GRANT AGREEMENT No 609035
FP7-SMARTCITIES-2013

Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications

Collaborative Project

D5.3 Smart City Environment User Interfaces

Document Ref.	D5.3
Document Type	Report
Work package	WP5
Lead Contractor	SIE
Author(s)	Dan Puiu (SIE), Daniel Kümper (UOAS), Marten Fischer (UOAS), Sefki Kolozali (UNIS), Nazli Farajidavar (UNIS), Feng Gao (UNIS), Thorben Iggena (UOAS), Thu-Le Pham (NUIG), Daniel Puschmann (UNIS), Joao Fernandes (AI), Bogdan Serbanescu (SIE), Cosmin Marin (SIE)
Contributing Partners	ALL
Planned Delivery Date	M30
Actual Delivery Date	M31
Dissemination Level	Public
Status	Delivered.
Version	1.0
Reviewed by	João Fernandes (AI) and Stefan Bischof (SAGO)

Executive Summary

The CityPulse project proposes a framework for large-scale data analytics to provide information in (near-)real-time, transform raw data into actionable information, and to enable creating “up-to-date” smart city applications. The CityPulse framework components can be divided in three main categories as follows: large scale data stream processing modules; reliable information processing modules; and adaptive decision support modules. The components which are generic and reusable in different application domains, are provided as open-source (https://github.com/CityPulse).
This document presents the results of the activity *A5.3 Visualization in Dynamic Smart City Environments.* The report contains the description of the CityPulse framework user interfaces and user manual (including API description) of each component.

Contents

1. Introduction

The CityPulse project aims to design, develop and test a distributed framework for semantic discovery, processing and interpretation of large-scale real-time Internet of Things and relevant social data streams for knowledge extraction in a city environment.

The CityPulse WP5 intends to support the development of a framework, which enables user-centric, dynamicity-aware and adaptive smart city applications, including support for visualization.

The visualization and interpretation of data, in particular of massive, dynamic and heterogeneous data, is a challenging topic. The visualization is strongly linked to the user interfaces. User interfaces have to address data manageability when dealing with large amounts of objects.

The *Task 5.3 Visualization in Dynamic Smart City Environments* aims to investigate how to visualise smart city data in end user applications coming from different sources. The CityPulse consortium has considered different devices, e.g. smart phones and tablets, together with the users (user driven methodology). The consortium has created prototypes of the functionalities and mechanisms that these applications should provide to the user. The user interface mock-ups served as prototypes for the demonstration of CityPulse applications to city stakeholders.

Furthermore, this activity has defined a common API the CityPulse framework that is exposed to external users. The API has a generic access layer that enables the use of the different functionalities provided by the framework, that can be used by 3rd party applications and services.

This document reports on the activity done within task 5.3 and it has two main components as follows:

- Visualization in dynamic smart cities environments: this chapter presents the various user interfaces of the CityPulse framework (CityPulse dashboard, CityPulse travel planner mobile application, CityPulse QoI monitoring, CityPulse components performance monitoring, CityPulse social media analysis, CityPulse 3D map);
- CityPulse framework API description: The chapter is dedicated to present the CityPulse framework APIs exposed to the 3rd party application developers.

A summarised description of the CityPulse framework components and APIs is presented in the paper entitled *CityPulse: Large Scale Data Analytics Framework for Smart Cities* [Puiu, D. et all 2016] published in the IEEE Access journal.

2. Visualization in Dynamic Smart Cities Environments

This chapter is dedicated to present the graphic user interfaces (GUI) of the CityPulse framework, as follows:

- CityPulse dashboard: supports visual analytics for the data sources registered in the framework;
- CityPulse travel planner mobile application: mobile application dedicated to the citizens, for obtaining user-centric travel and parking recommendations;
- CityPulse QoI explorer: GUI to monitor in real-time the quality and the trust of the data sources;
- CityPulse components performance monitoring: GUI which can be used to monitor in real-time the performance (processing times) of various CityPulse components;
- CityPulse social media analysis: a GUI which displays the events extracted from the social media (Twitter) streams;
- CityPulse 3D map: support 3D visual analytics for the data sources registered in the framework.

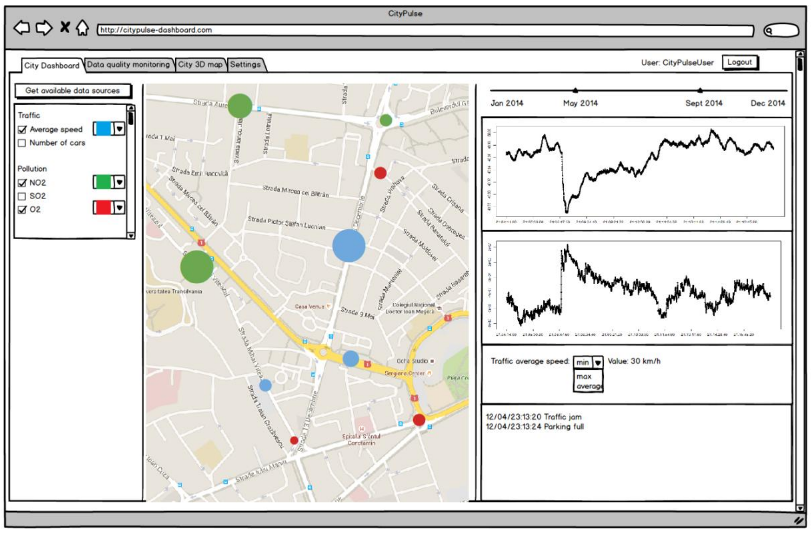
The main points presented for each graphic user interface are:

- the scope of the GUI;
- the technologies used for developing the GUI and the development process;
- the connection to the CityPulse framework (which components are interrogated, what is the format of the data, etc.).

2.1 CityPulse dashboard

The CityPulse dashboard provides immediate and intuitive visual access to the results of its intelligent processing and manipulation of data and events. The ability to record and store historical (cleaned and summarised) data for post-processing makes it possible to analyse the status of the city not only on the go but also at any point in time. In addition to these, the dashboard enables the diagnosing and “post mortem” analysis of any incidents or relevant situation that might have occurred. To facilitate that, a dashboard for visualising the dynamic data of smart cities is provided on top of the CityPulse framework. Based on this dashboard, the user has the possibility to visualise a holistic and summarised view of data across multiple contexts or a detailed view of data of interest, as well as to monitor the city life as it evolves and as things happens. The investigation of past city events or incidents can be conducted from different perspectives, e.g. by observing the correlations between various streams, since the streaming data is stored in the framework for a period of time which can be configured, and it can be retrieved for visualisation and analysis at any moment.

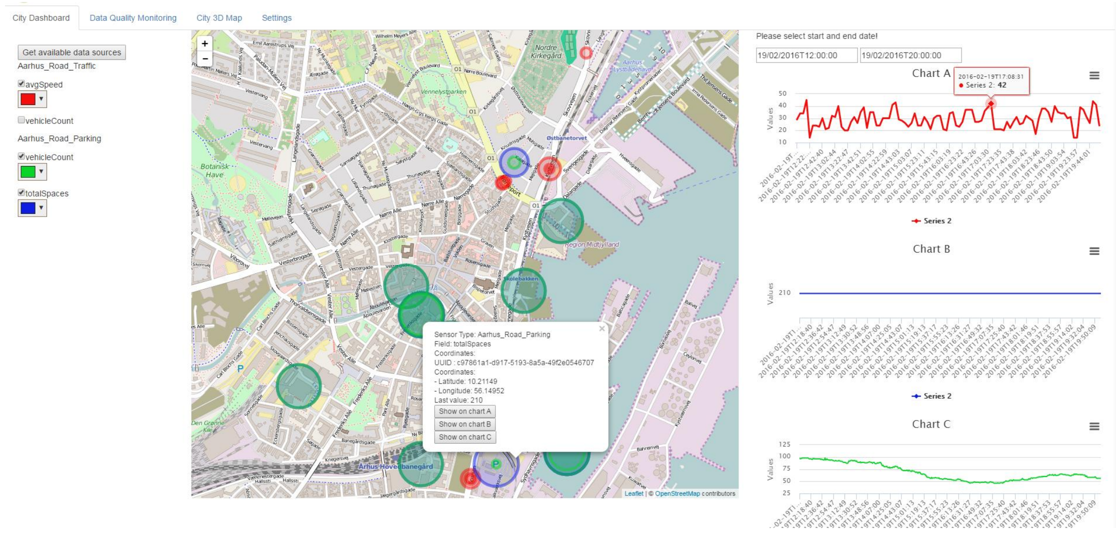
Before starting the development of the CityPulse dashboard, the application end-user requirements have been defined in collaboration with the 2 pilot cities. In that sense, two workshops have been organised in September 2015 with IT representatives from Brasov and Aarhus municipalities. A snapshoot of the agreed mock-up is presented in Figure 1. At he end resulted a list of specification for the CityPulse dashboard.



12/04/23:13:20 Traffic jam
12/04/23:13:24 Parking full

Figure 1 The mock-up of the CityPulse dashboard.

Figure 2 depicts a snapshot of the CityPulse dashboard application [Puiu, D. Et al. 2016]. Using the dashboard the user can select a certain area from the city for which he/she whant to perform visual analytics. The dashboard will display all the data sources from the selected area. After that the user is able to visualuse in real time the eveolution of the fenomenon, or can the evolution of the measuruements from a historic perspective.



Sensor Type: Aarhus_Road_Parking
Field: totalSpaces
Coordinates:
UUID : c97861a1-d917-5193-8a5a-49f2e0546707
Coordinates:
- Latitude: 10.21149
- Longitude: 56.14952
Last value: 210
Show on chart A
Show on chart B
Show on chart C

Figure 2 CityPulse dashboard.

In order to display the status of the city, the dashboard application connects directly to the to the Resource Management (see section 3.3) or to the Data Bus for fetching the description of the available streams or the real-time/historic observations. The dashboard application can be used out of the box and there are no configuration or development steps that have to be done by the application developer.

Figure 3 depicts the workflow executed by the CityPulse components when the user wants to visualize the real-time or historic data.

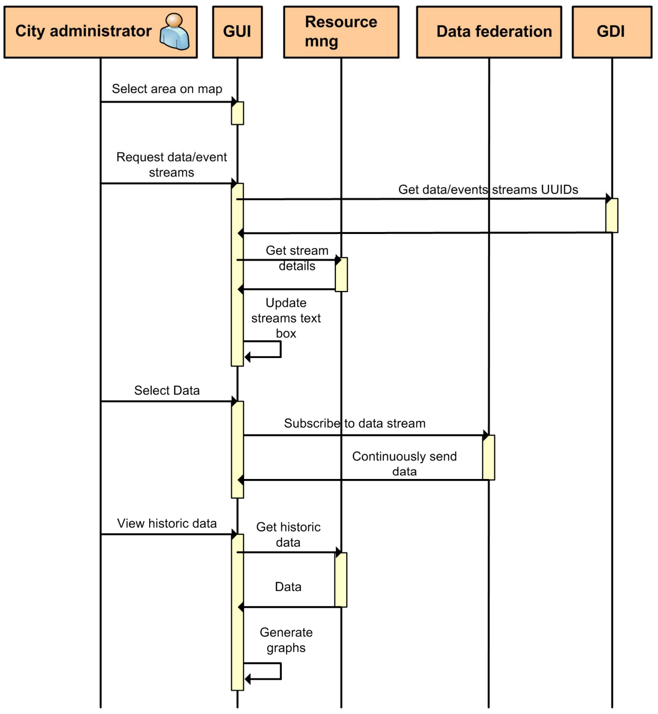


Figure 3 CityPulse dashboard workflow.

The CityPulse dashboard workflow steps are briefly described below:

First, the user has to select an area of interest on the map;

- When the user clicks the *Get available data sources* button, a request is made to the geospatial database, which contains the GPS coordinates of the area selected by the user (a list containing all the available sensors and their UUIDs is received);
- The user selects the data sources which he wants to display on the map (and also pick a colour for their representation). Now a new request is generated to the data federation component in order to receive in real-time the selected data sources observations;
- After the data federation sends back the responses, the map will be populated with markers for each sensor. The size of the marker is proportional to the observation value and the colour is the one selected by the user previously;
- If a sensor on the map is selected, a popup will appear that will contain the description of the sensor (its type, field name, coordinates and last value);
- The user then has three options of displaying the historical data: Show on chart A, B or C from the left side of the dashboard;
- If one of these 3 options is selected, a request to the resource management will be made for obtaining the historical data for the time period specified in the upper region of the left section of the dashboard;
- At the end, a graph is going to be created and displayed on the right part of the GUI.

2.2 CityPulse travel planner mobile application

In order to demonstrate how the CityPulse framework can be used to develop applications for smart cities and citizens, we have implemented a context-aware real time Travel Planner using the live data from the city of Aarhus, Denmark. The scenario was defined in collaboration with the IT departments of the Aarhus municipality and the following criteria have been considered: the impact of the application for the citizens and the data availability. Et al.This scenario aims to provide travel-planning solutions, which go beyond the state of the art solutions by allowing users to provide multi dimensional requirements and preferences such as air quality, traffic conditions and parking availability. In this way the users receive parking and route recommendations based on the current context of the city. In addition to this, Travel Planner continuously monitors the user context and events detected on the planned route. The user will be prompted to opt for a detour if the real time conditions on the planned journey do not meet the user specified criteria anymore [Puiu, D. Et al. 2016].

For the considered scenario, the real-time adaptive urban reasoning components are used to provide answers, when a user generates routes and parking recommendation requests. Figure 4 depicts the workflow executed by the components for providing an answer when route a recommendation is requested.

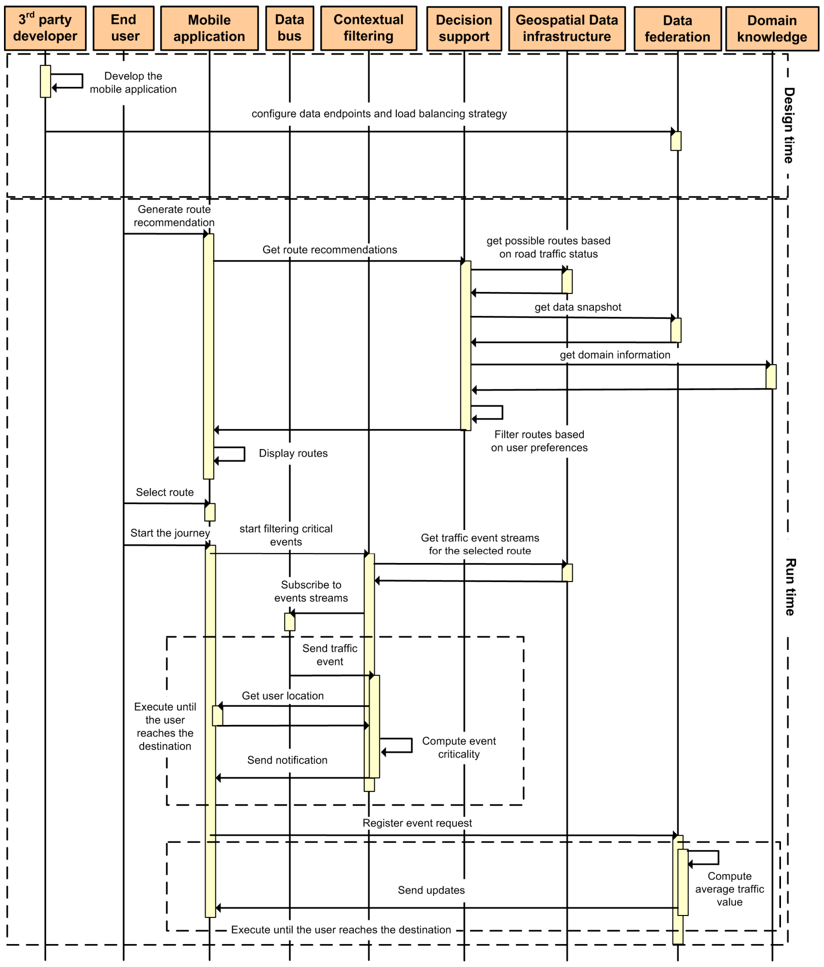

3rd party developer
End user
Mobile application
Data bus
Contextual filtering
Decision support
Geospatial Data infrastructure
Data federation
Domain knowledge
Develop the mobile application
configure data endpoints and load balancing strategy
Design time
Generate route recommendation
Get route recommendations
get possible routes based on road traffic status
get data snapshot
get domain information
Filter routes based on user preferences
Display routes
Select route
Start the journey
start filtering critical events
Get traffic event streams for the selected route
Subscribe to events streams
Run time
Send traffic event
Get user location
Execute until the user reaches the destination
Compute event criticality
Send notification
Register event request
Compute average traffic value
Send updates
Execute until the user reaches the destination

Figure 4. Real-time Adaptive Urban Reasoning workflow for Travel Planner [Puiu, D. Et al. 2016].

Figure 5 depicts the user interfaces used by the end user to set the travel preferences and the destination point.

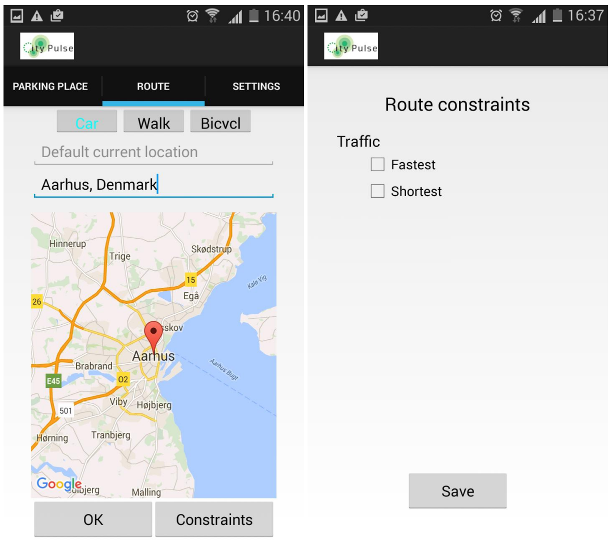


Figure 5 The user interfaces of the Android application used to select the starting point (left) and the travel preferences (right) [Puiu, D. Et al. 2016].

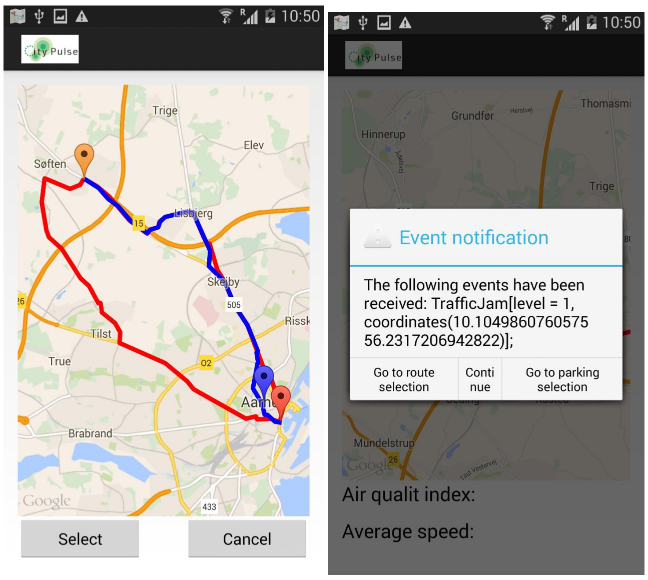
After the user has filled in the details and made the request using the user interface, the mobile application generates the appropriate request for the Decision support component which has the following main fields:

- Type: indicating what decision support module is to be used for this application (“TRAVEL-PLANNER” in this case);
- Functional details: specifying possible values of user’s requirements, including:
    - Functional parameters: mandatory information that the user provides such as starting and ending locations, starting date and time, and transportation type (car, bicycle, or walk).
    - Functional constraints: numerical thresholds for cost of a trip, distance, or travel time.
    - Functional preferences: the user can specify his preferences along selected routes, which hold the functional constraints. These preferences can be the minimization or the maximisation of travel time or distance.

Figure 6 (left) depicts the user interface where the routes computed by the Decision support are displayed to the end user.

After the user selects the preferred route, a request is generated to the *Contextual filtering* component in order to identify the relevant events for the use while he/she is traveling. The request includes the following properties:

- Route of interest: the current route of the user



The following events have been received: TrafficJam[level = 1, coordinates(10.1049860760575 56.2317206942822)];

Figure 6 The user interfaces of the Android application: select the preferred route (left); notification of a traffic jam which appeared on the selected route while the user is travelling (right) [Puiu, D. Et al. 2016].

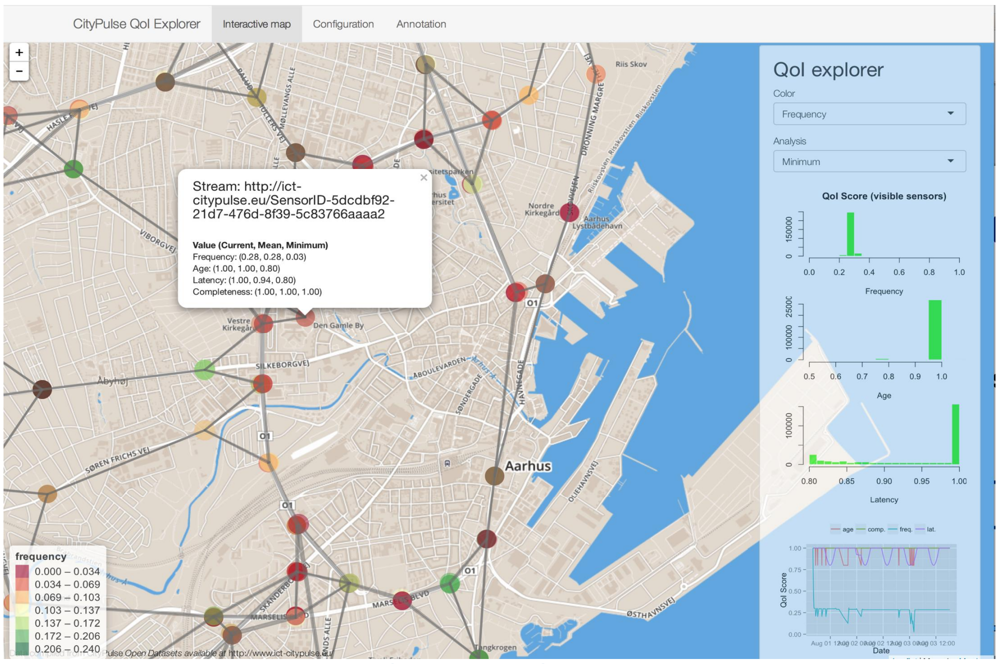
- Filtering Factors: used to filter unrelated (unwanted) events out, and include the event's source, event's category, and user's activity, as specified in a user-context ontology. For this particular scenario, the activities included in our ontology include CarCommute (user is traveling by a car), or Walk, or BikeCommute (user is traveling by a bike).
- Ranking Factor: identifies which metric is preferred by the user for ranking the criticality of incoming events. We have currently implemented the Ranking Factor based on two metrics: distance, and gravity of the event. In order to combine these two metrics, we use the linear combination approach, where the user can identify weights (or importance) for each metric.

Figure 6 (right) depicts the notification received by the end user, while she is traveling and a traffic event is detected on her route.

