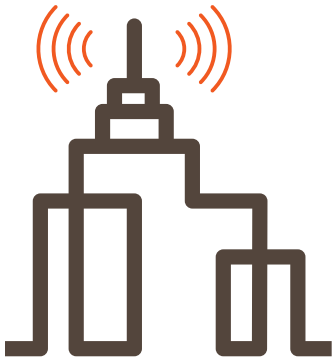
VaVeL
H2020 - 688380



VaVeL

D8.1 - First Report on Warsaw Use Case

Orange Polska S.A, Warsaw University of Technology, City of Warsaw

May 30, 2017

Status: Final

Scheduled Delivery Date: 31/05/2017

Document History

- (April 1th, 2017) Version 1.0
- (April 20th, 2017) Version 1.1
- (April 25th, 2017) Version 1.2 after internal review
- (May 29th, 2017) Version 1.3 Submitted to the EC and uploaded to VaVeL website.

Executive summary

The aim of this report is to provide overall description of the City of Warsaw prototype of VaVeL platform. The key aspects related to CoW use-cases are: (i) processing of live stream of tram data, (ii) processing of live stream of bus data and (iii) linking live streams with schedules and other static information. The report gives overview on the test beds and production CoW VaVeL platform. The test beds are installed in Orange Polska premise ('bajorko' cluster) and Warsaw University of Technology, and are both based on Apache Hadoop ecosystem, mostly on customized Horton Data Platform installation. The production platform used within Warsaw pilot will be hosted at the City of Warsaw IT Department and will follow the same guidelines and design pattern on both test beds. The report provides deep technical description of component developed especially for CoW pilot, including, but not limited to:

(a) Apache Flink-based components for live stream processing:
    (a) Integration of live streams with schedule data
    (b) Filtering of tram and bus data
    (c) Estimating delays in trams/buses
    (d) Estimating type of movement for trams/buses
(b) Apache Flume based components devoted to data collection, especially Flume's components acting as a gateways to external systems, data sources etc., including but not limited to:
    (a) Polling data source about live trams and buses data
    (b) Collecting ZTM (Warsaw's Public Transport Authority) RSS channel news
(c) The backend components for mobile application in Warsaw pilot

The technical description of the modules provides the guidelines on modules installation, running and detailed description of processed data. Moreover the expected inputs and outputs are included in order to simplify the verification of proper functioning of the modules in new environment. This directly addresses the VaVeL project interoperability goal which aims the platform to be transferable to large number of cities after the project is finished. The mobile application backend (MAB) is based on components out of standard Hadoop ecosystem, but its main goal is to support mobile application, eliminate unnecessary load on the platform and minimize network traffic. There is necessity of providing expected scalability and performance. MAB consumes processed data streams from the core of VaVeL platform and makes them available on a subscription-based mechanism to the mobile application users. The document was prepared as part of Task T8.1- Prototype Development.

Document Information

Contract Number	H2020-688380	**Acronym**	VaVeL
Name	Variety, Veracity, VaLue: Handling the Multiplicity of Urban Sensors		
Project URL	http://www.vavel-project.eu/		
EU Project Officer	Francesco Barbato		

Deliverable	**D8.1**	First Report on Warsaw Use Case		
Work Package	**Number**	WP8		
Date of Delivery	31/05/2016	**Actual**	31/05/2016	
Status	Final			
Nature	Report			
Distribution Type	Public			
Authoring Partner	Orange Polska S.A.			
QA Partner	OPL			
Contact Person	Jarosław Legierski	jaroslaw.legierski@orange.com		
	Izabella Krzeminska	izabella.krzeminska@orange.com		
	Phone		**Fax**	

List of Contributors: Jarosław Legierski (OPL), Marcin Luckner (WUT), Tomasz Zaremba (WUT), Paweł.Zawistowski (WUT), Karolina Kwasiborska (WUT), Przemysław Biecek (WUT), Izabella Krzeminska (OPL), Robert Kunicki (CoW), Piotr Wawrzyniak (OPL)

Project Information

This document is part of a research project funded by Horizon H2020 programme of the Commission of the European Union as project number 688380. The beneficiaries in this project are:

No.	Name	Short Name	Country
1	National and Kapodistrian University of Athens	UoA	Greece
2	Technische Universität Dortmund	TUD	Germany
3	Technion - Israel Institute of Technology	Technion	Israel
4	Fraunhofer-Gesellschaft Zur Förderung Der Angewandten Forschung E.V.	Fraunhofer	Germany
5	IBM Ireland Limited	IBM	Ireland
6	AGT International	AGT GROUP (R&D) GMBH	Germany
7	Orange Polska S.A.	OPL	Poland
8	Dublin City Council	DCC	Ireland
9	City of Warsaw	CoW	Poland
10	Warsaw University of Technology	WUT	Poland

Table of Contents

Index of Figures

1 Introduction

In this document the VaVeL consortium presents information about VaVeL framework platform components dedicated for City of Warsaw use case realization developed in Work Package 8 in first half of project schedule. The City of Warsaw (CoW) use case is concentrated on usage public transport related data sets such as: public transport timetables, vehicles location, information from Public Transport Authority (ZTM) portal (RSS) and Twitter account and data corelated with citizen activity (data related to non emergency issues reported by citizens - from special number: 19115).

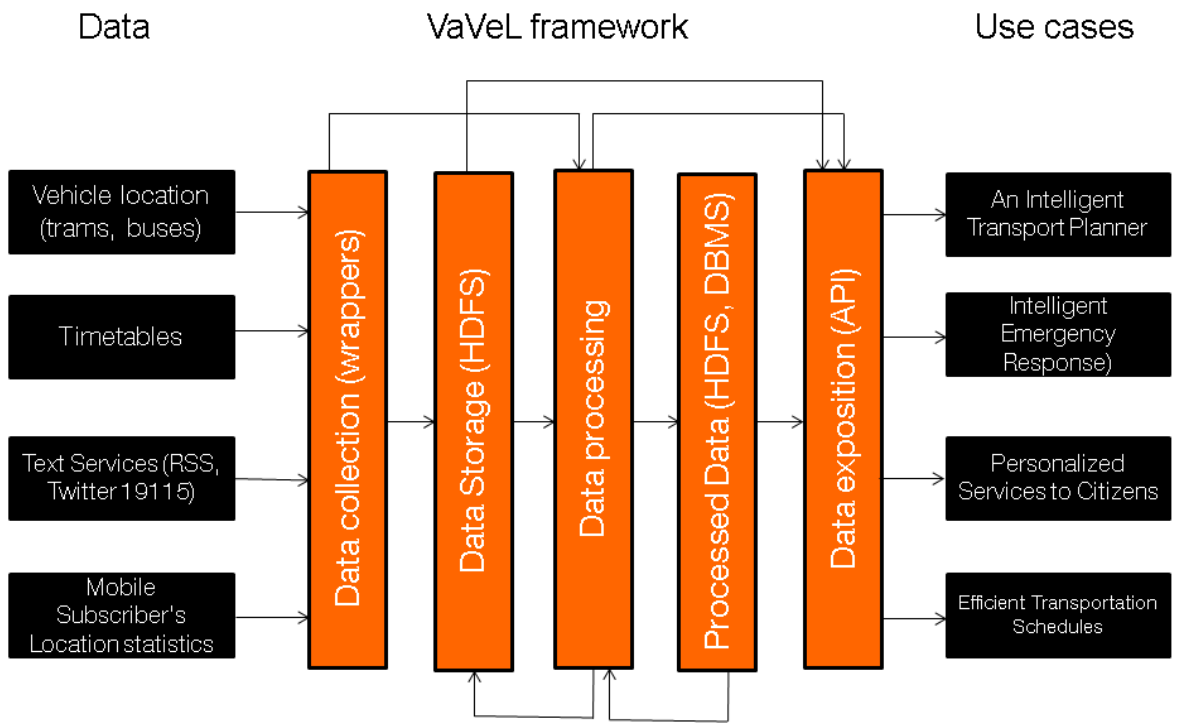


Figure 1: VaVel framework implementation for CoW use case

From end users point of view, in Warsaw the VaVeL framework implements four use cases:

- An Intelligent Transport Planner - situation aware trip planner for Warsaw citizens
- Intelligent Emergency Response. - context aware notification system
- Personalized Services to Citizens - personalized and context aware citizen service
- Efficient Transportation Schedules - dedicated application for Warsaw Public Transport Authority (ZTM)

Detailed functional and non functional requirements listed above use cases contains chapter User Requirements in D6.1 Report on Requirements and Architecture. The implementation details are described in the document D6.2 First Report on system Integration.

2 Detailed CoW use case architecture

The document D6.1 Report on Requirements and Architecture contains high level of VaVeL framework system architecture dedicated for realization two different use cases for Dublin and Warsaw. This report is concentrating on Warsaw' specific implementation. The detailed architecture of platform dedicated for CoW use case realization is presented on Fig.2.

