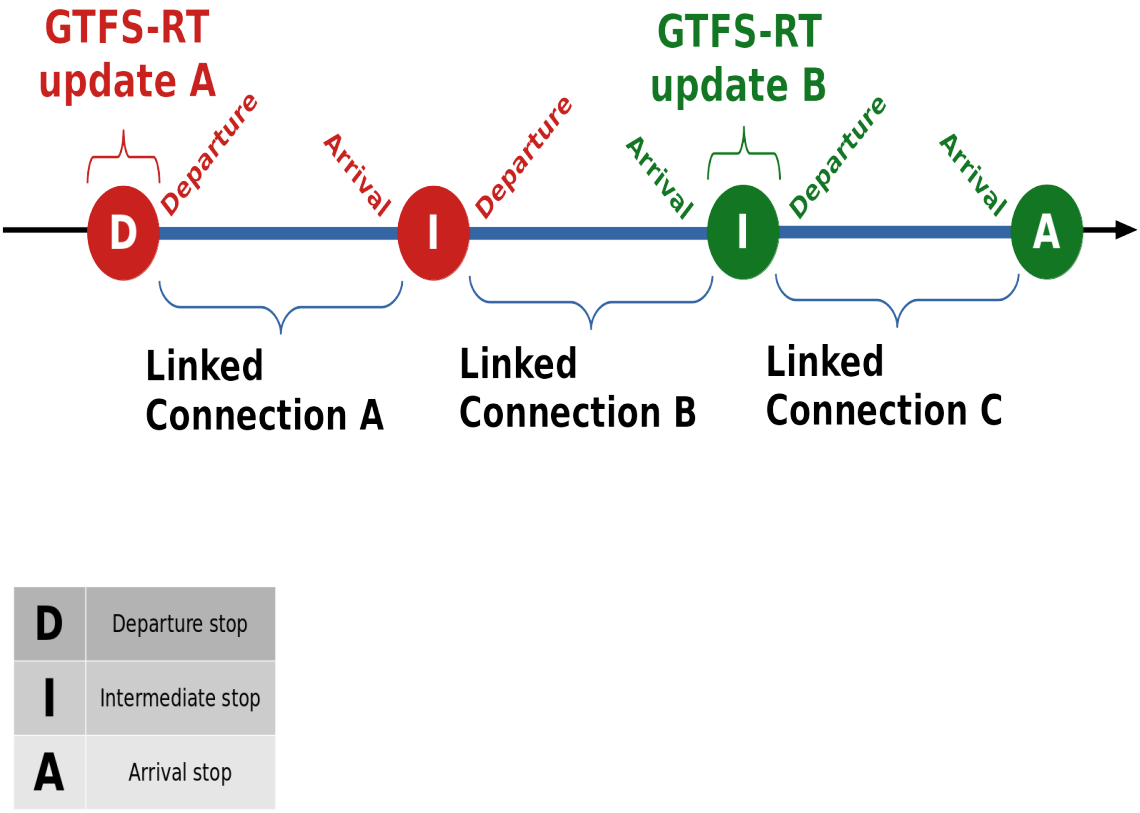


Figure 3.3: The GTFS-RT update B has a different arrival delay than the GTFS-RT update A. This will introduce a conflict: Connection B will normally have the arrival delay of the update A, but it has to be the arrival delay of update B.

By checking the conditions for this conflict before we reach this next update stop, we can update the arrival delay of the last connection correctly. The source of this conflict relies in the fact that the GTFS-RT specification specifies every stops with their accompanying arrival and departure delays. However, Linked Connections describes each connection between 2 stops (figure 3.2). Thanks to this new algorithm, the Linked Connections Server can now correctly update all the affected connections of a vehicle's trip when a GTFS-RT update is received.

```
getConnectionType(entity) {
        // ScheduleRelationship.CANCELED = 3
        let schedule_relation = entity.trip_update.trip.schedule_relationship;
        let is_deleted = entity.is_deleted;
        if (is_deleted || schedule_relation == 3) {
                return 'CanceledConnection';
        }
        else {
                return 'Connection';
        }
}
```

Snippet 3.3: Supporting cancellations in the gtfsrt2lc library.

3.3.2 Cancellations support

If a trip or a part of a trip has been cancelled by an agency, this information will be made available through the GTFS-RT stream. As mentioned before, we use the `gtfsrt2lc` library to parse these data. Unfortunately, the library did not support this feature yet. According to the GTFS-RT specification, the cancellation of a trip is advertised by `schedule_relationship` and `is_deleted` properties. The last property is only used when a complete trip has been cancelled. The `schedule_relationship` property can have multiple values, according to the GTFS-RT specification[4]:

- **SCHEDULED = 0**: Normal operation, the vehicle is following the GTFS schedule.
- **ADDED = 1**: Additional trips are added for the vehicle.
- **UNSCHEDULED = 2**: A vehicle that isn't associated with the GTFS schedule.
- **CANCELED = 3**: The vehicle was cancelled.

We created a small function that converts this information into Linked Connections vocabulary. By default, each connection has the`"@type": "Connection"` type. However, if the connection was cancelled, it gets the `"@type": "CanceledConnection"` type. With the `@type` property, the client can check if the connection was cancelled or not during routing.

[4] https://developers.google.com/transit/gtfs-realtime/reference

3.3.3 Publication of events

Every time a GTFS-RT update is received by the Linked Connections Server, it contains the real time data of the last couple of minutes. However, we do not want to publish the same event more than once to the clients. To overcome this issue, we created the Events Manager. This manager integrates with the Dataset Manager[5]. Every time the Dataset Manager receives a GTFS-RT update, the generated Linked Connections are passed to the Events Manager.

The Events Manager keeps track of all pending events of every agency. These events are received by the Events Manager, but they are not published yet. Every time an event is received by the Events Manager from the Dataset Manager, the pending events are updated. If the event is already pending, it is updated. In the other case, it is added to the pending events list (snippet 3.4).

The Events Manager processes the pending events as soon as the Dataset Manager received the complete GTFS-RT stream. All the pending events are encapsulated into a data structure with the right Linked Data Vocabulary and written to a JSON-LD file. We use the Sensor, Observation, Sample and Actuator (SOSA) ontology to publish the events. The main reasons for using the SOSA ontology for events data publishing are:

- **W3C ontology**: This vocabulary is widely spread, actively used and still developed by the W3C.
- **Designed for observations**: Each event we publish is an observation of a connection.
- **Reusable**: Clients that are able to visualize SOSA data, can easily pickup our observations of each connection.

We also looked into the Timeline ontology and the Event ontology by Yves Raimond and Samer Abdallah for the events publication. However, these vocabularies are less suitable than the SOSA ontology. The Timeline and Event ontologies are more used for describing events in songs or videos. In comparison to the SOSA ontology, which was developed for publishing events in real time situations.

We created an additional type (`Event`) for the Linked Connections vocabulary to describe an event. We applied here again the Hydra[6] vocabulary to navigate between pages. The Hydra vocabulary (`hydra:view`) is also used to provide a link to the page where the updated Linked Connection can be found on the `/connections` resource[7]. The observation is available through the `sosa:hasResult` property, while the time of the observation can be retrieved from the `sosa:resultTime` property.

[5] The Dataset Manager handles all the data set processing in the Linked Connections Server.

[6] Linked Data vocabulary to create Hypermedia-Driven Web APIs, more information is available at: https://www.hydra-cg.com/spec/latest/core/

[7] The developer can decide to fetch the page or update the cache using the provided connection. The first approach is easier than the second one. However, it requires more bandwidth than using the updated connection directly.

```
let existingEvent = this.pendingEvents[agency].find((event) =>
                {event['connection']['@id'] === newEvent['@id']});
let isNew = false;

// Only replace when event already exists
if(typeof existingEvent !== 'undefined') {
        // Pending event has been updated
        if(type !== newEvent['@type']
                || departureDelay !== newEvent['departureDelay']
                || arrivalDelay !== newEvent['arrivalDelay']) {
                // Existing event updated, removing the old one
                let index = this.pendingEvents[agency].indexOf(existingEvent);
        this.pendingEvents[agency].splice(index, 1);
        isNew = true;
        }
}
// New event
else {
        isNew = true;
}

// Adding new event if needed
if(isNew) {
        this.pendingEvents[agency].push({
                id: new Date(),
                connection: connection,
        });
}
```

Snippet 3.4: Events Manager pending events algorithm. The order of the events is the same as the order in which the events are received from the GTFS-RT stream.

```
{
    "@id": "http://irail.be/connections/IC1717/20190325/8814357#2019-
                03-25T18:14:00.739Z",
    "@type": "Event",
    "hydra:view": "http://lc.dylanvanassche.be/sncb/connections?
                                departureTime=2019-03-25T18:34:00.000Z",
    "sosa:resultTime": "2019-03-25T18:14:00.739Z",
    "sosa:hasResult": {
        "@type": "sosa:hasResult",
        "Connection": {
                "@id": "http://irail.be/connections/IC1717/20190325/8814357",
                "@type": "Connection",
                "departureStop": "http://irail.be/stations/NMBS/008814357",
                "arrivalStop": "http://irail.be/stations/NMBS/008814365",
                "departureTime": "2019-03-25T18:34:00.000Z",
                "arrivalTime": "2019-03-25T18:36:00.000Z",
                "departureDelay": 240,
                "arrivalDelay": 240,
                "direction": "Liege-Guillemins",
                "gtfs:trip": "http://irail.be/vehicle/IC1717/20190325",
                "gtfs:route": "http://irail.be/routes/IC1717",
                "gtfs:pickupType": "gtfs:NotAvailable",
                "gtfs:dropOffType": "gtfs:NotAvailable"
        }
    }
}
```

Snippet 3.5: Data structure of an event using the SOSA ontology.

3.3.4 Real time resource

Our last addition to the Linked Connection Server is the real time resource: `/{agency}/events`. One of the goals of this thesis is to evaluate whether Linked Connections should use a pushing or polling approach for real time data publishing. For this reason, we implemented this resource twice: once with HTTP polling and once with SSE.

We only publish the latest events to the clients since they can get all the necessary data from the `/{agency}/connections` resource anyway. In case the client's connection is lost, the client can just refetch its pages. The real time resource can't be used in this case. Allowing clients to navigate through past events will result in a significant increase of the server's workload. Refetching the pages does not have a big impact on the client because it can make use of HTTP caching in this case. Each refetch that results in the same page, returns immediately a HTTP 304 `Not Modified` status.

3.4 QRail library

We will work with our own written QRail library[8] to consume Linked Connections data in an easy way. This Qt[9] library currently focuses on the NMBS, but can be extended with other public transport agencies as soon as the data are published. This is only possible thanks to the standardisation of the Linked Connections specification.

The QRail library provides the developers of Linked Connection applications to implement intelligent clients without any effort. QRail contains a modified version of the CSA algorithm[10] in order to improve the amount of transfers and transfer stops, based on the previous work of Bert Marcelis [6].

For example, the NMBS stations: Brussels-South, Brussels-Central and Brussels-North sometimes introduce a loop[11] into the routes calculated by the original version of the CSA profile variant.

The CSA implementation is also improved by adding the Earliest Arrival Time CSA variant in front of the CSA profile variant. By doing this, we can already skip non reachable connections before applying the slower CSA profile variant on the connections. This approach provides a reduction in the connections that needs to be scanned by the algorithm. In the beginning, the amount of connections is reduced from more than 100 to less than 10 connections. This advantage is lost as soon as the CSA profile variant goes deeper in the routes calculation. The deeper the algorithm goes, the more combinations of connections it encounters. The algorithm scans the connections faster at the beginning than at the end because of this optimisation.

[8] https://github.com/DylanVanAssche/QRail

[9] Qt is a cross-platform C++ framework (https://www.qt.io).

[10] https://www.github.com/DylanVanAssche/QRail/blob/master/src/engines/router/routerplanner.cpp

[11] Due to the fact that these stations are so close to each other and they have a lot of passing trains, the CSA profile variant from the paper [5] may calculate routes with connections going from Brussels-South to Brussels-North and back to Brussels-South for example.

Besides the implementation of the CSA, the QRail library also contains a *liveboard* fetcher[12]. A liveboard is the list of arriving or departing vehicles at a stop. The liveboard fetcher uses a filter to scan each Linked Connections page for any arriving or departing vehicles for a given stop. Thanks to this approach, the liveboard fetcher is much faster than the CSA route planner.

Finally, the library provides an interface to gather all the information about a stop. The information of each stop is retrieved from iRail[13] and included in the library. When the library is started, the information is inserted into a SQLite database. Thanks to this database, we can easily retrieve the information about a stop using the QRail library.

We wrote the library in several modules, each module has a certain amount of basic functionality. We give an overview of these modules and their basic functionality:

1. **Network**: Interacts with the WWW through HTTP. This module has been build around the Qt's `QNetworkAccessManager` library. It allows QRail to fetch Linked Connects pages and perform HTTP caching.
2. **Fragments factory**: Converts the data from the Linked Connections server into objects. Pages, connections, ... are parsed as JSON-LD data into Qt objects.
3. **CSA**: Provides the routing algorithm of QRail. The CSA uses the parsed data to generate a journey of routes based on the user's request.
4. **Liveboard generator**: Generates a liveboard from the parsed data on request.
5. **Stop search**: Developers can implement a stop search with this module. Users can search the stop database through the QRail interface.

Below, we discuss several improvements we did to each module, after we wrote the basic functionality. Each improvement focuses on the real time data aspect of this thesis, for example:

1. **Polling and pushing support**: SSE and HTTP polling support for the *Network* module.
2. **Routing calculation with rollback support**: Reusing the previous calculated routes when real time updates are received.
3. **Liveboard regeneration**: QRail can efficiently regenerate a liveboard by only applying the update to the liveboard.
4. **Update watchers**: Automatically updating liveboards and journeys without any interaction of the user. The developer never has to poll the library to check if a liveboard or journey is out-of-date.
5. **Real time data caching**: Caching real time data using a page cache on the client. The real time data is automatically processed to keep the cache up-to-date.

[12] https://www.github.com/DylanVanAssche/QRail/blob/master/src/engines/liveboard/liveboardfactory.cpp
[13] https://www.github.com/iRail/stations

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 10000

data: Simple message

data: {"message": "JSON data"}

event: apple
data: Message of type "apple"

id: 33
event: pear
data: Multi-line message of
data: type "pear" and id "33"