2.3 CityPulse QoI explorer

To allow the domain expert to observe the quality of data streams the CityPulse framework provides a tool called QoI explorer to monitor deployed sensors in the city. Within this tool, a map visualises the location of each sensor (see Figure 7). By selecting a quality metric in the menu at the right side the state for this metric is shown by a colour for each sensor on the current map. A legend in the left bottom corner indicates the corresponding quality value for the sensors colour. By clicking a sensor on the map a detailed view for the selected sensor is shown. In addition, the menu at the right side provides a selection of the time range for shown quality values and to switch between average, minimum, or maximum values. A histogram for each metric allows a quick overview about the distribution of quality values for the sensors currently shown within the map.

The QoI explorer interacts mainly with the quality monitoring component and it was developed using the library R Shiny. Some additional configuration options (e.g. an offline mode) are available via the *Configuration* tab.



Stream: http://ict-citypulse.eu/SensorID-5dcdbf92-21d7-476d-8f39-5c83766aaaa2

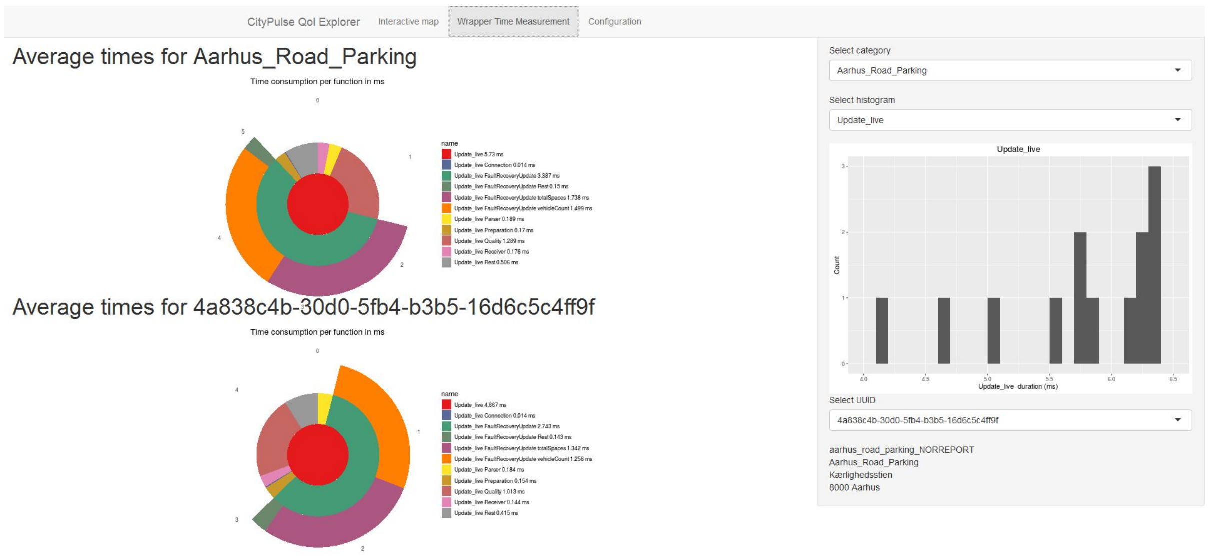
Value (Current, Mean, Minimum)
Frequency: (0.28, 0.28, 0.03)
Age: (1.00, 1.00, 0.80)
Latency: (1.00, 0.94, 0.80)
Completeness: (1.00, 1.00, 1.00)

frequency
0.000 – 0.034
0.034 – 0.069
0.069 – 0.103
0.103 – 0.137
0.137 – 0.172
0.172 – 0.206
0.206 – 0.240

Figure 7 QoI Explorer

2.4 CityPulse components performance monitoring

The QoI Explorer offers an additional tool to measure the current performance of the CityPulse framework. All components, which are integrated into the Resource Management can be monitored for their current average time consumption. A snapshot of this user interface is presented in Figure 8.



Average times for Aarhus_Road_Parking

name
🟥 Update_live 5.73 ms
🟦 Update_live Connection 0.014 ms
🟩 Update_live FaultRecoveryUpdate 3.387 ms
🟩 Update_live FaultRecoveryUpdate Rest 0.15 ms
🟪 Update_live FaultRecoveryUpdate totalSpaces 1.738 ms
🟧 Update_live FaultRecoveryUpdate vehicleCount 1.499 ms
🟨 Update_live Parser 0.189 ms
🟫 Update_live Preparation 0.17 ms
🟥 Update_live Quality 1.289 ms
🟪 Update_live Receiver 0.176 ms
⬜ Update_live Rest 0.506 ms

Average times for 4a838c4b-30d0-5fb4-b3b5-16d6c5c4ff9f

name
🟥 Update_live 4.667 ms
🟦 Update_live Connection 0.014 ms
🟩 Update_live FaultRecoveryUpdate 2.743 ms
🟩 Update_live FaultRecoveryUpdate Rest 0.143 ms
🟪 Update_live FaultRecoveryUpdate totalSpaces 1.342 ms
🟧 Update_live FaultRecoveryUpdate vehicleCount 1.258 ms
🟨 Update_live Parser 0.184 ms
🟫 Update_live Preparation 0.154 ms
🟥 Update_live Quality 1.013 ms
🟪 Update_live Receiver 0.144 ms
⬜ Update_live Rest 0.415 ms

Select category: Aarhus_Road_Parking
Select histogram: Update_live
Select UUID: 4a838c4b-30d0-5fb4-b3b5-16d6c5c4ff9f

aarhus_road_parking_NORREPORT
Aarhus_Road_Parking
Kærlighedsstien
8000 Aarhus

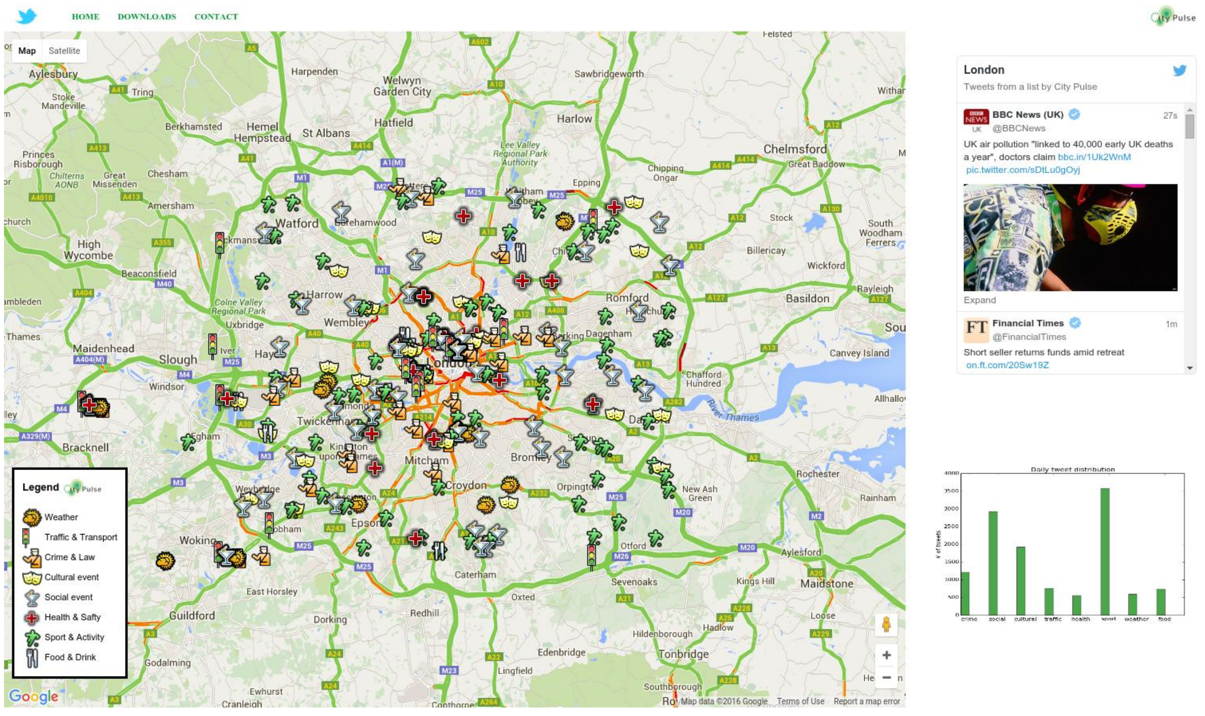
Figure 8 CityPulse components performance monitoring

The snapshot depicts the average processing times for the *Aaarhus Road Parking* data stream. The layered pie plots show the processing time for each measured component and its subcomponents. The menu at the right sides provides an additional histogram to see the distribution of processing times for a selected component. The *Time Measurement* component uses an internal statistic interface provided by the Resource Management and is completely realised by using R.

2.5 CityPulse social media analysis

Having connected to the Twitter Streaming API, the CityPulse dedicated Data wrapper collects a live stream of data in the form of tweets and automatically detect the source language and translate the tweets to English to facilitate the data processing step. The data processing unit then uses a multi-view learning approach to extract events by connecting to the raw-tweet database. Given a tweet, the processing unit assigns it to one of the event classes from the pre-defined class set: {Transportation and Traffic, Weather, Cultural Event, Social Event, Sport and Activity, Health and Safety, Crime and Law, Food and Drink}.

The web interface (see Figure 9) facilitates the visualisation of the extracted city events on a Google map in near real-time. The interface is written in HTML and Javascript and composed of (a) Google map canvas layer on which the processed and annotated Tweets are displayed with their class-identical icons (b) a live London traffic layer from google traffic API - code coloured paths on the map (c) a bar chart panel which presents the class distribution histogram of daily Tweets and (d) a panel for displaying Twitter timeline.



London
Tweets from a list by City Pulse

BBC News (UK) @BBCNews 27s
UK air pollution "linked to 40,000 early UK deaths a year", doctors claim bbc.in/1Uk2WnM pic.twitter.com/sDtLu0gOyj
Expand

Financial Times @FinancialTimes 1m
Short seller returns funds amid retreat on.ft.com/20Sw19Z

Figure 9 CityPulse social media analysis GUI.

The map data is updating in 60 second time windows by adding the past minute's Tweets to the existing ones upto a 60-minutes time window. Meaning that, the map data will be updated on an hourly basis. Clicking on each event a dialogue box is shown on the map which reveals the underlying Tweet content along with its time-stamp. The twitter user id and the name are anonymised for privacy purposes.

The web interface is written in Javascript and HTML languages and reads the data from a CSV file of the live NLP processing component.

2.6 CityPulse 3D map

The 3D map tool provides an innovative way of visualising real-time data related to a city in a 3D model. By using this tool municipality employees, city planners or citizens in general, are able to have a perception of different data and its effect on the 3D model. This way users are also able to identify data patterns and other phenomena in the different areas of the city.

The 3D map (see Figure 10) has been developed initially for the city of Aarhus, but it can be easily used to display other cities. In order to add a new city to be tool the user needs to provide a KML file with the buildings and roads for that city.

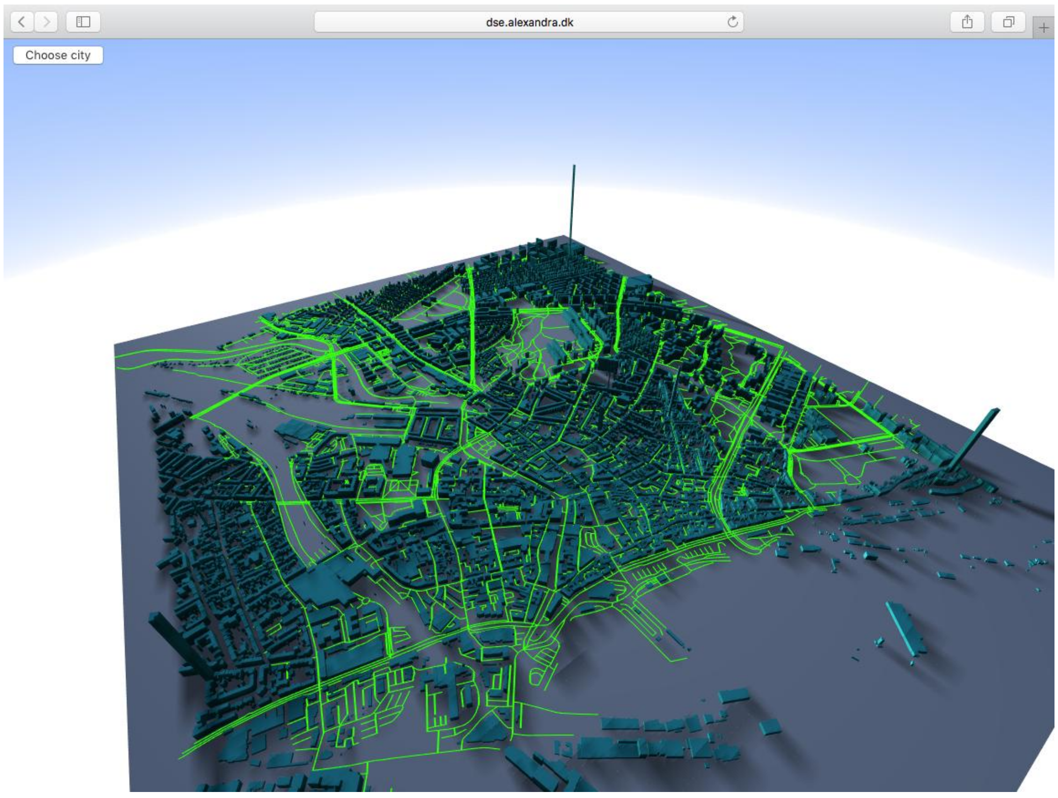


Figure 10 CityPulse 3D map.

The phenomena visualized in the tool includes detected events from the CityPulse middleware, such as traffic jams, public parking and twitter related events, as well as pollution and noise factors and energy consumption in buildings. For the CityPulse events the user can visualize a geo-located bubble and click on it to have the details of the event. For the pollution, noise and energy consumption in buildings the user can visualize building height changes according to the received values and color changes to the buildings following a color code according to the severity of the detected event.

The tool has been developed using Javascript and libraries such as three.js for the visual part. For the backend and the integration with the CityPulse middleware a simple component was developed using Node.js. This component connects to the Message Bus component (AMQP consumer) listening to all the messages of type “event”. The events are received in RDF format, parsed and forwarded in JSON format to the 3D map using web sockets so that the visualization can be updated.

3. CityPulse framework API description

3.1 Overview

The CityPulse framework contains several components, which can be used for performing real time data analytics on the data generated by the smart cities. Some of the main operations, which can be done using the frameworks' components, are:

- Smart city sensors virtualization;
- Semantic annotation;
- Data quality monitoring;
- Data aggregation;
- Data federation;
- Event detection;
- User centric decision support;
- Smart city data visualisation.

Third party developers can use the framework to create new applications for smart-cities. Using the framework is simpler and faster than building the application from scratch. Based on the application requirements the developers can deploy the whole framework's stack of components or only a selection of them.

The scope of the framework's exposed APIs is to allow the developers to properly configure and adapt the components based on the processing requirements of the application which have to be developed. The developers can apply the following types of adaptations to the CityPulse framework:

- Register new data sources in the framework;
- Plug into the components custom made processing logic;
- Configure the components operations (e.g. turn on/off a certain feature)
- Configure which components to be used based on the application requirements.

The following sections of this chapter present the APIs exposed to third party developers of the CityPulse framework.

3.2 The CityPulse framework architecture

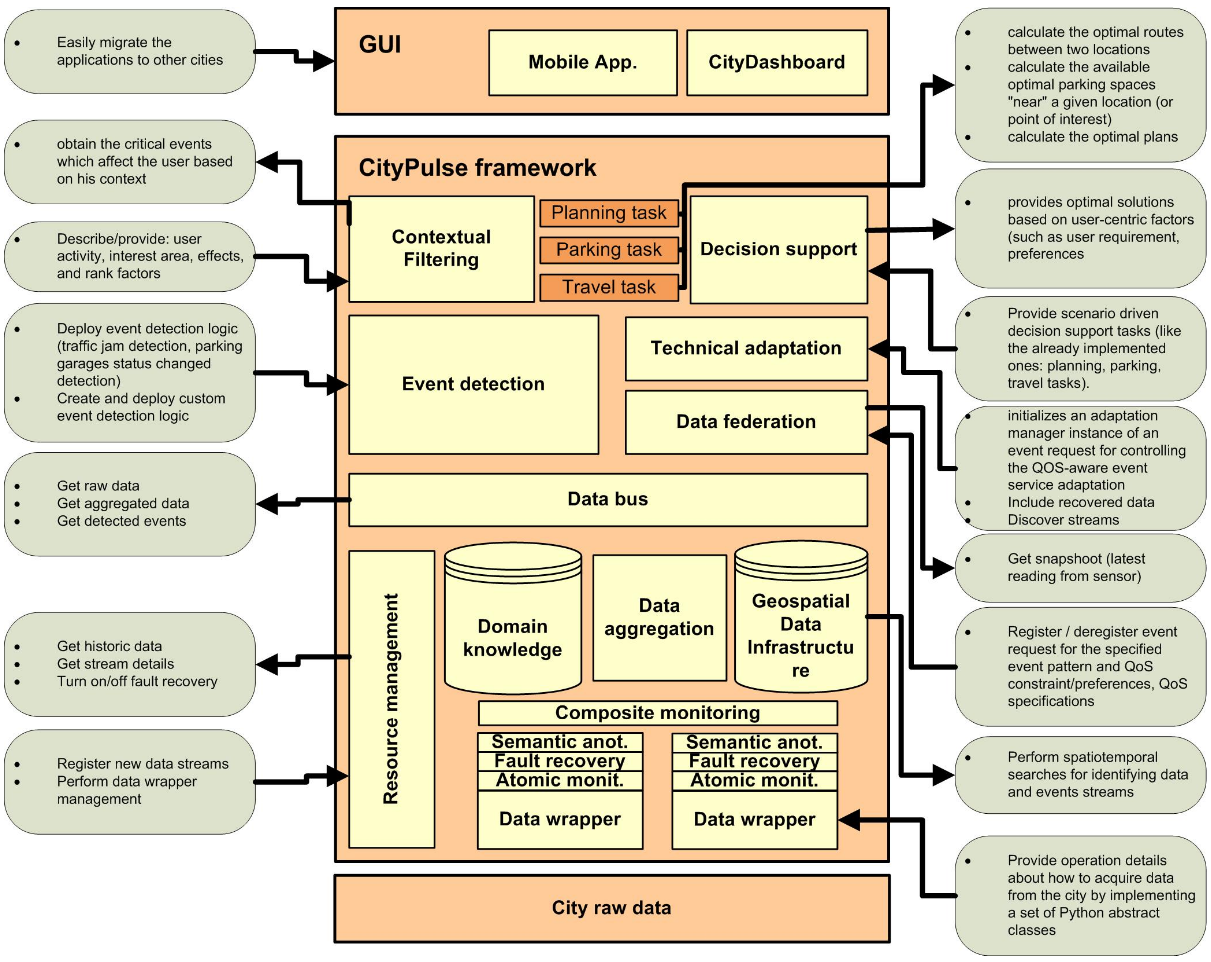
The CityPulse framework integrates and processes large volumes of streaming city data in a flexible and extensible way. Service and application creation is facilitated by open APIs that are exposed by the CityPulse components.

The CityPulse components are depicted in Figure 11 and can be divided in three main categories:

- Large-scale data stream processing modules: representing the tools, which allow the application developer to interact with the heterogeneous and unreliable data sources from the cities. The tools allow also discovering, summarizing and processing the data streams. The comomonents from this category are represented by: resource management, data wrapper, semantic annotation, data aggregation, geospatial data infrastructure, event detection and data federation.

- Reliable information processing modules: these tools can be used to continuously asses the plausibility, correctness and trustworthiness of the data sources. The comomonents from this category are represented by: atomic monitoring, composite monitoring, fault recovery, conflict resolution and technical adaptation.
- Adaptive decision support modules: containing the tools which can be used for making various recommendations based on the user context and the current status of the city. The comomonents from this category are represented by: contextual filtering, decision support and planning, parking, travel tasks.

The data bus is not presented in this document because it was not developed by the CityPulse consortium. For our presentations we have used RabbitMQ data bus which can be downloaded from this location: https://www.rabbitmq.com/ .



Easily migrate the applications to other cities
obtain the critical events which affect the user based on his context
Describe/provide: user activity, interest area, effects, and rank factors
Deploy event detection logic (traffic jam detection, parking garages status changed detection)
Create and deploy custom event detection logic
Get raw data
Get aggregated data
Get detected events
Get historic data
Get stream details
Turn on/off fault recovery
Register new data streams
Perform data wrapper management
GUI
Mobile App.
CityDashboard
CityPulse framework
Contextual Filtering
Planning task
Parking task
Travel task
Decision support
Event detection
Technical adaptation
Data federation
Data bus
Resource management
Domain knowledge
Data aggregation
Geospatial Data Infrastructure
Composite monitoring
Semantic anot.
Fault recovery
Atomic monit.
Data wrapper
Semantic anot.
Fault recovery
Atomic monit.
Data wrapper
City raw data
calculate the optimal routes between two locations
calculate the available optimal parking spaces "near" a given location (or point of interest)
calculate the optimal plans
provides optimal solutions based on user-centric factors (such as user requirement, preferences
Provide scenario driven decision support tasks (like the already implemented ones: planning, parking, travel tasks).
initializes an adaptation manager instance of an event request for controlling the QOS-aware event service adaptation
Include recovered data
Discover streams
Get snapshoot (latest reading from sensor)
Register / deregister event request for the specified event pattern and QoS constraint/preferences, QoS specifications
Perform spatiotemporal searches for identifying data and events streams
Provide operation details about how to acquire data from the city by implementing a set of Python abstract classes

Figure 11 CityPulse architecture and APIs.

The tools from the first category handle and process the city data streams. The CityPulse enabled applications will use cloud-based components to run the services. In this way the components continuously monitor and process the streams. From this point, any application can interact with the components, via the exposed APIs, in order to obtain at any moment, information about the current status of the city.

The reliable information processing modules ensure a reliable, consistent and fault tolerant processing of information.

In the CityPulse applications, the adaptive decision support components are triggered when a recommendation is needed or a certain context or situation needs to be monitored. Therefore these components are not running continuously as the ones from the first category. The remainder of this section presents the CityPulse components.

3.3 Resource management

3.3.1 Component description

The Resource Management component, as introduced in [CityPulse-D3.1], is responsible for managing all Data Wrappers. During runtime an application developer or the CityPulse framework operator can deploy new Data Wrappers to include data from new data streams. Furthermore, deployed Data Wrappers can be deactivated or removed from the system. There is a configuration file to specify the SPARQL endpoint and MessageBus grounding used by all Data Wrappers, managed by this component. The API is implemented as HTTP server.

The Resource Management component can be used for the following types of scenarios:

- Fetch live stream data via one or more Data wrappers (see section 3.4)
- Replay historic data embedded in a Data wrapper

Known limitations: The minimum sampling rate to pull data from an external resource is 1 second. Fetching data more frequently is only possible if the external resource pushes the data into the corresponding Data Wrapper.

3.3.2 Component location on GitHub

This component can be found at the following GitHub location:
https://github.com/CityPulse/CP_Resourcemanagement.git

3.3.3 Component prerequisites

The Resource management is implemented in the programming language Python. The following Python packages have to be installed before using the component. The packages are available either in the package repository of the Ubuntu operating system or can be installed using PIP.