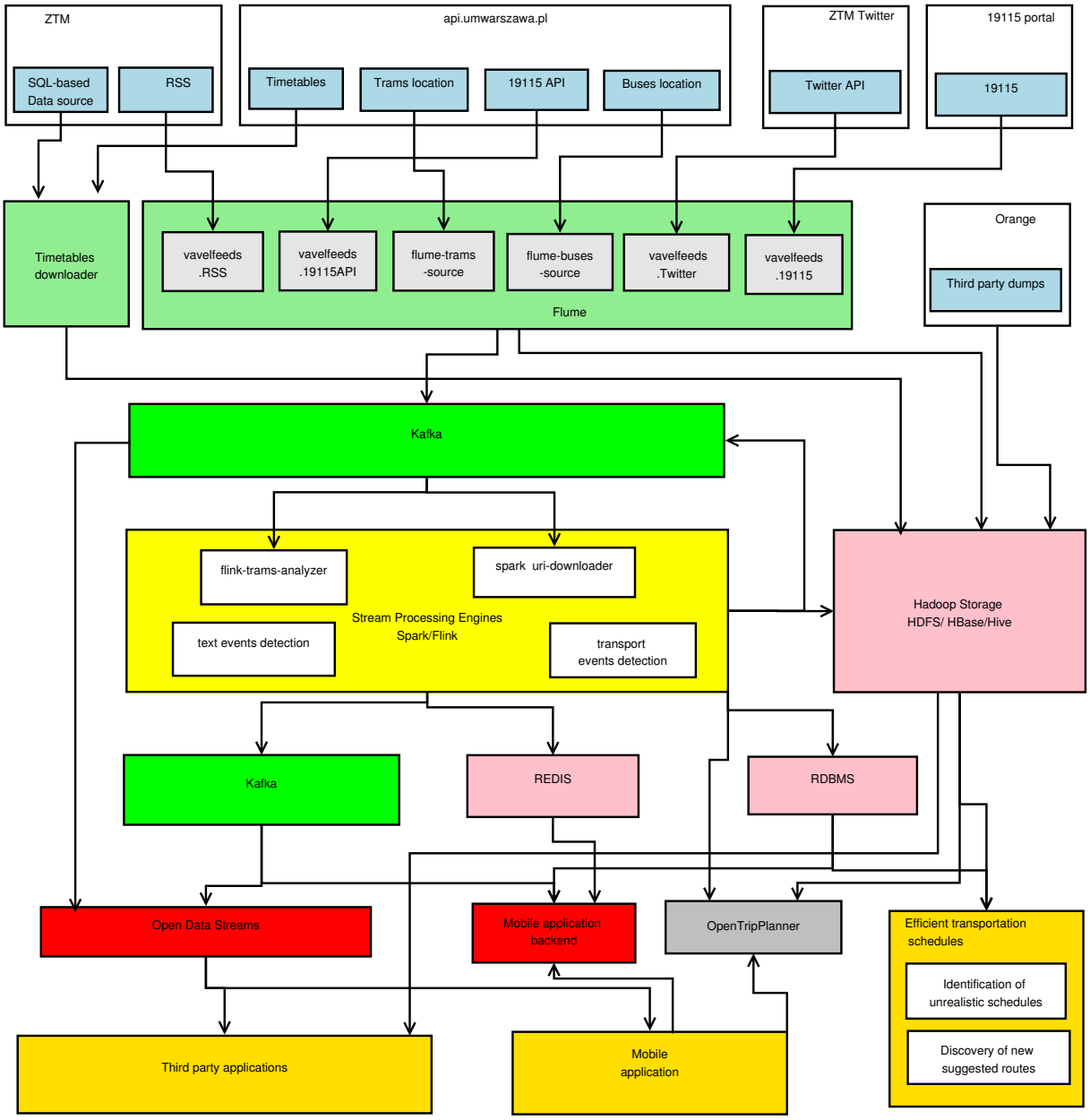


Figure 2: Scheme of CoW detailed architecture

In presented on Fig.2 system architecture following elements and layers can be specified:

1. **Data sources layer:**
   - SQL - database oriented services from ZTM (e.g. public transport timetables details)
   - RSS - news services from ZTM web portal
   - Services exposed by api.um.warszawa.pl platform using Web Services (timetables, trams and buses locations)
   - text information from citizens from non emergency issuses reporting system 19115 (Web Services)
   - text information from citizens from 19115 system (from web page)
   - ZTM Twitter - social network API - information from ZTM Twitter account
   - Orange data - PLMN statistics
2. **Data acquisition layer:**
   - Real time data collection modules (Apache Flume agents)
   - Timetables downloader - dedicated application for timetables acquisition
3. **Data processing layer:**
   - Stream processing engines - dedicated for processing stream data (based on Apache Spark and Apache Flink)
   - Message queuing system based on Apache Kafka
4. **Data Storage layer:**
   - Hadoop storage - dedicated for big data storage and batch processing (HDFS/Map Reduce, Apache HBase, Apache Hive)
   - RDBMS- Relational Database Management System - database dedicated for aggregated data storage (PostgreSQL)
   - Redis - in memory DB - database dedicated for mobile application used for notifications
5. **Data Exposition layer:**
   - Open Data Streams - CoW streams dedicated for exposition in Open Data model (e.g using existing CoW portal api.um.warszawa.pl)
   - Web Application - web application dedicated for CoW citizens
   - Mobile application backend - server part of mobile application
   - Open Trip Planner - an open source multi-modal trip planner for mobile application
6. **Application layer:**
   - Third party applications - application 3rd parties open data consumers
   - Mobile application - mobile application for CoW citizens
   - Efficient transport schedule - dedicated application for ZTM use case

3 Preparation of data gathering mechanisms and preprocessing of transport data

3.1 Description

The aim of this part of the project is to create a mechanism that will collect information about location of trams and buses in Warsaw and perform analysis on it. Due to the fact that the data will be coming in regular, short intervals, it is necessary to ensure scalability and efficiency of the solution. Data analysis will allow us to predict the movement of vehicles, detect lacks of data and to draw basic statistics about traffic and efficiency of trams and buses' movement in Warsaw. Then this information could be used to improve the operation of public transport in the city.

3.2 Data

Location of trams

Information about location of trams are downloaded from Warsaw API open data platform `http://api.um.warszawa.pl` New data appears every thirty seconds approximately. They are available in JSON format and contains the following information for each tram:

- *FirstLine*: string -first line number
- *Brigade*: string - brigade number
- *Time*: datetime - record timestamp
- *Lat*: float- latitude GPS
- *Lon*: float - longitude GPS
- *Lines*: string- the numbers of all lines (for multiline brigades there will be more than one line)
- *Status*: string- determines whether a tram is "RUNNING" or the ride is "FINISHED"
- *LowFloor*: bool- determines whether the tram is low-floor (1 - yes, 0 - no)

The new information about locations and statuses of trams appears in the source every 30 seconds, however, to ensure that no data is misse, the system queries the API more often. The frequency of polling the service `http://api.um.warszawa.pl` can be parametrized (default value is 20 seconds). Fragment of an example JSON array is presented on figure 3.

Location of buses

The information about bus location is exposed as open data. The data is available from begining of April 2017 on the websisite `https://api.um.warszawa.pl` and contains the following fields for each bus:

- *Lines*: string - line number
- *Brigade*: string - brigade number
- *Time*: datetime - record timestamp
- *Lat*: float - latitude GPS
- *Lon*: float - longitude GPS

```
{
   "result":[
      {
         "Status":"RUNNING",
         "FirstLine":"6",
         "Lon":21.0585594,
         "Lat":52.2468872,
         "Lines":"6",
         "Time":"2016-01-10T13:26:53",
         "LowFloor":false,
         "Brigade":"7"
      },
      ...
   ]
}
```


Figure 3: Trams location data - fragment of an example JSON array

3.3 Module description

The raw location data is polled by a custom data source ran on Apache Flume. Then they are saved on a distributed file system, HDFS, which is a part of Apache Hadoop framework, and forwarded to Apache Kafka messaging system. The data from the message queue is consumed by Apache Flink which is used to clean and pre-process information about trams and buses in streaming mode. Finally results are stored in HDFS.

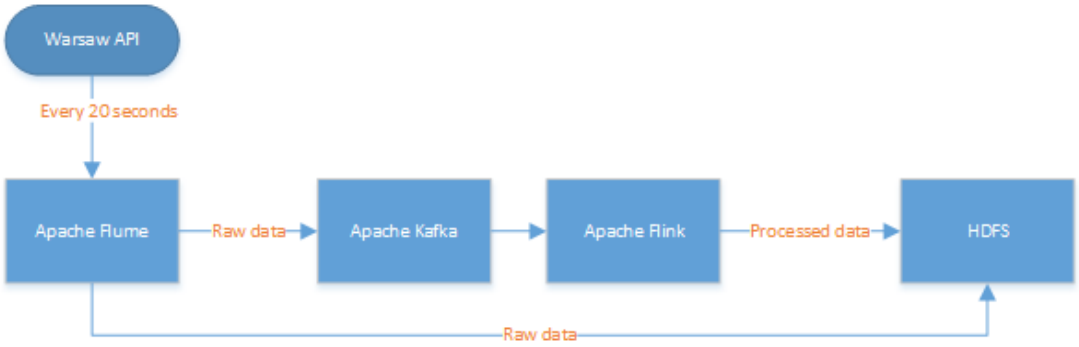


Figure 4: Trams and buses locations data flow

3.4 Data acquisition

The Flume Source (Trams downloader) periodically download new information about location of vehicles. The data is converted to the list of objects, one for each vehicle, which contain data about position of each vehicle and timestamp. The each vehicle is identified by a combination

of fields "line number" and "brigade number". The raw data is written to HDFS and passed to Flink by a message queue inside Kafka.

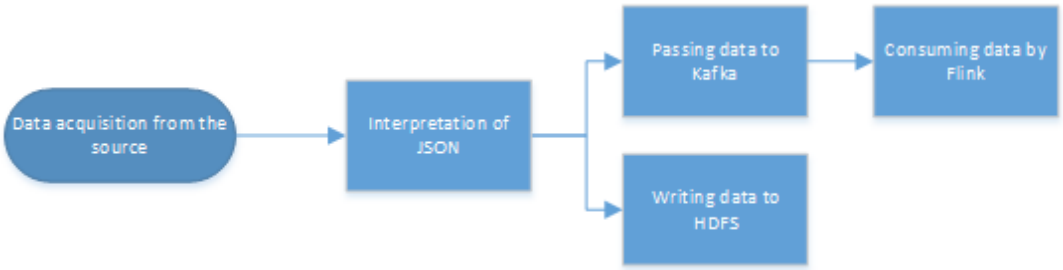


Figure 5: Scheme of data acquisition

3.5 Data analysis

Data analysis is performed by Apache Flink in streaming mode.