id: 34
data: Bye! id "34"
```

Snippet 3.6: SSE data frame.

3.4.1 SSE and HTTP polling in Qt

Qt has integrated support for HTTP using the `QNetworkAccessManager` library. However, Qt does not support SSE or HTTP polling out of the box. In order to use SSE and HTTP polling in Qt, we created an universal interface. The interface is based on SSE's EventSource from JavaScript.

The `QNetworkAccessManager` has already support for streaming applications. We leveraged this feature to create our SSE interface in Qt. Each time a chunk of data is received by Qt, the `bytesReceived(data)` signal is emitted by the `QNetworkAccessManager`. If the data is received from a SSE resource, the data will be a `QString`. Such data frame can be parsed like HTTP headers, which make it very easy to implement it in any programming language (snippet 3.6).

We can divide each SSE frame in the following pieces:

1. **id**: Every chunk of data can be marked with an ID. In case of a network failure for example, SSE can reconnect to the server with this ID. The server will send all the missed chunks of data to the reconnected client.
2. **event**: Clients can also subscribe to specific types of events. Other events are ignored by the clients.
3. **retry**: In case the connection is lost, the server specifies in advance the reconnection interval for the clients in milliseconds.
4. **data**: The actual data of the event.

Only the `data` part is required, all the other parts are optional. Each chunk is ended by a double new line character while each part is splitted by a single new line character.

In Qt, we simply parse each new line in the SSE frame string. Each part of the SSE frame is handled differently by the interface. For example, the `retry` property is only used internally. When the interface lost the connection with the server, it will use the last saved retry time to reconnect with the server.

In order to make the integration easier with HTTP polling, we extended the interface to support HTTP polling too. The difference between with the SSE implementation lies in the fact that the HTTP polling implementation uses an internal timer to poll the server.

3.4.2 Real time data caching using updates

In the previous chapter, we explained that caching real time data is almost impossible since the data is continuously changing in time. However, we should be able to cache these data if all the changes (updates) are immediately applied on our cache. In figure 3.4, we propose a way to achieve this goal. An example flow follows these steps:

- The client downloads the complete data set and caches it following the `Cache-control` header.
- The client subscribes to the update resource of the server.
- Every time the server has a new update available, it pushes the update to the client.
- The client process the update and updates its cache.

The client does not have to do any effort to keep its cache up to date. The server will not receive any HTTP conditional requests from the client thanks to the push approach. The real time data in the client's cache will still have slight delay in comparison to the server's version of the data.

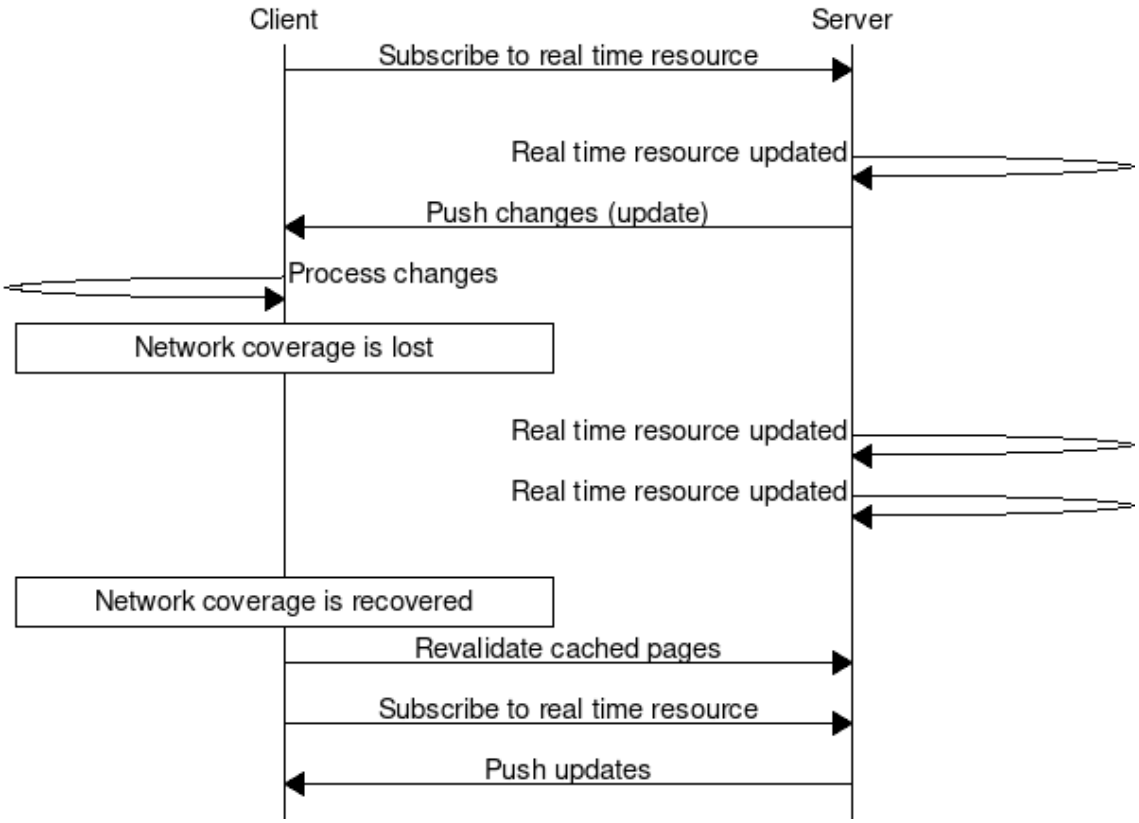


Figure 3.4: Caching real time data using updates, only the changes are pushed to the client to reduce the amount of bandwidth and other resources. If the connection is lost with the server, the client revalidates its pages and subscribe again to the real time resource.

This delay is a combination of:

- **Server's processing time**: The time between the data change and the push of the update to the client.
- **Transfer time**: The amount of time to transfer the data over the Internet.
- **Client's processing time**: The time between receiving the update and updating its cache.

This approach would reduce the amount of resources since not all the changed pages must be refetched from the server. Only the difference of all these pages in fetched from the real time resource. In case the connection with the server is lost, the client can revalidate its pages when the connection is re-established.

This way, both parties can reduce the amount of resources to keep the client's cache up to date since no polling takes place. This is very important if the client is a mobile device with a limited battery and bandwidth.

3.4.3 CSA snapshot rollback

As explained in section 2.4, the CSA profile variant depends on the previous state of the T and S array. In order to recalculate only the changed part of a route, we must be able to look up the previous state of both arrays. We achieve this by using *snapshots* of the arrays. Every time a page has been processed, a snapshot is made of these arrays.

A page consists of a list of connections and some meta data (snippet 3.7). The connections are presorted by departure time as required by the CSA. Using the meta data, we can identify the page and navigate to the previous or the next page.

A snapshot is not the same as a page. A snapshot is a *picture* of the T and S arrays at a certain point in time. The order of profiles and the profiles in each array is saved into a snapshot. The URI of the last processed page is used to identify each snapshot. We can use the URI to match each snapshot with a page. We should be able to rollback before the moment where the algorithm added the changed connection to both arrays. If we do this, we can reuse the T and S array since these connections are still valid. This way, only the connections after our update have to be recalculated.

```
{
        "@context": {...}
        "@id": "https://graph.irail.be/sncb/connections?
                departureTime=2019-04-24T18:40:00.000Z",
        "@type": "hydra:PartialCollectionView",
        "hydra:next": "https://graph.irail.be/sncb/connections?
                                departureTime=2019-04-24T19:01:00.000Z",
        "hydra:previous": "https://graph.irail.be/sncb/connections?
                                departureTime=2019-04-24T18:20:00.000Z",
        "hydra:search": {
                "@type": "hydra:IriTemplate",
                "hydra:template": "https://graph.irail.be/sncb/connections
                                        {?departureTime}",
                "hydra:variableRepresentation": "hydra:BasicRepresentation",
                "hydra:mapping": {
                        "@type": "IriTemplateMapping",
                        "hydra:variable": "departureTime",
                        "hydra:required": true,
                        "hydra:property": "lc:departureTimeQuery"
                }
        }
        "hydra:view": "http://lc.dylanvanassche.be/sncb/connections?
                        departureTime=2019-03-25T18:34:00.000Z",
        "@graph": [
        {
                "@id": "http://irail.be/connections/8894433/20190424/IC1841",
                "@type": "Connection",
                "departureStop": "http://irail.be/stations/NMBS/008894433",
                "arrivalStop": "http://irail.be/stations/NMBS/008894425",
                "departureTime": "2019-04-24T18:40:00.000Z",
                "arrivalTime": "2019-04-24T18:40:00.000Z",
                "gtfs:trip": "http://irail.be/vehicle/IC1841/20190424",
                "gtfs:route": "http://irail.be/vehicle/IC1841",
                "direction": "Ostende",
                "gtfs:pickupType": "gtfs:NotAvailable",
                "gtfs:dropOffType": "gtfs:NotAvailable"
        },
        ...
    ]
}
```

Snippet 3.7: Data structure of a Linked Connections page. The context contains all the necessary Linked Data vocabulary, the `@id` identifies the page and the `@type` gives more information about the data type. The `hydra` properties provide the navigation between the pages. The actual data is available in the `@graph` property.

We illustrate this using an example:

1. **Initial routing**: The client runs the CSA for the first time to retrieve all the possible routes for a certain time range.
2. **Update received**: The client received a changed connection from the server and processes it. It checks if the page of the connection is used in the processed routes. If so, a rollback is needed.
3. **Rolling back**: All the affected routes are rolled back to the state before the affected connection was being processed.
4. **Rerouting**: The rerouting process is started using the new connection. Since we do not have to start the routing process from scratch, we can update the routes much more efficiently.

This approach can be very effective if the affected connection is close to the departure stop of the journey. The reason for this is the fact that the CSA profile variant calculates the routes backwards in time. This is visualised in figure 3.5.

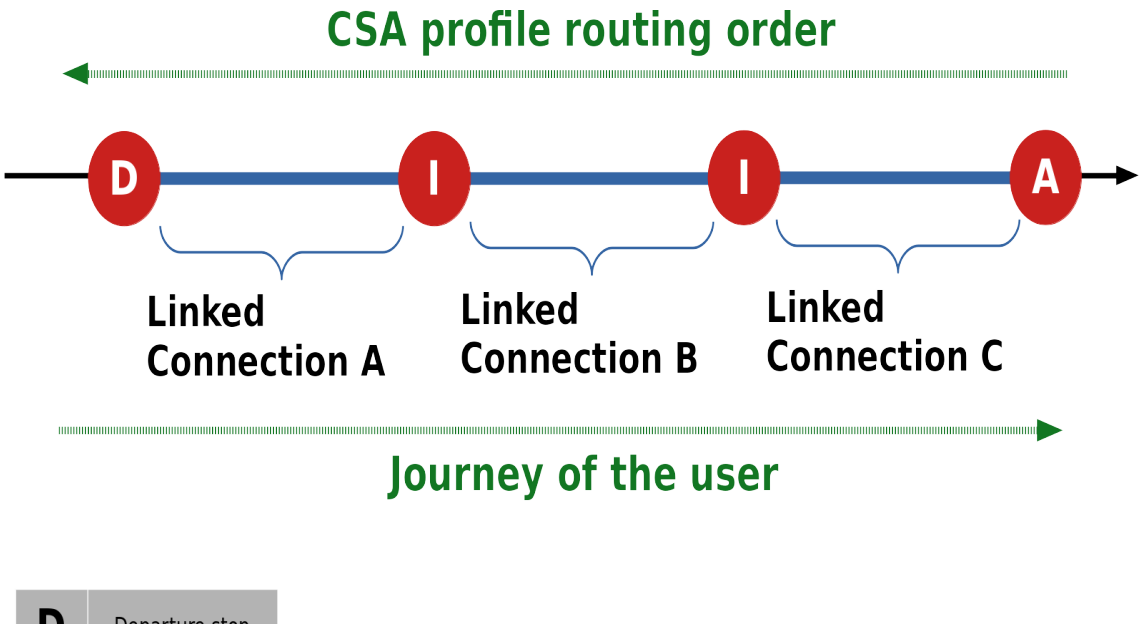


Figure 3.5: The CSA profile variant calculates the routes in the opposite direction of the user's journey.

When the affected connection contains the arrival stop, we need to rollback to almost the beginning state of the arrays. We should skip the rollback step completely because this operation is useless in this case.

In order to rollback any changes to a route, we must keep track of the steps that the CSA takes to construct a route. Several approaches can be used to keep a log of every step of the route construction, for example:

1. **Snapshot of each connection**: Every step generates a new version of the arrays. We save them completely using the Memento design pattern [48]. Rolling back to the right state of both arrays is relative easy when an update is received.
2. **Save every change of the arrays**: Only the changes to the arrays are saved. Since we only save the changes of each step, less memory is consumed. However, rolling back both arrays requires a more complex algorithm and resources.
3. **Snapshot of each page**: After a couple of steps, the arrays are saved. If we only save snapshots of the whole arrays, we need to recalculate more legs of each route, but less memory is consumed in the process.

With these approaches, we are able to rollback any changes to both arrays when an update is retrieved for a certain connection. Since we are running QRail on mobile devices too, we implemented the snapshot of each page approach. The other approaches are too complex or they use too much memory on these low power devices.

Using the Memento design pattern [48], we implemented this approach using journeys in the QRail library. Each page that has been processed by the CSA, gets a JourneySnapshot object. We save the *T_profile*, *S_profile*, *T_earliest_arrival*, *S_earliest_arrival* and the *page URI* we are processing at that time. The earliest arrival arrays are used to optimise the profile CSA, as explained in section 2.4. In snippet 3.8, we explain how such a restoration process works[14].

The rollback mechanism is triggered when a journey received an update. We register our journey with the update watcher. This update watcher will trigger a rollback when an update changes a connection in the journey. In case a route is changed, the watcher will notify us. There's no polling mechanism needed in QRail to check if the journey is up-to-date.

3.4.4 Liveboard regeneration

QRail can reroute more efficiently by reusing the previous results with a rollback algorithm (section 3.4.3). We applied almost the same idea to the liveboard generation. Each vehicle entry in the liveboard is watched by QRail. When an update is received, the update is parsed. In case the update contains a vehicle from the liveboard, the liveboard is updated by replacing the vehicle.

This operation is handled by the update watcher of the liveboard algorithm. We register the liveboard with the update watcher, the watcher will notify us as soon as the liveboard has been updated. We do not have to poll QRail each time to check if the liveboard is up-to-date.

[14] The lookup of the saved snapshots can be improved by using a binary search algorithm. We didn't implement this since this out of the scope of this thesis.

```
// Restoring is useless since we have to start from the beginning
if first saved snapshot == affected page:
        clear all data of the journey
        return

// Look through all snapshot and restore the snapshot before
// the snapshot of the affected page
for all saved snapshots:
        if saved snapshot == affected page:
                restore journey with all the data from
                the snapshot before the affected page
                return
```

Snippet 3.8: Pseudo code to restore a CSA journey snapshot. We always restore the snapshot which was taken just before the affected page. The CSA will automatically fetch the affected page again and start the rerouting process.

3.4.5 Page cache on the client

One of the benefits of caching real time data using updates is that we can use a local page cache on the client. The client does not have to fetch or revalidate each page when it wants to perform a routing operation for example[15]. The client can just use the local page cache which is kept in its RAM (Random Access Memory). Thanks to this approach, the lookup time of a page is heavily reduced.

This is – of course – only possible if we can keep this cache up-to-date. Fortunately, processing the real time updates can achieve this by using this page cache (section 3.4.6). If we need to cache a page, the page is saved in its RAM map with the URI of the page as key. If we need to retrieve a page, we can look it up by using the URI of the page. We do not have to go through the whole cache. In order to keep the cache persistent, each page is also saved as JSON-LD on the client's disk. If the library is restarted, the page cache checks if a requested page is in the RAM cache. If not, the cache will transparently access the disk to search for the page. If the page is not available from disk either, the page is retrieved from the Linked Connections Server.

3.4.6 Updates processing on the client

As mentioned in the previous section, processing real time updates requires us to use a modifiable page cache. Each time we receive a real time update, we must be able to lookup the page for the updated connection. Afterwards, we need to update this page in our cache. This last operation can create a conflict: the departure time of each connection must be greater than or equal to the one of the page and smaller than the next page. The URIs of the current, next and previous page can be retrieved from the meta data of the page (snippet 3.7).

[15] In case the client lost its network coverage, the client can revalidate its cached pages with the server (as mentioned in section 3.3.4).

If the connection has an updated departure delay, we must move the connection to the right page. If the arrival delay is updated or the connection is cancelled, we only have to modify the connection.

If we ignore this conflict, the CSA will produce incorrect results because the algorithm expects that each page is sorted by departure time. In case nothing has been changed, we skip the connection (figure 3.6).

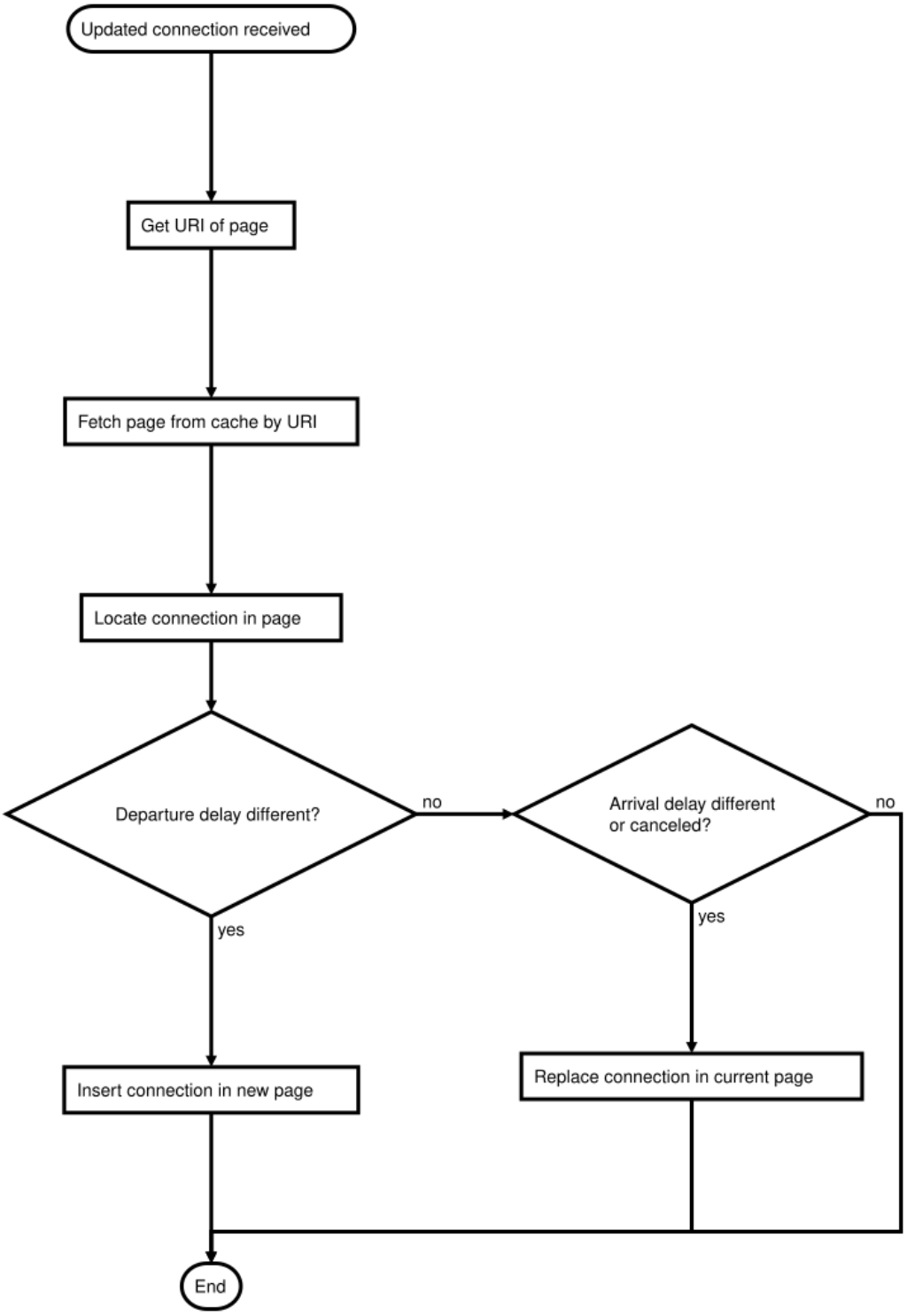


Figure 3.6: Page cache update algorithm. Pages with a different departure delay can move to another page. Otherwise, the connections are not sorted anymore by departure time. In all other cases, the connection can just be replaced in its current page.

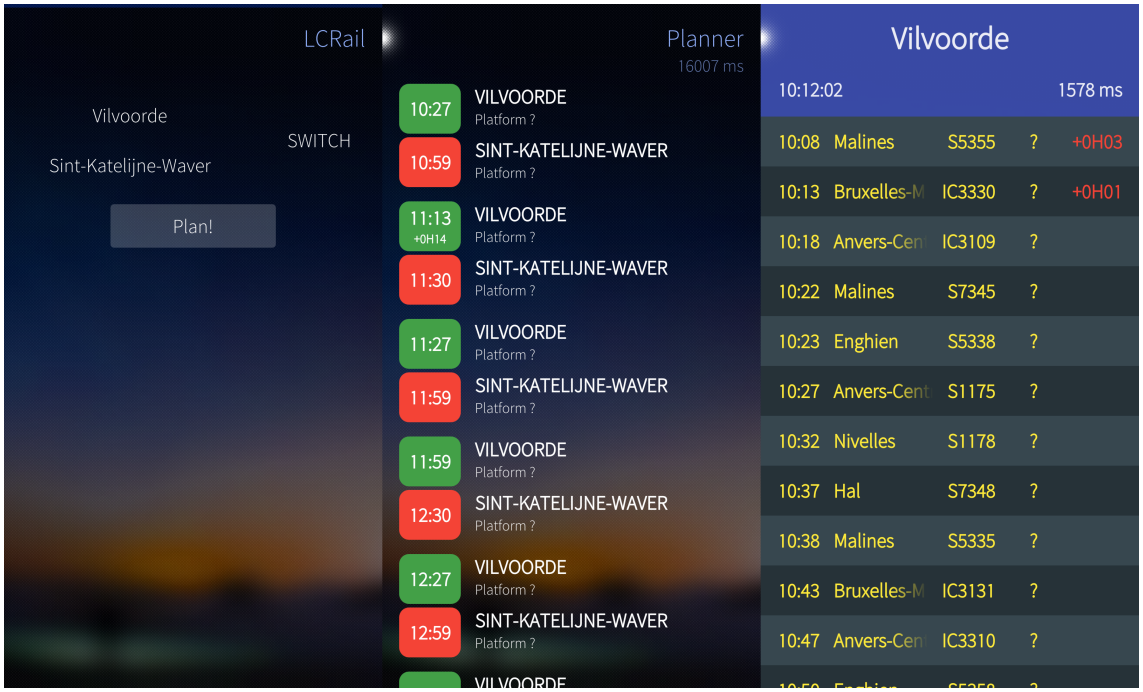


Figure 3.7: Screenshots of the LCRail application on a Sailfish OS smartphone. On the left, we see the selection of the stops for the route planner. In the middle, the results of the route planner and on the right the liveboard visualisation.

3.5 LCRail client

The visualisation of QRail is done by our own written LCRail client[16]. The LCRail client is also a Qt application that uses QRail as a library. We tested the stability of QRail in the real world by using this client on a Sailfish OS smartphone (figure 3.7).

LCRail's architecture is visualised in figure 3.8. We describe the architecture in detail in the next paragraphs. It makes use of QML (Qt Modeling Language) for the UI and C++ to interface with QRail. As QML library, we used the Sailfish OS's Silica components. These components provide a native experience on Sailfish OS devices. However, they do not provide a visualisation for a journey or a liveboard. We wrote those views with standard QML components as `ListItem` and `Label`. We provide an example of such a QML component in snippet 3.9.

[16] https://www.github.com/DylanVanAssche/LCRail