- Pika
- CherryPy
- Psycopg2
- NumPy
- SciPy
- SKLearn
- RDFlib

- Requests (version > 2.8)
- Requests-oauthlib
- Chardet

In addition, the following Ubuntu packages need to be installed in order to run the Resource management:

- python-pip
- libgeos++-dev
- libgeos-3.4.2
- python-dev
- libpq-dev
- git
- zip

3.3.4 CityPulse framework dependencies

For the Resource management in order to run properly it needs to have access to the following CityPulse components: Message Bus; Geospatial Data Infrastructure and the Knowledge Base (Triplestore).

3.3.5 Component deployment procedure

As mentioned before, the CityPulse Resource management requires additional libraries, which can be installed using the following command on an Ubuntu Linux installation. The Resource management is not limited to Ubuntu Linux, but no other Linux distribution has been tested so far.

```
$> sudo apt-get install python-pip libgeos++-dev libgeos-3.4.2 python-dev libpq-dev
python-scipy git automake bison flex libtool gperf unzip python-matplotlib
```

In addition, using the following command required python packages will be installed:

```
$> sudo pip install pika cherrypy shapely psycopg2 numpy sklearn rdflib chardet
requests requests-oauthlib
```

The Resource management uses the Virtuoso triplestore to store annotated observations. As of February 2016, the virtuoso provided with the apt-repository in Ubuntu 14.04 LTS is outdated and lacks required features. Therefore, an installation from the sources is necessary. This can be achieved with the followings commands:

```
$> wget --no-check-certificate -q https://github.com/openlink/virtuoso-
opensource/archive/stable/7.zip -O virtuoso-opensource.zip
$> unzip -q virtuoso-opensource.zip
$> cd virtuoso-opensource
$> ./autogen.sh
$> ./configure
$> make
$> sudo make install
```

After that start the virtuoso:

```
$> sudo /etc/init.d/virtuoso-opensource-7 start
```

NOTE: the make command may hang after "VAD Sticker vad_dav.xml creation ..." if there is a virtuoso process running. Check with "ps ax|grep virtuoso" and kill if a virtuoso is running.

Afterwards you can download the Resource Management source code from the Github repository:

```
$> git clone https://github.com/CityPulse/CP_Resourcemanagement.git
```

The next step is to edit the configuration file with your favourite editor. An example configuration can be found in virtualisation/config.json. For details about the configuration file see Table 1.

When running the Resource management in replay mode the python process may require a lot of file descriptors to read the historical data. Users may be required to increase a limit for file descriptors in the operating system. To change the limit on Mac OS X 10.10 and higher run the following command in a terminal:

```
$> sudo launchctl limit maxfiles 2560 unlimited
```

This will set the limit to 2560. On Linux

```
$> ulimit -n 2560
```

should do the trick. Add the line into the .bashrc in your home directory to make it permanent.

The Resource management uses a configuration file to store the connection details to other components of the framework. The configuration is provided as JSON document named “config.json” within the “virtualisation” folder. The configuration consists of a dictionary object, where each inner element holds the connection details to one component of the framework, also as a dictionary. The following table lists all inner dictionary keys (bold) and their content.

Table 1: Segments of the configuration file

triplestore	
driver	The Resource Management supports the use of either Virtuoso or Apache Fuseki as triplestore. The value “sparql” tells the Resource Management to use Virtuoso. “fuseki” for Fuseki.
host	The hostname/IP of the triplestore as string.
port	The port of the triplestore as integer.
path	The path to the sparql-endpoint.
base_uri	The base URI is used to create the graph name
rabbitmq	
host	The hostname/IP of the message bus as string.
port	The port of the message bus as integer.
interface	The configuration of the HTTP based API interface. The API is realised using the CherryPy framework. The configuration here is directly passed to CherryPy’s ‘quickstart’ method. Therefore all configuration options CherryPy provides are available. For more details see https://cherrypy.readthedocs.org/en/3.2.6/concepts/config.html#configuration.
gdi_db	
host	The hostname/IP of the geo-spatial database as string.

port	The port of the geo-spatial database as integer.
username	The username for the database as string.
password	The user's password for the database as string.
database	The name of the database to use as string.

An example configuration can be found in virtualisation/config.json.

3.3.6 Component Start

The Resource management is started via command line terminal. There are a series of command line arguments available to control the behaviour of the Resource management. The following Table 2 lists all command line arguments and their purpose.

Table 2: Command Line Arguments for the Resource management

Argument	Purpose
-replay	Start in replay mode. In replay mode historic sensor observations between the time frame START and END are used instead of live data. Also the replay speed can be influenced by the speed argument. Requires that the Resource Management has been started at least once before.
-from START	In replay mode determines the start date. The format is "%Y-%m-%dT%H:%M:%S".
-to END	In replay mode determines the end date. The format is "%Y-%m-%dT%H:%M:%S".
-messagebus	Enable the message bus feature. The Resource Management will connect to the message bus and publish new observation as soon as they are made.
-triplestore	Enable the triplestore feature.
-aggregate	Use the aggregation method, as specified in the SensorDescription, to aggregate new observations.
-speed SPEED	In replay mode determines the speed of the artificial clock. The value range is [0-1000]. An artificial second within the replay will take 1000 – SPEED milliseconds.
-gdi	Geospatioal Database Injection. Newly registered Data wrappers are reported to the Geospatioal Database.
-gentle	Reduces the CPU load in replay mode, but slower.
-cleartriplestore	Deletes all graphs in the triplestore (may take up to 300s per wrapper!)
-restart	Restarts the Resource Management with the same arguments as last time.
-eventannotation	The Resource Management will listen on the message bus for new events to semantically annotate them and store them into the triplestore. Last feature requires the triplestore argument.

3.3.7 Component APIs

The Resource Management provides a HTTP based API to control deployed Data wrappers and get access to the sensor observations made by the Data wrapper. The API can be accessed according to the configuration provided in the configuration file (seeTable 1) under the path "api/". Thus, a specific function is accessible via "http://<IP>:<PORT>/api/<FUNCTION>". The reminder of this section lists the available functions, required parameters and answer messages, which are usually in JSON format.

Function	listwrapper	Method	GET
Parameter	None		
Answer	A list of abbreviated SensorDescriptions of deployed wrapper. The SensorDescriptions contain only parts of the general part of a SensorDescription (see section "Data wrappers and semantic annotation") namely: uuid, sensorName, author, sensorID, fullSensorID, source, sensorType, sourceFormat, messagebus, and information.		

Function	listwrapperfull	Method	GET
Parameter	None		
Answer	A list of abbreviated SensorDescriptions of deployed wrapper. In contrast to listwrapper to complete SensorDescriptions are returned here.		

Function	get_static_stream_data	Method	GET
Description	Returns a JSON document containing the static information about a Data wrapper/stream deployed.		
Parameter	uuid	The UUID of a Data wrapper/stream deployed in the Resource Management	
Answer	status	In case an error occurred "Fail", otherwise "Ok".	
	message	In case an error occurred a description of the error, otherwise empty.	
	data	The static information about the Data wrapper as Notation 3 (N3) document.	

Function	deploy	Method	POST
Description	This function can be used to deploy new Data wrappers during runtime.		
Parameter	deployunit	The new Data wrapper packed as deployable unit (see section 3.4.5 for details about the deployable unit.	
Answer	status	In case an error occurred "Fail", otherwise "Ok".	
	message	In case an error occurred a description of the error, otherwise empty.	
	sensordescriptions	The SensorDescriptions of the newly deployed Data wrappers.	

Function	snapshot	Method	GET
Description	This function searches in the triplestore for observations of a specific Data wrapper.		
Parameter	uuid	The UUID of a Data wrapper/stream	
	start	A start date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. The parameter is optional. If left out the Resource Management will include the earliest observation.	

	end	An end date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. This parameter is optional. If left out the Resource Management will search till the most current observation.
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.
	message	In case an error occurred a description of the error, otherwise empty.
	data	The observations (within the time frame, if any) in Notation 3 (N3) format.

Function	snapshot_last		Method	GET
Description	This function searches in the triplestore for the very last observations made by a specific Data wrapper.			
Parameter	uuid	The UUID of a Data wrapper/stream		
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.		
	message	In case an error occurred a description of the error, otherwise empty.		
	data	The last observation of the Data wrapper in Notation 3 (N3) format.		

Function	snapshot_sql		Method	GET
Description	This function searches in the SQL database for observations of a specific Data wrapper. Compared to the snapshot function this function is more performant and allows specifying the return format.			
Parameter	uuid	The UUID of a Data wrapper/stream		
	start	A start date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. The parameter is optional. If left out the Resource Management will include the earliest observation.		
	end	An end date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. This parameter is optional. If left out the Resource Management will search till the most current observation.		
	format	The return format for the observations found in the database. Available formats are „n3“, „nt“, „xml“, „turtle“, „pretty-xml“, „trix“, and „json“. If none or an invalid value is provided „json“ is used.		
	last	If this Boolean parameter is true (not empty) only the last observation is returned.		
	fields	The fields parameter allows to specify a comma separated list of fieldnames to filter the observations.		
Answer	If successful the observations in the specified format, otherwise the word “error”.			

Function	uuids_by_servicecategory	Method	GET
Description	This function returns all observations of multiple Data wrappers belonging to the specified service category (aka sensorType).		

Parameter	service_category	The name of the service category.
	start	A start date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. The parameter is optional. If left out the Resource Management will include the earliest observation.
	end	An end date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. This parameter is optional. If left out the Resource Management will search till the most current observation.
Answer	The observations (within the time frame, if any) in “nt” format.	

Function	snapshot_quality		Method	GET
Description	This functions returns the quality of information (QoI) values for a specified Data wrapper.			
Parameter	uuid	The UUID of a Data wrapper/stream. Multiple UUIDs can be provided as comma separated list.		
	start	A start date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. The parameter is optional. If left out the Resource Management will include the earliest observation.		
	end	An end date to search for observations in the format *%Y-%m-%dT%H:%M:%S*. This parameter is optional. If left out the Resource Management will search till the most current observation.		
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.		
	message	In case an error occurred a description of the error, otherwise empty.		
	result	List of dictionary objects for each UUID. If for a single UUID quality information could be received, such entry is build as follows:		
		uuid	The UUID of the Data wrapper with the quality.	
		dataset	The quality information for this Data wrapper as N3 document.	
		If a specific UUID caused an error this entry is build as follows:		
		uuid	The UUID of the Data wrapper causing the error.	
		error	A description of the error.	
	uuid	The uuid parameter as used in the request.		

Function	data_timeframe and data_timeframe_sql		Method	GET
Description	These functions return the dates of the oldest and newest observation stored for a specific sensor. Using “data_timeframe” will search in the triplestore whereas “data_timeframe_sql” will search in the SQL database.			
Parameter	uuid	The UUID of a Data wrapper/stream		
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.		
	message	In case an error occurred a description of the error, otherwise empty.		
	data	The minimum and maximum dates for observations available in the triplestore.		

Function	activate_fault_recovery	Method	GET
Description	This function activates the fault recovery for the specified Data wrapper.		
Parameter	uuid	The UUID of a Data wrapper/stream	
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.	
	message	In case an error occurred a description of the error, otherwise empty.	

Function	deactivate_fault_recovery	Method	GET
Description	This function deactivates the fault recovery for the specified Data wrapper.		
Parameter	uuid	The UUID of a Data wrapper/stream	
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.	
	message	In case an error occurred a description of the error, otherwise empty.	

Function	get_description	Method	GET
Description	Returns the SensorDescription for the specified Data wrapper.		
Parameter	uuid	The UUID of a Data wrapper/stream	
Answer	status	In case an error occurred “Fail”, otherwise “Ok”.	
	message	In case an error occurred a description of the error, otherwise empty.	
	data	If successful the SensorDescription.	

Function	remove_wrapper	Method	GET
Description	Removes a Data wrapper specified by an UUID.		
Parameter	uuid	The UUID of the Data wrapper to be removed.	
Answer	“Ok” is successful, otherwise “Fail”.		

Function	getAllLastQualities	Method	GET		
Description	Returns current/last QoI values calculated by the Atomic Monitoring for all streams registered to the Resource Management.				
Parameter	None				
Answer	If success				
	“list”	Contains a list of dict elements for every stream			
		“QoI metric name”	The name of the QoI metric (e.g. “Latency”)		
			CURRENT	States that QoI values are the current (last) ones.	
				ratedValue	QoI value normalised to 0-1 by Reward and Punishment algorithm.
				absoluteValue	The absolute QoI value (e.g. 10 for latency
			unit	The unit of the QoI metric (e.g.	

				“http://purl.oclc.org/NET/muo/ucum/unit/time/second” for latency)
		uuid	The uuid of the stream the QoI values belong to.	
	If not successful.			
	status	In case an error occurred “Fail”.		
	message	In case an error occurred a description of the error.		
	uuid	The uuid parameter as used in the request.		

Function	getQualityValues		Method		GET
Description	Returns QoI values calculated by the Atomic Monitoring for all streams registered to the Resource Management.				
Parameter	uuid	The UUID of a Data wrapper/stream. Multiple UUIDs can be provided as comma separated list.			
	types	Provides a possibility to get the average quality for timespan. Possible options are “HOURLY” and “DAILY”. Additional options “WEEKLY” and “MONTHLY” are implemented but not configured. The selected fields are added to the preconfigured “CURRENT” field for QoI values.			
	avg	Set this parameter to “True” to display average QoI values.			
	minimum	Set this parameter to “True” to display minimum QoI values.			
	maximum	Set this parameter to “True” to display maximum QoI values.			
Answer	If success				
	“list”	Contains a list of dict elements for every stream			
		“QoI metric name”	The name of the QoI metric (e.g. “Latency”)		
			CURRENT	States that QoI values are the current (last) ones.	
				ratedValue	QoI value normalised to 0-1 by Reward and Punishment algorithm.
				absoluteValue	The absolute QoI value (e.g. 10 for latency
			“type”	One type entry for every type in parameter “types”.	
				absoluteAvg	If parameter “avg==True” displays the average value for absolute QoI values.
				ratedMax	If parameter “maximum==True” displays the maximum value for normalised QoI values.
				ratedAvg	If parameter “avg==True” displays the average value for normalised QoI values.
				absoluteMin	If parameter “minimum==True” displays the minimum value for absolute QoI values.
				absoluteMax	If parameter “maximum==True” displays the maximum value for absolute QoI values.

				ratedMin	If parameter “minimum==True” displays the minimum value for normalised QoI values.
			unit	The unit of the QoI metric (e.g. “http://purl.oclc.org/NET/muo/ucum/unit/time/second” for latency)	
		uuid	The uuid of the stream the QoI values belong to.		
	If not successful.				
	status	In case an error occurred “Fail”.			
	message	In case an error occurred a description of the error.			
	uuid	The uuid parameter as used in the request.			

In addition to the functions accessible at the “api/” path there is a statistic function under the path “stat_api/”. The function can be used to measure the performance of individual processing steps performed by a deployed data wrapper identified by the uuid parameter or by the category including all wrappers belonging to it. It returns a list of values containing elements with name and value fields. Every element can have an additional values field containing a list of elements. Name is the name of the measured section and value is the average time of the last measurements in seconds.

Function	avg_processing_time		Method	GET
Description	This function is the API for the time measurement module integrated in the data wrapper. The integrated measurement module calculates the average time, which is consumed for configured sections within the wrapper.			
Parameter	uuid/category	The UUID of a Data stream or the category stream. Multiple UUIDs or categories can be provided as comma separated list. If empty returns processing times for all wrappers by UUID.		
Answer	If success			
	“uuid”/”category”	UUID of the stream or name of the category.		
		name	The name of the measured function/module.	
		value	The average processing time of the function/module.	
		values	An optional list, which might contain new elements of name, value, and values.	
	If not successful			
	status	“Fail”.		
	message	“UUID/Category not found.”		

3.3.8 Error handling

Depending on the log level specified in the command line parameter the Resource management creates a log file under virtualisation/virtualisation.log. Log messages will also be printed in the terminal running the Resource management.

Table 3 present the list of errors, which might appear during the deployment of new Data wrappers or the execution of the Resource management component.

Table 3: Possible Errors during Runtime

Error name	Error description	Possible solution
„Could not connect to MessageBus server. Disabling MessageBus feature.“	The Resource management was unable to connect to the message bus broker. No messages will be sent or received over the message bus by the Resource management. This is only a warning – the Resource management will continue its operation, just without the message bus features.	• Check the configuration file if message bus details are correct • Check if the network operates properly - the maschine running the Resource management can reach the maschine running the message bus broker • Check if the port for the message bus broker is open/forewarned (if needed)
„Deployment of wrapper ... failed.“	This error may occur during the deployment of a Data wrapper as deployable unit.	• Check format of deployment descriptor • Check if deployment descriptor points to the correct module and class • Ensure class to be instantiate has a default constructor • Check if ZIP file not broken
“Error deploying wrapper: ...”	This error may occur during the deployment of a Data wrapper as deployable unit. Here, the sensor description is validated for mandatory information.	• Check if all manadatory information is provided in the sensor description
“Problem parsing start- or end date: ...”	The values for the command line parameters “-start” and/or “-end” could not be parsed.	• Ensure the provided values are in the “%Y-%m-%dT%H:%M:%S” format
„start- and enddate required for replay mode“	The command line parameter “-replay” was given, but not “-start” and/or “-end”.	• Ensure to provide the parameters “-start” and “-end”
„Exchange could not be declare: ...“	On startup the Resource management tries to register several exchanges on the message bus broker.	• Use the RabbitMQ admin panel, remove the exchange and try again • Restart RabbitMQ
“Error while updating sensor ...”	This error indicates, that a Data wrapper had a problem to process an observation. The problem can be occurred somewhere in the processing chain (connection,	• Check if sensor description is correct • Check your wrapper code • Check the network connection is up and

	parser, QoI monitoring, …). The error message contains the Data wrapper ID and the exception's message to guide the developer in fixing the problem.	running
"there is no data, ask fault recovery" (+ "2")	This message indicates that a Data wrapper's connection did not return a newly fetched observation. Since this is an expected behavior by unreliable external data sources, this is logged with the level "INFO". If the problem is caused due to the fact, that the external resource has gone offline, this problem will fix itself once the resource comes back online again.	• Check the "source" property in the sensor description • Check if the network is operational • Check the extension of the class AbstractConnection, if you created on.
"trying to add a wrapper which is not an instance of AbstractWrapper"	This error occurs, if you use the addWrapper(<wrapper>) method provided by the abstract class AbstractComposedWrapper and the parameter "wrapper" is not an extension of the abstract class AbstractWrapper.	• Check your code and provide an instance of AbstractWrapper for this method.
"Sparql Endpoint HTTPError in …"	An error occurred during the use of the Sparql HTTP endpoint.	• Check configuration file if triplestore details are correct • Check if the triplestore is running • Check if triplestore is reachable; required ports forwarded
"no historic data beginning at … found"	The Resource management was started in replay mode for a specific time interval. This warning appears when a Data wrapper has no historic data for this time interval. In this case the Data wrapper is excluded from the replay.	• Change the time span you want to replay
"ThreadedTriplestoreAdapter was unable to save data, store might be offline!"	The Resource management was unable to store triples into the triplestore.	• Check configuration file if triplestore details are correct • Check if the triplestore is running • Check if triplestore is reachable; required ports forwarded

3.4 Data wrappers and semantic annotation

3.4.1 Component description

A Data Wrapper as introduced in [CityPulse-D3.1] is the gateway of a data stream into the CityPulse framework. Depending on the type of stream, on the transport technology and the stream data format, many different Data Wrappers may exist. Basic information about the stream is provided via a SensorDescription. The SensorDescription also holds all required information for the semantic annotation of the stream and observations.

Operational details are implemented extending a set of abstract classes in the programming language Python. Often required features, such as periodic pulling of data from a HTTP source, have been developed within the CityPulse project and can be used directly. The implementation along with a deployment descriptor can be bundled and transferred to the Resource Management via API.

It can be used for the following types of scenarios:

- Scenario 1: Fetch data periodically from external resources.
- Scenario 2: Receive data pushed by an external resource.

3.4.2 Component location on GitHub

This component can be found at the following GitHub location:
https://github.com/CityPulse/CP_Resourcemanagement/tree/master/wrapper_dev

3.4.3 Component prerequisites

Data wrappers are not executed by themselves, but loaded by the Resource management. Thus Data wrappers can only be used if the Resource management and all its prerequisites have been deployed.

3.4.4 CityPulse framework dependencies

For a Data wrapper in order to run properly it needs to have access to the following CityPulse components: Resource management, Data aggregation, Data Bus, Knowledge base (triplestore).

3.4.5 Component deployment procedure

In order to be able to deploy a Data wrapper its code and SensorDescription must be packed into a ZIP archive along with a deployment descriptor. Such archive is called deployable unit (DU). The deployment descriptor is a JSON file named 'deploy.json', placed in the root of the DU and containing a dictionary object. This object must have the keys 'module' naming the Python module (name of the source code file without extension) and 'class' naming the class to instantiate when the wrapper is loaded. It is also possible to pack multiple Data wrappers into a single DU. In that case the deployment descriptor contains a list of dictionary objects as explained before. A DU can be deployed using the Resource Management's REST API or by storing it into the folder named 'autodeploy'. There is also an additional tool, the "Deployment Manager" available, able to create a DU and move it to the correct place.

A SensorDescription consists of two parts. The first part contains general properties describing the whole stream. Table 4 gives an overview of the properties in this part. The second part contains details about the fields within a sensor observation. Properties in this part must be repeated for each field of a sensor observation. Table 5 gives an overview about the properties in this part. The SensorDescription can be provided as JSON document or programmatically as Python object.

Table 4: Overview of general properties in a SensorDescription

Name	Type	Description
source	String	The source of the data stream, e.g. the HTTP URL for a REST service.
author	Stirng	Name of the stream provider/ID of a User Profile
sensorName	String	The name of the sensor. **
sensorID	String	A human readable ID for the sensor.
fullSensorID	URI	Long version of the sensorID as IRI. Used to semantically annotate and store observations into the triplestore.
sensorType	String	The type of sensory data this sensor provides. Also known as ServiceCategory.
graphName	String	The name of the graph used to store observations in the virtuoso.
sourceType	String	<pull|push> + "_" + <http|…> *
information	String	A free text description about the sensor. *
cityName	String	The name of the city the sensor is located in. *
countryName	String	The name of the country the sensor is located in. *
streetName	String	The name of the street the sensor is located in. *
postalCode	Integer	The zip code of the city the sensor is located in. *
maxLatency	Float	The maximum latency (in seconds) allowed on the network to fetch new raw data.
updateInterval	Integer	If the source type is a pull connection this field specifies the number of seconds between two requests performed by the framework. If the source type is a push connection the field specifies the maximum number of seconds an update must be sent to the framework. If no update was received the Fault Recovery is used to produce an estimation value.
messagebus	Dict	Dictionary with the fields 'routingKey' and 'exchange'. The framework will publish new observations on the message bus using these details.
uuid	UUID	A unique identifier for the wrapper. The value is used to identify the wrapper while using the Resource Managements API. ***
namespace	URI	An URI used to create instances of concepts during the annotation process. The URI should end with a slash '/'.
fields	List	A list of field names present in an observation by the data stream.
field	Dict	A dictionary describing the fields in an observation. The key elements have to be identical with the elements in the 'fields'-list. The values are dictionary objects as described in Table 5.