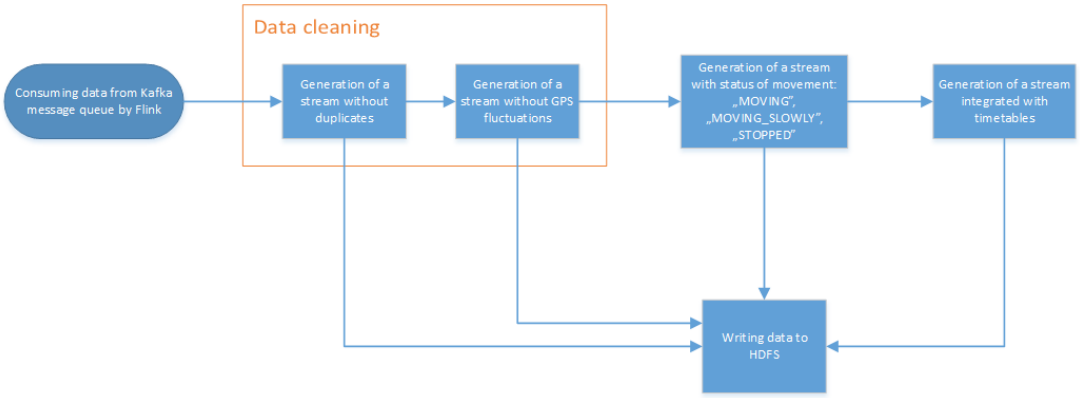


Figure 6: Scheme of data pre-processing

Data cleaning

Every time new data appears with the information about vehicles, the following steps are performed for each record: The record about a vehicle (identified by a line-brigade key) which appears in a given day for the first time is added to an auxiliary map. If the map contains information about this vehicle, the system is checking if timestamp of a new record is the same as timestamp of a record from the map. If yes the processed record is treated as a duplicate and missed (not stored). Otherwise we calculate the distance between two measurements. If it is less than 1 meter we treat it as GPS fluctuation and save it with old longitude and latitude.

Data with movement status

For clean data we determine a movement status of a vehicle based on a distance calculated during data cleaning.

- If a vehicle travelled less than 1 meter than a status is "STOPPED"

- If a distance is between 1 and 10 meters than a status is "MOVING_SLOWLY"
- Otherwise a status is "MOVING"

3.6 Output data

Raw and deduplicated data

Trams

- *FirstLine*: string - first line number
- *Brigade*: string - brigade number
- *Time*: datetime - record timestamp
- *Lat*: float - latitude GPS
- *Lon*: float - longitude GPS
- *Lines*: string - the numbers of all lines (for multiline brigades there will be more than one line)
- *Status*: string - determines whether a tram is "RUNNING" or the ride is "FINISHED"
- *LowFloor*: bool - determines whether the tram is low-floor (1 - yes, 0 - no)
- *ReceivedTime*: - cluster server time of getting the data from API

Buses

- *Line*: string - first line number
- *Brigade*: string - brigade number
- *Vehicle*: string - vehicle number
- *Time*: datetime - record timestamp
- *Lat*: float - latitude GPS
- *Lon*: float - longitude GPS
- *ReceivedTime* - cluster server time of getting the data from API

Clean data

Deduplicated raw data with denoisified location

The same data as above plus:

- *Lat*: float - latitude GPS
- *Lon*: float - longitude GPS
- *ProcessingFinishedTime* - local host time of writing the data to HDFS

Data with movement status

The same data as above plus:

- *TramStatus*: string - determines whether a tram is "STOPPED" (did not move by at least 1 meter since the last record), "MOVING_SLOWLY" (moved by 1-10 meters) or "MOVING" (moved by more than 10 meters).

4 Preparation of stream integration of location and schedule data

4.1 Data

The data used in the project comes from the Warsaw API open data platform `http://api.um.warszawa.pl` and contains information about location of trams and buses in Warsaw as well as timetables for all vehicles. The location data which serves as input for this algorithm is cleaned beforehand in a process described in the chapter "Preparation of data gathering mechanisms and preprocessing of transport data". The timetables for all the vehicles are downloaded separately once a day in the dedicated process (Timetablesdownloader) described below.

4.2 Timetables

The Warsaw API does not provide timetables for all lines and brigades directly in one place. The following information can be downloaded using the API: the list of all stops in Warsaw. Each stop is described by the following fields:

- *zespol* - identifier of a group of stops.
- *slupek* - post identifier.
- *nazwa_zespolu* - name of a group of stops.
- *id_ulicy* - street identifier.
- *szer_geo* - GPS latitude.
- *dlug_geo* - GPS longitude.
- *kierunek* - route direction.
- *obowiazuje_od* - date from which the stop is valid.

List of all lines that pass through certain stop. Timetables for specific stops and lines that stop at them described by the following fields:

- *czas* - precise time at which a vehicle should stop at given stop.
- *trasa* - name of the route.
- *kierunek* - direction in which a vehicle is moving.
- *brygada* - vehicle's brigade number.

The algorithm integrating location data with timetables requires full timetables per line-brigade with precise hours of stopping at each stop on every route. A dedicated process (application Timetablesdownloader) was created to provide timetables in such form using information available at Warsaw API. The process works in the following way:

1. Information about all stops in Warsaw is downloaded.
2. For each stop all lines passing through the stop are downloaded.
3. For each stop-line combination information about all brigades and precise hours is downloaded.

4. The data is transformed in order to obtain full timetables per line and brigade.

The example timetable after all transformations looks as follows:
For each line-brigade combination we have the list of all stops within the timetable with their positions, identifiers and information about direction of the route.

4.3 Integration with timetables

The cleaned data about location of vehicles is integrated in real time with timetables. The dedicated process based on Apache Flink Streaming API is used for integration vehicles location data with timetables. For each vehicle the following information is computed regularly:

- Precise route of the vehicle (understood as a series of stops passed)
- Direction in which the vehicle is moving
- Vehicle's delay

Algorithm

1. At the beginning of every day current timetables are loaded into memory from the file system.
2. When the first location of a vehicle on given day is downloaded we check if the time reported by the vehicle is within time boundaries of the first timetable. If so, we check which stop on its route is the closest to vehicle's current GPS position and since that stop we start tracking vehicle's route on given day. All the information computed for every vehicle is put in the auxiliary map.
3. If the time reported by the vehicle as the first one on given day is not within time boundaries of the first timetable, there are two possible options. If it is before the time of starting the first timetable (vehicle started reporting its location before starting the first route), we do not assign any timetable to the vehicle and do not put anything in the auxiliary map. Instead we wait for a location report within time boundaries of the first timetable. If it is after the first timetable finish time, we assign a timetable according to current time but mark it as unsafe, put this information in the auxiliary map and verify our assumption when the next location data arrives using the algorithm described later.
4. If given location report is not the first one for a certain vehicle on given day and we already have vehicle's previous position in the auxiliary map, there are several possible options:
   (a) If an entry in the map for this vehicle is marked as unsafe we verify this timetable using the algorithm described below in point Timetables verification.
   (b) If the time between receiving two subsequent GPS positions of the same vehicle is greater than 60 seconds we apply the algorithm of handling lacks in point Handling lacks in data
   (c) If an entry in the map for this vehicle was marked as safe and there was no lack in data, we compute all the information in a standard way.

The process of assigning timetables to vehicles assumes that the auxiliary map is cleared at night when the vehicles are not moving.

Handling lacks in data

The Warsaw API open data platform provides the new information about location of the vehicles approximately every 30 seconds or even more often (around 20 seconds). If there is no information about location of a vehicle for more than 60 seconds we consider such situation erroneous and apply a special algorithm of handling lacks in data. We assume that a vehicles are not more than 5 minutes ahead of their timetable. If the time passed between receiving two subsequent GPS locations of a vehicle is over 60 seconds but less than the time left till the end of vehicle's route minus 5 minutes, we assume that the vehicle is still running according to its previously assigned timetable and such assumption can be considered safe. If, on the other hand, the lack in data occurred somewhere around the time when the vehicle should be finishing its route, we assign a timetable based on current time or the previous timetable if the vehicle should be in the depot at that moment. Such timetable is then marked as unsafe and is being verified when the next record arrives.

If the time passed between receiving two subsequent positions of given vehicle is greater than 1 hour (this can be parametrized) we do not assign any timetable for this vehicle till the end of the day as we cannot say according to which timetable the vehicle is moving at the moment and we will not be able to verify it on that day. The timetable status is then set to missing. In practice, such situation practically never occurs.

Timetable verification

If the next GPS location of a vehicle indicates that it has moved towards the next stop according to the assumed timetable, it means that our assumption was correct and such timetable can be marked as safe. If the next GPS location indicates that the vehicle has moved towards the previous stop according to the assumed timetable, it means that our assumption was wrong and the previous timetable has to be assigned as the vehicle is late. If the next GPS location does not allow us to make any judgment (the vehicle has not moved close enough to any of the stops), we continue to mark such timetable as unsafe and attempt to verify it when the next record arrives.

5 Stream-based gathering of raw and preprocessed tram data integrated with tram schedules

5.1 Description

The aim of this part of the project is to gather raw and pre-processed data integrated with tram schedules. The dedicated flink-trams-analyzer process is used for linking location data with timetables.