```
import QtQuick 2.2
import Sailfish.Silica 1.0

ListItem {
    property string searchString

    id: stationItem

    // Animations on add and remove
    ListView.onAdd: AddAnimation {
        target: stationItem
    }

    ListView.onRemove: RemoveAnimation {
        target: stationItem
    }

    // Stop name
    Label {
        anchors {
            left: parent.left
            right: parent.right
            leftMargin: Theme.horizontalPageMargin
            rightMargin: Theme.horizontalPageMargin
            verticalCenter: parent.verticalCenter
        }

        // Highlight matching characters during search
        color: searchString.length > 0 ?
          (highlighted ? Theme.secondaryHighlightColor : Theme.secondaryColor)
          : (highlighted ? Theme.highlightColor : Theme.primaryColor)

        // Follow Sailfish OS UI design rules
        truncationMode: TruncationMode.Fade
        textFormat: Text.StyledText
        text: Theme.highlightText(model.name,
                                  searchString,
                                  Theme.highlightColor)
    }
}
```

Snippet 3.9: QML component for displaying the stop entry in the stop search of LCRail.

```
int main(int argc, char *argv[])
{
    // Initiliase the QRail library
    initQRail();

    // Expose the QRail C++ models to QML
    qmlRegisterUncreatableType
    <QRail::LiveboardEngine::Board>("LCRail.Models.Liveboard.Board",
                                    1, 0, "Board", "read only");
    qmlRegisterUncreatableType
    <QRail::StationEngine::Station>("LCRail.Models.Station",
                                    1, 0, "Station", "read only");
    qmlRegisterUncreatableType
    <QRail::RouterEngine::Route>("LCRail.Models.Route",
                                 1, 0, "Route", "read only");
    qmlRegisterType<Liveboard>("LCRail.Views.Liveboard",
                               1, 0, "Liveboard");
    qmlRegisterType<Router>("LCRail.Views.Router",
                            1, 0, "Router");
    qmlRegisterType<Stations>("LCRail.Views.Stations",
                              1, 0, "StationsSearch");