Table 5: Overview of field specific properties in a SensorDescription

Name	Type	Description
propertyName	String	The name of the concept in the city ontology to use for the

		semantic annotation (static).	
propertyURI	URI	A URI to create instances of the concept identified by 'propertyName'. The instance's URI will be appended by a dash '-' and the sensors uuid.	
propertyPrefix	String	By default an instance of a property, identified by 'propertyName' uses the City Ontology prefix 'ct'. The 'propertyPrefix' can be used to specify a different one. *	
Optional	Boolean	Indicates if the field is optional. If set to False or unset the field has to be delivered with the data stream. If set to True it is allowed to be missing.	
unit	URI	An URI to create an instance of muo:UnitOfMeasurement. *	
min	Float	The minimum value for a valid measurement.	
max	Float	The maximum value for a valid measurement.	
dataType	String	The data type of this field. Values for this property must correspond with a python data type. For example a field containing a timestamp could use 'datetime.datetime' to use the datetime class in the datetime python module.	
showOnCityDashboard	Boolean	This property specifies if this field will be shown on the city dashboard. The default value is false. *	
format	String	In case the field contains a timestamp this property may be used to specify the format. The value 'UNIX' can be used if the timestamp is in Unix format (i.e. number of seconds since 01.01.1970) or 'UNIX5' if it is the number of milliseconds since 01.01.1970.	
aggregationMethod	String	This property can be used to specify the method of aggregation to use for numeric values. If this optional this optional property is left out no aggregation will be performed. Following values are allowed: *	
		dft	Discrete Fourier Transformation
		paa	Piecewise Aggregation Approximation
		sax	Symbolic Aggregation Approximation
		sensorsax	SensorSAX
aggregationConfiguration	Dict	Configuration parameters for the selected aggregation method. *	
no_publish_messagebus	Boolean	Flag to avoid that observations are published on the message bus or not. Default value is false. *	

** optional*
*** no spaces allowed*
**** If no value is specified the framework will generate one.*

The following steps have to be achieved in order to deploy a Data wrapper:

- Step 1: change into the folder "wrapper_dev"
- Step 2: run python deploy_mgt.py
- Step 3: follow the instructions on screen

3.4.6 Component programmatic APIs

Table 6 contains that list of classes, which have to be extended during the development of new Data wrappers. The rest of the section explains the methods of each class, which are either available to the developer or must be overridden by the developer. In all methods the implicit 'self' reference (or 'cls' for classmethods) are left out in the parameter descriptions.

Table 6: The classes to develop a Data wrapper

Class name	Class scope
AbstractWrapper	This class representes a Data wrapper. Each geo-spatially distinct external resource that needs to be connected to the CityPulse framework must be modelled as an instance of AbstractWrapper.
AbstractComposedWrapper	In case multiple geo-spatially distinct sensor observations can be fetched or received with one single request, an AbstractComposedWrapper can be used to split and distribute the external resource's answer. The AbstractComposedWrapper holds references to multiple AbstractWrapper and acts as container for the distinct sensors. An AbstractComposedWrapper possesses a reference to an implementation of AbstractSplitter, which holds the logic how to split and distribute response messages.
AbstractConnection	A derivation of this class is responsible to fetch or receive new sensor observations from the external resource.
AbstractSplitter	AbstractSplitter is only used in combination with AbstractComposedWrapper. An implementation of an AbstractSplitter gets a response message from an external resource for multiple geo-spatially distinct sensor observations as input and divides these messages for each embedded AbstractWrapper within the AbstractComposedWrapper. Each embedded AbstractWrapper will have a special kind of AbstractConnection (named SplitterConnection), which collects the messages or messageparts corresponding to the AbstractWrapper.
AbstractParser	An instance of the AbstractParser class is responsible to interpret the plain response messages from the external resource received by the AbstractConnection (or SplitterConnection). An AbstractWrapper may have two implementations of an AbstractParser, one for real data and one for historic data.
AbstractReader	An AbstractReader provides historic data for an AbstractWrapper. If no AbstractReader is implemented the corresponding Data wrapper is not available for the Resource management's replay

	mode.

The class AbstractWrapper has the following methods:

- Method 1: getSensorDescription()
- Method 2: getFileObject(file currentfile, str filename, str mode='r')
- Method 3: run()
- Method 4: stop()
- Method 5: update()

Method 1 is an abstract method and needs to be overridden by the Data wrapper developer. It is possible to call the parent's implementation, if the sensor's description is provides as JSON document named "sensordescription.json" and located in the root of the Data wrapper. Otherwise the developer must create an instance of SensorDescription by himself and ensure required attributes (see Table 5 and Table 6) are provided. The method has no parameters and returns an instance of the SensorDescription class.

Method 2 is a class method providing access to files. The benefit of using this method compared to opening a file in the file system yourself is, that it provides a homogenous way of accessing a file, regardless if the Data wrapper is packaged in a Deployable Unit (basically a ZIP archive) or not. When working with files to access meta- or historic-data this method should be preferred.

The method getFileObject has the following parameters:

- file currentfile: a file object in the context of the calling Python document. In most cases this will be __file__.
- str filename: a relative path to the filename on the file system or within a ZIP archive.
- str mode: the mode to open the file, as documented here https://docs.python.org/2.7/library/functions.html#open.

The method returns a file like object.

Method 3 and 4 have no parameters and need to be overridden in case the source type is 'push', that means the wrapper needs to establish and hold open a connection to the external resource. The wrapper's developer is responsible to start a new thread, which establishes the connection (using the wrappers AbstractConnection) in the run-method. In the stop-method he must close the connection and end the created thread.

Method 5 starts the whole processing chain: fetching the data via the connection, parsing it, annotating it semantically, storing the observations in the triple store and publishing the observations on the message bus. Using the default implementation of the run-method, the update-method is called periodically in the specified interval ('updateInterval in the SensorDescription). However, when overriding the run-method (usually if the sensor type is 'push'), the update-method must be called explicitly. A good opportunity to do so is when the Data wrapper's Connection has received new data.

The class AbstractComposedWrapper has the following methods:

- Method 1: getSensorDescription()
- Method 2: getFileObject(file currentfile, str filename, str mode='r'); identical as in the AbstractWrapper class.
- Method 3: addWrapper(AbstractWrapper w)

Method 1 has the same purpose as in the AbstractWrapper class, but instead of returning a single instance of a SensorDescription it returns a Python list of all contained Data wrapper's SensorDescriptions. This behaviour is already implemented; a Data wrapper developer does not need to override this method.

Method 3 is used to add an implementation of an AbstractWrapper to the AbstractComposedWrapper. Since all Data wrappers within an AbstractComposedWrapper need to be accessible shortly after instantiation, the addWrapper method should only be used in the constructor of the implementation of the AbstractComposedWrapper. Error! Reference source not found. presents the signature of the method 3.

The method has the following parameters:

- AbstractWrapper w: An implementation of an AbstractWrapper to be added to the AbstractComposedWrapper

The method returns nothing.

The class AbstractConnection has the following methods:

- Method 1: __init__(AbstractWrapper wrapper); the constructor
- Method 2: next()

Method 2 is abstract and needs to be overridden by a Data wrapper developer. The purpose is to provide the next raw sensor observation for the Data wrapper, fetch from the external source. If no new observation was received this method must return None. The method has no parameters.

The method __init__ has the following parameters:

- AbstractWrapper wrapper: A reference to the Data wrapper using this Connection. In the default implementation this reference is stored in an attribute called "wrapper".

The method returns an instance of AbstractConnection.

The class AbstractSplitter has the following methods:

- Method 1: __init__(AbstractComposedWrapper wrapper); the constructor
- Method 2: next(SensorDescription sensorDescription). The behavior and signature of this method is identical to the one in the AbstractConnection.

- Method 3: update(str data). Divides raw sensory data for each Data wrapper contained in the AbstractComposedWrapper

The update method is called when the connection of the AbstractComposedWrapper has fetched new raw data. It is an abstract method and needs to be overridden by a Data wrapper developer. The task of this method is to divide the raw data for each AbstractWrapper within its AbstractComposedWrapper.

The method __init__ has the following parameters:

- AbstractComposedWrapper wrapper: A reference to the AbstractComposedWrapper using this Splitter. In the default implementation this reference is stored in an attribute called "wrapper".

The method returns an instance of the AbstractSplitter.

The method update has the following parameters:

- str data: The raw data as fetched by the AbstractComposedWrapper's connection.

The method returns nothing.

The method next has the following parameters:

- SensorDescription sensorDescription: A reference to the SensorDescription of the Data wrapper requesting the raw data.

An implementation of this method must return the raw data for a specific Data wrapper identified by the provided SensorDescription. The raw sensory data is provided via the update method and must be cached in this class.

The class AbstractParser has the following methods:

- Method 1: __init__(AbstractWrapper wrapper, str timestampfield=None); the constructor
- Method 2: parse(str data, AbstractClock clock)

The parse method gets the raw data from the Connection (or the Splitter, in case the Data wrapper is part of a AbstractComposedWrapper) and parses it.

The method __init__ has the following parameters:

- AbstractWrapper wrapper: A reference to the Data wrapper using this Parser. In the default implementation this reference is stored in an attribute called "wrapper".
- str timestampfield: The name of the field that contains a timestamp, in case a sensor observation of this sensor provides a timestamp. The name must correspond to one field described in the SensorDescription. The default value is None. By default the value is stored in an attribute name "timestampfield".

The method returns an instance of an AbstractParser.

The method parse has the following parameters:

- str data: the raw data as returned by the Data wrapper's Connection/Splitter
- AbstractClock clock: An instance to the Clock implementation currently used by the Resource management. It provides the method "now()" to get a datetime.datetime instance of the current time. In replay it is simulated and will differ from the system's time.

The method returns the parsed data as Python dict object. The object has the structure as shown in the following table. If the parameter named data is None, this method should return None.

Name	Content	
fields	List object containing the names of the fields as provided in the SensorDescription.	
<fieldname>	Dict-object with the parsed data belonging to a field with the name of fieldname. Each fieldname is an element of *fields*, thus len(fields) of these dict-objects exist in the parsed data. Each of these dict-objects has the following entries:	
	propertyName	As specified in the SensorDescription. *
	propertyURI	As specified in the SensorDescription. *
	unit	As specified in the SensorDescription. If no unit was specified this entry is omitted. *
	sensorID	The fullSensorID, as specified in the SensorDescription. *
	observationSamplingTime	The time the observation was parsed.
	observationResultTime	If an observation provides a field with the timestamp, the time the observation was observed, otherwise the time the observation was parsed (same as observationSamplingTime).
	value	The value for this field if found in the raw data, otherwise None.

* see Table 5: Overview of field specific properties in a SensorDescription Table 5 and Table 6

The class AbstractReader has the following methods:

- Method 1: __init__(AbstractWrapper wrapper, str timestampfield); the constructor
- Method 2: tick(AbstractClock clock)

The Resource management when executed in replay mode calls the tick method periodically. The tick method's purpose is similar to a Connection's next method. But instead of simply returning the next raw sensor observation, this method must return an observation O_t out of the historic data H fulfilling the following condition:

$$O_t \in H \mid t > t\text{-}1 \wedge \min(T\text{-}t) \wedge t < T,$$

where T is the current time in the Resource management, t is the time the historic observation O was made and t-1 is the time the previous returned observation was made. To determine the current time, the method gets a reference of the Clock used by the Resource management as parameter.

The method tick has the following parameters:

- AbstractClock clock: A reference to the Resource management's current clock.

The method returns the raw data of an observation selected from the historic data.

The method __init__ has the following parameters:

- AbstractWrapper wrapper: A reference to the Data wrapper using this AbstractReader.
- str timestampfield: The name of the field containing the timestamp. With this information it is possible to parse the timestamps and fulfill the tick-method's condition.

The method returns an instance of the AbstractReader.

Notes on data types of parameters

As Python is not strict on data types and they are not enforced in the Data wrapper, the passage of raw sensor data as string (str) from the Connection to the Parser (over the Splitter) is only a suggestion. If for example the Splitter requires loading a JSON serialised list in order to divide the raw data, it makes more sense to pass elements of this list, instead of serialising to string and loading again in the parser.

Notes on the Clock

Some methods described above have an implementation of AbstractClock as parameter. There exist two implementations of AbstractClock in the Resource management, namely RealClock and ReplayClock. Which implementation is used depends if the Resource management was started in replay mode or not. Both implementations provide a method "now()" which returns the current time in the Resource management as instance of datetime.datetime. A Data wrapper developer does not need and should not implement his own version of a clock.

3.4.7 Component APIs

A Data wrapper has no API to influence its execution, instead the Resource management provides the necessary API. However, a Data wrapper developer determines the behaviour of a Data wrapper by providing the mandatory descriptions file called SensorDescription introduced above.

3.4.8 Error handling

Since Data wrappers are executed within the Resource management, errors and possible solutions are listed in the Resource management section.

3.5 Data aggregation

3.5.1 Component description

The Data aggregation component allows coping with large volumes of data and reducing the size of real-time data streams. On the one hand, sensory observations can be very dynamic and their analysis can be computationally expensive. While it is desirable to reduce the computational cost, general practice in sensory environments is to reduce the communication overhead. So on the other hand, it is important to use a lightweight, adaptive, yet effective time series analysis approach in stream processing.

In time-series data analysis, Symbolic Aggregate Approximation (SAX) is one of the most computationally low cost data aggregation approaches. It transforms a time series into a discretised series of letters e.g. a word. However, it uses a non-adaptive window size in the segmentation process (i.e. fixed window size) of time-series analysis. While it is common to use the same parameterisation for all sensors in the analysis of the same type of data streams (e.g. traffic, parking), SAX affects the performance of the time series analysis approach. Utilisation of an adaptive window size in segmentation enables to overcome this issue with a function to dynamically predict the window size depending on the distribution or deviation of time-series data streams. In this context, we use SensorSAX that has been introduced in [Ganz et. all 2016] as an extension of SAX. It uses an adaptive window size in the segmentation of time series analysis. It adapts the window size based on the standard derivation of the data segment.

It can be used for the following types of scenarios:

- Scenario 1: Data aggregation component can be applied on regardless of the type of use-case scenario as long as it involves time series numerical data streams, such as traffic data streams, parking data streams. Utilisation of data aggregation component enables to avoid high memory consumption and can benefit the smart city framework to perform anomaly and event detection on a small set of aggregated data stream instead of performing on large volumes of raw data streams.

It cannot be used in the following situations:

- Scenario 2: It cannot be used for string data streams unless they are converted to numerical values.

Known limitations: Although we use SensorSAX to aggregate data streams, there still remains a need to take decision regarding the right parameterisation of the algorithm. Therefore, the parameters have to be decided by a domain expert after careful analysis of the data streams.

3.5.2 Component location on GitHub

The component can be found as part of the KAT toolkit on the following Github link: https://github.com/CityPulse/KAT

3.5.3 Component prerequisite

The Data Aggregation component is not executed by itself, but loaded by the Resource management. Thus Data Aggregation can only be used if the Resource management and all its prerequisites have been deployed.

3.5.4 CityPulse framework dependencies

The Data Aggregation needs to have access to the following CityPulse components: Resource Management, Data Wrapper, Data Bus

3.5.5 Component deployment procedure

The Data Aggregation component is automatically deployed when the Resource Management is deployed. Which Aggregation method is used for which Data source is specified in the configuration of the specific Data Wrapper (see Section Data Wrappers and Semantic Annotation). Currently there are four different Data Aggregation methods implemented in the CityPulse framework: Discrete Fourier Transformation (DFT), Piecewise Aggregate Approximation (PAA), Symbolic Aggregate approXimation (SAX) and SensorSAX (an extension for SAX).

3.5.6 Component programmatic APIs

Table 6Table 7 contains that list of classes, which have to be extended/implemented during the development of new Data Aggregation methods. The rest of the section explains the methods of each class, which are either available to the developer or must be overridden by the developer. In all methods the implicit 'self' reference (or 'cls' for classmethods) are left out in the parameter descriptions.

Table 7: The classes of the Data aggregation component

Class name	Class scope
GenericAggregator	This class is the base method for all the aggregation methods. Each currently implemented aggregation method is inheriting from this class and new aggregation methods have to inherit from this class.
AbstractAggregationControl	Each aggregation method hast to implement this class, which controls the programmatic flow of the aggregation method.
AggregatorControlFactory	Factory class to generically select the aggregation method. Whenever a new method is implemented by a developer, a new entry has to be made in the AggregatorControlFactory. Currently the entries are: entries = {"sax": SaxControl, 'sensorsax': SensorSaxControl, 'dft': DftControl, 'paa': PaaControl}

The class GenericAggregator has the following methods/functions/procedures:

- aggregate: defines the computation of the aggregation method, has to be overwritten by inheriting classes
- wrapper_added: Attaches the in the SensorDescription of the Data Wrapper specified Aggregation method to the fields of the data that have to be aggregated.

Listing 1 presents the signature of the method used for *wrapper_added*

```
aggregate(dict data, SensorDescription sensordescription)
```

Listing 1: the signature of the method aggregate

The method *wrapper_added* has the following parameters:

- dict data: the data that will be aggregated by the aggregation method
- SensorDescription sensordescription: Description of the data source, here it is specified which fields of the data have to be aggregated

The method returns aggregated data

Listing 2 presents the signature of the method used for *aggregate*

```
wrapper_added(SensorDescription sensordescription)
```

Listing 2: the signature of the method wrapper_added

The method *aggregate* has the following parameters:

- SensorDescription sensordescription: Description of the data source, here it is specified which fields of the data have to be aggregated by which aggregation method

The method returns nothing

The class AbstractAggregationControl has the following methods/functions/procedures:

- control: defines the work flow of the aggregation method (e.g. buffering incoming data until enough data is there to aggregate)

Listing 3 presents the signature of the method used for *control*

```
control(dict data)
```

Listing 3: the signature of the method control

The method *control* has the following parameter:

- dict data: the incoming data from the data stream

The method returns aggregated data

The class AggregatorControlFactory has the following methods/functions/procedures:

- make: uses the entries (entries = {"sax": SaxControl, 'sensorsax': SensorSaxControl, 'dft': DftControl, 'paa': PaaControl}) to assign the correct Aggregator to the Data Wrappers as specified in the sensor descriptions

Listing 4 presents the signature of the method used for *make*

```
make(str aggregator)
```

Listing 4: the signature of the method make

The method *make* has the following parameter:

- str aggregator: the name of the aggregation method, has to be included in entries

The method returns an instance of GenericAggregator

3.5.7 Component APIs

Table 8 contains the list of REST/websocket/other endpoints which can be used to interact with the component.

Table 8: The end points of the Data aggregation component

End point name	End point scope
Message bus	The aggregated data is published via the message bus on the exchange name: “aggregated_data”

Listing 5 presents the signature of the end point used for getting data from the message bus

```
Subscribe to the message bus on exchange “aggregated_data” at Demo server: 131.227.92.56:8007, at Development server:
131.227.92.55:8007
Routing key: “#” to subscribe to aggregated data from all data sources, specific routing keys can be found here:
http://131.227.92.55:8017/overview
```

Listing 5 the signature of the end point used for getting data from the message bus

The end point returns aggregated data in JSON format

3.6 Data federation

3.6.1 Component description

The Data Federation component is integrated as part of the key functionalities in the Automatic Complex Event Implementation System (ACEIS) as described in [citypulse d3.2]. ACEIS is a middleware for complex event services and the data federation component is responsible for event service discovery, composition, deployment and execution. In the CityPulse project, event service discovery and composition refer to finding a data stream or a composite set of data streams to address the user-defined event request. Event service deployment refers to transforming the event service composition results (a.k.a., composition plans) into RDF Stream Processing (RSP) queries and register these queries to relevant RSP engines, e.g., CQELS [LePhouc11] and C-SPARQL [Barbieri09]. Event service execution refers to establishing the input and output connections for the RSP engines to allow them consuming real-time data from lower-level data providers and delivering query results to upper-level data consumers.

It can be used in scenarios where on-demand discovery and composition of data streams are needed, and RSP is used for evaluating queries over the data streams e.g.,:

Scenario 1: monitoring traffic conditions over different streets and regions in a city for a city administrator. The data federation component can find the relevant data streams for observing the traffic and select the most optimal combination of streams in order to satisfy the QoS constraints specified by the users.

Scenario 2: planning the travel routes for a citizen based on his/her functional and non-functional requirements. The data federation components will find the relevant streams according to the routes computed from the start and end location specified by the users. It will deploy a query on the estimated travel time and notify the user about the approximate travel time during the trip. Alternatively, if a user is concerned with the air quality along his/her route, the data federation component can deploy a query to monitor the air pollution level.

It cannot be used in situations where queries or streams used for the queries are fixed.

3.6.2 Known limitations

The data federation components has the following limitations:

- Event Semantics: The expressiveness of event requests with regard to temporal and logical correlations is limited to AND, OR, SEQUENCE and REPETITIONS, as described in [citypulse d3.2], and the events modelled in ACEIS are instant events, i.e., only one timestamp allowed for each event. Interval-based events are not supported.
- RSP Engine Types: Currently only CQELS and C-SPARQL engines are supported for event service execution. However, a third-party developer can integrate new engines by extending the query transformation module.
- Concurrent Queries: Existing RSP engines are still in their early stages and there is room for performance optimisation. Currently, the data federation component can handle approximately 1000 CQELS or 90 C-SPARQL queries in parallel, by applying a load-balancing

technique. Higher number of concurrent queries *may* (depending on the query complexity) result in unstable engine status.

3.6.3 Component location on GitHub

This component can be found at the following GitHub location: https://github.com/CityPulse/ACEIS

3.6.4 Component prerequisite

The following applications have to be installed before using the component:

- JDK 1.7
- Web Server (e.g., Jboss 7.1.2)
- CQELS engine (available at:)
- CSPARQL engine (available at:)
- Other Java libraries located at https://github.com/CityPulse/ACEIS

3.6.5 CityPulse framework dependencies

The data federation component needs to have access to the following CityPulse components in order to run properly: resource management, data aggregation, data bus, data quality analysis etc.

3.6.6 Component deployment procedure

The following steps have to be achieved in order to deploy the component :

- Step 1: download resources provided at https://github.com/CityPulse/ACEIS
- Step 2: edit configuration file aceis.properties which includes the following fields
  hostIp: the IP address of the server hosting data federation server
  port: the main port used by the data federation server
  ontology: the folder storing the ontology files (used only if local *dataset* files are used)
  streams: the folder storing the stream data files for simulation (used only if operating on simulated streams)
  dataset: the location of stream meta-data, could be a local file or an URI for the virtuoso endpoint provided by the resource management component.
  request: the location of sample test queries.
- Step 3: run ACEIS.jar or compile from Main.java to start ACEIS server, with the following program parameters:
  cqelsCnt: number of CQELS engine instances
  csparqlCnt: number of CSPARQL engine instances
  smode: load balancing strategy, could be "elastic", "balancedLatency", "balancedQueries", "rotation" or "elastic"
  qCnt: number of concurrent queries (used only in simulation mode)
  step: interval between registering new queries (used only in simulation mode)
  query: path to the query file (used only in simulation mode)
  duration: life-time of the ACEIS server instance, 0 means indefinite
- Alternatively to Step 3: run $sh run.sh to skip step 3 and start ACEIS with default settings.