5.2 Data

Raw data:

The information about location of trams is downloaded from the Warsaw API open data platform `http://api.um.warszawa.pl`. The new data appears every thirty seconds. It is available in JSON format and contains the following information for each tram:

- *FirstLine* - string - first line number
- *Brigade* - string - brigade number
- *Time* - datetime - record timestamp
- *Lat* - float - latitude GPS
- *Lon* - float - longitude GPS
- *Lines* - string - the numbers of all lines (for multiline brigades there will be more than one line)
- *Status* - string - determines whether a tram is "RUNNING" or the ride is "FINISHED"
- *LowFloor* - bool - determines whether the tram is low-floor (1 - yes, 0 - no)

At the beginning of the processing an additional column to raw data is added:

- *ReceivedTime* - VM time of getting the data from API

Clean data

The deduplicated raw data with denoisified location. The same data as above plus:

- *Lat* - float - latitude GPS
- *Lon* - float - longitude GPS Data with movement status

The data with movement status The same data as above plus:

- *TramStatus* - string - determines whether a tram is "STOPPED" (did not move by at least 1 meter since the last record), "MOVING_SLOWLY" (moved by 1-10 meters) or "MOVING" (moved by more than 10 meters).

Data integrated with timetables

- *ProcessingFinishedTime* - cluster server time of writing the data to HDFS
- *delayedBy* - the difference in seconds between planned time of arrival at a particular stop and actual arrival time, based on brigade+line+iteration matching, where iteration is the iteration of passing the route. Checked when a vehicle is less than 30 meters away from a stop. If the distance is greater than 30 meters, delay at the previous stop is displayed.
- *nearestStop* - the closest stop to the vehicle (stop is selected from the list of stops assigned to the current timetable in geographical terms, Euclidean distance). Related fields:
  - *nearestStopName* - the name of the nearest stop.
  - *nearestStopDistance* - the Euclidean distance between the vehicle and the nearest stop.
  - *nearestStopLon* - the longitude of the nearest stop.
  - *nearestStopLat* - the latitude of the nearest stop.

- *previousStop* - the recently visited stop (stop is selected from the list of stops assigned to the current timetable). Related fields:
  - *previousStopName* - the name of the previous stop.
  - *previousStopDistance* - the distance between the vehicle and the previous stop.
  - *previousStopLon* - the longitude of the previous stop.
  - *previousStopLat* - the latitude of the previous stop.
  - *previousStopLeaveTime* - the time when the vehicle leaved the last visited stop (checked when a vehicle is less than 30 meters away from the stop). This time is the last time of observing the vehicle within the distance of no more than 30 meters from the stop.
  - *previousStopArrivalTime* - the time when the vehicle arrived at the stop (checked when a vehicle is less than 30 meters away from the stop). This time is the first time of observing the vehicle within the distance of no more than 30 meters from the stop.
  - *plannedLeaveTime* - the leave time as planned in the schedule, a reference time based on which delayedBy time is calculated.
  - *delayAtStop* - the name of the stop, delayedBy refers to.
- *nextStop* - the next stop on the current route (stop is selected from the list of stops assigned to the current timetable). Related fields:
  - *nextStopName* - the name of the next stop.
  - *nextStopLon* - the longitude of the next stop.
  - *nextStopLat* - the latitude of the next stop.
  - *nextStopDistance* - the Euclidean distance between the vehicle and the next stop.
  - *nextStopTimetableVisitTime* - the planned time of arrival at the next stop.
- *courseDirection* - the course direction stop name (the last stop from the timetable).
- *timetableIdentifier* - the identifier of the currently assigned timetable (start time and end time of a timetable e.g. 03:42:00-04:16:00).
- *timetableStatus*:
  - *"SAFE"* - the standard status set when we might be almost sure that we correctly assigned a timetable to the vehicle.
  - *"UNSAFE"* - the status indicating that assigned timetable and all computed information might be incorrect. Set in case of some lack in data at a crucial point, for instance close to vehicle's ride destination, when we cannot be sure whether the vehicle is still on its previous route or it already started the next ride. In case of "UNSAFE" status a timetable according to given time is assigned (or the previous timetable if no timetable matches given time) and correctness of this assignment is verified when the next record arrives. Details of this algorithm are explained in point 2.2.
  - *"MISSING"* - no timetable assigned.

Data gathering and storage

The data is gathered on a hadoop cluster (HDFS). The following streams are created from raw data:

- Deduplicated data (without duplicate events)
- Clean data (without duplicates and GPS fluctuations)
- Clean data with status MOVING/STOPPED
- Data integrated with timetables

The all data is stored in HDFS in directories listed in the table below:

Data type	Directory
Raw	/vavel/trams/data/raw
Deduplicated	/vavel/trams/data/deduplicated
Clean	/vavel/trams/data/clean
Clean with status	/vavel/trams/data/clean-with-status
Clean integrated with timetables	/vavel/trams/data/clean-with-timetables

Table 1: HDFS directories for trams data storage.

The data in all the folders listed above are grouped by day and stored in dedicated folders, as in the example below

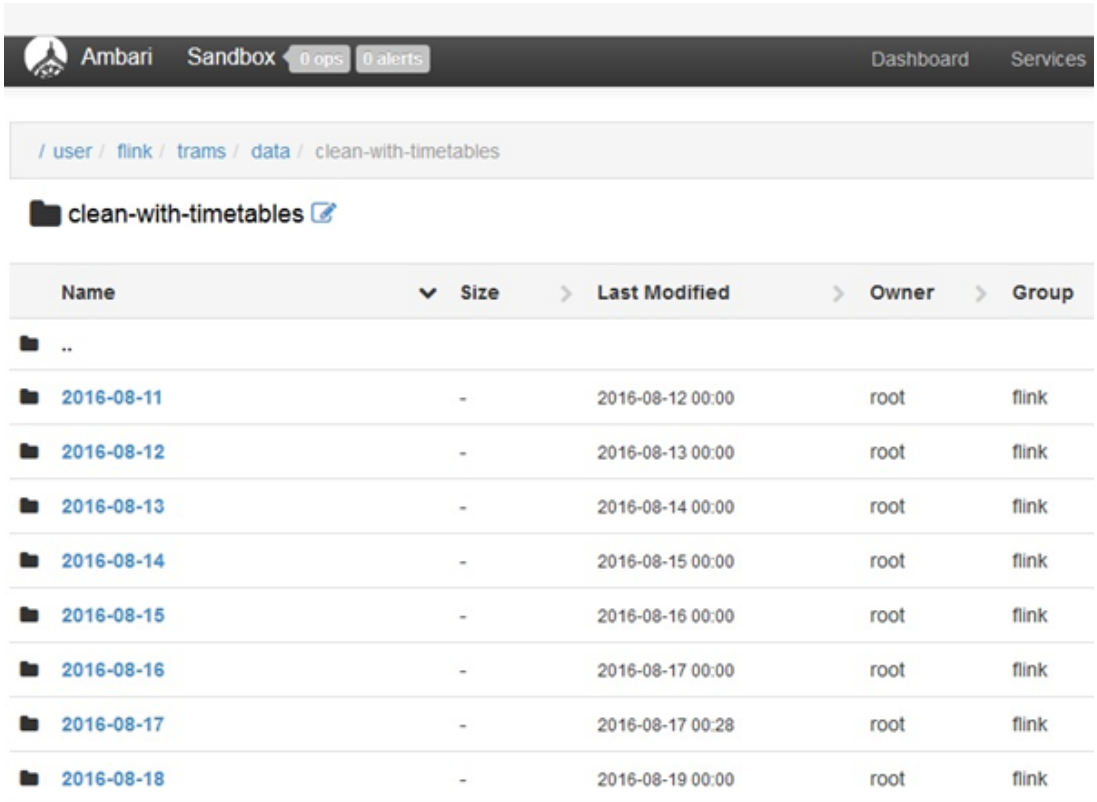


Figure 7: Data structure

6 Stream-based gathering of bus data integrated with the schedules

6.1 Description

The aim of this part of the project is to gather and store raw and preprocessed data integrated with the bus schedules in real time using Apache Flink stream processing.

6.2 Data

The information about location of buses is not public. It is made available by the Warsaw City Hall for the purposes of this project only. The data is available via a RESTlike Web Service and contains the following fields for each bus:

- *Line*: string - the first line number
- *Brigade* - string - the brigade number
- *Time* - datetime - the record timestamp
- *Lat* - float - latitude GPS
- *Lon* - float - longitude GPS

These data are referred to as "raw" in this document and form the basis of all other data types. At the beginning of the processing is added an extra column for raw data

- *ReceivedTime* - local PC time of getting the data from API

Clean data
The deduplicated raw data with denoisified location. The same data as above plus:

- *Lat* - float - latitude GPS
- *Lon* - float - longitude GPS Data with movement status
- *ProcessingFinishedTime* - server time of writing the data to HDFS

Data with movement status
The same data as above plus:

- *BusStatus* - string - determines whether a bus is "STOPPED" (did not move by at least 1 meter since the last record), "MOVING_SLOWLY" (moved by 1-10 meters) or "MOVING" (moved by more than 10 meters).

The data integrated with timetables

- *delayedBy* - the difference in seconds between the planned time of arrival at a particular stop and actual arrival time, based on brigade+line+iteration matching, where iteration is the iteration of passing the route. Checked when a vehicle is less than 30 meters away from a stop. If the distance is greater than 30 meters, delay at the previous stop is displayed.