    /*
     * Finish initilisation with the SailfishApp version of
     * QApplication and QML engine
     */
    return SailfishApp::main(argc, argv);
}
```

Snippet 3.10: Linking QRail with LCRail's QML UI. We expose the QRail's data types to the QML engine.

LCRail uses Qt signals to stream the results of the QRail library into the UI. Qt will expose all the C++ signals and methods automatically to the QML UI when the C++ class is registered in the *main* routine. Qt also handles the conversion between C++ and QML data types (in both ways). For example: a QML `Date` object is automatically translated into a `QDateTime` object in C++. However, if we want to expose a `QList` from C++ to QML, we need to make use of the `QAbstractListModel` class. This class allows us to create interactive lists in QML with data from C++. We used this abstract class for exposing the liveboard's entries, the routes of a journey and the stop search. The `QAbstractListModel` class provides several methods for adding, removing and updating items in a list.

We started by linking the QRail library to our visualisation. By linking the library, we can use all the exposed methods of the CSA and liveboard algorithm directly in LCRail. This is done in the `main` function (snippet 3.10). We can now call each method depending on the need of the user.

If the user wants to plan a journey, the CSA `getConnections` method is called. The method will use the given stops and the current time as parameters for the CSA. The CSA in QRail will generate all the possible routes for the journey. The data is available through the Qt signal and slots system. LCRail can connect to the signals of the QRail library. As soon as the CSA has a new result, a signal is emitted from within the QRail library. LCRail will automatically get the data through the Qt's signal and slots system. The technology behind this is the event loop of Qt. With an event loop, we can program our application in an asynchronous way. No polling is required to get the data from the QRail library into LCRail. If QRail is waiting on the Linked Connection server for example to get a page, QRail parses the previous page for example. When a event on the event loop has been completed, Qt will start to handle the next event. Since the UI is also event driven, the UI stays responsive, even under heavy load. Thanks to an event driven approach, embedded devices can even run Qt[17].

In case the user wants to retrieve the liveboard of a stop, the `getLiveboard` method will be called. QRail will start to fetch the necessary Linked Connections pages to complete the request of the user. Each page is filtered by the stop, provided by the user. Every time a valid connection is found, a signal is emitted from within QRail. LCRail is connected to those Qt signals. LCRail can immediately retrieve the new data, as mentioned in the previous paragraph.

In order to show how powerful Linked Connections can be, we added a *previous* and *next* buttons to the UI. With those buttons, the user can extend the liveboard. Instead of regenerating the liveboard, QRail will add an additional connection to the liveboard. QRail keeps searching in the pages until at least one valid connection is found to extend the liveboard. Instantly, the user will see an additional connection in the liveboard.

LCRail has also access to the QRail's stop search. QRail exposes the internal stop database to LCRail. When the user wants to select a certain stop, only a few characters are needed to show stop suggestions. Since iRail provides the stop names in 4 languages[18], the user can search in the 4 languages at the same time.

[17] https://www.qt.io/qt-for-device-creation/

[18] Each stop of the NMBS has a name in Dutch, French, German and English.

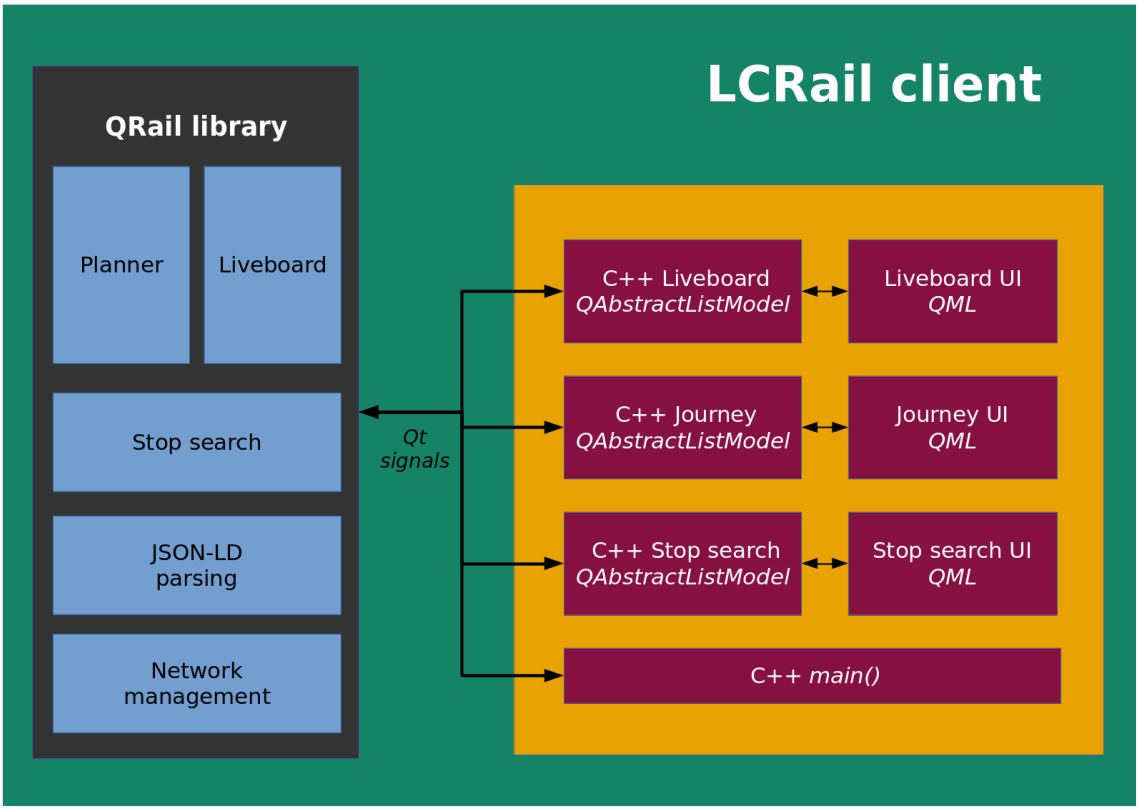


Figure 3.8: The architecture of the LCRail client. The main function initialise QRail and the QML engine. The QAbstractListModel class allows us to create interactive QML lists with C++ data. Qt signals are used to communicate between each part of the client.

3.5.1 Features

- **Stop list**: The list of stops is retrieved from the QRail library and can be searched in 4 different languages (English, Dutch, French and German).
- **Liveboard**: A liveboard can be generated for a given stop. All the departing vehicles for the stop are streamed to the UI.
- **Route planner**: The routes of a journey are streamed to the UI as soon as the CSA finds a possible route for a given journey.
- **User informed time**: The user can see in the UI how long it took to complete his or her request.
- **Cache**: All the Linked Connections data is heavily cached to improve the user experience (section 3.4.5).

3.5.2 Automatic UI updates

Most applications today are working with a refresh button in order to update their view. If you want to plan your journey with the official NMBS application, you have to go through the whole process each time you want to update your journey view. This is not the case anymore for LCRail. We adapted our streaming algorithm in LCRail to show updates directly into the journey view (figure 3.9).

Our streaming algorithm operates as follows: When a user starts to plan its journey, LCRail will register this journey with the QRail update watcher. As discussed above, QRail will check if the journey has been affected by an update. If that is the case, LCRail will receive this information and show it directly in the UI. This feature is also implemented for the liveboard view. In order to improve the experience, the user will receive a notification if one of the views are updated.

3.6 Conclusion

In this chapter, we gave an overview of the modifications we did to the Linked Connections Server. We had to add support for the real time resource to the Linked Connections Server. We also modified the `gtfsrt2lc` library to handle cancelled vehicles and parse the delay information correctly.

Afterwards, we discussed our own written QRail library and the LCRail client. The QRail library was written in a modular way. This approach allowed us to modify it later to use the real time information in a more efficient way.

With the LCRail client, we visualised our results from the QRail library in a Qt QML UI on a Sailfish OS device.

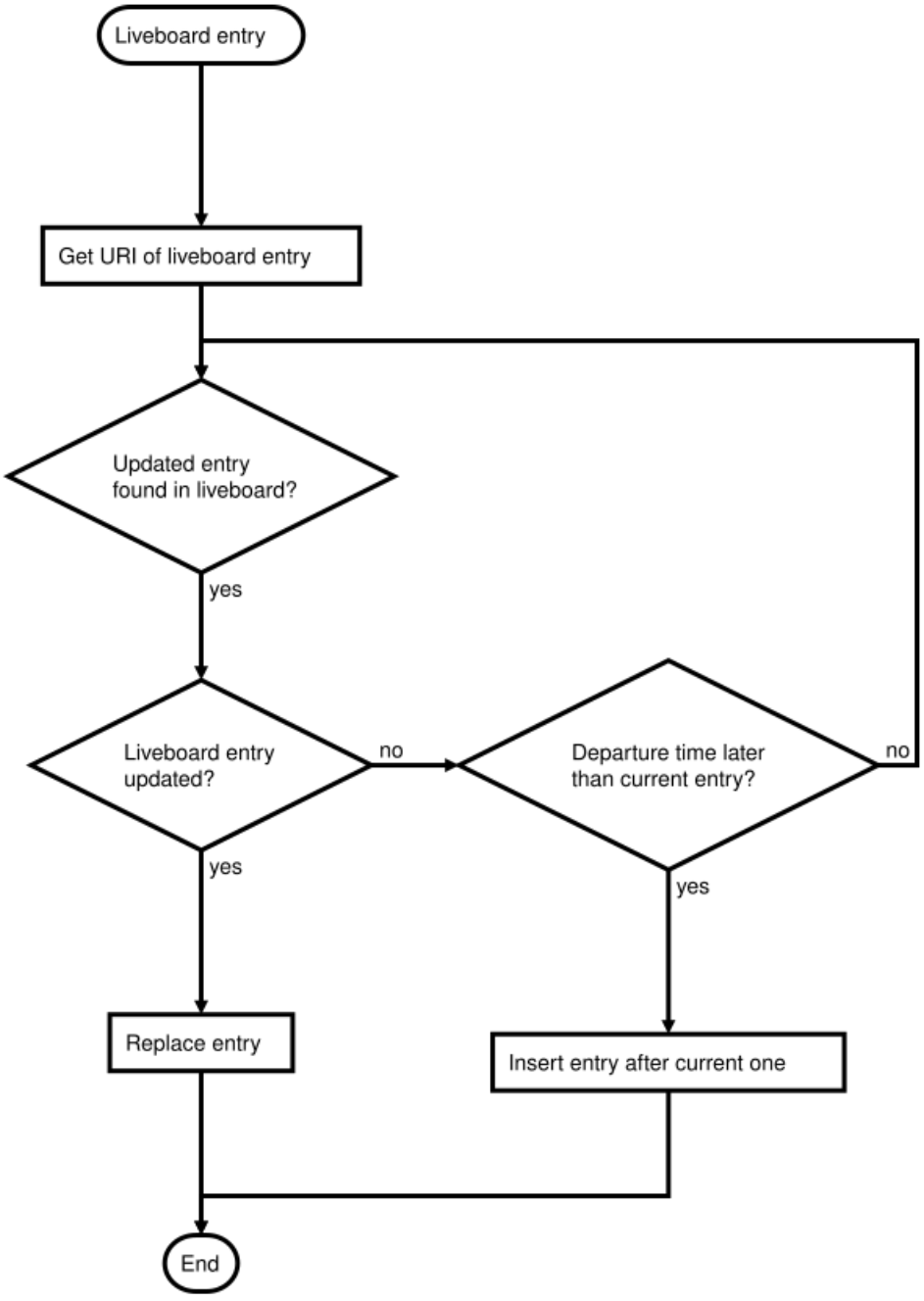


Figure 3.9: Stream processing of the liveboard data in LCRail.

4 Results

In this chapter, we first describe our test environment and how we created our benchmarks. Afterwards, we discuss the results of these benchmarks.

4.1 Test environment

The test environment is divided in 2 different parts. The first part is used to benchmark the workload on the server with a pushing approach and a polling approach. In the second part, we benchmark the client to see how fast an update is shown to the user.

In order to create a reproducible environment, we created a file server (Linked Connections Reproduction)[1]. We can not use the GTFS and GTFS-RT stream directly since we have to compare each approach in a reproducible way. The file server provides a repeatable GTFS and GTFS-RT feed to the Linked Connections Server during our tests.

With a Python script, we download several hours of GTFS and GTFS-RT data from the NMBS. We apply a round-robin scheduling to determine which GTFS-RT file is served to the Linked Connections Server. When we run out of GTFS-RT files, we restart from the beginning. This way, we can reproduce the real time events during our tests.

[1] https://github.com/DylanVanAssche/Linked-Connections-Reproduction

Table 4.1 An overview of the hardware for the pushing or polling experiment.

Device	Specifications
Virtual Wall I	• Ubuntu 18.04 LTS • 2x Intel E5520 • 2x Quad-core 4x 2.2 GHz • 12 GB RAM • 160 GB hard disk • 2x-4x Gigabit Ethernet
Digital Ocean droplet	• Ubuntu 18.04 LTS with NGINX • Intel Xeon Gold 6140 • Single-core 1x 2.30 GHz (virtual) • 2 GB RAM • 50 GB SSD • Gigabit Ethernet

4.1.1 Pushing or polling

The impact of our real time resource on the server's workload must be as low as possible. As mentioned in chapter 2.2.2, multiple approaches exist to achieve this goal. The main difference between these approaches is the method of delivery. Some approaches require the client to poll the server, others let the server push the information to the clients.

The comparison of pushing and polling is executed on the Virtual Wall of IMEC (Interuniversitair Micro-Elektronica Centrum). The Virtual Wall is a server cluster which can be configured to simulate a part of the Internet. Using the Virtual Wall, we are able to test our own written software in real production-like environments[2].

[2] https://doc.ilabt.imec.be/ilabt/virtualwall/index.html

Figure 4.1: The Virtual Wall at IDLab Ghent. The Virtual Wall I has 206 nodes and the Virtual Wall II has 161 nodes at their disposal. Each node can be configured over SSH with the jFed tool.