3.6.7 Component programmatic APIs

The complete API documentation (as Javadoc) is available online at: http://fenggao86.github.io/ACEIS/doc/index.html . Table 9 summarises the list of key classes used during the development of the data federation component, we also list the key public methods of those classes in the following.

Table 9: The classes of the data federation component

Package name	Class name	Class scope
org.insight_centre. aceis.engine	ACEISEngine	Modelling the ACEIS engine.
	ACEISScheduler	The scheduler for ACEIS engines, controlling the load balancing and performance optimisation.
	Comparator	Provides a list of static methods comparing the reusability of event services.
	CompositionPlanEnumerator	Provides methods to create composition plans based on the reusability between event services
	GeneticAlgorithm	Implements an heuristic algorithm based on genetic evolution to create QoS-optimised composition plans
	ReusabilityHierarchy	A hierarchy of event services with servces as nodes and resuable relations as edges, the hierarchy is used as an index for event services.
org.insight_centre. aceis.eventmodel	EventDeclaration	Modeling event services.
	EventPattern	Modeling event patterns for complex event services.
	EventRequest	Modeling event requests from users.
org.insight_centre. aceis.io	EventRepository	Modeling event repositories.
org.insight_centre. aceis.io.rdf	VirtuosoDataManager	Handling RDF data reading/writing via virtuoso server.
org.insight_centre. aceis.io.streams.cqels.rabbitmq	CQELSAarhusRabbitmqStream	CQELS data streams over rabbitmq.
org.insight_centre. aceis.io.streams.csparql.rabbitmq	CSPARQLAarhusRabbitmqStream	CSPARQL data streams over rabbitmq.
org.insight_centre. aceis.querytransformation	QueryTransformer	Interface for query transformers
	CQELSQueryTransformer	QueryTransformer for CQELS
	CSPARQLQueryTransformer	QueryTransformer for CSPARQL
	TransformationResults	Modeling the query transformation results
org.insight_centre.	SubscriptionManager	Handles the query registration/deregistration and

aceis.subscriptions		subscription for CQELS/CSPARQL engines.
org.insight_centre. citypulse.main	MultipleInstanceMain	Main class for starting ACEIS engines for citypulse project.
org.insight_centre. citypulse.server	MultipleEngineInstanceServerEndpoint	Server endpoint with multiple ACEIS engines deployed.

The class ACEISEngine has the following methods:

ACEISEngine

`public ACEISEngine(ACEISEngine.RspEngine engine)`
Creates an ACEIS engine with a specific RSP engine type.

Parameters:

`engine` - the type of RSP engine used by ACEIS

initialize

```
public void initialize(java.net.URI uri)
throws java.lang.Exception
```
initialize the ACEIS engine using the stream meta-data stored at the given uri

Parameters:

`uri` - the location of the stream meta-data

Throws:

`java.lang.Exception`

getContext

`public org.deri.cqels.engine.ExecContext getContext()`

Returns:

the cqels engine instance

getCsparqlEngine

`public eu.larkc.csparql.engine.CsparqlEngine getCsparqlEngine()`

Returns:

the csparql engine instance

getRepo

`public EventRepository getRepo()`

Returns:

the event repository used by this ACEIS engine

The class ACEISScheduler has the following methods:

initACEISScheduler

```
public static void initACEISScheduler(int cqelsNum,
int csparqlNum,                        java.lang.String dataset)
throws java.lang.Exception
```

Initializes the scheduler.

Parameters:

`cqelsNum` - max number of cqels engine instances

`csparqlNum` - max number of csparql engine instances

`dataset` - location of stream meta-data

Throws:

`java.lang.Exception`

addEngine

```
public static void addEngine(ACEISEngine engine)
```

Deploys a new ACEIS engine.

Parameters:

`engine` - ACEIS instance to be added

getAllACEISIntances

```
public
static java.util.Map<java.lang.String,ACEISEngine> getAllACEISIntances()
```

Returns:

the map of ACEIS engines, in each entry of the map, id is the unique engine id and engine is the instance

getBestEngineInstance

```
public static ACEISEngine getBestEngineInstance(ACEISEngine.RspEngine engineType)
throws java.lang.Exception
```

Gets the best engine instance according to the specified SchedulerMode

Parameters:

`engineType` - the specific type of RSP engine

Returns:

the most suitable engine instance

Throws:

`java.lang.Exception`

The class Comparator has the following methods:

calculateUtility

```
public static java.lang.Double calculateUtility(QosVector constraint,
WeightVector weight, QosVector capability)
throws java.lang.Exception
```

Calculates QoS utility based on given constraints, weights and capability.

Parameters:

`constraint` - QoS constraints

`weight` - QoS preferences

`capability` - QoS performance of a composition plan

Returns:

QoS utility calculated

Throws:

`java.lang.Exception`

constraintEvaluation

```
public
static java.util.List<EventPattern> constraintEvaluation(java.util.List<EventPat
tern> results, QosVector constraint)
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

directReusable

```
public static boolean directReusable(EventPattern ep1,
EventPattern ep2)
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```

Checks if two event patterns are directly reusable

Parameters:

`ep1` - the first ep

`ep2` - the second ep

Returns:

whether is directly reusable

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

getSameDSTs

```
public static java.util.List<java.lang.String> getSameDSTs(EventPattern ep1,
EventPattern ep2)
throws NodeRemovalException,
java.lang.CloneNotSupportedException
```

Get the list of DSTs in ep1 shared by ep2

Parameters:

`ep1` - first event pattern

`ep2` - second event pattern

Returns:

list of direct sub trees

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

inDirectReusable

```
public static boolean inDirectReusable(EventPattern ep1,
EventPattern ep2)
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```

Checks if two event patterns are in-directly reusable

Parameters:

`ep1` - first event pattern

`ep2` - second event pattern

Returns:

whether two patterns are in-directly reusable

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

isCanonicalSubstitute

```
public static boolean isCanonicalSubstitute(EventPattern ep1,
EventPattern ep2)
throws java.lang.Exception
```

Checks if the canonical event patterns of two event patterns are substitutes to each other

Parameters:

`ep1` - first event pattern

ep2 - second event pattern

Returns:

if two canonical patterns have identical meanings

Throws:

`java.lang.Exception`

isSubstitute

```
public static boolean isSubstitute(EventPattern ep1,
EventPattern ep2)
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```
Checks if two event patterns are isomorphic

Parameters:

ep1 - first event pattern

ep2 - second event pattern

Returns:

whether they are isomorphic

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

isSubstitutePrimitiveEvent

```
public static boolean isSubstitutePrimitiveEvent(EventDeclaration ed1,
EventDeclaration ed2)
```
Checks if two primitive event services are functionally equivalent

Parameters:

ed1 - first primitive event service

ed2 - second primitive event service

Returns:

reusable

```
public static java.lang.String reusable(EventPattern ep1,
EventPattern ep2)
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```
Derive the reusability code for two event patterns

Parameters:

ep1 - first event pattern

ep2 - second event pattern

Returns:

reusability code

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

sortDSTs

```
public
static java.util.List<EventPattern> sortDSTs(java.util.List<EventPattern> dstList
,                                           EventPattern parentEp)
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```

Sort the direct sub-trees for a given event pattern based on their temporal relations

Parameters:

`dstList` - list of DSTs

`parentEp` -

Returns:

sorted DSTs

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

The class CompositionPlanEnumerator has the following methods:

CompositionPlanEnumerator

```
public CompositionPlanEnumerator(EventRepository repo,
EventPattern query)
```

Creates an composition plan enumerator for a query and a repository

Parameters:

`repo` - event service repository

`query` - event service request/query

calculateBestFitness

```
public EventPattern calculateBestFitness(WeightVector weight,
QosVector constraint)                                    throws
java.lang.Exception
```

Finds the best composition plan via brute-force enumeration

Parameters:

`weight` - QoS preferences

`constraint` - QoS constraints

Returns:

composition plan with highest QoS utility

Throws:

`java.lang.Exception`

getACPs

```
public java.util.List<EventPattern> getACPs()
throws java.lang.Exception
```

Creates abstract composition plans (ACP) for the given query, an ACP is a composition plan that do not care about the actual event service bindings for the leaf nodes in the pattern.

Returns:

list of abstract composition plans

Throws:

`java.lang.Exception`

getCCPsforACP

```
public java.util.List<EventPattern> getCCPsforACP(EventPattern acp)
throws java.lang.Exception
```

Creates all possible concrete composition plans (CCPs) for a given ACP. A CCP is concrete in the sense that it has complete event service binding information for all of its leaf nodes.

Parameters:

`acp` - abstract composition plan

Returns:

list of concrete composition plans

Throws:

`java.lang.Exception`

getCCPsForACPs

```
public java.util.List<EventPattern> getCCPsForACPs(java.util.List<EventPattern> a
cps)                                            throws java.lang.Exception
```

Creates concrete composition plans (CCPs) for a list of ACPs.

Parameters:

`acps` - list of abstract composition plan

Returns:

list of concrete composition plans

Throws:

`java.lang.Exception`

The class GeneticAlgorithm has the following methods:

GeneticAlgorithm

```
public GeneticAlgorithm(CompositionPlanEnumerator cpe,
int selectionMode,                 int populationSize,
int numberOfIteration,                 java.lang.Double crossoverRate,
java.lang.Double mutationRate,                 java.lang.Double factor,
boolean fixedPopulationSize,                 WeightVector weight,
QosVector constraint)                 throws
java.lang.CloneNotSupportedException,
NodeRemovalException
```

Creates an instance of the genetic algorithm

Parameters:

`cpe` - composition plan enumerator

`selectionMode` - population selection mode

`populationSize` - initial population size

`numberOfIteration` - number of maximum iterations

`crossoverRate` - crossover rate

`mutationRate` - mutation rate

`factor` - selection factor

`fixedPopulationSize` - whether uses fixed population size

`weight` - QoS preferences

`constraint` - QoS constraints

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

createNextGen

```
public java.util.List<EventPattern> createNextGen(java.util.List<EventPattern> in
itialPopulation)                 throws
java.lang.Exception
```

Creates the next generation based on the current population

Parameters:

`initialPopulation` - current population

Returns:

next generation

Throws:

`java.lang.Exception`

evolution

```
public java.util.List<EventPattern> evolution()
throws java.lang.Exception
```

Starts genetic evolution

Returns:

final generation

Throws:

`java.lang.Exception`

getBestEP

```
public EventPattern getBestEP(java.util.List<EventPattern> population)
throws java.lang.Exception
```

Picks the best composition plan from the population

Parameters:

`population` -

Returns:

best composition plan

Throws:

`java.lang.Exception`

The class ReusabilityHierarchy has the following methods:

ReusabilityHierarchy

```
public ReusabilityHierarchy(java.util.List<EventPattern> queries,
java.util.List<EventPattern> candidates)
```

Creates an instance of an Event Reusability Hierarchy (ERH) with a set of queries and candidates (event patterns)

Parameters:

`queries` - list of queries

`candidates` - list of composition candidates

buildHierarchy

```
public void buildHierarchy()                    throws java.lang.Exception
```

Builds the ERH

Throws:

`java.lang.Exception`

getAllDescendants

```
public java.util.List<java.lang.String> getAllDescendants(java.lang.String
epId)
```

Gets all descendants of an event pattern in the ERH

Parameters:

`epId` - queried pattern id

Returns:

list of descendants

getIdenticalEp

```
public java.util.List<java.lang.String> getIdenticalEp(java.lang.String epI
d)
```

Gets all patterns with identical meaning of an event pattern

Parameters:

`epId` - id of the queried pattern

Returns:

list of identical patterns

getRootsFromHierarchy

```
public java.util.List<java.lang.String> getRootsFromHierarchy()
```

Gets the root nodes in the ERH

Returns:

list of root nodes

insertIntoHierarchy

```
public void insertIntoHierarchy(EventPattern ep)
throws java.lang.Exception
```

Inserts an event pattern into the ERH

Parameters:

`ep` - inserted pattern

Throws:

`java.lang.Exception`

The class EventDeclaration has the following methods:

EventDeclaration

```
public EventDeclaration(java.lang.String iD,
java.lang.String src,                java.lang.String eventType,
EventPattern ep,                java.util.List<java.lang.String> payloads,
java.lang.Double frequency,                QosVector internalQos)
```

Creates an event service declaration

Parameters:

`iD` - service id

`src` - source location

`eventType` - primitve event type

`ep` - event pattern, null if it is primitive

`payloads` - event payloads

`frequency` - update frequency

`internalQos` - internal QoS capability, i.e., QoS of service I/O

getExternalQos

```
public QosVector getExternalQos()
throws java.lang.CloneNotSupportedException,
NodeRemovalException
```

Calculates the overall QoS for this event service

Returns:

QoS vector

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

The class EventPattern has the following methods:

EventPattern

```
public EventPattern(java.lang.String iD,
java.util.List<EventDeclaration> eds,
java.util.List<EventOperator> eos,
java.util.Map<java.lang.String,java.util.List<java.lang.String>> provenance
Map,
java.util.Map<java.lang.String,java.lang.String> temporalMap,
int timeWindow,             int trafficDemand)
```

Creates an event pattern

Parameters:

`iD` - pattern id

`eds` - list of event delcarations involved in the pattern

`eos` - list of operators in the pattern

`provenanceMap` - map of provenance relationships

`temporalMap` - map of temporal relationships

`timeWindow` - time window

`trafficDemand` - estimated traffic demand

addDSTs

`public void addDSTs(java.util.List<EventPattern> dsts, EventOperator root)`
Adds a list of direct sub trees

Parameters:

`dsts` - list of DSTs

`root` - root operator

aggregateQos

`public QosVector aggregateQos() throws java.lang.CloneNotSupportedException, NodeRemovalException`
Aggregates qos vector for the event pattern.

Returns:

aggregated Qos vector

Throws:

`java.lang.CloneNotSupportedException`

`NodeRemovalException`

appendDST

`public void appendDST(EventDeclaration ed) throws NodeRemovalException`
Append DST to a leaf node (event delcaration)

Parameters:

`ed` - leaf node to append

Throws:

`NodeRemovalException`

getAncestors

```
public java.util.List<java.lang.String> getAncestors(java.lang.String nodeI
d)
```

Gets all ancestor nodes

Parameters:

`nodeId` - investigated node id

Returns:

list of ancestors

getCanonicalPattern

```
public EventPattern getCanonicalPattern()
throws java.lang.Exception
```

Gets tha canonical form of the event pattern

Returns:

canonical pattern

Throws:

`java.lang.Exception`

getCompletePattern

```
public EventPattern getCompletePattern()
```

Gets the complete form of the event pattern

Returns:

complete event pattern

getReducedPattern

```
public EventPattern getReducedPattern()                                 throws
java.lang.Exception
```

Gets irreducible pattern of this pattern

Returns:

irreducible event pattern

Throws:

`java.lang.Exception`

getSubtreeByRoot

```
public EventPattern getSubtreeByRoot(java.lang.String rootId)
throws java.lang.CloneNotSupportedException
```

Gets the sub-tree of this pattern

Parameters:

rootId - root of the sub-tree

Returns:

Throws:

`java.lang.CloneNotSupportedException`

insertParent

`public void insertParent(`EventOperator` eo, java.util.List<java.lang.String> childIds)`
Inserts an event operator for a set of nodes, updates the idcnt when done

Parameters:

`eo` - operator to insert under

`childIds` - list of child node ids

lift

`public boolean lift(java.lang.String rootId) throws java.lang.Exception`
Removes vertical redundant nodes

Parameters:

`rootId` - root node for the lifting

Returns:

whether removed redundant operators

Throws:

`java.lang.Exception`

liftTree

`public void liftTree(java.lang.String rootId) throws java.lang.Exception`
Invokes lift method for each node in a tree

Parameters:

`rootId` - root of the tree to be lifted

Throws:

`java.lang.Exception`

merge

`public boolean merge(java.lang.String rootId) throws java.lang.Exception`
Removes horizontal redundant nodes by merging direct sub trees for a node

Parameters:

`rootId` - root node for the subtree to be merged

Returns:

whether the merging function results in a modification.

Throws:

`java.lang.Exception`

replaceED

```
public void replaceED(java.lang.String edID,
EventDeclaration replacement)                   throws java.lang.Exception
```

Replace an event declaration with another without changing semantics of this pattern

Parameters:

`edID` - id of the event declaration to be replaced

Throws:

`java.lang.Exception`

replaceSubtree

```
public void replaceSubtree(java.lang.String nodeId,
EventPattern replacement)                            throws java.lang.Exception
```

Replaces a subtree with another without changing semantics

Parameters:

`nodeId` - root of the subtree to be replaced

`replacement` - replacement pattern

Throws:

`java.lang.Exception`

The class EventRequest has the following methods:

EventRequest

```
public EventRequest(EventPattern ep,                QosVector constraint,
WeightVector weight)
```

Creates an event request with a query pattern, a set of qos constraints and qos preferences

Parameters:

`ep` - queried event pattern

`constraint` - qos constraints

`weight` - qos preferences

The class EventRepository has the following methods:

buildHierarchy

`public void buildHierarchy()                    throws java.lang.Exception`
Builds the reusablity hierarchy

Throws:

`java.lang.Exception`

The class VirtuosoDataManager has the following methods:

buildRepoFromLocalFile

```
public static EventRepository buildRepoFromLocalFile(java.lang.String uri,
int simSize)                                               throws
java.lang.Exception
```
Builds the repository from local file

Parameters:

`uri` - path to local file storing event service descriptions

`simSize` - number of simulated event services for each real service retrieved

Returns:

created event repository

Throws:

`java.lang.Exception`

buildRepoFromSparqlEndpoint

```
public static EventRepository buildRepoFromSparqlEndpoint(java.net.URI uri,
int simSize)                                                    throws
java.lang.Exception
```
Builds the event repository from remote virtuoso endpoint

Parameters:

`uri` - endpint uri

`simSize` - number of simulated sensors

Returns:

created event repository

Throws:

`java.lang.Exception`

getSnapShot

```
public
static java.util.HashMap<java.lang.String,java.lang.String> getSnapShot(Eve
```

ntDeclaration ed)
throws java.lang.Exception
Gets the latest readings (snapshot) for a specific sensor

Parameters:

ed - the queried primitive event service

Returns:

latest reading

Throws:

`java.lang.Exception`

createEDModel

```
public
static com.hp.hpl.jena.rdf.model.Model createEDModel(com.hp.hpl.jena.rdf.mo
del.Model m,                                              EventDeclaration ed)
throws NodeRemovalException
```
Writes service description into RDF model

Parameters:

`m` - target RDF model

ed - event service description to write

Returns:

result RDF model

Throws:

`NodeRemovalException`

The class CQELSAarhusRabbitmqStream has the following methods:

CQELSAarhusRabbitMQStream

```
public CQELSAarhusRabbitMQStream(org.deri.cqels.engine.ExecContext context,
com.rabbitmq.client.Channel channel,
java.lang.String queueName,                          EventDeclaration ed)
throws java.io.IOException
```
Creates an CQELS stream over rabbitmq

Parameters:

`context` - target CQELS engine

`channel` - rabbitmq channel

`queueName` - rabbitmq queue name

ed - primtive event service declaration

Throws:

`java.io.IOException`

handleDelivery

```
public void handleDelivery(java.lang.String consumerTag,
com.rabbitmq.client.Envelope envelope,
com.rabbitmq.client.AMQP.BasicProperties properties,
byte[] body)                        throws java.io.IOException
```

Specified by:

`handleDelivery` in interface `com.rabbitmq.client.Consumer`

Throws:

`java.io.IOException`

stop

```
public void stop()
```

Specified by:

`stop` in class `org.deri.cqels.engine.RDFStream`

The class CSPARQLAarhusRabbitmqStream has the following methods:

CSPARQLAarhusRabbitMQStream

```
public CSPARQLAarhusRabbitMQStream(java.lang.String uri,
com.rabbitmq.client.Channel channel,
java.lang.String queueName,                              EventDeclaration ed)
throws java.io.IOException
```

Creats a CSPARQL stream over rabbitmq.