- *nearestStop* - the closest stop to the vehicle (stop is selected from the list of stops assigned to the current timetable in geographical terms, Euclidean distance). Related fields:
  - *nearestStopName* - the name of the nearest stop.
  - *nearestStopDistance* - the Euclidean distance between the vehicle and the nearest stop.
  - *nearestStopLon* - the longitude of the nearest stop.
  - *nearestStopLat* - the latitude of the nearest stop.
- *previousStop* - the recently visited stop (stop is selected from the list of stops assigned to the current timetable). The related fields:
  - *previousStopName* - the name of the previous stop.
  - *previousStopDistance* - the distance between the vehicle and the previous stop.
  - *previousStopLon* - the longitude of the previous stop.
  - *previousStopLat* - the latitude of the previous stop.
  - *previousStopLeaveTime* - the time when the vehicle leaved the last visited stop (checked when a vehicle is less than 30 meters away from the stop). This time is the last time of observing the vehicle within the distance of no more than 30 meters from the stop.
  - *previousStopArrivalTime* - the time when the vehicle arrived at the stop (checked when a vehicle is less than 30 meters away from the stop). This time is the first time of observing the vehicle within the distance of no more than 30 meters from the stop.
  - *plannedLeaveTime* - the leave time as planned in the schedule, a reference time based on which delayedBy time is calculated.
  - *delayAtStop* - the name of the stop, delayedBy refers to.
- *nextStop* - the next stop on the current route (stop is selected from the list of stops assigned to the current timetable). Related fields:
  - *nextStopName* - the name of the next stop.
  - *nextStopLon* - the longitude of the next stop.
  - *nextStopLat* - the latitude of the next stop.
  - *nextStopDistance* - the Euclidean distance between the vehicle and the next stop.
  - *nextStopTimetableVisitTime* - the planned time of arrival at the next stop.
- *courseDirection* - the course direction stop name (the last stop from the timetable).
- *timetableIdentifier* - the identifier of the currently assigned timetable (start time and end time of a timetable e.g. 03:42:00-04:16:00).
- *timetableStatus*:
  - *"SAFE"* - the standard status set when we might be almost sure that we correctly assigned a timetable to the vehicle.
  - *"UNSAFE"* - the status indicating that assigned timetable and all computed information might be incorrect. Set in case of some lack in data at a crucial point, for instance close to vehicle's ride destination, when we cannot be sure whether the vehicle is still on its previous route or it already started the next ride. In case of "UNSAFE" status a timetable according to given time is assigned (or the previous

timetable if no timetable matches given time) and correctness of this assignment is verified when the next record arrives. Details of this algorithm are explained in point 2.2.
  - *"MISSING"* - no the timetable assigned.

6.3 Data Storage

The following streams are created from the raw data:

- The deduplicated data (without duplicate events for every bus)
- The clean data (without duplicates and GPS fluctuations)
- The clean data with statuses MOVING/STOPPED
- The clean data integrated with the timetables

All data is stored in HDFS in directories listed in the table below:

Data type	Directory
Raw	/vavel/buses/data/raw
Deduplicated	/vavel/buses/data/deduplicated
Clean	/vavel/buses/data/clean
Clean with status	/vavel/buses/data/clean-with-status
Clean integrated with timetables	/vavel/buses/data/clean-with-timetables

Table 2: HDFS directories for buses data storage.

The data in all the folders listed above are grouped by day.

7 Determination of average speed of vehicles during a day

7.1 Description

The goal of this part of the project is to create a mechanism that calculates the average speed of a tram at different hours as well as the average speed of the fastest and the slowest tram line at different hours. The speed is calculated based on subsequent GPS positions of every tram provided by Warsaw API open data platform.

7.2 Data

The data is downloaded from Warsaw API open data platform in a form of JSON array. The API provides new information about location of trams approximately every 30 seconds. For each tram the following data is provided:

- *FirstLine* - string - first line number

- *Brigade* - string - brigade number
- *Time* - datetime - record timestamp
- *Lat* - float - GPS latitude
- *Lon* - float - GPS longitude
- *Lines* - string - the numbers of all lines (for multiline brigades there will be more than one line)
- *Status* - string - determines whether a tram is "RUNNING" or the ride is "FINISHED"
- *LowFloor* - boolean - determines whether the tram is low-floor (1 - yes, 0 - no)

The data is pooled from Warsaw API open data platform every 15 seconds to make sure that no record is missed.

7.3 Algorithm

The average speed of each tram in every hour is calculated using Apache Flink Stream Processing API. The data is aggregated in hour windows. The dedicated function counts total distance covered by a tram in every hour by summing distances covered between two subsequent GPS positions. The distance between two GPS positions is calculated using the haversine formula. The average speed in given hour is then calculated as the quotient of total distance covered during that hour and time passed between the first and last record for this tram in given hour.

7.4 Results

In this chapter are presented results of trams data analysis - average speed for the fastest and the slowest tram lines in Warsaw.

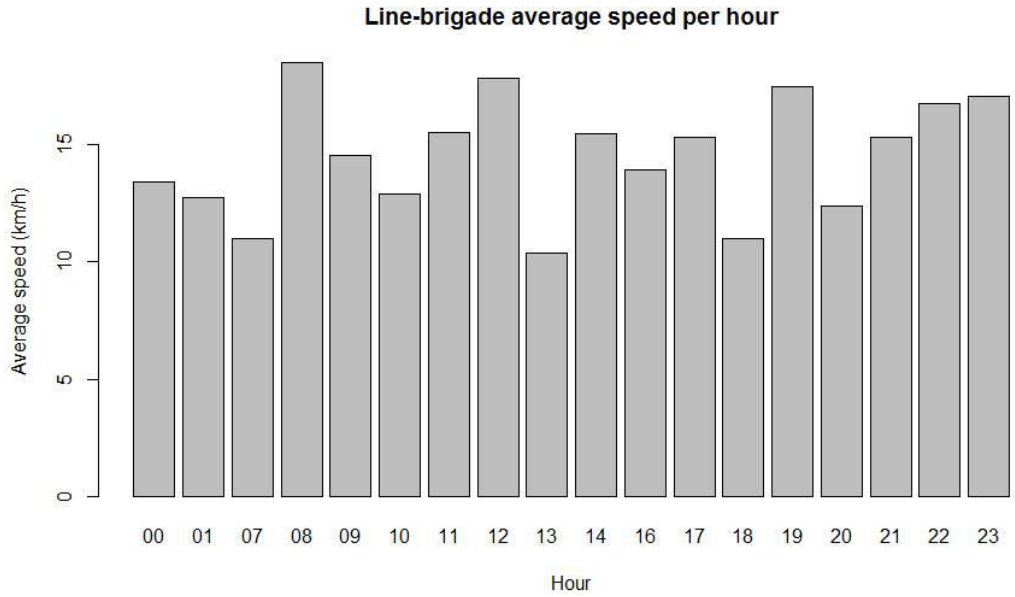


Figure 8: Average speed in every hour of the fastest line in km/h

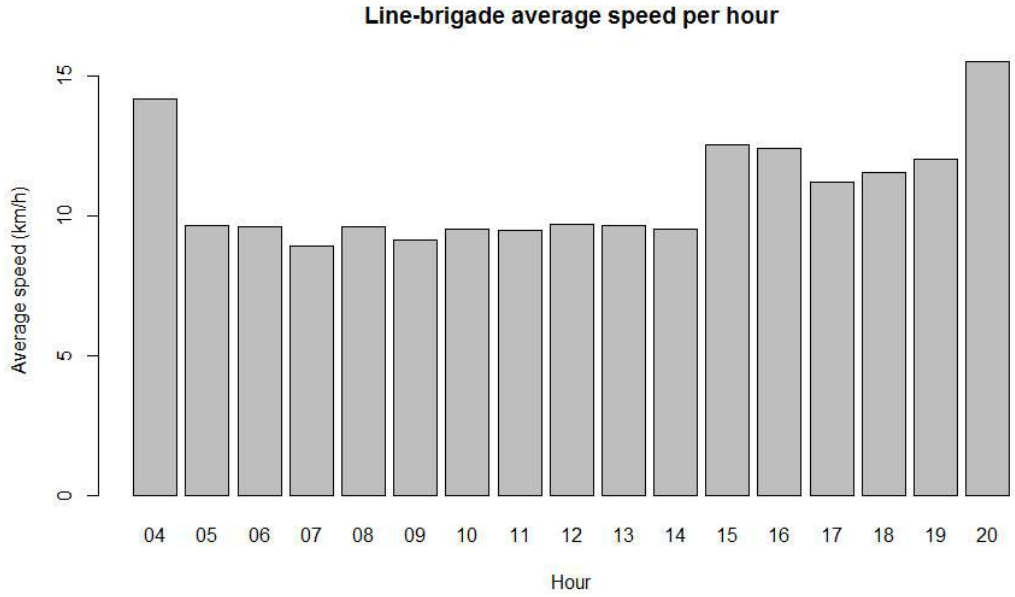


Figure 9: Average speed in every hour of the slowest line in km/h

8 Determination of ratio of moving and stopped vehicles

8.1 Description

The aim of this part of the project is to create a mechanism that will allow to determine whether a tram is moving or not. A status of movement is calculated when new data about a tram appear

8.2 Data

Th einformation about location of trams is downloaded from the Warsaw API open data platform `http://api.um.warszawa.pl`. The new data appears every thirty seconds. It is available in JSON format and contains the following information for each trams

- *FirstLine* - string - first line number
- *Brigade* - string - brigade number
- *Time* - datetime - record timestamp
- *Lat* - float - latitude GPS
- *Lon* - float - longitude GPS
- *Lines* - string - the numbers of all lines (for multiline brigades there will be more than one line)
- *Status* - string - determines whether a tram is "RUNNING" or the ride is "FINISHED"
- *LowFloor* - bool - determines whether the tram is low-floor (1 - yes, 0 - no)