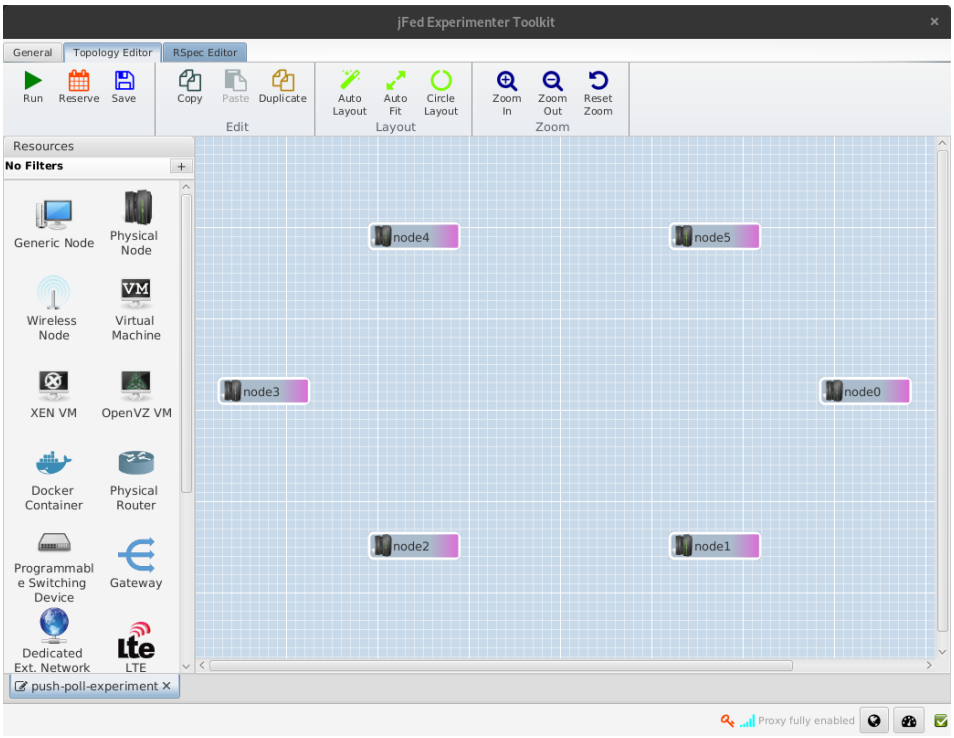


Figure 4.2: Our experiment in the Experimenter Toolkit UI of the jFed tool. We allocated 6 nodes with each 250 clients on the Virtual Wall.

We run our tests on the Virtual Wall I and a Digital Ocean droplet (table 4.1). This experiment is in collaboration with Brecht Van de Vyvere from IDLab. The collaboration was established because of our common research goals about SSE and HTTP polling. The NodeJS scripts for the server and the clients can be found in their Github repository[3].The server consist of a NodeJS script and a reverse proxy (NGINX). The NodeJS script only handles the data management. The NGINX reverse proxy exposes the NodeJS server to the privileged HTTP port (80) on the Digital Ocean droplet. We use NGINX to run the NodeJS script without root privileges on the server. Another benefit of NGINX is the resource usage. NGINX uses less resources for interaction with the client, performing HTTPS encryption and compression.

The client has 2 different operation modes. Depending on the environment variable `MODE`, we can select a `polling` or `pushing` mode:

1. **Polling**: The client will poll the server with a given polling interval.
2. **Pushing**: The client connects to the server. The server will push the data to the client.

Deploying on the Virtual Wall can be done through the *jFed tool* (figure 4.2). jFed is a testbed federation tool, developed by IMEC and the university of Ghent. We will use the Experimenter Toolkit UI to allocate the nodes on the Virtual Wall. In this benchmark, 6 nodes are used with 250 clients each. The Digital Ocean droplet receives requests from 1500 clients during this benchmark. The resource usage of the server is benchmarked using the Linux tool: `top`. The data is analyzed using a Python script.

Using this benchmark, we can see how much additional resources on the server are consumed when the number of clients increase. Because we also benchmark the clients, we can see which approach is consuming the least amount of resources for both sides.

4.1.2 Processing of updates

When it comes to real time information, it is important to deliver the data as fast as possible to the receiver. In our case, we would like to inform the user as fast as possible. This can be translated into the time between the reception of an update and when the update is shown to the user. During this experiment, we will compare the original implementation and our modified versions in terms of CPU, RAM, network usage and how fast an update is shown to the user (refresh time). The experiment is performed on our own LCRail client (section 3.5).

This experiment is performed on 2 different mobile devices as client and a Linked Connection server running on a Digital Ocean droplet (table 4.2).

[3] https://github.com/DylanVanAssche/realtime-open-data-benchmark

Table 4.2 An overview of the hardware for the processing of updates experiment.

Device	Specifications
Sony Xperia X	• Jolla's Sailfish OS 3.0.3.8 (Linux, Qt 5.6.3 LTS, GCC 4.9) • Qualcomm MSM8956 Snapdragon 650 (ARM) • Hexa-core 4x 1,4 GHz (Cortex-A53) and 2x 1,8 GHz (Cortex-A72) • 3 GB RAM • 32 GB MMC • 2G/3G/4G • Wi-Fi 802.11 a/b/g/n/ac, dual-band
Jolla 1	• Jolla's Sailfish OS 3.0.3.8 (Linux, Qt 5.6.3 LTS, GCC 4.9) • Qualcomm MSM8930 Snapdragon 400 (ARM) • Dual-core 2x 1,4 GHz (Krait 300) • 1 GB RAM • 16 GB MMC • 2G/3G/4G • Wi-Fi 802.11 b/g/n
Digital Ocean droplet	• Ubuntu 18.04 LTS with NGINX • Intel Xeon Gold 6140 • Single-core 1x 2.30 GHz (virtual) • 2 GB RAM • 50 GB SSD • Gigabit Ethernet

It is important to perform these tests on mobile devices due the fact that commuters have these kind of devices with them all the time. This way, we can reflect better the reality in our benchmark. In order to benchmark this in a reproducible way, we created a benchmark script. This script uses the following Linux tools: `top` and `nethogs` to log the CPU, RAM and network usage of our application. Additionally, we also log the processing time of an update. All of this data is parsed by a Python script and converted to graphs using the Matplotlib module[4].

Since real time data is not reproducible in the future, we also created a NodeJS server which serves a captured amount of data to the Linked Connections Server. The data was captured using a Python script. This script polls the SNCB GTFS-RT resource every 30 seconds[5] and saves the GTFS-RT data to the disk. We can assure now that we are able to reproduce the same results since the Linked Connections Server receives the same real time data over and over again.

4.2 Test results

In section 4.1, we explained how we designed our benchmarks for our modifications to LCRail and QRail. In this section, we show the results of these benchmarks. First of all, we discuss the “Pushing or polling” experiment. Afterwards, we look at the “Processing of updates” experiment.

4.2.1 Pushing or polling

During this experiment we benchmarked the server twice. In each benchmark we used 1500 clients and a single server. We ran each benchmark for approximately 30 minutes. At the start of each benchmark, the CPU usage is almost 0 % since the nodes are still booting. They become fully operational after approximately 200 seconds. To avoid the influence of the booting process, we will only discuss the results which are measured after 250 seconds.

The first benchmark used HTTP polling to retrieve the events from the server. Each client polled the server every 30 seconds. The server has each 30 seconds new data available for the clients.

In the second benchmark, we used SSE pushing. The clients connect to the server and the server pushes the data the clients. Each 30 seconds, the server has new data available and pushes it to the clients.

We measured the CPU and RAM usage of the server during the benchmark. The used network bandwidth is measured in the next experiment (section 4.2.2).

[4] https://matplotlib.org/

[5] The SNCB only publishes a new GTFS-RT file every 30 seconds.

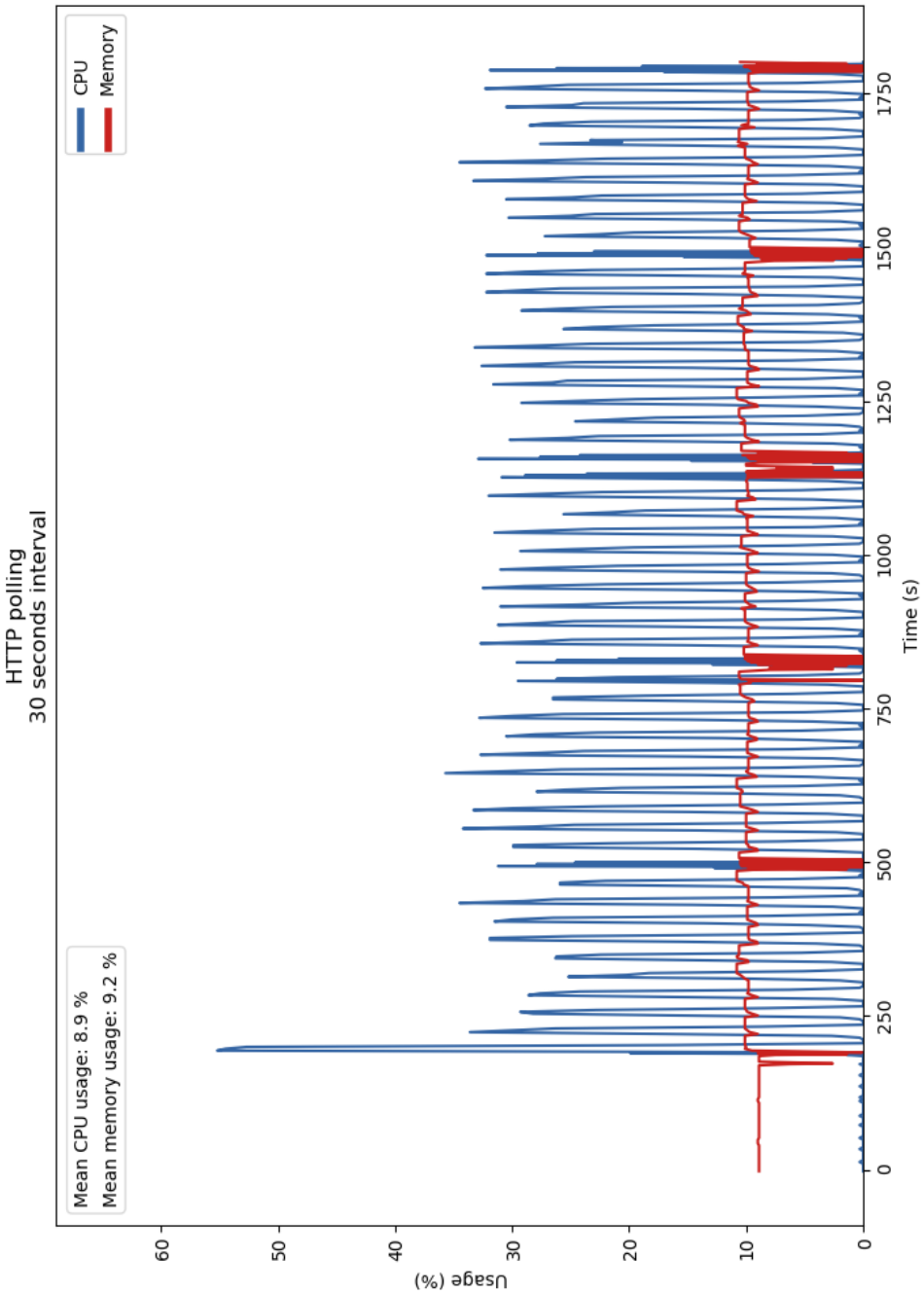


Figure 4.3: The CPU and memory usage of the server when 1500 clients are actively polling the server every 30 seconds (see spikes). The data on the server is refreshed every 30 seconds. The benchmark ran for 30 minutes.

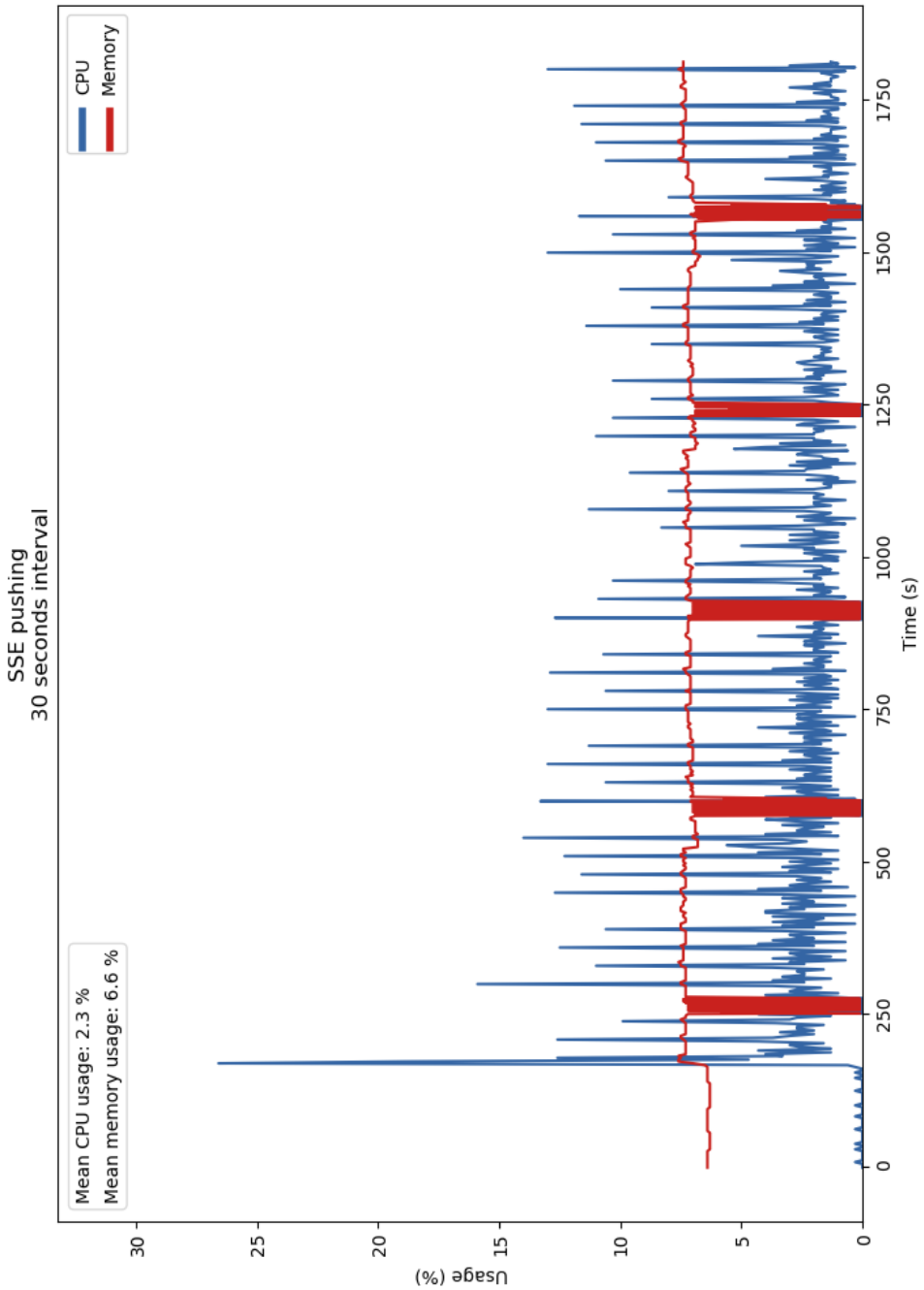


Figure 4.4: The CPU and memory usage of the server when 1500 clients are connected using SSE. The server pushes the new data every 30 seconds to the 1500 clients (spikes). The benchmark ran for 30 minutes.

4.2.2 Processing of updates

Our first benchmark was executed on the original version of LCRail. This version does not use any real time resource to alert the user if a journey or liveboard was changed. Instead, it uses a QML timer to trigger a refresh of the views every 30 seconds. This is the same interval as the one used in the real time resource.

Our second and third benchmarks were executed on the modified versions of LCRail. In the second benchmark, we used HTTP polling to retrieve the data from the real time resource. In the third benchmark, SSE is used instead of HTTP polling. In both modified versions of LCRail, we used a RAM cache for the Linked Connections pages. This RAM cache is kept up-to-date with an update algorithm as explained in chapter 3.

The first peak at the beginning of each CPU usage graph, is related to the loading of the stations database. We will not take this into account when discussing the results, since it is out of the benchmark's scope. Each benchmark was started with a clean cache, otherwise we can not compare the benchmarks with each other in terms of network usage and refresh time. In every benchmark, we used the same data to make the benchmark reproducible.

4.2.2.1 Liveboard benchmark

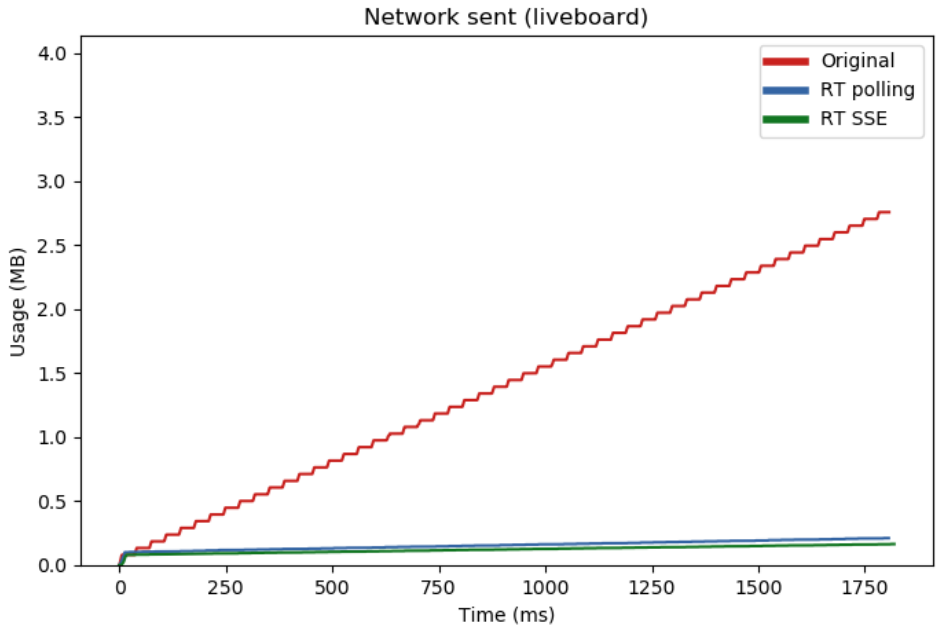


Figure 4.5: Amount of sent data of the liveboard for the 3 approaches on a single device in time. The benchmark ran for 30 minutes.

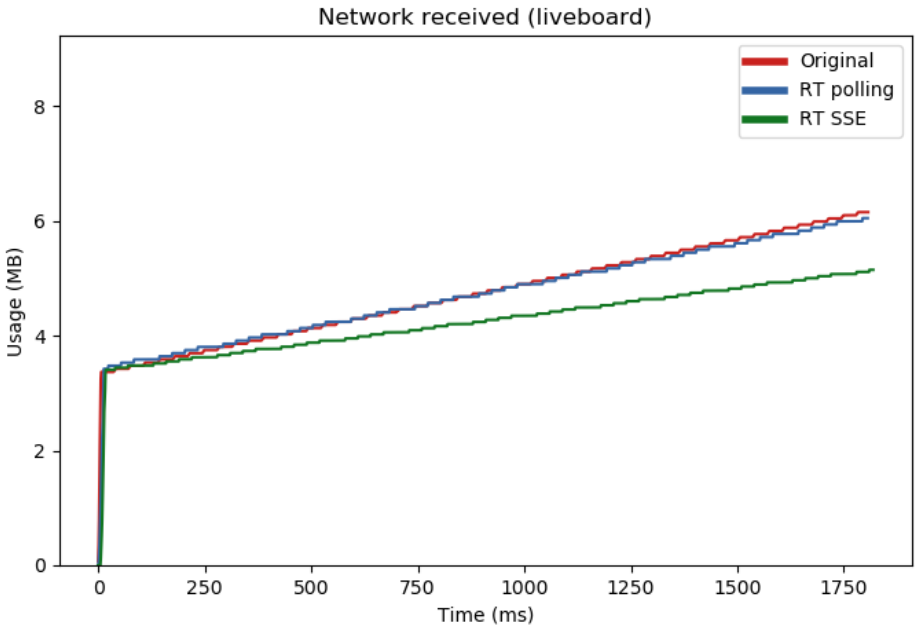


Figure 4.6: Amount of received data of the liveboard for the 3 approaches on a single device in time. The benchmark ran for 30 minutes.

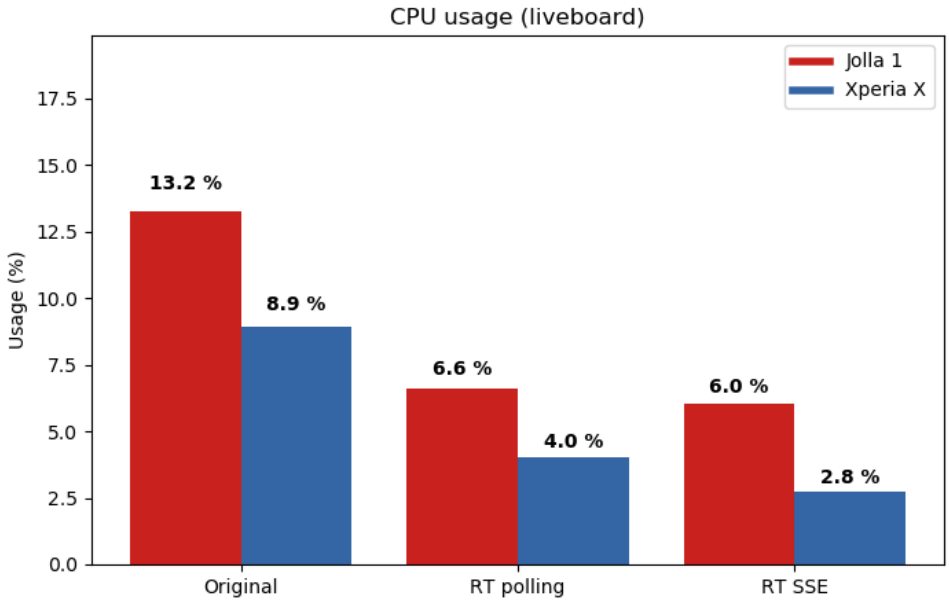


Figure 4.7: The mean CPU usage of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

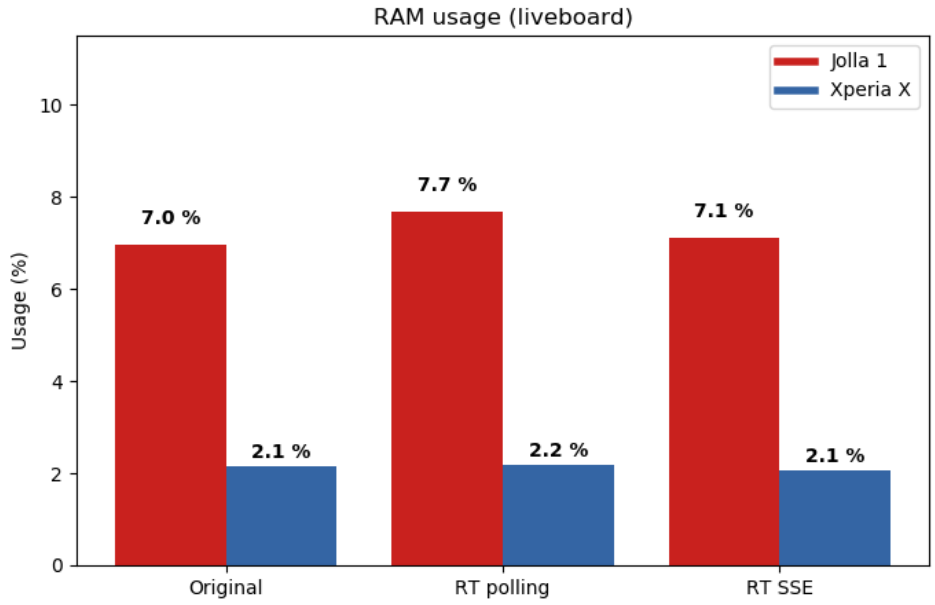


Figure 4.8: The mean RAM usage of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

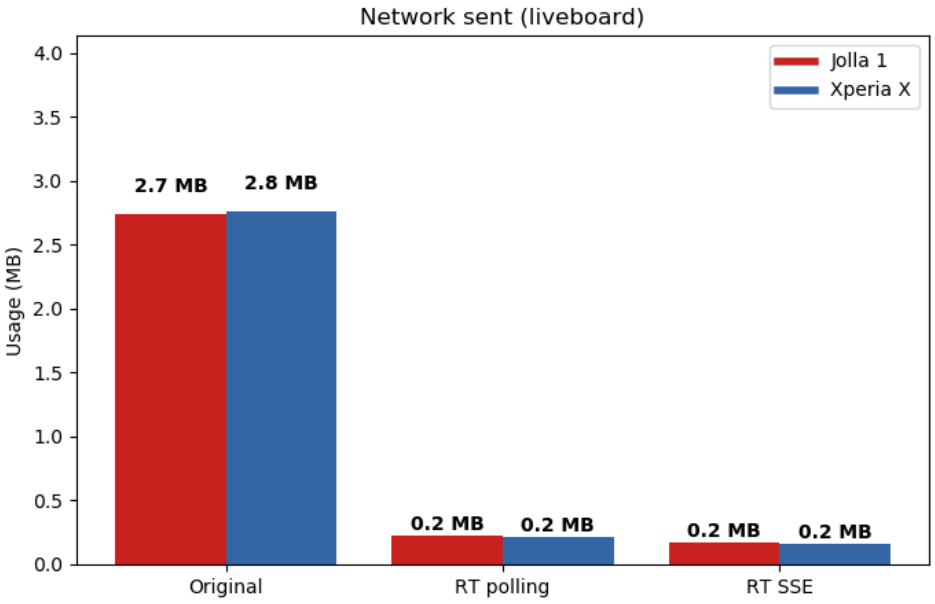


Figure 4.9: The sum of the amount of sent data of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

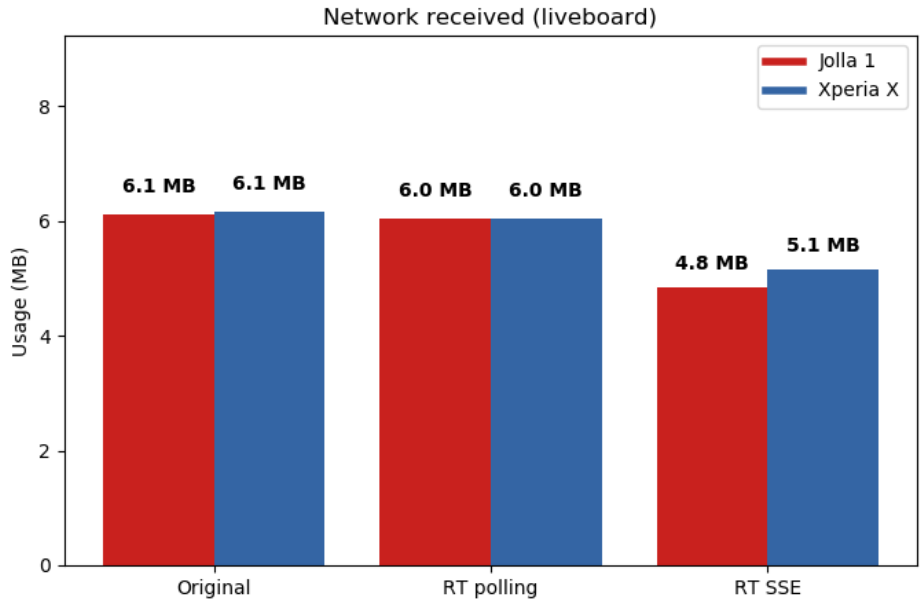


Figure 4.10: The sum of the amount of received data of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

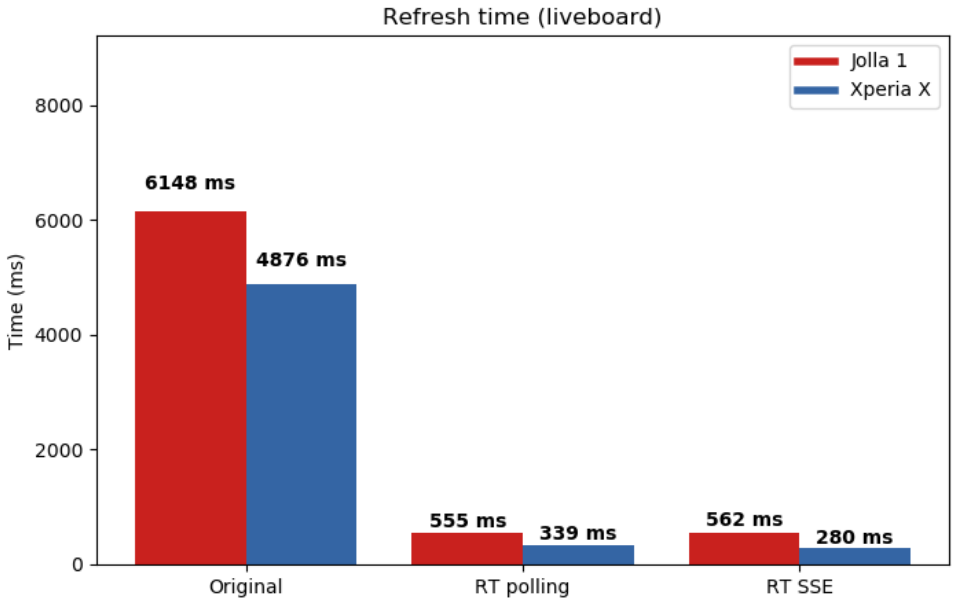


Figure 4.11: The mean refresh time of the liveboard for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

4.2.2.2 CSA benchmark

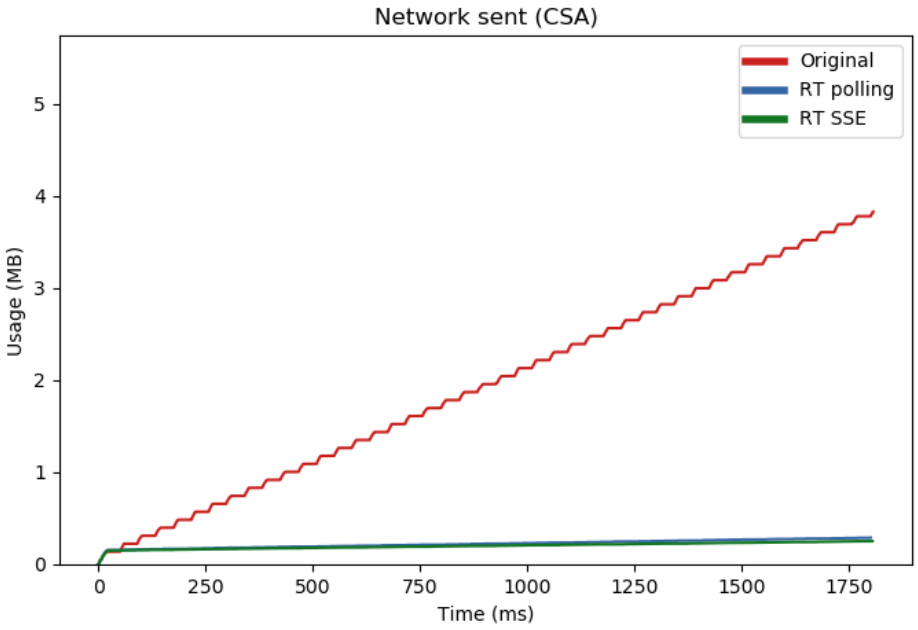