Parameters:

`uri` - stream uri

`channel` - rabbitmq channel

`queueName` - rabbitmq queue name

`ed` - primitive event service description

Throws:

`java.io.IOException`

handleDelivery

```
public void handleDelivery(java.lang.String consumerTag,
com.rabbitmq.client.Envelope envelope,
com.rabbitmq.client.AMQP.BasicProperties properties,
byte[] body)                        throws java.io.IOException
```

Specified by:

`handleDelivery` in interface `com.rabbitmq.client.Consumer`

Throws:

`java.io.IOException`

stop

```
public void stop()           throws java.io.IOException
Stops the stream
```

Throws:

`java.io.IOException`

The class QueryTransformer has the following methods:

transformFromED

TransformationResult `transformFromED(`EventDeclaration` m)`
`throws java.lang.Exception`
Creates transformation result from composition plan

Parameters:

`m` - composition plan

Returns:

transformation result

Throws:

`java.lang.Exception`

The class CQELSQueryTransformer has the following methods:

transformFromED

`public` TransformationResult `transformFromED(`EventDeclaration` plan)`
`throws java.lang.Exception`
Description copied from interface: **`QueryTransformer`**

Creates transformation result from composition plan

Specified by:

`transformFromED` in interface `QueryTransformer`

Parameters:

`plan` - composition plan

Returns:

transformation result

Throws:

`java.lang.Exception`

The class CSPARQLQueryTransformer has the following methods:

transformFromED

```
public TransformationResult transformFromED(EventDeclaration plan)
throws java.lang.Exception
```

Description copied from interface: **`QueryTransformer`**

Creates transformation result from composition plan

Specified by:

`transformFromED` in interface `QueryTransformer`

Parameters:

`plan` - composition plan

Returns:

transformation result

Throws:

`java.lang.Exception`

The class TransformationResults has the following methods:

TransformationResult

```
public TransformationResult(java.util.List<java.lang.String> serviceList,
java.util.List<java.lang.String> propertyList,
java.lang.String queryStr,
EventDeclaration transformedFrom)
```

Creates a transformation result

Parameters:

`serviceList` - list of services involved in the query

`propertyList` - list of properties selected in the query

`queryStr` - query string

`transformedFrom` - original composition plan

The class SubscriptionManager has the following methods:

createCompositionPlan

```
public EventDeclaration createCompositionPlan(ACEISEngine engine,
EventPattern queryPattern,
QosVector constraint,                                  WeightVector weight,
boolean forceRealSensors,
boolean useBruteForce)                                     throws
java.lang.Exception
```

Creats composition plan for a query

Parameters:

`engine` - ACEIS engine used for the composition plan

`queryPattern` - queried event pattern

`constraint` - qos constraints

`weight` - qos preferences

`forceRealSensors` - choose whether only real sensor are used for composition (i.e., ignoring simulated sensors)

`useBruteForce` - choose whether brute-force enumeration is used

Returns:

composition plan derived

Throws:

`java.lang.Exception`

deregisterAllEventRequest

```
public void deregisterAllEventRequest()
throws java.lang.Exception
```

De-register all event requests

Throws:

`java.lang.Exception`

deregisterEventRequest

```
public java.util.Date deregisterEventRequest(EventDeclaration plan,
boolean stopAllStream)
```

Deregister the cqels query, stop the result stream

Parameters:

`plan` - the plan to be de-registered

registerEventRequest

```
public void registerEventRequest(ACEISEngine engine,
EventDeclaration plan,                          JsonQuery jq,
javax.websocket.Session session,
java.lang.Double sensorFreq,                             java.util.Date start,
java.util.Date end,                             java.lang.String windowStr,
TechnicalAdaptationManager.AdaptationMode am)                         throws
java.lang.Exception
```

Register an event request to the ACEIS engine

Parameters:

`engine` - target ACEIS engine

`plan` - composition plan to register

`jq` - user query wrapped in Json

`session` - websocket session from server endpoint

`sensorFreq` - base sensor update frequency (used only in simulation)

`start` - start time for the request (used only in simulation)

`end` - end time for the request (used only in simulation)

`windowStr` - string specification for time window

`am` - adaptation mode

Throws:

`java.lang.Exception`

registerEventRequestOverMsgBus

```
public java.lang.String registerEventRequestOverMsgBus(ACEISEngine engine,
EventDeclaration plan,
TechnicalAdaptationManager.AdaptationMode am,
javax.websocket.Session session)
throws java.lang.Exception
```

Register event request over streams offered by message bus

Parameters:

`engine` - ACEIS engine to register

`plan` - composition plan

`am` - adaptation model

`session` - ACEIS server endpoint session

Returns:

registered query id

Throws:

`java.lang.Exception`

registerTechnicalAdaptionManager

```
public void registerTechnicalAdaptionManager(EventDeclaration plan,
QosVector constraint,                                        WeightVector weight,
java.lang.Double freq,
java.util.Date start,
java.util.Date end,
TechnicalAdaptationManager.AdaptationMode am)
throws java.lang.Exception
```

Initialzes adaptation manager

Parameters:

`plan` - composition plan

`constraint` - qos constraints

`weight` - qos preferences

`freq` - sensor frequency (used only in simulation mode)

`start` - request start time (used only in simulation mode)

`end` - request end time (used only in simulation mode)

`am` - adaptation mode

Throws:

`java.lang.Exception`

subscribeToQosUpdate

```
public void subscribeToQosUpdate(EventDeclaration plan,
javax.websocket.Session session,
java.util.List<java.lang.String> sids,
java.lang.Double freq,                          java.util.Date start)
```

Send qos update subscription commands to qos server endpoint

Parameters:

`plan` - composition plan

`session` - session used to send subscription

`sids` - list of sensor ids

`freq` - sensor frequency (used only in simulation)

`start` - sensor start time

unSubscribeToQosUpdate

```
public java.util.Date unSubscribeToQosUpdate(javax.websocket.Session sessio
n,
java.util.List<java.lang.String> sids,
java.lang.Double freq)
```

Unsubscribe to qos updates

Parameters:

`session` - session to send unsubscrption commands

`sids` - list of sensor ids

`freq` - sensor frequency

Returns:

time the qos update listening stops

The class MultipleInstanceMain has the following methods:

MultipleInstanceMain

```
public MultipleInstanceMain(java.util.Properties prop,
java.util.HashMap<java.lang.String,java.lang.String> parameters)
```
Initialzes ACEIS server with the configuration file and program parameters

Parameters:

`prop` - properties extracted from configuration file

`parameters` - program parameters

main

```
public static void main(java.lang.String[] args)                    throws
java.lang.Exception
```
Starts ACEIS server

Parameters:

`args` - program parameters

Throws:

`java.lang.Exception`

The class MultipleEngineInstanceServerEndpoint has the following methods:

onMessage

```
public java.lang.String onMessage(java.lang.String message,
javax.websocket.Session session)                                 throws
java.lang.Exception
```
When an event request is received, parse the request and take relevant actions

Parameters:

`message` - event request message wrapped in Json `session` - websocket session

Returns:

response message

Throws:

`java.lang.Exception`

onClose

```
public void onClose(javax.websocket.Session session,
javax.websocket.CloseReason closeReason)             throws
java.io.IOException
```

When a close message is received, deregister all event requests registered on the server

Parameters:

`session` - websocket session

`closeReason` - the close reason object

Throws:

`java.io.IOException`

3.6.8 Component APIs

Table 10 contains the list of websocket endpoint which can be used to interact with the component.

Table 10: The end points of the data federation component

End point name	End point scope
Data federation websocket server endpoint	The endpoint for registering continuous event requests as well as instant queries for sensor data snapshots.

Listing 6 presents the signature of the end point used for connecting to the data federation websocket server. Once the connection is established, the websocket server accepts a message wrapped from an org.insight_centre.aceis.eventmodel.EventRequest as an input. The data federation components gives two types of outputs: for a continuous event request, it produces continuous query results as Json messages wrapped from org.insight_centre.citypulse.commons.data.json.DataFederationResult. If the event request is not continuous it will produce a single DataFederationResult using the sensor data snapshot.

```
ws://[hostIP]:[port]/datafederation
```

Listing 6: the url example of the data federation end point

3.6.9 Error handling

The log file of the component can be found at EC-log.log

Table 11 present the list of errors which might appear during the deployment or the execution of the component.

Table 11: The error handling of the data federation component

Error name	Error description	Possible solution
Engine overloaded	When too many queries (e.g., >1000) are registered in parallel, the CQELS/C-SPARQL engines may stop responding and cease to produce results	Deploy more engine instance on more powerful server clusters.
Failed to de-register queries	Unable to deregister C-SPARQL queries, the server will send the message "FAULT: Error occur while stopping streams." to notify this error.	This error can be ignored unless engine is overloaded.
IOException during snapshot	Unable to connect to snapshot server, identified by the message starting with "FAULT: IOException".	Check the status of the snapshot server.
Invalid input format	Cannot parse the input message, identified by the message starting with "FAULT: JsonSyntaxException".	Check the input format and content.
Cannot create composition plan	The data federation server cannot create composition plans that satisfy the event request. Identified by the message "FAULT: Cannot create composition plans under specified constraints."	Check QoS constraints, try define more relaxed constraints.
Cannot register event requests	The data federation server cannot register the event request over message bus data streams. Identified by the message "FAULT: Failed to register event request."	Check connectivity to the message bus.

3.7 Event detection

3.7.1 Component description

The component is able to interpret raw (annotated) and aggregated data, in order to detect useful events for the user. The event detection component is generic and the user can deploy custom developed detection patterns. This code can be used to deploy or execute custom event detection logic. The application developer can, at any time, trigger the deployment of these mechanisms in order to analyse the data coming from the considered city. In addition to that these, the application developer can develop custom made event detection logic, by implementing a Java interface.

It can be used for the following types of scenarios:

- Scenario 1: processing numerical raw (annotated) and aggregated data; the focus is on temporal and structural constraints.

3.7.2 Component location on GitHub

This component can be found at the following GitHub location:

https://github.com/CityPulse/EventDetection

3.7.3 Component prerequisite

The following applications have to be installed before using the component:

- Esper from EsperTech: http://www.espertech.com/esper/

3.7.4 CityPulse framework dependencies

The Event detection component in order to run properly needs to have access to the following CityPulse components:

- *Geo-spatial data base (GDI)* in order to send it events that it has detected in its deployed nodes.
- Data bus from where it gets the annotated and aggregated data.
- Resource Manager Component to get the sensors description, information that is later used in the interpretation on aggregated and annotated data.

3.7.5 Component deployment procedure

The following steps have to be achieved in order to deploy the component

- Step 1: Locate the Event Detection(ED) .jar file.
- Step 2: Import the jar into a Java project.
- Step 3: Create a new class that will extend the EventDetectionNode class. This new class will represent your event detection node.
- Step 4: Inside the new class implement your CEP logics.
- Step 5: Deploy your newly created node (resulted after completing step 3 and 4) in your main method.
- Step 6: Repeat steps 3 to 5 for every new event detection node you want to deploy.
- Step 7: Inside the Event Detection .jar file you will find a config.properties file where you will have to change the parameters so that it will work in your scenario. The configuration parameters will be enumerated below.
- Step 8: After configuring the config.properties file, run your project.

An sample project can be found on GitHub.

The config.properties file has the following parameters:

- Resource Manager Configuration parameters: IP and Port.
- The Data bus configuration parameters: IP and Port.
- Geo-spatial data base configuration parameters: IP and Port.
- Testing mode parameter: testParameter (Boolean type)
- The URI for GDI parameter: GDI_AMQP_URI

3.7.6 Component programmatic APIs

Event detection component contains the following list of classes which can be used during the development:

Table 12: The classes of the Event detection component

Class name	Class scope
EventDetection	The goal of the event detection event detection is to process in near real-time a large volume of observations in order to identify added value events and to provide powerful mechanism for describing structural and temporal relationships between the observations coming from one or several data streams.
EventDetectionNode	Represents the generic event processing logic used for detecting special situations from the raw data such as traffic incidents or parking status incidents

The class EventDetection has the following methods:

- addEventDetectionNode: Method that adds a new event detection node.
- addEventDetectionNodeAggregated: Identical to addEventDetectionNode only it is designed for aggregated messages.
- removeEventDetectionNode: Method that removes a node by its eventDetectionNodeID, the list of already deployed nodes is updated with this change.

Listing 7 presents the signature of the method used for EventDetection:

```
void addEventDetectionNode(HashMap<String, String> eventDetectionLogicInputMaping, EventDetectionNode
eventDetectionNode)
```

Listing 7: the signature of the method addEventDetectionNode

The method addEventDetectionNode has the following parameters:

- HashMap<String, String> eventDetectionLogicInputMaping: is a HashMap linking the inputStreamName and it's UUID.
- EventDetectionNode eventDetectionNode: is the instance of EventDetectionNode you want to deploy.

The method returns void.

Listing 8 presents the signature of the method used for EventDetection:

```
void addEventDetectionNodeAggregated(HashMap<String, String> eventDetectionLogicInputMaping,
EventDetectionNodeAggregated eventDetectionNodeAggregated)
```

Listing 8: the signature of the method addEventDetectionNodeAggregated

The method addEventDetectionNodeAggregated has the following parameters:

- HashMap<String, String> eventDetectionLogicInputMaping: is a HashMap linking the inputStreamName to its UUID.
- EventDetectionNodeAggregated eventDetectionNodeAggregated: is the instance of EventDetectionNodeAggregated you want to deploy.

The method returns void.

Listing 9 presents the signature of the method used for EventDetection:

```
void removeEventDetectionNode(String eventDetectionNodeID)
```

Listing 9: the signature of the method removeEventDetectionNode

The method removeEventDetectionNode has the following parameters:

- String eventDetectionNodeID: the ID of the node we want to remove.

The method returns void.

The class EventDetectionNode has the following methods/functions/procedures:

- getListOfInputStreamNames: Method that returns the list of input streams.
- getEventDetectionLogic: Method inside which you will develop the CEP logics.

Listing 10 presents the signature of the method used for EventDetectionNode:

```
protected ArrayList<String> getListOfInputStreamNames()
```

Listing 10: the signature of the method getListOfInputStreamNames

The method getListOfInputStreamNames has the following parameters:

The method returns an ArrayList of strings representing the name of all the input streams.

Listing 11 presents the signature of the method used for EventDetectionNode:

```
public EPServiceProvider getEventDetectionLogic(EPServiceProvider epService)
```

Listing 11: the signature of the method getEventDetectionLogic

The method getEventDetectionLogic has the following parameters:

- epService: is the Esper object you want to refer to when creating your CEP logic.

The method returns Null.

3.7.7 Error handling

The log file of the component can be found in the same directory as the event.detection.jar under the name you used when running the component for the first time. In our example this is *EventDetectionLog.log*

Here is the list of errors which might appear during the deployment or the execution of the component.

Table 13: The end points of the Event detection component

Error name	Error description	Possible solution
Problems connecting to URI: amqp://guest:guest@ID:PORT	This error message means that the ED component cannot connect to GDI via AMQP connection	Check if the IP and PORT are correctly written
EventDetection: Cannot connect to the data bus	This error message means that the ED component could not create a new factory connection to the Message bus	Check the IP and PORT parameters
Error trying to remove all event streams from GDI.	This error means that the ED component could not access GDI or encountered problems while trying to remove all the event streams present at launch	Restart the application, if that does not resolve the problem try checking the connection to GDI. In the last case scenario, the GDI is not working correctly
Problems computing StreamDetails from UUID …	This error means that in the process of acquiring the streams	Check internet connection, check to see if the connection to Resource Manager is set

	description, something went wrong	correctly
Error registering new EventStream in GDI with UUID	This error means that there is a problem connecting/registering a new event stream to GDI	Check GDI connection parameters
Error while deploying event detection logic	This error message means that the inputStreams specified in the EventDetectionNode are not present in the eventDetectionLogicInputMaping.	Check the name of the inputStreams and the UUIDs
Error publishing an event on the databus for the sensor with UUID:...	This error means that there are some problems publishing a new event message on “events” echange	Check the message bus parameters
Error while cancelling the consumer with channelTag	This error message means that there were some problems while cancelling a consumer when the disconnectInputAdapter mehod was called	Check the UUID provided to the removeEventDetectionNode method

3.8 Twitter Event Detection

3.8.1 Component description

Having connected to the Twitter Streaming API, the CityPulse dedicated Data wrapper collects a live stream of data in the form of tweets and automatically detects the source language and translates the tweets to English to facilitate the data processing step. The data processing unit then uses a multi-view learning approach to extract events through connecting to the raw-tweet database. Given a tweet, the processing unit assigns it to one of the event classes from the pre-defined class set: {Transportation and Traffic, Weather, Cultural Event, Social Event, Sport and Activity, Health and Safety, Crime and Law, Food and Drink}.

The web interface facilitates the visualisation of the extracted city events on a Google map in near real-time. The interface is written in HTML and Javascript and composed of (a) Google map canvas layer on which the processed and annotated Tweets are displayed with their class-identical icons (b) a live London traffic layer from Google traffic API - code coloured paths on the map (c) a bar chart panel which presents the class distribution histogram of daily Tweets and (d) a panel for displaying Twitter timeline.

The map data is updating in 60s time windows by adding the past minute's Tweets to existing ones up to a 60-minutes time window. Meaning that, the map data will be updated on hourly bases. Clicking on each event a dialogue box is shown on the map which reveals the underlying Tweet content along with its time-stamp. The twitter user id and the name are anonymised for privacy purpose.

The web interface utilises JavaScript and HTML coding and reads the data from a CSV rest file of the live NLP processing component.

It can be used for the following types of scenarios (only if applicable):

- Scenario 1: visualizing twitter detected events and their underlying contents on a Google map
- Scenario 2: visualizing a live traffic layer

It cannot be used in the following situations (only if applicable):

- Scenario 3: It is live and accessible at all times as a social media dashboard of the city for social, cultural, weather, health & safety, law & crime, sport and transportation events.

3.8.2 Component location on GitHub

This component can be found at the following GitHub location: https://github.com/CityPulse/Twitter

3.8.3 Component prerequisite

The package needs the senna Convolutional Neural Network package to be installed in the main directory from: http://ronan.collobert.com/senna/
It also requires Conditional random field NER tagger which can be downloaded from: http://personal.ee.surrey.ac.uk/Personal/N.Farajidavar/Downloads.html
The following applications have to be installed before using the component:

- Twitter Stream API, https://dev.twitter.com/streaming/overview
- Google Translate API, https://cloud.google.com/translate/docs
- Google Map API, https://developers.google.com/maps/

3.8.4 CityPulse framework dependencies

The component in order to run properly needs to have access to the following CityPulse components: Twitter data wrapper

3.8.5 Component deployment procedure

The following steps have to be achieved in order to deploy the component:

- Step 1: Start the data collection component following the github instructions (%php filter-geo-Aarhis.php)
- Step 2: Start the NLP analysis component following the github instructions (%python Aarhus_V4.py)

3.8.6 Component programmatic APIs

The main class has the following methods/functions/procedures:

- Method 1: serializeEvent : to serialize the processed and detected events on the message bus
- Method 2: fix_SENNA_result: to align the PoS embedding results with the NER embedding
- Method 3: compte_level: to compute the detected event's level

- Method4: get_results: to finalize the two view results and make it ready for serialization
- Method 5: load_data_from_mysqlDB: to load twitter data from mysql database
- Method 6: check_end_of_day: to check the end of day for cleaning the database and preparing the component for the next day data

Listing 12 presents the signature of the Method 1:

```
<string> serializeEvent (eventID <string>, eventType<string> ,eventSource <string>,eventLevel <float>, eventLoc <float,float>,
eventCategory <string>,timestamp <datetime obj>)
```

Listing 12 the signature of the method 1:

The method has the following parameters:

- Event ID: which is the twitter id of the detected event
- Event Type: The event type
- Event source: is a constant string (Twitter)
- Event level: the computed event level
- Event location: the extracted location term
- Event category: the category representing the event class
- Time stamp: twitter time stamp

The method returns the string, response.

Listing 13 presents the signature of the Method 2:

```
<void> fix_SENNA_result(filename <string>,Sleep<string>):
```

Listing 13 the signature of the method 2:

The method has the following parameters:

- Filename: the filename where the PoS results are saved
- Sleep: the delay time for processing the results

The method returns void.

Listing 14 presents the signature of the Method 3:

```
<float> compute_level(temp_data <float>, threshold<float>):
```

Listing 14 the signature of the method 3:

The method 3 has the following parameters:

- Temp_data: temporal processed data in python dictionary format

- Threshold: geographical threshold for aggregating similar event reports

The method returns float level of the events.

Listing 15 presents the signature of the Method 4:

```
<string> get_result(NOW<datetime obj>,trans_df <string>,raw_df <string>,filename1<string>,filename2<string>,filename3<string>):
```

Listing 15 the signature of the method 4:

The method 4 has the following parameters:

- NOW: datetime object for time
- Tranf_df: string dataframe containing the translated raw data
- Raw_df: string dataframe containing the raw data for location extraction
- Filename1,2,3: 2 views of the processed trans_df + processed raw_df

The method returns a string of processed data.

Listing 16 presents the signature of the Method 5:

```
<string> load_data_from_mysqlDB(NOW <datetime obj>,step<datetime obj>,delay<daatetime obj>):
```

Listing 16 the signature of the method 5:

The method 5 has the following parameters:

- NOW: datetime object for time
- Step: datetime object for fetching the data from mysql databse
- Delay: datetime object to delay the fetching process according to the previous minute's processing time

Listing 17 presents the signature of the Method 6:

```
<string> Check_End_of_Day(current_time <datetime obj>,cursor<string>)
```

Listing 17 the signature of the method 6:

The method 6 has the following parameters:

- Current_time: datetime object for current time
- Cursor: string presenting the mysql cursor

The method returns a binary flag, which informs of the end of day.

3.9 Contextual Filtering

3.9.1 Component description

The main role of the Contextual Filtering component is to continuously identify and filter events that might affect the optimal results of the decision making task (performed by the Decision Support component). The users need to input their current context such as place of interest, filtering factors, and ranking factors. The Contextual Filtering will subscribe to events only in the place of interest and determine which event is critical to users based on filtering and ranking factors (see [CityPulse-D5.1], [CityPulse-D5.2]).

Known limitations:

- User Context Ontology: the current ontology about user's context includes only the activity of the user. The developer can extend this ontology for their specific scenarios.

3.9.2 Component location on GitHub

This component can be found at the following GitHub location:
https://github.com/CityPulse/DecisionSupport-ContextualFiltering

3.9.3 Component prerequisite

The following applications have to be installed before using the component:

- Clingo4 (available at: http://potassco.sourceforge.net)

3.9.4 CityPulse framework dependencies

The Contextual Filtering component needs to have access to the following CityPulse components:

- Geospatial Data Infrastructure (GDI)
- Event Detection

3.9.5 Component deployment procedure

The Contextual Filtering component and the Decision Support component are implemented in the same jar pakage as two different web-socket endpoints of the same server (we call it User-Centric Decsion Support server. Therefore, the Decision Support component is automatically deployed when the Contextual Filtering component is deployed and vice versa

The following steps have to be achieved in order to deploy the component

- Step 1: download the following resources provided at https://github.com/CityPulse/DecisionSupport-ContextualFiltering and place them in the same folder:
    - o jar package in folder */target* with title *CityPulseWP5-jar-with-dependencies.jar*
    - o folder *res*
- Step 2: edit the configuration file *config.properties* in the folder *res* which includes the following fields:
    - o *hostname*: the IP address of the server hosting the User-Centric Decision Support server.

- *port*: the main port used by the User-Centric Decision Support server
- *GDI_URI*: the URI of GDI component
- *eventRabbitURI*: the URI of rabbitMQ to subscribe events from the Event Detection component

- Step 3: run *CityPulseWP5-jar-with-dependencies.jar* to start the server.

3.9.6 Component programmatic APIs

Table 14 contains the list of classes which are provided by the Contextual Filtering component.

Table 14: The classes of the Contextual Filtering

Package name	Class name	Class scope
org.insight_centre.urq.citypulse.wp5.contextual_filtering	CityEventConsumer	This class implements the RabbitMq consumer in order to consume city events detected by the Event Detection component
	ContextualEventFilter	This class implements the Contextual Filter to identify the critical events based on the user's context
	ContextualEventRequestRewriter	This class implements the rewriter in order to map the ContextualEventRequest to Asp rules
citypulse.commons.contextual_filtering.contextual_event_request	ContextualEventRequest	Modeling the request for the Contextual Filtering component
citypulse.commons.contextual_filtering.user_context_ontology	UserContext	Modeling the context of the user

The class CityEventConsumer has the following methods:

CityEventConsumer

```
public CityEventConsumer(java.util.List<java.lang.String[]> exchange
Topics, ContextualEventFilter ceFilter)
```
This method starts a consumer

Parameters:
```
exchangeTopics - a list of exchange topics which used to subscribe
events
ceFilter - an instance of ContextualEventFilter
```

The class CityEventFilter has the following methods:

startCEF

```
public void startCEF(citypulse.commons.contextual_filtering.contextu
al_event_request.ContextualEventRequest contextualEventRequest)
```

This method starts the ContextualEventFilter

Parameters:

`contextualEventRequest` – the request of the ContextualEventFilter

stopCEF

```
public void stopCFF()
```

This method stops the ContextualEventFilter by closing the CityEventConsumer

subscribeCityEvents

```
public void subscribeCityEvents(citypulse.commons.contextual_filteri
ng.contextual_event_request.Place place)
```

This function connects to GeoDB in order to get information for subscribing events from MessageBus

Parameters:

`place - (of interest)`

performReasoning

```
public void performReasoning(java.util.List<citypulse.commons.contex
tual_filtering.city_event_ontology.CityEvent> cityEvents)
```

This method calls Clingo to peform reasoning over a list of new detected events in order to identify critical events

Parameters:

`cityEvents - a list of detected events`

The class ContextualEventRequestRewriter has the following methods:

getRules

```
public
final java.util.List<java.lang.String> getRules(citypulse.commons.co
ntextual_filtering.contextual_event_request.ContextualEventRequest r
equest)
```

This method maps a contextual event request to ASP rules

Parameters:

`request – the contextual event request`

Returns:

`a list of rules in ASP format`

3.9.7 Component APIs

Table 15 contains the websocket endpoint which can be used to interact with the Contextual Filtering.