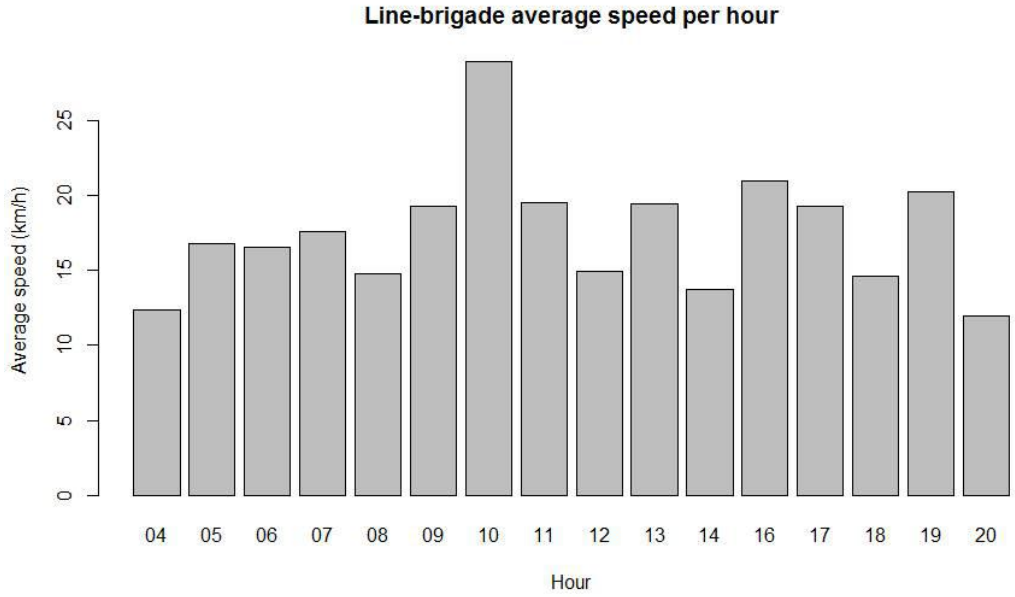


Figure 10: The average speed of a tram in every hour in km/h

8.3 Tram movement status

Defining tram movement status based on changes in its location is performed by Apache Flink in a streaming mode. This is a part of the project related to transport data analysis.

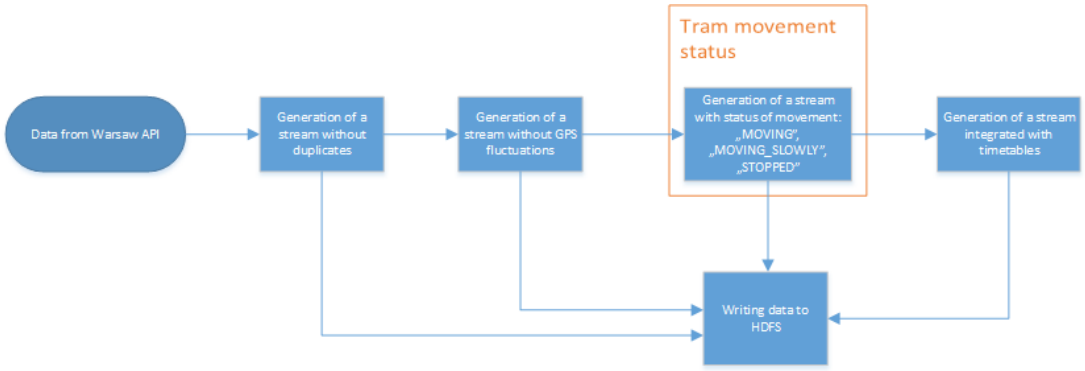


Figure 11: Tram movement status

After data cleaning (de-duplication and de-noising) a tram movement status is determined based on a distance between two measurements of tram location. Every time new data appears with information about the vehicles, the following steps are performed for each record:

- A record about a tram (identified by a line-brigade key) which appears in a given day for the first time is added to an auxiliary map with status "UNKNOWN"
- If the map contains information about this tram we calculate a distance between two measurements.
  - If a tram travelled less than 1 meter than a status is set to "STOPPED"

– If a distance is between 1 and 10 meters than a status is set to "MOVING_SLOWLY"
– Otherwise a status is set to "MOVING".

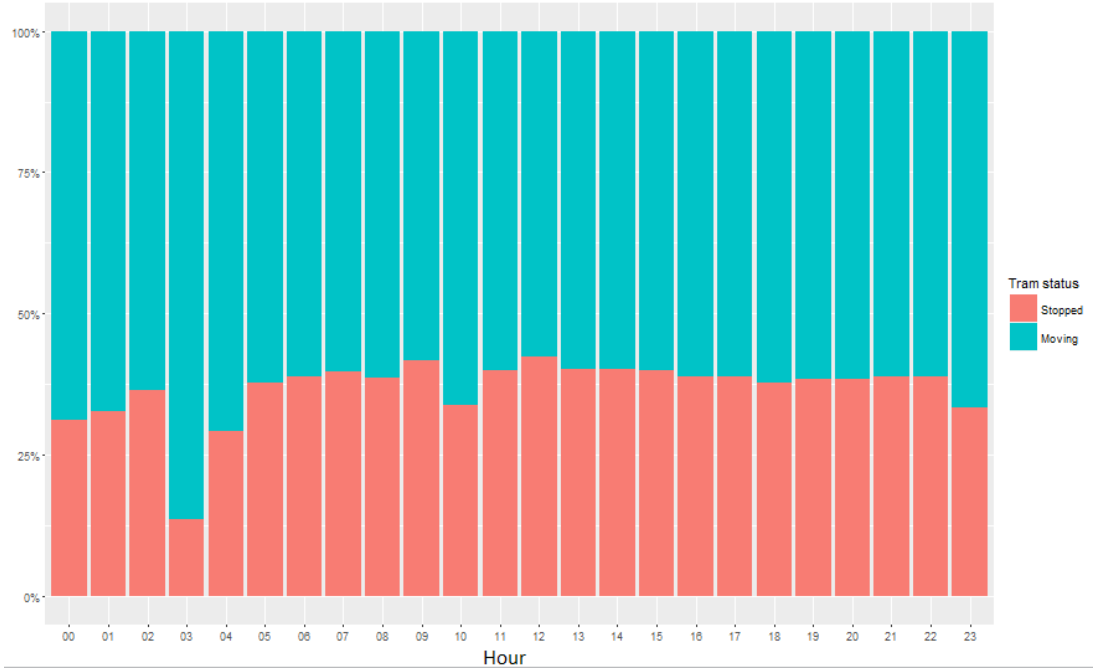


Figure 12: Tram status

The Figure12 shows the percentage of Moving ("MOVING" and "MOVING_SLOWLY") and Stopped ("STOPPED") statuses in every hour.

9 Collection of text data describing city events and produced by the City Hall

9.1 Overall architecture

The goal of this module development was to design and implement components required for fetching data from text-feeds defined in the requirements. The fetching process was split into two parts:

1. retrieving data directly from the feed sources and identifying external links - which is the responsibility of the implemented vavelfeeds python package (described in section VaVelFeeds - text-feed readers python package),
2. fetching the external links along with all their dependencies (i.e. links present on the fetched HTML pages) - which is handled by UriDownloader created using Spark Streaming (described in section UriDownloader - Spark Streaming link handler).

The way that these components interact with each other is depicted on the diagram below:

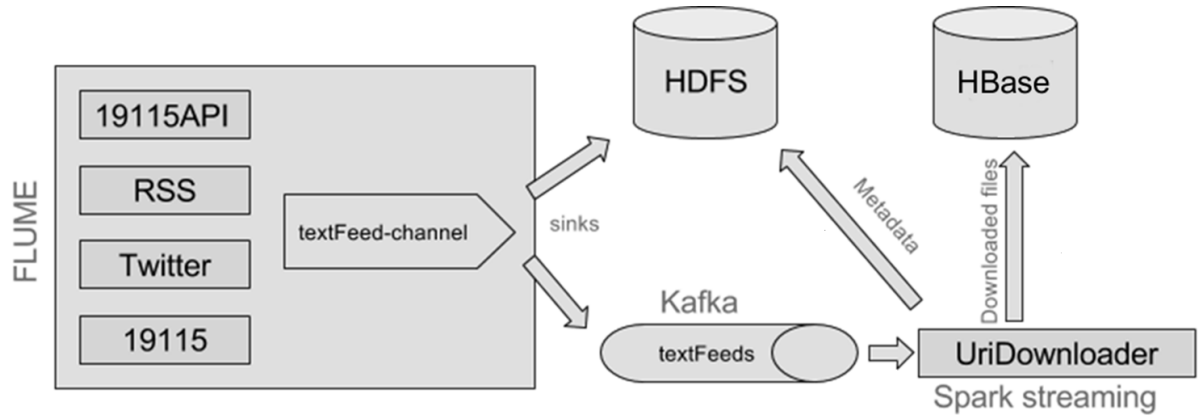


Figure 13: Text data fetching modules - architecture diagram

Flume is utilised to pass the output of the feed readers to both Kafka and HDFS sinks. Kafka's textFeeds channel provides a buffer for UriDownloader to prevent potential data loss. The fetched text-feed messages are passed in the following json format (with extra comments added for explanatory purposes):

```
{
 "feedType":"twitter",                          //which reader produced the message
 "feedSource":"ztm_warszawa",                   //message source
 "eventTimestamp":"2016-09-21 11:51:48+00:00", //when the message was fetched
 "links":["https://t.co/9BGxDtW0oJ"],           //ext links for UriDownloader
 "text":"Message text",                         //the main text extracted from the
                                                //message
 "rawData":{...}                                //raw message content
 }
```

9.2 VaVelFeeds - text-feed readers python package

VaVelFeeds python package provides a number of readers consuming text feeds and printing them out as json lines on stdout. The following readers have been implemented as parts of the task:

- 19115 - fetches posts from `http://warszawa19115.pl/glownawebsite`
- 19115API - integrates with `https://api.um.warszawa.pl`,
- RSS - follows the `http://www.ztm.waw.pl/rss.phpRSSfeed`,
- Twitter - fetches the messages from a dedicated Horyzont Twitter account.