Figure 4.12: Amount of sent data of the CSA for the 3 approaches on a single device. The benchmark ran for 30 minutes.

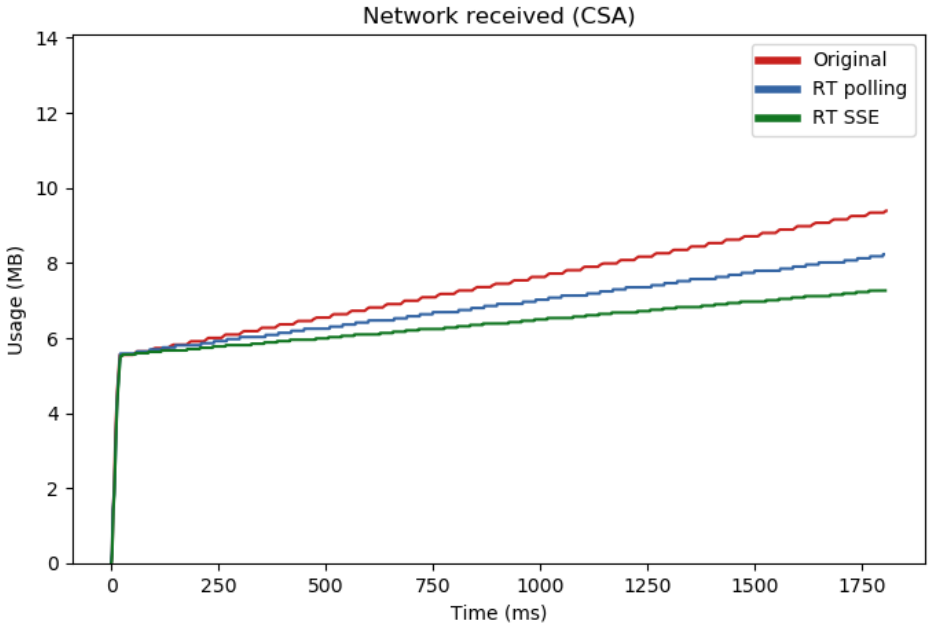


Figure 4.13: Amount of received data of the CSA for the 3 approaches on a single device. The benchmark ran for 30 minutes.

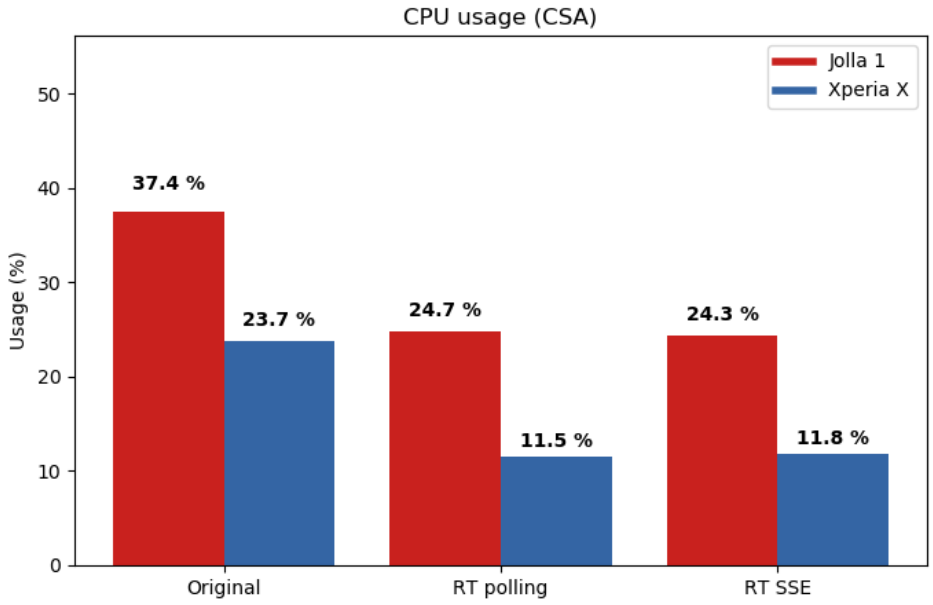


Figure 4.14: The mean CPU usage of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

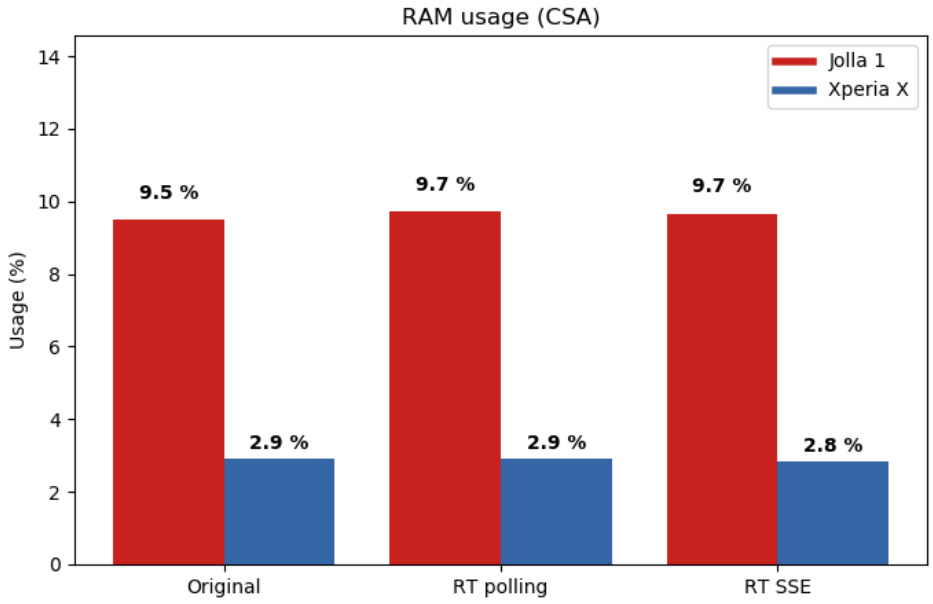


Figure 4.15: The mean RAM usage of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

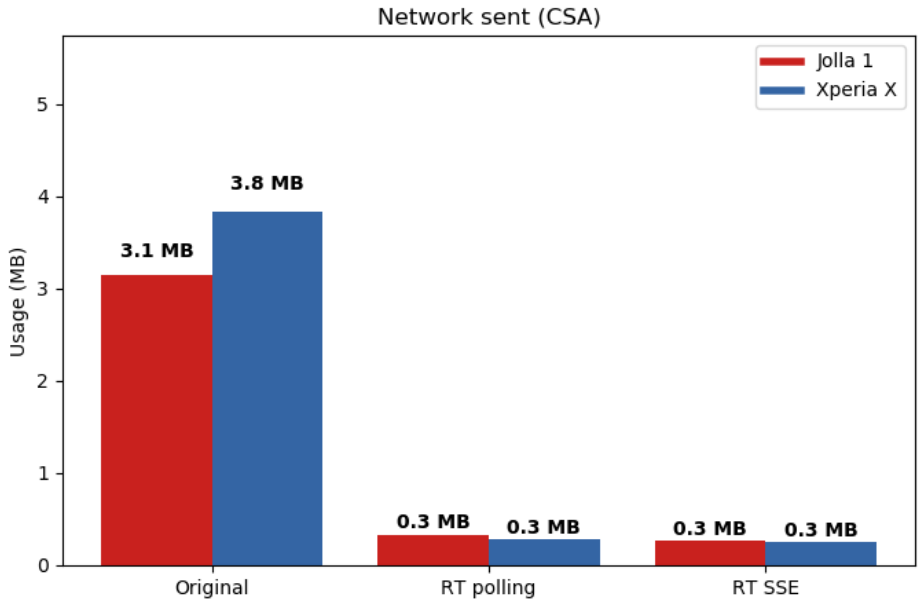


Figure 4.16: The sum of the amount of sent data of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

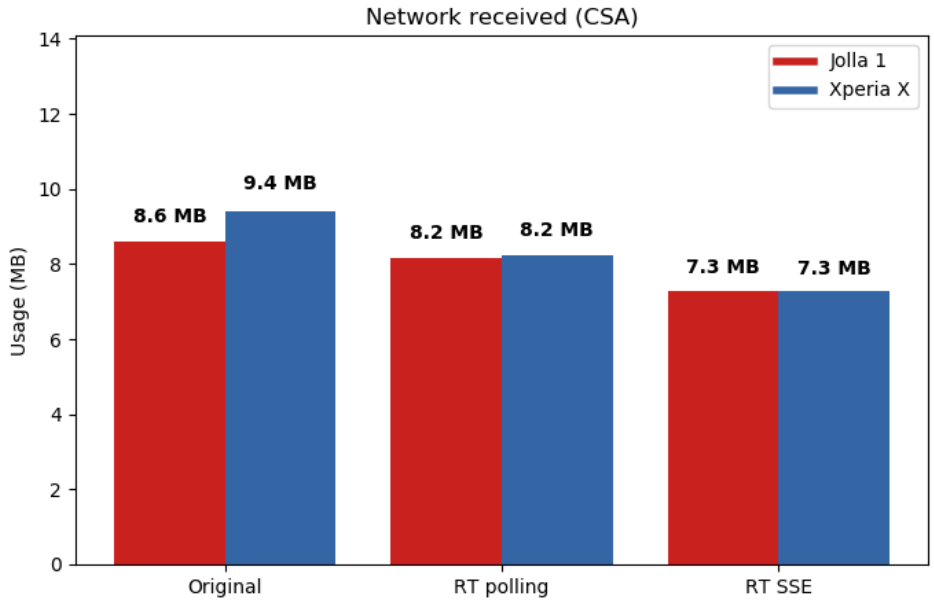


Figure 4.17: The sum of the amount of received data of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

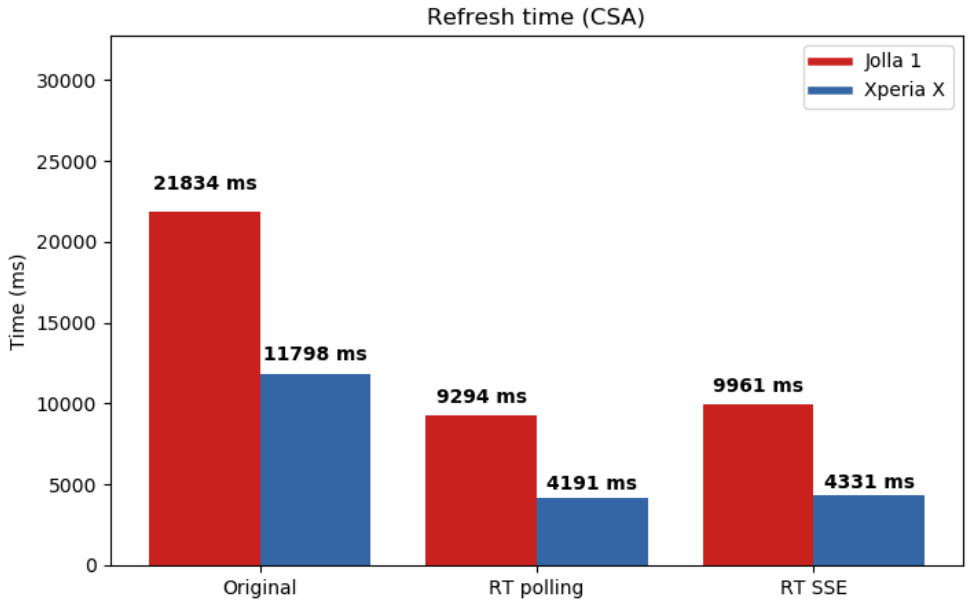


Figure 4.18: The mean refresh time of the CSA for the 3 approaches on 2 different devices. The benchmark ran for 30 minutes.

4.3 Discussion

4.3.1 Pushing or polling

The CPU and RAM usage of the server is important for the Open Data publisher. With real time Open Data, the amount of resources can increase significantly for the publisher. With the common HTTP polling approach, the server needs more resources to transfer the same data to its client, in comparison with the SSE pushing approach.

The server needs to setup a TCP/IP connection, read the client's request, look up the data and send it back to the client. In our case, we ran our test with 1500 clients. This means that the server has to do all these operations 1500 times. However, all these clients will perform the same operations again after a certain polling interval (30 seconds in our case). We can clearly see this behaviour in the CPU usage of the server (figure 4.3). Each time the server has to answer 1500 polling clients, the CPU usage spikes at 35 %. The RAM usage does not spike when performing these operations.

If we look at the SSE pushing approach, we still have spikes when the data is pushed to the 1500 clients. However, the height of the spikes are much lower than with HTTP polling (15 %). The amount of spikes is also lower than with HTTP polling. This behaviour can be explained by the fact that the server only have to send the data to the clients. The data is looked up once and transferred to all the connected clients. The TCP/IP connection setup and reading the client's request are only performed when the client connects to the server. Afterwards, the connection is kept open by the server. This result in the continuous CPU usage between the spikes. Fortunately, keeping a connection open uses less resources than setting up a new connection.

To compare both approaches even better, we calculated the mean CPU and RAM usage for both approaches. The mean CPU usage for SSE (2,3 %) is almost 4 times lower than with HTTP polling (8,9 %). The mean RAM usage is 28 % lower with SSE in comparison to HTTP polling.

4.3.2 Processing of updates

4.3.2.1 Liveboard benchmark

If we look at the results of our liveboard benchmark (section 4.2.2.1), we can clearly see that the hardware is an important factor. When faster hardware – like the Xperia X – is used, the refresh time of the liveboard is lowered by almost 2 seconds. Figure 4.7 shows clearly that the Xperia X can faster update the liveboard than the Jolla 1 in all implementations.

In the beginning of the benchmark, they both cache each Linked Connections page. When a refresh action is triggered, the HTTP cache is used. This is visible in the network usage graph. Only changed Linked Connections pages are completely fetched from the server. All other pages returned immediately a `HTTP 304 Not Modified` status code. This is not the case with the RAM cache, the update algorithm keeps the RAM cache up-to-date in the background. The liveboard does not have to perform any additional network queries to retrieve the latest version of a page. Since the network usage is almost the same for both devices, we illustrate this with a single device in figures 4.5 and 4.6.

If we compare the results of second and third benchmarks with the original implementation (figures 4.7, 4.8, 4.9, 4.10 and 4.11), we can see several improvements:

1. **CPU usage**: Since we only process the updates to the Liveboard, the CPU usage is heavily reduced. For the HTTP polling implementation, the Jolla 1 uses 50 % less CPU power than the original implementation. The Xperia X consumes 55 % less CPU power when using polling. However, the SSE implementation saves even more CPU usage. The Jolla 1 was able to save 54 % of its CPU power and the Xperia X was able to save 68 % of its CPU power. Besides a faster response to an update, a commuter should also gain a better battery life for his or her mobile devices.

2. **RAM usage**: The amount of RAM usage is almost the same in the 3 benchmarks for both devices.

3. **Network usage**: The amount of network bandwidth has been reduced. The amount of sent data is reduced by 92 % while the amount of received data has been reduced by 21 % for the SSE implementation and by 2 % for the HTTP polling implementation. The difference between both implementation is that the SSE implementation will not push any data when no events are available. The HTTP polling implementation will still receive an empty array[6] of events.

4. **Refresh time**: The refresh time for both devices is reduced by a factor of 10 in comparison with standard HTTP caching. Originally, it took about 10 seconds to refresh a liveboard. Our modifications reduced this to less than 1 second. Thanks to the RAM cache, the algorithms do not have to parse the JSON-LD each time. The Linked Connections pages are directly available as objects through the RAM cache. A commuter will be informed much faster about a change in a liveboard than with the original implementation.

4.3.2.2 CSA benchmark

The CSA is heavier than the liveboard algorithm to process for the clients. This can be seen in the CPU usage of the CSA. As said above, if faster hardware is used, the data processing requires less CPU power for the same result. In figure 4.14, we can see that the Xperia X can update the journey faster than the Jolla 1 in all implementations.

This is reflected in the refresh time of the CSA planner. The Jolla 1 can reroute a journey in approximately 20 seconds (with HTTP caching) while the Xperia X can handle the same operation in only 10 seconds. The initial peak in the refresh time is related to the fact that we always start with a clean cache.

In the beginning of each benchmark, they both cache each Linked Connections page. In the first benchmark, when a refresh action was triggered, the HTTP cache is used. This is visible in the network usage graph. Only changed Linked Connections pages are completely fetched from the server. All other pages returned immediately a `HTTP 304 Not Modified` status code. This is not the case with the RAM cache, the update algorithm keeps the RAM cache up-to-date in the background. The CSA does not have to perform any additional network queries to retrieve the latest version of a page. Since the network usage is almost the same for both devices, we illustrate this with a single device in figures 4.12 and 4.13.

If we compare the benchmarks for the CSA (figures 4.14, 4.15, 4.16, 4.17 and 4.18), we can draw almost the same conclusions as with the liveboard benchmark (section 4.2.2.1):

1. **CPU usage**: Thanks to the rollback mechanism we implemented in the CSA, less CPU power is used. The Jolla 1 could save almost 34 % of its CPU usage when using our modified version of LCRail. The Xperia X was able to save even more, the CPU usage was reduced by 50 %. The difference here is related to the small overhead the restoring process causes when a journey is restored. The Xperia X can locate the CSA snapshot faster then the Jolla 1.

[6] Including the necessary Linked Data vocabulary to understand the data.

2. **RAM usage**: The amount of RAM usage is almost the same in the 3 benchmarks for both devices.

3. **Network usage**: The amount of network bandwidth is reduced. For the same data, our modified versions sent 90 % less data to the Linked Connections server. The HTTP polling version receives 5 % less data, while the SSE implementation saves up to 15 % of data. The difference is here bigger than with the liveboard since the CSA uses more Linked Connections pages to calculate a route than the liveboard algorithm. Since the HTTP polling implementation receives empty events arrays[7] too, less data is saved than with the SSE implementation.

4. **Refresh time**: The refresh time is reduced by a factor of 2 in comparison with standard HTTP caching. On the Jolla 1, it took 20 seconds to refresh a journey. With our modifications, we were able to reduce this to less than 10 seconds. Since the Xperia X is a more powerful device, we were able to reduce the refresh time from less than 12 seconds to approximately 4 seconds. A commuter will be informed much faster about a change in a journey than with the original implementation.