Table 15: The endpoint of the Contextual Filtering component

End point name	End point scope
Contextual Filtering websocket server endpoint	The endpoint for registering the ContextualEventRequest

Listing 18 presents the signature of the end point used for connecting to the Contextual Filtering component. Once the connection is established, the websocket server accepts a message wrapped from an *citypulse.commons.contextual_filtering.contextual_event_request* as an input.

```
ws://[hostIP]:[port]/websockets/contextual_events_request
```

Listing 18: the signature of the Contextual Filtering end point

The end point returns the critical events as JSON messages wrapped from *citypulse.commons.contextual_filtering.city_event_ontology.CriticalEventResults*

3.9.8 Error handling

The log file of the component can be found at ./logs/<log_file_name>.log

Table 16 presents the list of errors which might appear during the deployment or the execution of the component.

Table 16: The end points of the component Contextual Filtering

Error name	Error description	Possible solution
Invalid input format	Cannot parse the input message, identified by the message starting with "com.google.gson.JsonSyntaxException".	Check the input format and content.
Fail to connect to other components	Cannot connect to other component, identified by the message starting with "org.postgresql.util.PSQLException: Connection to ... refused"	Check the URIs in the config.properties file in the res folder or make sure that other components are running

3.10 Decision Support

3.10.1 Component description

The main role of the Decision Support component is to enable reactive decision support functionalities to be easily deployed, providing the most suitable answers at the right time by utilizing contextual information available as background knowledge, user patterns and preferences from the application as well as real-time events. The reasoning capabilities needed to support users in making better decisions require handling incomplete, diverse input, as well as constraints and preferences in the deduction process. This expressivity in the Decision Support component is achieved by using a declarative non-monotonic logic reasoning approach based on Answer Set Programming. The Decision Support component produces a set of answers to the reasoning request that satisfy all user's requirements and preferences in the best possible way. These solutions are computed by applying sets of rules deployed as scenario-driven decision support modules. We currently support three different types of decision support modules, covering a broad range of application scenarios (see [CityPulse-D5.2]).

- Scenario 1 (Routing APIs): It provides the best solution(s) for a routing task. The users not only identify starting point and ending point but are also able to provide various selection criteria (including constraints and preferences) in order to select the optimal routes.
- Scenario 2 (Parking space APIs): It provides the best selection among a set of available parking places based on optimisation criteria, constraints and preferences of the users.

3.10.2 Component location on GitHub

This component can be found at the following GitHub location:
https://github.com/CityPulse/DecisionSupport-ContextualFiltering

3.10.3 Component prerequisite

The following applications have to be installed before using the component:

- Clingo4 (available at: http://potassco.sourceforge.net)
- Python 2.7 and package *interruptingcow* (Installation: $ pip install interruptingcow)

3.10.4 CityPulse framework dependencies

The Decision Support component in order to run properly needs to have access to the following CityPulse components:

- Routing component
- Triple store
- GDI
- Data Federation

3.10.5 Component deployment procedure

The Decision Support component and the Contextual Filtering component are implemented in the same jar pakage as two different web-socket endpoints of the same server (we call it User-Centric Decsion Support server. Therefore, the Contextual Filtering component is automatically deployed when the Decision Support component is deployed and vice versa.

The following steps have to be achieved in order to deploy the component:

- Step 1: download the following resources provided at https://github.com/CityPulse/DecisionSupport-ContextualFiltering and place them in same folder:
    - jar package in folder */target* with title *CityPulseWP5-jar-with-dependencies.jar*
    - folder *res*
- Step 2: edit the configuration file *config.properties* in folder *res* which includes the following fields:
    - *hostname*: the IP address of the server hosting the User-Centric Decsion Support server.
    - *port*: the main port used by the User-Centric Decsion Support server
    - *routing_uri*: the uri to access the Routing component (for Routing APIs only)
    - *data_federation_uri*: the uri to access the Data Federation component
    - *knowledge_base_uri*: the uri to access the triple store
    - *GDI_URI*: the URI of GDI component
    - *clingo*: the path of Clingo
- Step 3: run *CityPulseWP5-jar-with-dependencies.jar* to start the server.

3.10.6 Component programmatic APIs

Table 17 contains the list of classes which can be used during the development of Decision Support component

Table 17: The classes of the Decision support component

Package name	Class name	Class scope
org.insight_centre.urq.citypulse.wp5.decision_support_system	CoreEngine	This class manages the communication to the ASP solver (Clingo) using the EmbASP framework
	DSManager	The main class of the Decision Support System
	RequestRewriter	This class derives logic rules from a Reasoning Request
citypulse.commons.reasoning_request	ReasoningRequest	Modeling the request for the Decision Support component
	Answer	This abstract class models the answer of the Decision Support component

The class CoreEngine has the following method:

- *performReasoning*: Invokes the ASP solver and returns the answers obtained

Listing 19 presents the signature of the method used for *performReasoning*

```
citypulse.commons.reasoning_request.Answers performReasoning(citypulse.commons.reasoning_request.ReasoningRequest reason
ingRequest, java.util.List<java.lang.String> rules)
```

Listing 19: the signature of the method *performReasoning*

The method *performReasoning* has the following parameters:

- citypulse.commons.reasoning_request.ReasoningRequest reasoningRequest: a request to perform reasoning
- java.util.List<java.lang.String> rules: a list of logic rules which automatically mapped from the reasoning request by the RequestRewriter

The method returns the answers

The class DSManager has the following method:

- *startDSS*: starts an instance of the Decision Support system

Listing 20 presents the signature of the method used for *startDSS*

```
citypulse.commons.reasoning_request.Answers startDSS(citypulse.commons.reasoning_request.ReasoningRequest reasoningReques
t)
```

Listing 20: the signature of the method *startDSS*

The method *startDSS* has the following parameter:

- citypulse.commons.reasoning_request.ReasoningRequest reasoningRequest: a request to perform reasoning

The method returns the answers

The class RequestRewriter has the following method:

- *getRules*: maps a reasoning request into logical rules

Listing 21 presents the signature of the method used for *getRules*

```
java.util.List<java.lang.String> getRules(citypulse.commons.reasoning_request.ReasoningRequest request)
```

Listing 21: the signature of the method *getRules*

The method *startDSS* has the following parameter:

- citypulse.commons.reasoning_request.ReasoningRequest reasoningRequest: a request to perform reasoning

The method returns logic rules.

3.10.7 Component APIs

Table 18 contains the websocket endpoint which can be used to interact with the Decision Support.

Table 18: The endpoint of the Decision Support component

End point name	End point scope
Decision Support websocket server endpoint	The endpoint for registering the ReasoningRequest

Listing 22 presents the signature of the end point used for connecting to the Decision Support component. Once the connection is established, the websocket server accepts a message wrapped from an *citypulse.commons.reasoning_request.ReasoningRequest* as an input.

```
ws://[hostIP]:[port]/websockets/reasoning_request
```

Listing 22: the signature of the Decision Support endpoint

The end point returns the answers as JSON messages wrapped from citypulse.commons.reasoning_request.Answers

3.10.8 Error handling

The log file of the component can be found ./logs/<log_file_name>.log

Table 19 presents the list of errors which might appear during the deployment or the execution of the component.

Table 19: The end points of the Decision Support

Error name	Error description	Possible solution
Invalid input format	Cannot parse the input message, identified by the message starting with " com.google.gson.JsonSyntaxException".	Check the input format and content.
Fail to connect to other components	Cannot connect to other component, identified by the message starting with "org.postgresql.util.PSQLException: Connection to ... refused"	Check the URIs in the config.properties file in res folder or make sure that other components are running
Fail to get information from external	Can not get information from external components (such Routing, KnowledgeBase, DataFederation, ...)	Check the status of the corresponding external components

components within amount of time	within amount of time (60seconds), idenfitied by messages starting with: "ERROR get_routes ..." "ERROR get_parking_spaces ..."	
Fail to call Clingo	Can not trigger Clingo, identified by message starting with: "java.io.IOException: Cannot run program "... /clingo": error=2, No such file or directory"	Check the directory of Clingo in config.properties file

3.11 Fault Recovery

3.11.1 Component description

The Fault recovery component ensures the continuous and proper operation of a CityPulse enabled application by generating estimated values for the data stream when the quality drops or it has temporally missing observations. When the quality of the data stream is low for a longer time period, an alternative data source has to be selected. The selection can be performed automatically by the technical adaptation component. In other words, the technical adaptation process does not have to be triggered if the QoI of a stream is low only for a short period of time because the Fault recovery component provides estimated values.
The component can be used for the following types of scenarios:

- Scenario 1: estimate missing values from time series.

It cannot be used in the following situations:

- Scenario 2: Not working for time series with categorical values.
- Scenario 3: Not working if the time series does not have affixed interval for delivering the observations.

3.11.2 Component location on GitHub

This component can be found at the following GitHub location:

https://github.com/CityPulse/FaultRecovery

3.11.3 Component prerequisite

The following applications have to be installed before using the component:

- Python 2.7: https://www.python.org/download/releases/2.7
- SciLab library for Python

3.11.4 CityPulse framework dependencies

The Fault Recivery component in order to run properly needs to have access to the following CityPulse components:

- Data wrapper

3.11.5 Component deployment procedure

It is already integrated in the data wrapper.

3.11.6 Component programmatic APIs

Fault Recovery component contains that list of classes which can be used during the development:

Table 20: The classes of the Fault recovery component

Class name	Class scope
FaultRecovery	Generates estimated values for the data stream when the quality drops or it has temporally missing observations

The class FaultRecovery has the following methods/functions/procedures:

- __init__: The constructor method that will initialize the Fault Recovery component.
- addValidMeasurement: Method used for reporting a new valid measurement.
- reportInvalidMeasurement: Method used for reporting a new invalid measurement.
- getEstimation: Method used for optaining the estimations.
- isReady: Method that checks if the reference data set contains enough entries intorde to generate a prediction

Listing 23 presents the signature of the method __init__:

```
def __init__(self)
```

Listing 23: the signature of the method __init__

The method __init__ has the following parameters: none

The method returns void.

Listing 24 presents the signature of the method used for addValidMeasurement:

```
def addValidMeasurement(self,measurement)
```

Listing 24: the signature of the method addValidMeasurement

The method addValidMeasurement has the following parameters:

- measurement: is an integer or double value representig the latest measurement

The method returns void.

Listing 25 presents the signature of the method used for reportInvalidMeasurement:

```
def reportInvalidMeasurement(self)
```

Listing 25: the signature of the method reportInvalidMeasurement

The method reportInvalidMeasurement has the following parameters: none.

The method returns void.

Listing 26 presents the signature of the method used for getEstimation:

```
def getEstimation(self)
```

Listing 26: the signature of the method getEstimation

The method getEstimation has the following parameters: none

The method returns a double value representing the estimation.

Listing 27 presents the signature of the method used for isReady:

```
def isReady(self)
```

Listing 27: the signature of the method isReady

The method isReady has the following parameters: none

The method returns a Boolean value.

3.12 Conflict resolution

3.12.1 Component description

The Conflict Resolution component is used to resolve conflicting sensor values, i.e., two or more functionally equivalent sensors reporting different observation values at the same time. In such cases, the users needs to determine which sensor reading is more believable or trustworthy. The conflict resolution component leverages the QoS of sensor streams captured by techniques described in [CityPulse-D4.2] as well as the QoS aggregation and estimation methods described in [CityPulse-D3.2] to select better stream(s) and notify the users. There is more than one way to decide which stream is more reliable, in addition to the built-in algorithms developed in the component (see [CityPulse-D4.2] for details), the conflict resolution component provides extension mechanisms to allow a third-party developer to integrate his solutions easily.

The component can be used for the following types of scenarios:

- Scenario 1: two temperature sensors deployed in the same room reporting different air temperature values at the same time, and the user needs to know which value is more reliable.
- Scenario 2: a set of air pollution sensors deployed in the same area giving different readings on air quality.

It cannot be used in the situations where the conflicting values are produced by sensors observing different feature-of-interest, e.g., deployed in different areas, or the values are captured at different times.

3.12.2 Component location on GitHub

This component can be found at the following GitHub location: https://github.com/CityPulse/ACEIS

3.12.3 Component prerequisite

- See prerequisite for the data federation component (see section 3.6.).

3.12.4 CityPulse framework dependencies

- The technical adaptation component is deployed together with the data federation.
- See dependencies for the data federation component (see section 3.6.).

3.12.5 Component deployment procedure

- See deployment procedure for the data federation component (see section 3.6.).

3.12.6 Component programmatic APIs

Table 21 contains the list of classes which can be used during the development of the conflict resolution component.

Table 21: The classes of the conflict resolution component

Package name	Class name	Class scope
main.java.citypulse.commons.conflict_resolution	ConflictResolutionRequest	Modelling the conflict resolution request.
	ConflictResolutionResponse	Modelling the conflict resolution response.
org.insight_centre.	Solver	Abstract class for conflict resolution solver.

aceis.conflictresolution	SimpleSolver	An implementation of a solver.
org.insight_centre.citypulse.server	ConflictResolutionServerEndpoint	The server endpoint for handling conflicts.

The class ConflictResolutionRequest has the following methods:

ConflictResolutionRequest

```
public ConflictResolutionRequest(java.util.List<java.lang.String> streamIds
)
```
Creates an instance of conflict resolution request

Parameters:

`streamIds` - a list of conflicting stream ids

getStreamIds

```
public java.util.List<java.lang.String> getStreamIds()
```
Retrieves the stream ids

Returns:

stream ids in the request

setStreamIds

```
public void setStreamIds(java.util.List<java.lang.String> streamIds)
```
Sets the stream ids

Parameters:

`streamIds` - the conflicting stream ids

The class ConflictResolutionResponse has the following methods:

ConflictResolutionResponse

```
public ConflictResolutionResponse(java.lang.String streamId)
```
Creates an instance of conflict resolution response

Parameters:

`streamId` - the best stream id

getStreamId

```
public java.lang.String getStreamId()
```
Gets the best stream id

Returns:

best stream id

setStreamId

`public void setStreamId(java.lang.String streamId)`
Sets the best stream id

Parameters:

`streamId` - best stream id

The class Solver has the following methods:

Solver

`public Solver(ACEISEngine engine)`
Creates a solver for an ACEIS engine

Parameters:

`engine` - the ACEIS engine associated with the solver

solve

```
public
abstract java.lang.String solve(java.util.List<java.lang.String> streamIds)
throws java.lang.Exception
```
Chooses the best stream id when they report conflicts

Parameters:

`streamIds` - list of stream ids that have conflicts

Returns:

most reliable stream id, "" if cannot decide

Throws:

`java.lang.Exception`

The class SimpleSolver has the following methods:

SimpleSolver

`public SimpleSolver(ACEISEngine engine)`
Creates a solver for an ACEIS engine

Parameters:

`engine` - the ACEIS engine associated with the simple solver

solve

`public java.lang.String solve(java.util.List<java.lang.String> streamIds) throws java.lang.Exception`

Description copied from class: **`Solver`**

Chooses the best stream id when they report conflicts

Specified by:

`solve` in class `Solver`

Parameters:

`streamIds` - list of stream ids that have conflicts

Returns:

most reliable stream id, "" if cannot decide

Throws:

`java.lang.Exception`

getMode

`public org.insight_centre.aceis.conflictresolution.SimpleSolver.solverMode getMode()`

Gets the solver mode

Returns:

the mode of the solver

setMode

`public void setMode(org.insight_centre.aceis.conflictresolution.SimpleSolver.solverMode mode)`

Sets the solver mode

Parameters:

`mode` - solver mode, could be "globalRanking", i.e., comparing global QoS utility comparison of the streams, or "groupedRanking", which compares the aggregated stream QoS utilities over groups of streams which have the same observation values

The class ConflictResolutionServerEndpoint has the following methods:

ConflictResolutionServerEndpoint

`public ConflictResolutionServerEndpoint()`

Creates an instance of the conflict resolution server endpoint

onMessage

```
public java.lang.String onMessage(java.lang.String message,
javax.websocket.Session session)
throws java.lang.Exception
```

When a new conflict resolution request is received, trigger solving process

Parameters:

`message` - message received from client

`session` - websocket session established with the client

Returns:

Throws:

`java.lang.Exception`

3.12.7 Component APIs

Table 22 contains the list of websocket endpoint which can be used to interact with the component.

Table 22: The end points of the conflict resolution component

End point name	End point scope
Conflict Resolution websocket server endpoint	The endpoint for solving sensor reading conflicts.

Listing 28 presents the signature of the end point used for connecting to the conflict resolution websocket server. Once the connection is established, the websocket server accepts a message wrapped from an main.java.citypulse.commons.conflict_resolution.ConflictResolutionRequest as input. The conflict resolution component decides which stream in the request is more reliable, it produces the result as a Json messages wrapped from main.java.citypulse.commons.conflict_resolution.ConflictResolutionResponse.

```
ws://[hostIP]:8020/ConflictResolution
```

Listing 28: the url example of the conflict resolution end point.

3.12.8 Error handling

The log file of the component can be found at EC-log.log

Table 23 present the list of errors which might appear during the deployment or the execution of the component.

Table 23: The error handling of the conflict resolution component

Error name	Error description	Possible solution
IOException during websocket session	Triggered by unexpected client connection loss etc.	Restart component.
Invalid input format	Cannot parse the input message, identified by the a warning log starts with "FAULT: JsonSyntaxException".	Check the input format and content.

3.13 Quality Monitoring

3.13.1 Component description

The Quality Monitoring is split into two different layers: the Atomic and the Composite Monitoring. This document focuses on the Atomic Monitoring, which is integrated and part of the Resource management (see 3.3). For the Composite Monitoring no API is available.

The Atomic Monitoring follows an application independent approach, which enables it to work for every scenario. The only information, which is needed by the Atomic Monitoring, is a description of the used sensors or streams that is described in the following:

```
sensordescription.maxLatency = 2
sensordescription.updateInterval = 60
sensordescription.fields = ["v1", "v2", “v3”]
sensordescription.field.v1.min = 0
sensordescription.field.v1.max = "@v3"
sensordescription.field.v1.dataType = "int"
sensordescription.field.v2.dataType = "datetime"
sensordescription.field.v2.format = "%Y-%m-%dT%H:%M:%S"
sensordescription.field.v3.dataType = “int”
…
```

The description has to be provided by the stream provider. The provider has to annotate the maximum latency the provided data could have and the update interval in which the stream delivers new data. The following lines describe different data fields the stream provides. The first field, v1, is of the data type “int” and has a minimum value of 0. The maximum value is the value provided by another field, v3. This is stated by the “@fieldname”. The second field, v2, is a date type, which can have a specific data format. Supported by the Atomic Monitoring are Python`s basic data types.

With the help of the sensor description and the application independent approach the Atomic Monitoring can be used for every scenario.

Known limitations: Special data types might not work or require additional implementation work.

3.13.2 Component location on GitHub

As this component is directly integrated into the Resource management it can be found at https://github.com/CityPulse/CP_Resourcemanagement.git in the subfolder “Quality”.

3.13.3 Component installation

As the Atomic Monitoring is part of the Resource management, no separate installation is required. For further prerequisites, dependencies and deployment procedure see the Resource management guide (section 3.3).

3.13.4 Component APIs

The Atomic Monitoring is completely integrated into the Resource management. Therefore the available API functions are described in section 3.3.7.

3.13.5 Error handling

Error messages of the Atomic Monitoring are written into the "virtualisation.log" file of the Resource management. They have an informative character and the monitoring will work even if these messages occur, but it might be helpful for find misconfigurations for specific streams.

Table 24 presents the list of errors, which might appear during the execution of the component.

Table 24: The possible error messages for the Quality Monitoring

Error name	Error description	Possible solution
ReputationSystem: checkTimeRelevantMetrics called for Stream http://ict-citypulse.eu/sensor/XY ReputationSystem: There was no update, lets punish!	This first message indicates the timer based check for quality and is no error message. If this message is followed by the second one, the timer based check found out that there was no update for the stream in the annotated update interval. Therefore, the QoI for the stream is lowered. This is only a warning that no new observation could be received. As a consequence, the QoI for this data stream will be reduced.	Check if the update interval within the sensor description is right and/or check if the stream pushes data or is pulled in the annotated interval.
Completeness missing fields: 1 (*fieldname*)	Indicates that the value for the "fieldname" is missing.	Check if the stream sends out all annotated fields or if the sensor description is wrong.
Completeness wrong fields: 1 (*fieldname*)	Indicates that the value for the "fieldname" is wrong, e.g. "NA", "NULL", or "".	Check if the stream sends out all annotated fields with real values.
Correctness wrong fields: 1 (*fieldname*)	Indicates that the value for the "fieldname" is not within the specified range or data type.	Check if the stream sends out annotated fields with values within the range and data type annotated in the sensor description.

3.14 Geo-spatial data base

3.14.1 Component description

To overcome the limitations of semantic data storage and processing, a gesospatial data infrastructure is used to support geospatial analysis and implement spatiotemporal reasoning processes. The GDI is used to analyse dependencies between infrastructure, sensors and events. This goal is achieved by utilising an optimised search index for geospatial objects. Since most city sites and elements (e.g., streets or places) in cities can't just be described as a coordinate (e.g. longitude, latitude in the WGS84 system of GPS) it is necessary to support more complex geometric types like Linestrings and Polygons to represent roads, public places, polluted areas, etc. Processing these kinds of complex geometries needs a qualified geospatial indexing process to find nearby objects, process overlapping areas and generate adapted routing graphs.

CityPulse utilises a PostgreSQL database with a PostGIS extension to enable those technologies in the project. Using the Generic Index Structure (GIST) overcomes the limitation of the default index type B-Tree [Hwang 2003]. B-Tree indices are not lossy (inexact) in the way a GIST index can be. This means that while the GIST index only indexes the bounding box of the geometry, the B-Tree must index the entire geometry, which can often be larger than the index can cope with. Therefore a first selection of query-results can be found by simply using the min/max longitude instead of always comparing the full values/geometries. Figure 12 depicts the architecture of the Geospatial Data Infrastructure (GDI) connection.