The package is written in python 2.7 in order to be compatible with the environment on the hadoopp cluster used in the project.

9.3 Flume integration

The readers provided by vavelfeeds are to be used as flume sources.In this configuration, text messages are retrieved by the appropriate vavelfeeds scripts and passed to Flume, which links

these sources to the Kafka sink. This makes the feed messages available in near-realtime through Kafka's textFeed-channel. To be precise, the latency is mostly influenced by polling frequencies of the individual text feed sources - these may of course be set very low to make the feed messages available as soon as possible, however certain restrictions regarding for example the usage of Twitter's API apply here. The package strives to provide exactly-once processing semantics on a best-effort basis (without dependencies on external systems). Because of this, the history of recently used messages is stored on disk for each reader. This allows you to reject possibly repeated messages early in the processing pipeline. This approach, however, is not bulletproof - too small history storege or deletion of the contents of these files can cause duplicate messages to be processed. On the other hand, disabling the channel readers for a long time will result in the loss of the message.

Apart from the above, certain quirks also exist for each of the sources:

- 19115 - fetches data from posts placed on `http://warszawa19115.pl/glowna` - these are updated quite infrequently - the period between consecutive messages is usually counted in days, so the polling period for this reader defaults to a larger value than in the other cases,
- 19115API - deals with the source which enables fetching notifications from a given time period - unfortunately if the number of available notifications is too large - the returned list is truncated. For this purpose a dynamic fetching strategy has been developed which divides the requested period in smaller pieces if the number of notification is greater than 90,
- RSS - the `http://www.ztm.waw.pl/rss.php` RSS feed contains no timestamps connected with the provided posts, therefore a hashing algorithm had to be used to prevent duplicates.
- Twitter - the reader is restricted by connection limits imposed by Twitter's API - when the server responds that the rate limit has been exceeded the reader needs to sleep for 15 minutes before making another connection attempt.

9.4 UriDownloader - Spark Streaming link handler

A Spark streaming job called UriDownloader is responsible for downloading links extracted from text-feed messages. As depicted in Figure 13, this job consumes JSON messages from Kafka, processes them and saves the results to external storage. Processing a single message proceeds as follows. First the URIs from the message's links field are extracted. Each of them is then downloaded - if they contain HTML, further links are extracted from their content and also downloaded (the link extraction process is NOT executed for these '2nd level' links). Each downloaded link is stored under in a HBase database. The row id is generated as a hash of the file's content - this approach helps to avoid the problem of storing and multiple copies of the same binary content. As having too many files stored in the same directory may pose problems, a nested directory structure is utilized in which for example web page id: aafeb5e78a9c00e0368ace316b19c6049e8380bd gets stored using row id: aaf/eb5/aafeb5e78a9c00e0368ace316b19c6049e8380bd Apart from the downloaded content, a record containing metadata is stored in HDFS in CSV format. The sample of the record is given below:

```
2016-09-02T11:04:03.195+02:00,http://localhost:8089/foo.html,TEXT,
88a/383/88a383fa691,88a383fa691
```

where the consecutive columns are:

- download timestamp,
- URI,
- file type (BINARY or TEXT),
- file path,
- file content checksum (calculated using sha1).

983/5000 Because the same URIs appear in multiple channel messages at different times, it is necessary to handle potentially duplicate content. Storing each downloaded link using a separate write identifier leads to wasted disk space, so the mixed approach (described above) is used to prevent such problems. Although the content will not be stored many times with this solution, it still needs to be downloaded many times, potentially wasting bandwidth. It would be naive to download the contents of each URI once, but this idea falls for links containing content updates - for example, news services. Another approach that can reduce this problem is to use ETAG HTTP to detect whether the content has been altered or not. Unfortunately, this feature is not available for all potential external links.

Feeding sources currently used in the project generate rather a small number of messages. Because the project assumes that these messages should be processed with a small latency, these sources are often searched and the messages sent are very small (in HDFS standards). Because these batches are stored in the HDFS file system, this leads to the generation of large numbers of small files, which is not optimal for HDFS performance. This problem should be solved to prevent potential scalability issues.

10 Development of server backend for mobile application

10.1 Description

The aim of this project is to create REST and websocket back-end service. This report will describe design deccisions made in the course of application prototype development: services, frameworks and libraries used to build project components:

- REST API backend service
- Real-time websocket server
- Open Trip Planner
- Asynchroneous task queue
- Push notifications service

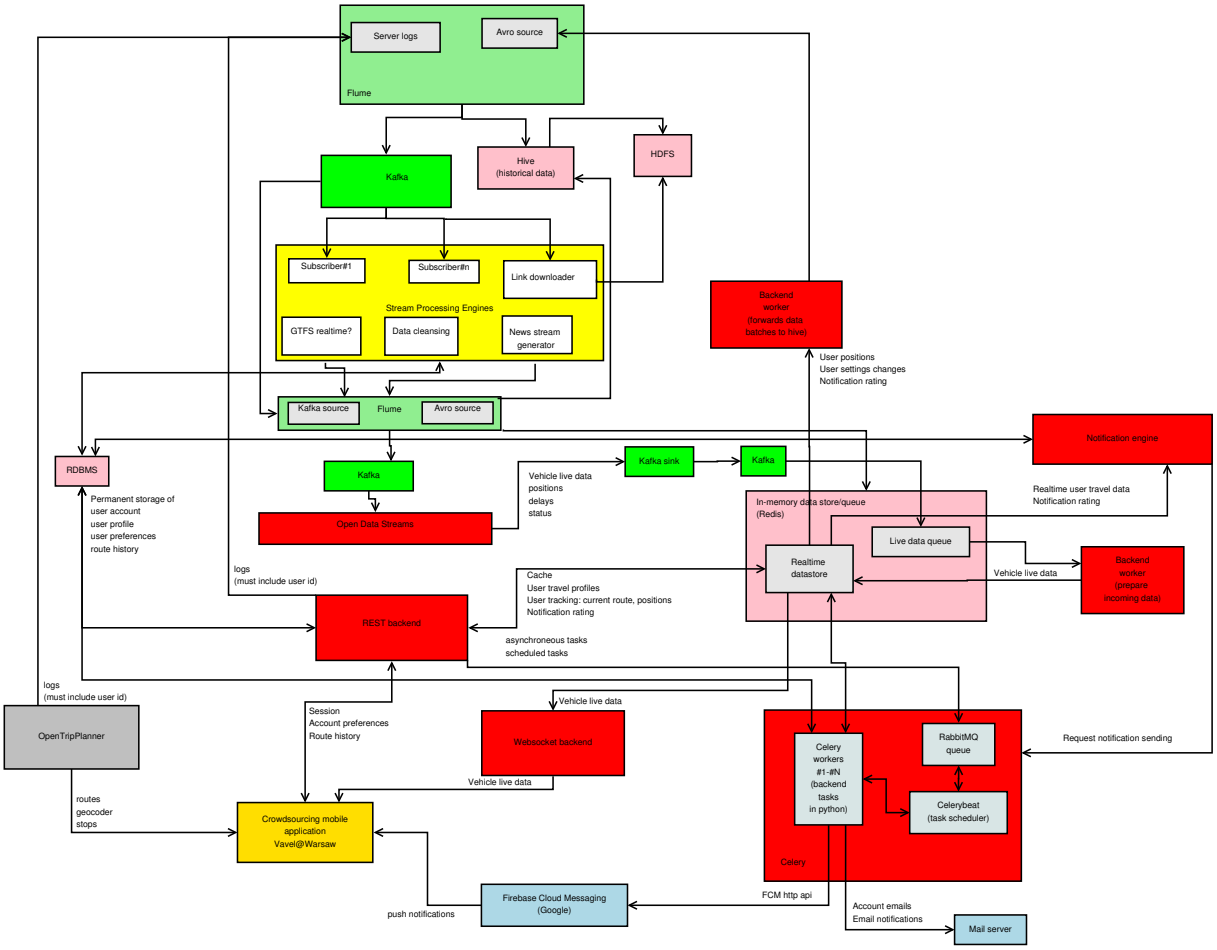


Figure 14: Scheme of CoW detailed architecture of mobile part

10.2 REST API backend service

Django was chosen as the framework to create the main backend component which is the REST API service.

The latest python 3.5.2 was chosen as it is already production-ready. During the application development there were minor issues with selected python libraries, however overall it has proved to be a good choice. Precise library descriptions and versions will be described in deployment folder included in the application repository.

10.3 Real-time websocket service

To provide a vehicle positions and delays in a real-time manner appropriate transport mechanism must be adopted. The continuous HTTP requests or long polling technique are not enough to deliver time-sensitive data to the clients, moreover those solutions require significant resources and should be treated as a last-resort and/or fallback solution for clients that do not support such transmission method.

Websocket is a widely adopted standard for such purpose. It must be noted that though this standard has existed for some time now and even despite high demand for such solution it

is still a volatile method emulated on top of regular HTTP transmission. Regular HTTP or HTTPS session is established and an Upgrade header is sent to request a websocket session. Most proxies do not handle Upgrade header properly, likely stripping. Therefore it is important to use HTTPS (WSS) connection which is terminated by a websocket server on the backend side.

Popular reverse-proxy just as Nginx does easily support such solution. Other services such as Apache can support websockets as well, although they will require some attention and/or additional plugins. Since the target deployment relies heavily on Apache webserver, such capability must be researched. This may require upgrading Apache to more recent versions and/or installing plugins. A separate instance of Apache (or Nginx) for hosting websocket server alone could be considered.

Websockets are perfect for sending real-time information, but they do not provide a delivery guarantee nor sessions stability. The API must be designed in a way that allows easy recovery in case of transmission interruption.