4.4 Conclusion

In this chapter, we gave an overview of the test environment for our benchmarks and discussed our results.

We used 6 nodes of the Virtual Wall as clients during the *pushing or polling* experiment. During the *processing of updates* experiment, 2 Sailfish OS devices acted as clients. In every benchmark, a Digital Ocean droplet was used as the Linked Connections Server.

We saw that our modified version of the LCRail client was faster than the original implementation. The modified version also used less CPU and network resources when using the real time resource. This was the case for both of our experiments.

[7] Including the necessary Linked Data vocabulary to understand the data.

5 Conclusion

We now provide an answer to our research questions and hypotheses from section 1.2 with our literary study from chapter 2 and the results from chapter 4.

How can we keep a cache for route planners up to date in a cost-efficient way for both Open Data publishers and consumers?

We compared several technologies in our literary study and benchmarks. The main difference between all these technologies is how they deliver the data to the client. Some used a pushing approach, others a polling approach. With pushing, the server will push the data to the client on its own. In case of polling, the client will repeatedly ask the server if there is new data available or not.

We can conclude from our results that a pushing approach is more cost-efficient in terms of bandwidth for the Open Data publishers and consumers. With pushing, we only have to transfer data when it is really needed. If there are no updates available, we do not transfer any data. This is not the case with polling, we still have to transfer some data even if no updates are available yet.

Another reason for using a pushing approach, is that less processing power is needed. Since less data is transferred, we do not have to parse anything. Thanks to pushing (SSE), we reduced the mean CPU usage almost 4 times (2,3 % instead of 8,9 %). The spikes in the CPU usage are lower (15 %) in comparison to HTTP polling (35 %). The spikes are less frequent and shorter.

The results confirm our hypothesis about this research question:

"We suspect that we can improve the cache of route planners in a more cost-efficient way by using updates. Instead of fetching the complete data set again, we only fetch the changes to the data set. This way, we should be able to reduce the network bandwidth and computing resources for Open Data publishers and consumers."

The cache of the route planner is indeed updated in a more cost-efficient way. Only the changed data is fetched and updated. The network bandwidth and computing resources are reduced for Open Data publishers and consumers.

How can we consume real time Open Data updates in route planner algorithms in an efficient way?

Consuming real time Open Data updates in route planner algorithms without any additional logic is a waste of resources. We clearly saw in our benchmarks that refreshing a liveboard or a journey takes almost the same amount of time as constructing one from scratch. However, if we only process the updates to the liveboard or journey, the refresh time decreases enormously.

For the liveboard, we were able to reduce the refresh time with a factor of 10. Each updated connection in a Linked Connection page is replaced in the liveboard. All the other entries in the liveboard remain unaffected.

By applying a rollback mechanism to the CSA, we only had to recalculate a part of the journey. Only the part that is affected by a Linked Connections page update, is recalculated. Thanks to this approach, we were able to reduce the refresh time of the CSA by a factor of 2-3. The reduction is depending on the hardware used in the test. The faster the hardware, the higher the refresh time reduction. This is related to the small overhead of restoring a journey in the CSA.

The results confirm our hypothesis about this research question:

"Instead of recalculating a complete new route if the data has been updated, we only update the affected data. Since not all the data is changed at once, the routing algorithms must be able to reuse their previous calculations. With this approach, the results must be faster updated than in the current approach. The user of the data should be faster informed about any changes in his or her trip."

Only a part of the route is recalculated with the CSA. The changed liveboard entries are the only entries that are updated. The user is informed much faster than in the original approach. We could indeed reuse the previous calculations to make the algorithms faster (for example: the CSA rollback mechanism).

Future work

In this thesis, we created a client side library for Linked Connections (QRail). This library is consumed by the LCRail client. However, our work does not end here. The library can be optimised even further by applying several improvements.

If we use smart pointers, we can reduce the memory usage even further. The reason for this, is the fact that C++ will take over the managing of the heap memory. As soon as an object is not used anymore, C++ will automatically delete the object it is holding.

Another optimisation would be the use of a faster JSON parser. The default Qt JSON parser is slower than other C++ JSON parsers. Changing the JSON parser to a faster one, would improve the processing time when data needs to be fetched from the server.

We could lower the initial fetching time of the Linked Connections pages by using a prefetch algorithm. With prefetching, QRail can load the Linked Connections pages directly from the disk cache. If we make the prefetching concurrent by downloading multiple pages at once, we can lower the prefetching time even more.

Last but not least, the CSA rollback mechanism could be made more efficient by using a binary search algorithm to locate the snapshot which we need to restore.

Bibliography

[1] Pieter Colpaert et al. Intermodal public transit routing using linked connections. 2015. URL https://linkedconnections.org/lc_demo_paper.pdf.

[2] Tim Berners-Lee. Tim berners-lee. https://www.w3.org/People/Berners-Lee/, 2018. [22 October 2018].

[3] Open Knowledge International. What is open data? http://opendatahandbook.org/guide/en/what-is-open-data/, . [22 October 2018].

[4] Open Knowledge International. The open definition. http://opendefinition.org/, . [22 October 2018].

[5] Julian Dibbelt et al. Connection scan algorithm. *CoRR*, abs/1703.05997, 2017. URL http://arxiv.org/abs/1703.05997.

[6] Bert Marcelis. De door de gebruiker ervaren performantie van routeplanning-api's. Master's thesis, University of Ghent, 2018.

[7] Marek Obitko. Rdf graph and syntax. http://www.obitko.com/tutorials/ontologies-semantic-web/rdf-graph-and-syntax.html, 2007. [5 October 2018].

[8] What is server-sent events? https://streamdata.io/blog/server-sent-events/, 2017. [8 October 2018].

[9] Ilya Grigorik. Http caching. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching, 2018. [14 October 2018].

[10] Open Knowledge Belgium. irail. https://irail.be/. [22 October 2018].

[11] Open Knowledge International. Why open data? http://opendatahandbook.org/guide/en/why-open-data/, . [22 October 2018].

[12] Joost Vennekens and Herman Crauwels. Webtechnologie, 2016.

[13] Florian Bauer and Martin Kaltenböck. *Linked Open Data: The Essentials*. edition mono/monochrom, Vienna, Austria. ISBN 978-3-902796-05-9.

[14] Michael Hausenblas. 5 star open data. https://5stardata.info/en/, 2012. [22 May 2018].

[15] Tim Berners-Lee. Linked data. https://www.w3.org/DesignIssues/LinkedData.html. [15 November 2018].

[16] The Linked Data Fragments collaborators. Linked data fragments. http://linkeddatafragments.org/. [15 November 2018].

[17] Ruben Verborgh et al. Querying datasets on the web with high availability. 2014. URL http://linkeddatafragments.org/publications/iswc2014.pdf.

[18] Google and GTFS contributers. Gtfs static specification. http://gtfs.org/, 2016. [8 November 2018].

[19] Google and GTFS contributers. General transit feed specification. http://gtfsrt.com/. [6 November 2018].

[20] EACOMM Corporation. General transit feed specification real time. http://gtfsrt.com/. [6 November 2018].

[21] Transmodel Update Project Team. Transmodel cen standard. http://www.transmodel-cen.eu/. [7 November 2018].

[22] NeTEx working group. Netex cen standard. http://netex-cen.eu/. [8 November 2018].

[23] CEN group. Standard interface for real-time information. http://www.transmodel-cen.eu/standards/siri/, . [8 November 2018].

[24] CEN group. Cen ts 15531 service interface for real time information (siri). https://www.vdv.de/siri.aspx, . [8 November 2018].