Websocket solutions

There is a wide variety of websocket-capable engines in Python. A number of libraries was considered for this purpose. Some of them were rejected based on limited functionality (providing just the transport, but no application logic):

- ws4py
- aiohttp

There is a wide range of solutions that provide asynchronous web servers:

- twisted - network programming framework, a little low level for a desired solution,
- TwistedWeb - solution based on twisted
- tornado - popular websocket capable python server, a decent choice for this kind of application,
- wide selection of django + tornado or twisted - solutions that often require sacrificing some django solutions due to incompatibility with django'a synchroneous approach, often lack good reach and/or are backed by little or no team,
- autobahn.ws - provides WAMP protocol implementation, provides easy RPC and publish-subscribe solutions
- crossbar.io - networking platform built on top of autobahn.ws
- django-channels - very recent project, but well-coupled with the django webserver and plans on full integration with django in near future.

10.4 Prototype #1: crossbar.io

Crossbar was selected as the first test solution. Underlying Autobahn server library guarantees great scalability. WAMP2 implementation would cover message delivery, leaving only message handling logic to be implemented.

Unfortunately, at the moment there is no feasible WAMP2 implementation for android. Current workaround involves including a web server library in the application, which is problematic for several reasons. WAMP2 implementation in c was considered but assessed as a similarly problematic.

Creating WAMP2 protocol implementation would be too much effort compared to how simple websocket application is required. Simple tornado application would be more feasible in this case.

Therefore, crossbar.io was rejected for a websocket server solution.

10.5 Prototype #1: django-channels

The django-channels project was created about a year ago. Currently in beta, but the API seems well designed and integrates with Django framework very nicely allowing reuse of django ORM and session engines.

Due to high demand for such solution and given involvement of a Django core developer it is very probable this solution will become a popular solution for realtime communication for django web applications. The API has proven friendly and easily integrates with the remaining backend modules.

Solution was created based on version 0.17.2 of django-channels.

10.6 Open Trip Planner

A number of existing OTP applications was investigated to asses good practices used in the field. Research was done keeping in mind both mobile and future web (javascript) applications.

There is a number of components to assess for such applications:

- route search
- geocoder
- station (stops) data
- map servers

The early assumptions suggested involving backend server in OTP queries, including geocoder and proxying search queries to the server. However after some research a decision was made to rely on OTP's own facilities for that purpose.

The route search is done by contacting the server directly. Referencing individual queries to a user/device would be by sending an extra tracking token in each search query. Currently no attempt is made to enforce search queries authentication that would be based on token received via user login. Sensitive OTP API methods are protected by default by requiring https connection and basic auth password, which should be only setup if required and certainly not for the end users. See more `http://docs.opentripplanner.org/en/latest/Security/`

In order to perform any route queries, user input location must be converted to latitude and longitude coordinates. This task should be performed by a geocoder service. Currently we are using default OTP geocoder that is provided with the instance, however that is a basic solution providing only static map data. While sufficient for a prototype, further investigation should be done on this topic to (if possible) include more context to the map. Existing planner instances that were investigated often used a geocoder service written in different web technologies (suggested even by the endpoint names like .asp, .php), therefore designing own geocoder service may be a task for future development.

Geocoder service could be created using a search index such as Apache Solr or elasticsearch. However due to the limited dataset of map locations inside the city of Warsaw and surroundings

which is characterized by rather rare updates of the source data, a postgresql full text search solution could be used. It is already a mature solution that could be just sufficient for our purpose, while bringing relational database benefits when preparing the data. Stations (stops) data if queried as a separate request, the response is handled by OTP.

Consider also that data returned by: routes, geocoder, stops endpoint is that we must be ensure there are no discrepancies between those sources (location names and positions). Therefore, one initial data source would be mandatory. While investigation this issue, it was revealed that a process for continuous data re-generations must be researched as a part of the VaVel project.

Another important topic to consider is creating own map server. Just recently a popular provider has removed free developer licenses that were used by many example on-line application and free plugins.

For the purpose of mobile application, Google maps can be used according to their license even if the application is password-protected. Therefore Google maps are used at the moment in the prototype application.

However the ZTM application module would be password protected web application which violated the API license. Therefore an application based on OTP upcoming javascript libraries or directly - leaflet.js must be created. For such application a map server would be required. From initial research it is possible to run such server for the area in scope of the project (Warsaw and surroundings) without significant computation expense. There is a number of open-source solutions available. This topic should be investigated further as a prerequisite of ZTM web application.

10.7 Asynchroneous task queue

The Celery is a scalable and feature rich task queue for python with good django integration. It is used used to perform any task that should be done outside of django web request, such as long-running tasks or tasks involving remove services such as email or notification sending.

The typical celery setup consists of a queue service, tasks broker and groups of workers. The setup may be extended by adding results storage (ex. redis) and dedicated monitoring tools.

RabbitMQ is a superior solution for the the task queue, other services mentioned are include in celery package.

- Tasks were created for:
- simple vehicle movement simulation
- email sending
- push notification sending

10.8 Push notifications service

The push notifications must be sent using a dedicated Google service. Recently, there was some change in the API and all new applications are encouraged to start using FCM (firebase Cloud Messaging) in favor of GCM (Google Cloud Messaging).

There is a slight difference in how the API operate, but still there isn't much more than sending a proper payload to certain API endpoint.

The process is as follows:

1. mobile device registers for notifications from certaing app receiving a token
2. token is passed to the app backend
3. app backend sends notifications using own api key and recipients token
4. application can de-register at any time, while tokens are removed at logout

There were large delays reported at first (15-30 minutes), but the problem seems to have disappeared and thus we are attributing it to handling of fresh/small applications. Due to small overall number of events observed, this may require further investigation during development of the notification module.nt of the notification module.

11 Hackathons

The created infrastructure will be tested during hackathons. This section presents our plans according to the hackathons.

11.1 Overview

The hackaton will be held on Kaggle as Research Competition (`https://www.kaggle.com/host/research`) or similar platform. This will increase the range of the hackaton.

The final task will be formulated during workshops that will precede the hackaton. An example task is to predict what time trams will arrive on tram stops in a given day. In the hackaton users can make use of a static data dump or can use the live API (up to them, but using API should give them more data about history).

Workshops will be held in two cities, in Warsaw (a day before R users conference in September) and in Athens.

11.2 Goals

G.1. International audience, (participants from at least two European countries, the more the better)

G.2. Many participants that actively access VaVeL API (no KPI, the more the better)

Secondary goals:

S.1. Brainstorming of ideas how to use VaVel API in innovative data-based solutions (smart city), (no KPI)

S.2. Generation of Use Cases - success stories of data based solutions that make use of VaVel data (at least one, the more the better).

11.3 Phases

In order to meet G and S goals, the following activities will be performed

A.1. Organization of an one/two days open hands-on workshop in Warsaw, Poland. 30 percent of the workshop will be devoted to big-data technologies like hadoop/hive/similar techniques. 30 percent will be devoted to learning of VaVel API, exploration and understanding of the VaVel data. 40 percent of time will be devoted to a creation of a data-based solution that makes use of VaVel API. Expected number of participants 30 60 (two lecturers). Materials will be prepared in English. A draft version of web page with a proposed agenda is published on http://whyr.pl/hackathon/.

A.2. Organization of two-days open workshops in Athens, Greece. Similar structure as the Warsaw workshop.

A.3. Kaggle competition for the VaVel data. Price for the winner $1000–$2000

Competition is based on a sample of a data - flat file. Testing data will be published by created API.

11.4 Date and place

Ad A.1. Since the Polish R users conference is going to be held in September 2017 in Warsaw, the workshop will be just before the conference at26.08.2017.

Ad A.2. Materials (slides, ideas) from A.1. and A.2. may be shared. This workshop may be organized in the second part of the year 2017.

Ad A.3. Kaggle expects that a very well defined problem is going to be considered. We plan to run the Kaggle competition in January-March of 2018. The problem that will be submitted to Kaggle may be defined during the Warsaw/Athens workshops.

11.5 Other notes

Workshop materials should contain:

- materials that introduce to hadoop/hive technologies
- materials that introduce to VaVeL API
- examples of simple things that can be done with VaVeL API

Call for participants for a workshop should start III-VI of 2017.

We shall consider small fee in order to limit number of people that register but not arrive to the workshop. The total fee may be spent as a prize for the best solution.

The structure of the data and some examples should be available at least a month prior to workshop. To make people aware what we are going to work with.

At the end of the workshop-hackaton each team will present their results.

12 Summary

This document describes the prototype VaVeL platform installed in the City of Warsaw. This report covers the design and implementation of VaVeL framework components tailored for the CoW use case, in particular customized components for Apache Flume and Apache Flink, that are designed to process live streams of data generated by various entities of the CoW including, but not limited to Public Transport Authority and Citizen Emergency Services. Despite purely stream-oriented processing in its batch layer the VaVeL platform relies on Apache Spark, therefore the components developed for this tool are also described within the Report. Moreover mobile application and it's backend are described in details providing deep technical understanding on the technology being used for communication purposes, the data being processed and the way they are exposed and managed by the Mobile Application Backend.

This report do not covers the details on the system architecture as the prototype architecture outline was developed as a part of Work Package 6 and documented in D6.1 Report on Requirements and Architecture. Moreover, the details on data flows and data dependencies within VaVeL platform are described in D6.2- First Report on System Integration, also being a part of Work Package 6.

References

[1] M. Grover, T. Malaska, J. Seidman and G. Shapira, "Hadoop Application Architectures", O'Reilly, Sebastopol, CA, United States, 2015

[2] T. White, "Hadoop: The Definitive Guide.", O'Reilly, Sebastopol, CA, United States, 2015

[3] Apache Flume project, http://flume.apache.org

[4] Apache Kafka project, http://kafka.apache.org

[5] Apache Flink project, http://flink.apache.org

[6] Apache Spark project, http://spark.apache.org