[25] Pieter Colpaert et al. The impact of an extra feature on the scalability of linked connections. 2016. URL http://ceur-ws.org/Vol-1666/paper-05.pdf.

[26] K. Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pages 177–182, May 2012. doi: 10.1109/DICTAP.2012.6215346.

[27] Douglas Crockford. Introducing json. http://www.json.org/. [5 October 2018].

[28] Saurabh Zunke and Veronica D'Souza. Json vs xml: A comparative performance analysis of data exchange formats. http://ijcsn.org/IJCSN-2014/3-4/JSON-vs-XML-A-Comparative-Performance-Analysis-of-Data-Exchange-Formats.pdf. [10 December 2018].

[29] W3C JSON-LD Community Group. Json for linking data. http://www.json-ld.org, 2014. [5 October 2018].

[30] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. [5 October 2018].

[31] Sadayuki Furuhashi. Messagepack. https://msgpack.org/index.html. [5 October 2018].

[32] Google. Protocol buffers. https://developers.google.com/protocol-buffers/. [5 October 2018].

[33] Bruno Krebs. Beating json performance with protobuf. https://auth0.com/blog/beating-json-performance-with-protobuf/. [6 October 2018].

[34] Apache Software Foundation. Apache thrift. https://thrift.apache.org/. [4 May 2019].

[35] Mozilla contributors. Connection management in http/1.x. https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x, 2018. [13 October 2018].

[36] Browserscope contributors. Browserscope. https://www.browserscope.org/?category=network&v=top, 2018. [13 October 2018].

[37] WHATWG community. Server-sent events. https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events, 2018. [6 October 2018].

[38] The gRPC Authors. Frequently asked questions. https://grpc.io/faq/. [5 October 2018].

[39] Aaron Krauss. How websockets work – with socket.io demo. https://thesocietea.org/2016/04/how-websockets-work-with-socket-io-demo/, 2016. [5 October 2018].

[40] Mozilla contributors. The websocket api (websockets). https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API, 2018. [5 October 2018].

[41] MQTT contributors. Mqtt. https://mqtt.org/. [8 October 2018].

[42] HTTPbis Working Group. *Hypertext Transfer Protocol Version 2 (HTTP/2)*, 2015. [7 October 2018].

[43] Mozilla contributors. Http caching. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching, 2018. [14 October 2018].

[44] Internet Engineering Task Force (IETF). *Hypertext Transfer Protocol (HTTP/1.1): Caching*, 2014. [8 October 2018].

[45] Internet Engineering Task Force (IETF). *HTTP Immutable Responses*, 2017. [7 October 2018].

[46] Pieter Colpaert. Linked connections specification 1.0. https://linkedconnections.org/specification/1-0, 2018. [12 October 2018].

[47] Pieter Colpaert et al. Public transit route planning through lightweight linked data interfaces. 2017. URL https://pietercolpaert.be/papers/icwe2017-lc/paper.pdf.

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *1994 Patterns: Elements of Reusable Object-Oriented Software*. 2018. ISBN 0201633612.

Attachments

The code of this thesis is available on the included CD and through Github. In A, we give an overview of all our contributions and the code we wrote.

Attachment A: Code overview
Attachment B: Source code (on the CD)

A

Code overview

In this thesis, we modified the Linked Connections Server, created our own QRail library and LCRail client. With these modifications, we added a real time resource to Linked Connections.

This way, the user is informed as soon as possible about events on the public transport network. The user gets automatically an alternative for his or her journey with our QRail library and LCRail client.

In this attachment, we give an overview of the actual work that we did to achieve this goal (chapter 1). To give an idea about the amount of work, we added an overview of each contribution and the amount of lines we changed.

A.1 Server side

The gtfsrt2lc library We modified the `gtfsrt2lc` library to support canceled vehicles from the GTFS-RT stream. We also noticed that the library was not capable of handling the delay information of the NMBS. They used a feature of the GTFS-RT specification which was not implemented by the library. We updated the `gtfsrt2lc` library to support this feature of the GTFS-RT specification. More information about these modifications can be found in section 3.3.1.

All our changes our upstreamed to the Github repository of the `gtfs2lc` library:

Contribution	Added lines	Removed lines
https://github.com/linkedconnections/gtfsrt2lc/pull/8	+32	-6
https://github.com/linkedconnections/gtfsrt2lc/pull/9	+33	-7
https://github.com/linkedconnections/gtfsrt2lc/pull/10	+3891	-1094

Linked Connections Server The Linked Connections Server uses the `gtfsrt2lc` library to parse the GTFS-RT stream. We created an Events Manager to manage the received events from the GTFS-RT stream. The Events Manager generates the required JSON-LD files for the real time resource.

The server publishes the events as a JSON-LD file for HTTP polling clients and pushes it to the connected SSE clients. The Linked Connection Server modifications are explained in sections 3.3.3 and 3.3.4.

We added the real time resource to the Linked Connections Server with the following contributions:

Contribution	Added lines	Removed lines
https://github.com/julianrojas87/linked-connections-server/pull/22	+7	-3
https://github.com/julianrojas87/linked-connections-server/pull/26	+7068	-3

Linked Connections Reproduction The Linked Connections Reproduction server is a file server to reproduce real time events in our test environment. We created this file server to be able to compare different approaches in a reproducible way.

More information can be found in section 4.1.

Repository	Added lines
https://github.com/DylanVanAssche/Linked-Connections-Reproduction	+655

A.2 Client side

The QRail library In order to achieve our goal, we must have a modular client for Linked Connections. With this modular approach, we can change how Linked Connections is used by the client.

Since no such client was available yet, we created the QRail library. We wrote this library from scratch in Qt C++. We modified it afterwards to use the real time resource of the Linked Connections Server.

More information about QRail and how it was written, can be found in section 3.4.

Repository	Added lines
https://github.com/DylanVanAssche/QRail	+19993

LCRail The LCRail client is a user friendly UI for the QRail library. To test our implementation, we wrote a Sailfish OS client in Qt QML. LCRail interfaces with the QRail library and visualises the results from the library. The original implementation of LCRail can be found on the `lcrail-original` branch on Github[1] or on the CD. The modified version can be found on the `lcrail-modified` branch on Github[2] or on the CD.

LCRail is further explained in section 3.5.

Repository	Added lines
https://github.com/DylanVanAssche/LCRail	+3590

A.3 General flow

In this section, we give an overview of the general flow of our algorithms. First, the server side algorithms are illustrated with a flowchart. Afterwards, the algorithms for the client side processing are visualised.

A.3.1 Server side

The Linked Connections Server and the `gtfsrt2lc` library are using the algorithms below to provide a real time resource to the Linked Connections clients.

[1] https://github.com/DylanVanAssche/LCRail/tree/lcrail-original
[2] https://github.com/DylanVanAssche/LCRail/tree/lcrail-modified

A.3.1.1 GTFS-RT processing

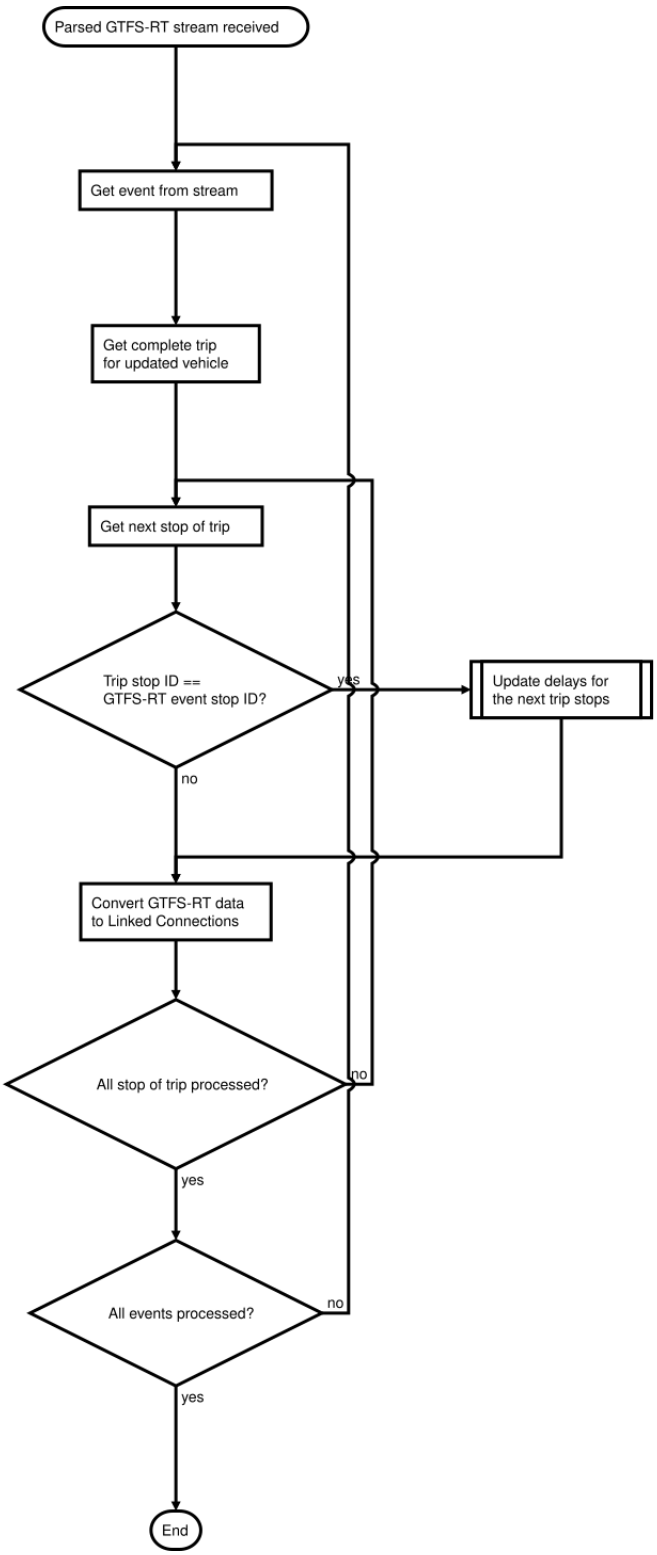


Figure A.1: Processing of the GTFS-RT stream in the `gtfsrt2lc` library.

A.3.1.2 Real time data publishing

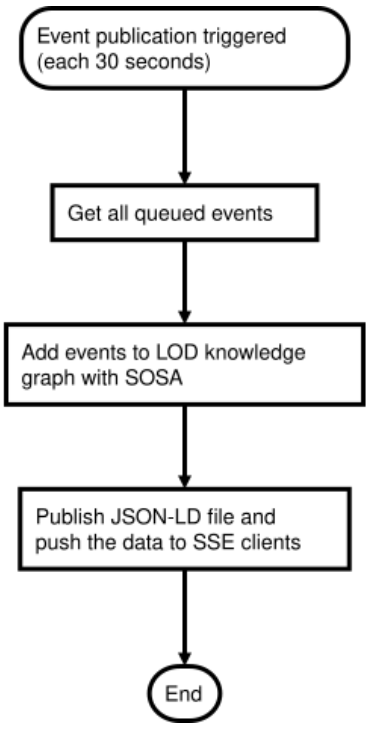


Figure A.2: Real time resource publishing in the Linked Connections Server.

A.3.2 Client side

The QRail library and the LCRail client make use of the algorithms below. These algorithms process the real time data, generate journeys and liveboards and to provide alternatives, in case of a vehicle cancellation or delay.

A.3.2.1 Searching for a stop

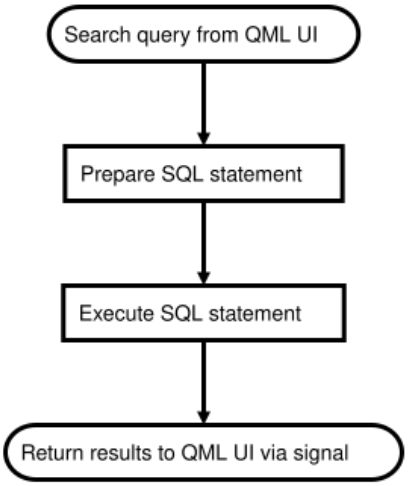


Figure A.3: Stop search in the LCRail client.

A.3.2.2 Update processing

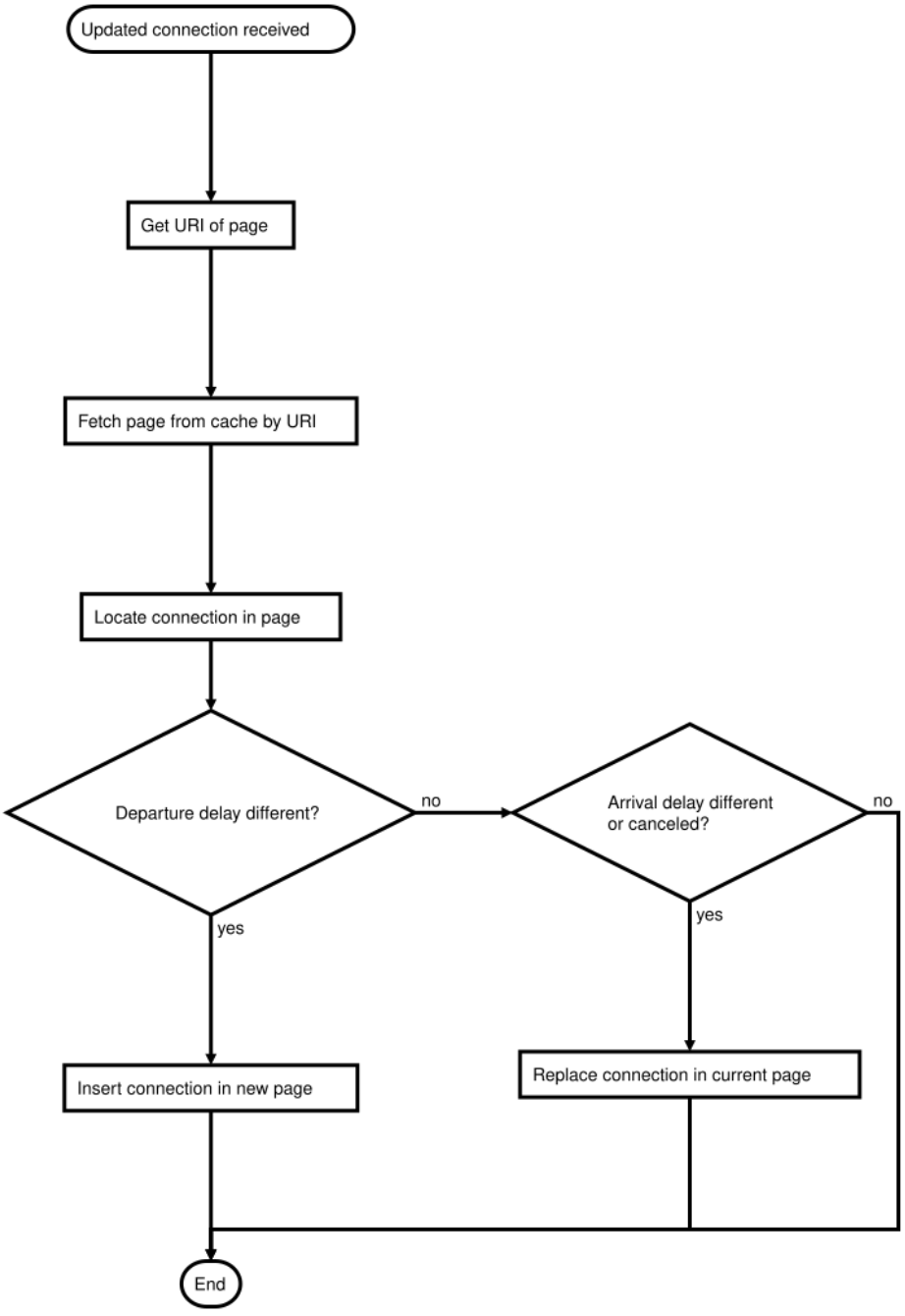


Figure A.4: Processing of updates, retrieved from the real time resource.

A.3.2.3 Retrieving a liveboard

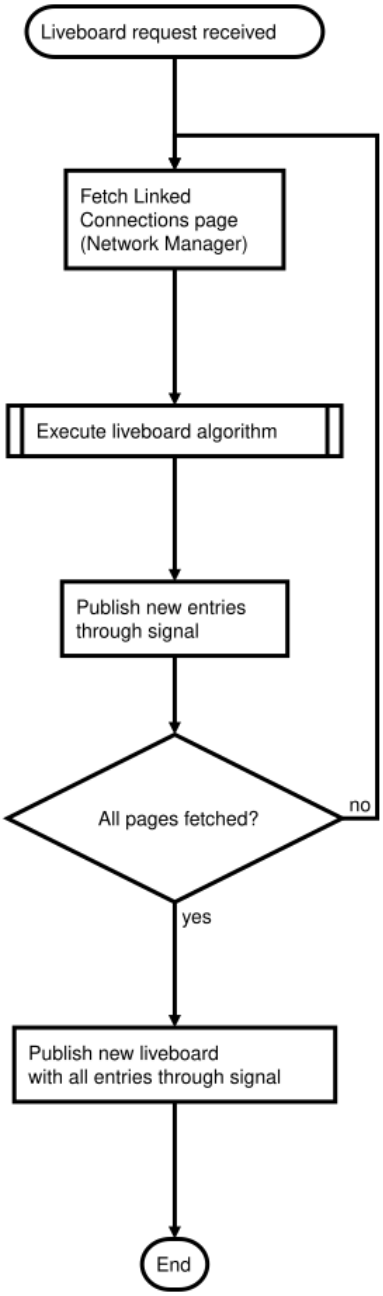


Figure A.5: Retrieving a liveboard for a given stop.

A.3.2.4 Planning a journey

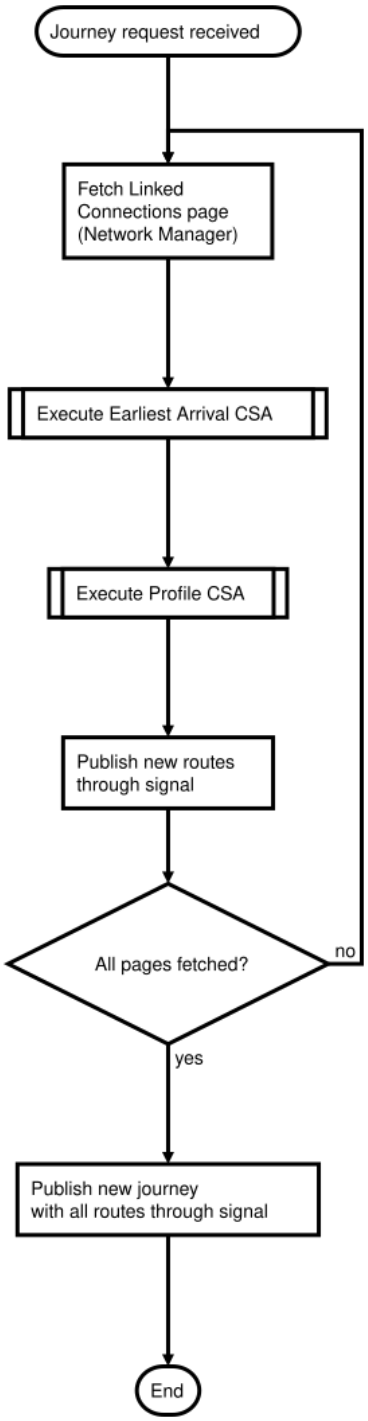


Figure A.6: Planning a journey for a given departure, arrival stop and departure time.

A.3.2.5 Updating a liveboard

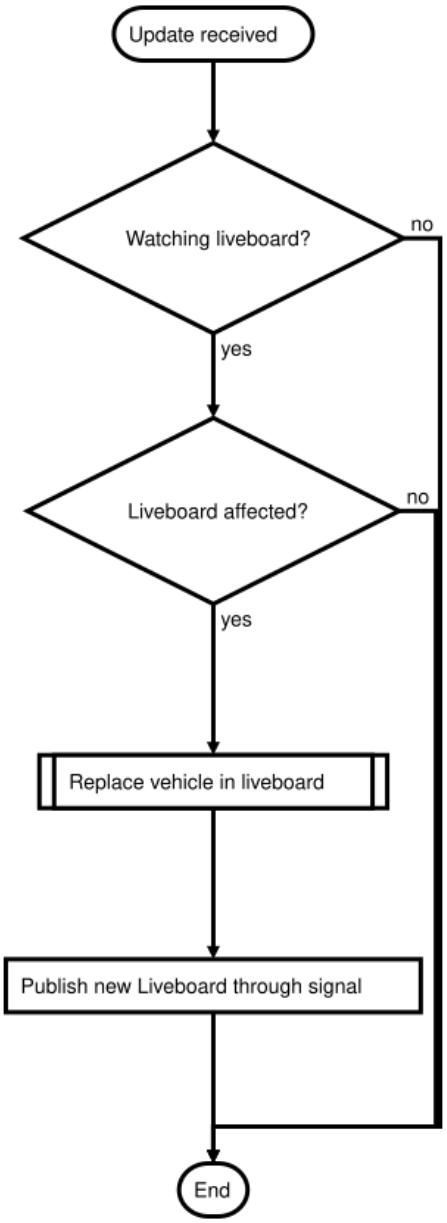


Figure A.7: Updating a watched liveboard.

A.3.2.6 Updating a journey

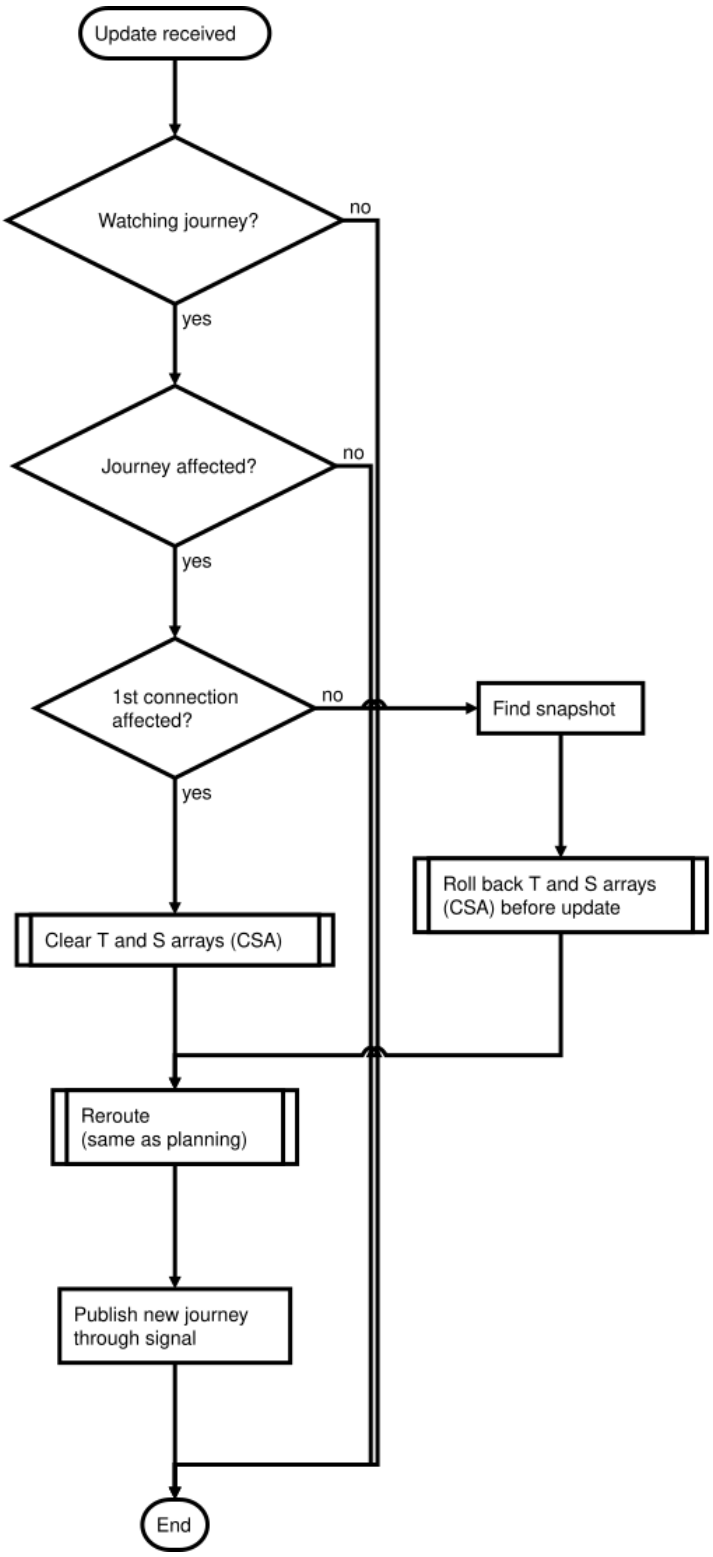


Figure A.8: Updating a watched journey.

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS DE NAYER SINT-KATELIJNE-WAVER
J. De Nayerlaan 5
2860 SINT-KATELIJNE-WAVER, België
tel. + 32 15 31 69 44
iiw.denayer@kuleuven.be
www.iiw.kuleuven.be