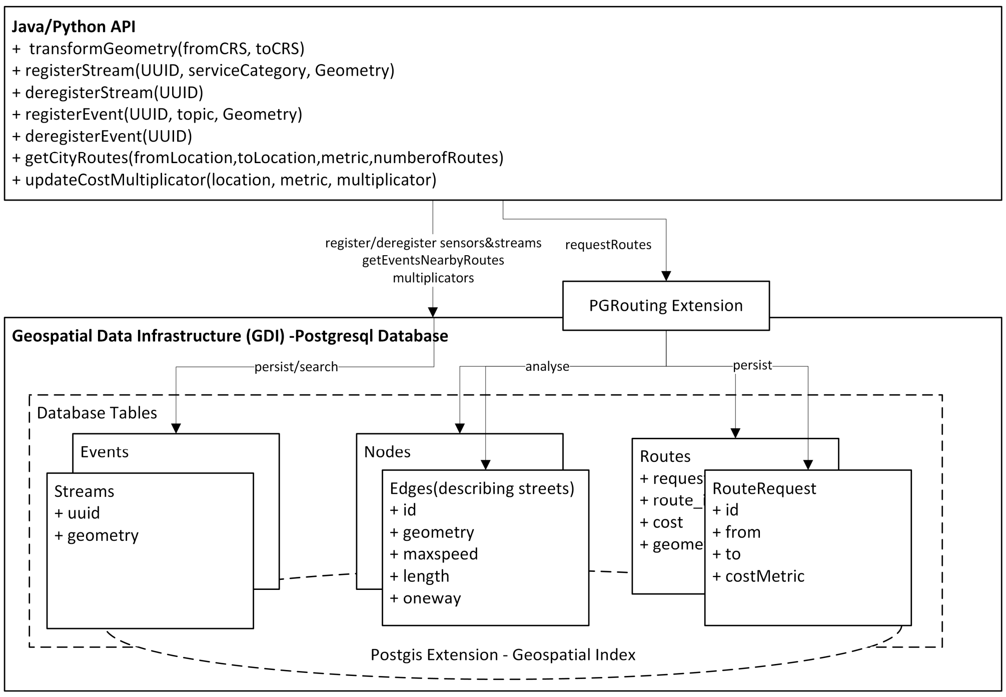

Java/Python API
+ transformGeometry(fromCRS, toCRS)
+ registerStream(UUID, serviceCategory, Geometry)
+ deregisterStream(UUID)
+ registerEvent(UUID, topic, Geometry)
+ deregisterEvent(UUID)
+ getCityRoutes(fromLocation,toLocation,metric,numberofRoutes)
+ updateCostMultiplicator(location, metric, multiplicator)
register/deregister sensors&streams
getEventsNearbyRoutes
multiplicators
requestRoutes
PGRouting Extension
Geospatial Data Infrastructure (GDI) -Postgresql Database
persist/search
analyse
persist
Database Tables
Events
Streams
+ uuid
+ geometry
Nodes
Edges(describing streets)
+ id
+ geometry
+ maxspeed
+ length
+ oneway
Routes
+ reques
+ route_
+ cost
+ geome
RouteRequest
+ id
+ from
+ to
+ costMetric
Postgis Extension - Geospatial Index

Figure 12 - Overview of the Geospatial Data Infrastructure

3.14.2 Component location on GitHub

This component can be found at the following GitHub location: https://github.com/CityPulse/GDI

3.14.3 Component prerequisite

The following applications have to be installed before using the component:

- PostgreSQL Database http://www.postgresql.org
- GESO Library: https://trac.osgeo.org/geos/
- Postgis Spatial Extension: http://postgis.net
- pgRouting Library: http://pgrouting.org
- Geospatial Infrastructure from Openstreetmap: e.g. http://download.geofabrik.de/europe/denmark.html

3.14.4 Component programmatic APIs

The Java API is described in the following. Since the component is mainly based on SQL Statements it is possible to easily adapt it to further languages.

Table 25: The classes of the Geo-spatial data base component

Class name	Class scope
R –API (no class)	Websocket access for decision support
eu.citypulse.uaso.gdiclient.CpGdiInterface	Geospatial Data Infrastructure (GDI) main interface for outside usage
eu.citypulse.uaso.gdiclient.CpClientExample	Test and Example class for Geospatial Data Infrastructure(GDI) client of CityPulse
eu.citypulse.uaso.gdiclient.routes.CpRoute	Output class of a single route
eu.citypulse.uaso.gdiclient.routes.CpRouteRequest	One request of a set of routes

- Method 1: short description
- closeConnection(): discard database connection
- closeEvent(java.util.UUID uuid, java.sql.Timestamp closeTime) :Set a stop / end time of the validity of the event
- deregisterEvent(java.util.UUID uuid): Deregister an event / Remove it from the GDI
- getAllEventStreams() : Get all Event Streams
  getCityRoutes(CpRouteRequest cprr, int count), getCityRoutes(double fromLong, double fromLat, double toLong, double toLat, java.lang.String costMetric, int count), getCityRoutes(java.lang.String fromWkt, java.lang.String toWkt, java.lang.String costMetric, int count) :
  Request a number of routes between two points
- getEventsForRoute(CpRoute cpr, java.sql.Timestamp fromTime, int duration_s, double buffer_m) : Get Events nearby the route which were created in duration_s before fromTime timestamp and have not been closed.

- getEventStreamsForRoute(CpRoute cpr, double buffer_m): Get Event streams in a distance of a specific route
- getEventStreamsForRoute(org.postgis.PGgeometry pgeo, double buffer_m) :Get Event streams in a distance of a specific route
- getEventStreamsForRoute(java.lang.String wktGeometry, int epsg, double buffer_m): Get Event streams in a distance of a specific route
- getNearestNodeID(double lon, double lat): Get the nearest node it in the graph for routing
- getnNextLocationsByAmenity(double lon, double lat, java.lang.String amenity, int count, int repetitionId): return nearby locations by euclidean distance for a special type of place (e.g. hospital, parking space or ATM)
- getSensorsForRoute(CpRoute cpr, double buffer_m): Get Sensor streams in a distance of a specific route
- registerEvent(java.util.UUID uuid, java.lang.String topic, java.sql.Timestamp event_time, double lon, double lat): Register a new Event in the GDI
- registerEvent(java.util.UUID uuid, java.lang.String topic, java.sql.Timestamp event_time, double x, double y, int epsg): Register a new Event in the GDI
- registerEvent(java.util.UUID uuid, java.lang.String topic, java.sql.Timestamp event_time, org.postgis.PGgeometry geom): Register a new Event in the GDI
- registerEvent(java.util.UUID uuid, java.lang.String topic, java.sql.Timestamp event_time, java.lang.String wkt, int epsg): Register a new Event in the GDI
- registerEventStream(java.util.UUID uuid, java.lang.String routingKey, double lon, double lat): Register a new EventStream in the GDI
- registerEventStream(java.util.UUID uuid, java.lang.String routingKey, double x, double y, int epsg): Register a new EventStream in the GDI
- registerEventStream(java.util.UUID uuid, java.lang.String routingKey, java.lang.String wkt, int epsg): Register a new EventStream in the GDI
- registerStream(java.util.UUID uuid, java.lang.String sensorId, java.lang.String serviceCategory, double lon, double lat): Register a new sensor/stream
- registerStream(java.util.UUID uuid, java.lang.String sensorId, java.lang.String serviceCategory, double x, double y, int epsg): Register a new sensor/stream and convert coordinates to WGS84 in the database
- registerStream(java.util.UUID uuid, java.lang.String sensorId, java.lang.String serviceCategory, java.lang.String wkt, int epsg): Register a new sensor/stream and convert Geometry to WGS84 in the database
- removeAllEventStream(): Remove all Event Streams from DB
- removeAllStreams(): Remove all Streams from DB
- removeEventStream(java.util.UUID uuid): Deregister previously registered EventStream (delete it from the database)
- removeStream(java.util.UUID uuid): Deregister/Remove previously registered Sensor/stream (delete it from the database)
- resetCostMultiplicators(): Reset all cost multiplcators for all metrics to 1.0

- resetCostMultiplicators(java.lang.String costMetric): Reset cost multiplicators of a specific type
- updateCostMultiplicatorArea(double minLon, double minLat, double maxLon, double maxLat, java.lang.String costMetric, double costMultiplicatorValue): Update the cost multiplicator in an rectengular area specified by minimum and maximum bounds
- updateCostMultiplicatorRadial(double lon, double lat, double radius_m, java.lang.String costMetric, double costMultiplicatorValue): Update one of the costMultiplicator Layers around a certain point, e.g.

Constructor Detail:

- ***CpGdiInterface***

```
public CpGdiInterface() throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

Initialisation of the database connection and prepared statements with localhost and port 5432

Throws:

```
java.lang.ClassNotFoundException - If pg-driver is not available
java.sql.SQLException - If it cant be connected...
```

- ***CpGdiInterface***

```
public CpGdiInterface(java.lang.String hostname, int port)
throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

Initialisation of the database connection and prepared statements.

Parameters:

```
hostname - Hostname of the database (localhost)
port - Port of the database (depends on if you are using a tunnel
or connect directly standard port is 5432)
```

Throws:

```
java.lang.ClassNotFoundException - If pg-driver is not available
java.sql.SQLException - If it cant be connected...
```

Method Details of CpGdiInterface:

- ***closeConnection***

```
public void closeConnection()                      throws
java.sql.SQLException
```

discard database connection

Throws:

```
java.sql.SQLException - If connection can't be closed
```

- ***getPointGeometry***

```
public static org.postgis.PGgeometry getPointGeometry(double lon,
double lat)  throws java.sql.SQLException
```

Throws:

```
java.sql.SQLException
```

- ***getPointGeometry***

```
public static org.postgis.PGgeometry getPointGeometry(double x,
double y, int epsg) throws java.sql.SQLException
```

Throws:

```
java.sql.SQLException
```

- ***getGeometry***

```
public       static org.postgis.PGgeometry getGeometry(java.lang.String
wktWithoutEpsg,  int epsg)  throws java.sql.SQLException
```

Throws:

```
java.sql.SQLException
```

- ***getLineGeometry***

```
public static org.postgis.PGgeometry getLineGeometry(double fromX,
double fromY,  double toX,  double toY, int epsg)
throws java.sql.SQLException
```

Throws:

```
java.sql.SQLException
```

- ***simpleQuery***

```
public int simpleQuery()
```

just a test function

Returns:

```
1 if gdi is available
```

- ***registerStream***

```
public boolean registerStream(java.util.UUID uuid,
java.lang.String sensorId, java.lang.String serviceCategory,
double lon, double lat) throws java.sql.SQLException
```

Register a new sensor/stream

Parameters:

`uuid - UUID of sensor/stream`

`sensorId - Internal id for debg purposes (eg. the number of the aarhus traffic sensors, or parkinggarage name)`

`serviceCategory - for extended searches like (return all parking garages)`

`lon - longitude in wgs84`

`lat - latitude in wgs84`

Returns:

`true if sensor was registeres`

Throws:

`java.sql.SQLException - If something went wrong`

- ***registerStream***

```
public boolean registerStream(java.util.UUID uuid,
java.lang.String sensorId, java.lang.String serviceCategory,
double x,  double y, int epsg) throws java.sql.SQLException
```

Register a new sensor/stream and convert coordinates to WGS84 in the database

Parameters:

`uuid - UUID of sensor/stream`

`sensorId - Internal id for debug purposes (eg. the number of the aarhus traffic sensors, or parkinggarage name)`

`serviceCategory - for extended searches like (return all parking garages)`

`x - 1st WKT-coordinate-Value of reference system`

`y - 2nd WKT-coordinate-Value of reference system`

`epsg - EPSG of coordinate reference system (eg. 4326 for WGS84/GPS, or 31700 for stereo70, http://spatialreference.org/ref/epsg/wgs-84/)`

Returns:

`true if sensor was registeres`

Throws:

`java.sql.SQLException - If something went wrong`

- ***registerStream***

```
public boolean registerStream(java.util.UUID uuid,
java.lang.String sensorId, java.lang.String serviceCategory,
java.lang.String wkt, int epsg) throws java.sql.SQLException
```

Register a new sensor/stream and convert Geometry to WGS84 in the database

Parameters:

`uuid` - UUID of sensor/stream

`sensorId` - Internal id for debug purposes (eg. the number of the aarhus traffic sensors, or parkinggarage name)

`serviceCategory` - for extended searches like (return all parking garages)

`wkt` - WellKnownText geometry description without Coordinate Reference System

`epsg` - EPSG of coordinate reference system (eg. 4326 for WGS84/GPS, or 31700 for stereo70, http://spatialreference.org/ref/epsg/wgs-84/)

Returns:

true if sensor was registered

Throws:

`java.sql.SQLException` - If something went wrong

- ***removeStream***

```
public boolean removeStream(java.util.UUID uuid)
throws java.sql.SQLException
```

Deregister/Remove previously registered Sensor/stream (delete it from the database)

Parameters:

`uuid` - uuid of stream/sensor

Returns:

true if entry was deleted

Throws:

`java.sql.SQLException` - If something went wrong

- ***removeAllStreams***

```
public boolean removeAllStreams()                          throws
java.sql.SQLException
```

Throws:

`java.sql.SQLException`

- ***registerEventStream***

```
public boolean registerEventStream(java.util.UUID uuid,
java.lang.String routingKey, double lon, double lat)  throws
java.sql.SQLException
```

Register a new EventStream in the GDI

Parameters:

`uuid - Event Stream UUID`

`routingKey - Event routingKey`

`lon - longitude in wgs84`

`lat - latitude in wgs84`

Returns:

`true if added`

Throws:

`java.sql.SQLException - If something went wrong`

- ***registerEventStream***

```
public boolean registerEventStream(java.util.UUID uuid,
java.lang.String routingKey, double x, double y,
int epsg) throws java.sql.SQLException
```

Register a new EventStream in the GDI

Parameters:

`uuid - Event Stream UUID`

`routingKey - Event routingKey`

`x - 1st WKT-coordinate-Value of reference system`

`y - 2nd WKT-coordinate-Value of reference system`

```
epsg - EPSG of coordinate reference system (eg. 4326 for
WGS84/GPS, or 31700 for stereo70,
http://spatialreference.org/ref/epsg/wgs-84/)
```

Returns:

`true if EventStream was registered`

Throws:

`java.sql.SQLException - If something went wrong`

- ***registerEventStream***

```
public boolean registerEventStream(java.util.UUID uuid,
java.lang.String routingKey, java.lang.String wkt,  int epsg)
throws java.sql.SQLException
```

Register a new EventStream in the GDI

Parameters:

`uuid` - Event Stream UUID

`routingKey` - Event routingKey

`wkt` - WellKnownText geometry description without Coordinate Reference System

`epsg` - EPSG of coordinate reference system (eg. 4326 for WGS84/GPS, or 31700 for stereo70, http://spatialreference.org/ref/epsg/wgs-84/)

Returns:

true if sensor was registered

Throws:

`java.sql.SQLException` - If something went wrong

- ***removeEventStream***

```
public boolean removeEventStream(java.util.UUID uuid)
throws java.sql.SQLException
```

Deregister previously registered EventStream (delete it from the database)

Parameters:

`uuid` - uuid of stream/sensor

Returns:

true if entry was deleted

Throws:

`java.sql.SQLException` - If something went wrong

- ***removeAllEventStream***

```
public boolean removeAllEventStream()
throws java.sql.SQLException
```

Throws:

`java.sql.SQLException`

- ***registerEvent***

```
public boolean registerEvent(java.util.UUID uuid,
java.lang.String topic,  java.sql.Timestamp event_time, double lon,
double lat) throws java.sql.SQLException
```

Register a new Event in the GDI

Parameters:

```
uuid - uuid

topic - topic

event_time - time the event occured

lon - longitude wgs84

lat - latidute wgs84
```

Returns:

```
true if the event was registered in the gdi
```

Throws:

```
java.sql.SQLException - if something went wrong
```

- ***registerEvent***

```
public boolean registerEvent(java.util.UUID uuid,
java.lang.String topic, java.sql.Timestamp event_time,
double x, double y, int epsg) throws java.sql.SQLException
```

Register a new Event in the GDI

Parameters:

```
uuid - uuid

topic - topic

event_time - time the event occured

x - x coordinate of the coordinate referene system defined by
epsg

y - y coordinate of the coordinate referene system defined by
epsg

epsg - EPSG code of the Coordinate reference system
```

Returns:

```
true if the event was registered in the gdi
```

Throws:

```
java.sql.SQLException - if something went wrong
```

- ***registerEvent***

```
public boolean registerEvent(java.util.UUID uuid,
java.lang.String topic, java.sql.Timestamp event_time,
java.lang.String wkt, int epsg) throws java.sql.SQLException
```

Register a new Event in the GDI

Parameters:

`uuid - uuid`

`topic - topic`

`event_time - time the event occured`

`wkt - WellKnownText description of the geometry/location`

`epsg - EPSG code of the Coordinate reference system`

Returns:

`true if the event was registered in the gdi`

Throws:

`java.sql.SQLException - if something went wrong`

- ***registerEvent***

```
public boolean registerEvent(java.util.UUID uuid,
java.lang.String topic, java.sql.Timestamp event_time,
org.postgis.PGgeometry geom) throws java.sql.SQLException
```

Register a new Event in the GDI

Parameters:

`uuid - uuid`

`topic - topic`

`event_time - time the event occured`

`geom - Geometry`

Returns:

`true if the event was registered in the gdi`

Throws:

`java.sql.SQLException - if something went wrong`

- ***deregisterEvent***

```
public boolean deregisterEvent(java.util.UUID uuid)
throws java.sql.SQLException
```

Deregister an event / Remove it from the GDI

Parameters:

`uuid - UUID of the event`

Returns:

`true if removed`

Throws:

java.sql.SQLException - e.g. if uid is not regeistered

- ***closeEvent***

```
public boolean closeEvent(java.util.UUID uuid,
java.sql.Timestamp closeTime) throws java.sql.SQLException
```

Set a stop / end time of the validity of the event

Parameters:

`uuid - UUID of the event`

`closeTime - timestamp when t ended / was not valid anymore`

Returns:

`true if time could be set`

Throws:

`java.sql.SQLException - e.g. if uid is not regeistered`

- ***getNearestNodeID***

```
public int getNearestNodeID(double lon, double lat) throws
java.sql.SQLException
```

Get the nearest node it in the graph for routing

Parameters:

`lon - WGS 84 longitude`

`lat - WGS 84 latitude`

Returns:

`NodeId of nearest Node iin the Graph, -1 if nothing is found`

Throws:

`java.sql.SQLException - If something went wrong`

- ***getNearestNodeID***

```
public int getNearestNodeID(org.postgis.PGgeometry geom)
throws java.sql.SQLException
```

Throws:

`java.sql.SQLException`

- ***getCityRoutes***

```
public eu.citypulse.uaso.gdiclient.routes.CpRouteRequest getCityRoutes(
double fromLong, double fromLat, double toLong, double toLat,
java.lang.String costMetric, int count) throws java.sql.SQLException
```

Request a number of routes between two points

Parameters:

`fromLong` - origin longitude (WGS84)

`fromLat` - origin latitude (WGS84)

`toLong` - destination longitude (WGS84)

`toLat` - destination latitude (WGS84)

`costMetric` - cost metric that should be used, e.g. CpRouteRequest.ROUTE_COST_METRIC_DISTANCE

`count` - number of routes that should be returned

Returns:

Route Request including all routes

Throws:

`java.sql.SQLException` - If something went wrong

- ***getCityRoutes***

```
public eu.citypulse.uaso.gdiclient.routes.CpRouteRequest getCityRoutes(
java.lang.String fromWkt, java.lang.String toWkt,
java.lang.String costMetric, int count) throws java.sql.SQLException
```

Get a CpRouteRequest for two locations and a metric

Parameters:

`fromWkt` - From PGGeometry in WKT

`toWkt` - To PGGeometry in WKT

`costMetric` - Metric that should be used to find optimal route

`count` - number of routes

Returns:

CpRouteRequest

Throws:

`java.sql.SQLException` - If invalid positions or geometries are given

- ***getCityRoutes***

```
public eu.citypulse.uaso.gdiclient.routes.CpRouteRequest getCityRoutes(
eu.citypulse.uaso.gdiclient.routes.CpRouteRequest cprr, int count)
throws java.sql.SQLException
```

Get a number of city Routes for a route request

Parameters:

`cprr` - get city routes from a pre-generated request count - number of routes that should be returned

Returns:

updated CpRouteRequest object

Throws:

java.sql.SQLException - If something went wrong

- ***getnNextLocationsByAmenity***

```
public java.util.ArrayList<eu.citypulse.uaso.gdiclient.objects.CpAmenit
y> getnNextLocationsByAmenity(double lon, double lat,
java.lang.String amenity, int count, int repetitionId)
throws java.sql.SQLException, java.lang.NullPointerException
```

Get ilst of n next Amenities (like Parking, Atm, Hospital)

Parameters:

lon - Longitude (WGS84)

lat - Latitude (WGS84)

amenity - Amenity Type

count - Number of returned Values

repetitionId - (Id for persistence Monitoring)

Returns:

ArrayList of CpAmenity objects

Throws:

java.sql.SQLException - If request is not successfull

java.lang.NullPointerException - If no routes are available

- ***getnNextLocationsByAmenity***

```
public java.util.ArrayList<eu.citypulse.uaso.gdiclient.objects.CpAmenit
y> getnNextLocationsByAmenity(org.postgis.PGgeometry fromGeom,
java.lang.String amenity, int count, int repetitionId)
 throws java.sql.SQLException, java.lang.NullPointerException
```

Parameters:

fromGeom - PGGeometry in WKT

amenity - Amenity Type

count - Number of returned Values

repetitionId - (Id for persistence Monitoring)

Returns:

ArrayList of CpAmenity objects

Throws:

```
java.sql.SQLException - If request is not successfull

java.lang.NullPointerException - If no routes are available
```

- ***updateCostMultiplicatorRadial***

```
public int updateCostMultiplicatorRadial(double lon,
double lat, double radius_m, java.lang.String costMetric,
double costMultiplicatorValue) throws java.sql.SQLException
```

Update one of the costMultiplicator Layers around a certain point, e.g. to set higher routing cost for pollution

Parameters:

```
lon - WGS 84 longitude

lat - WGS 84 latitude

radius_m - radius around this point where edges of the graph will
get a higher cost multiplicator

costMetric - cost metric that should be used, e.g.
CpRouteRequest.ROUTE_COST_METRIC_DISTANCE

costMultiplicatorValue - a value higher than 1.0 to set a higher
cost. I.e. 2.0 will double the cost.
```

Returns:

```
number of edges that were updated
```

Throws:

```
java.sql.SQLException - If something went wrong
```

- ***updateCostMultiplicatorArea***

```
public int updateCostMultiplicatorArea(double minLon,
double minLat, double maxLon, double maxLat,
java.lang.String costMetric,
double costMultiplicatorValue throws java.sql.SQLException
```

Update the cost multiplicator in an rectengular area specified by minimum and maximum bounds

Parameters:

```
minLon – minimal WGS 84 longitude

minLat – minimal WGS 84 latitude

maxLon – maximum WGS 84 longitude

maxLat – maximum WGS 84 latitude

costMetric - cost metric that should be used, e.g.
CpRouteRequest.ROUTE_COST_METRIC_DISTANCE
```

costMultiplicatorValue - a value higher than 1.0 to set a higher cost. I.e. 2.0 will double the cost.

Returns:

number of edges that were updated

Throws:

java.sql.SQLException - If something went wrong

- ***resetCostMultiplicators***

```
public int resetCostMultiplicators(java.lang.String costMetric)
throws java.sql.SQLException
```

Reset cost multiplicators of a specific type

Parameters:

costMetric - costMetric cost metric that should be reset, e.g. CpRouteRequest.ROUTE_COST_METRIC_DISTANCE

Returns:

number of edges that were updated

Throws:

java.sql.SQLException - If something went wrong

- ***resetCostMultiplicators***

```
public int resetCostMultiplicators()  throws java.sql.SQLException
```

Reset all cost multiplcators for all metrics to 1.0

Returns:

number of edges that were updated

Throws:

java.sql.SQLException - If something went wrong

- ***getEventStreamsForRoute***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiEventStream[] getEv
entStreamsForRoute(eu.citypulse.uaso.gdiclient.routes.CpRoute cpr,
double buffer_m) throws java.sql.SQLException
```

Get Event streams in a distance of a specific route

Parameters:

cpr - Route that should be checked

buffer_m - the distance between events and routes

Returns:

Array of CpGdiEventStream elements containing the nearby EventStreams

Throws:

java.sql.SQLException - If something went wrong

- ***getEventStreamsForRoute***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiEventStream[] getEv
entStreamsForRoute(java.lang.String wgs84WktGeometry, double buffer_m)
throws java.sql.SQLException
```

Get Event streams in a distance of a specific route

Parameters:

wgs84WktGeometry - WKT Text describing geometry in WGS84

buffer_m - the distance between events and routes

Returns:

Array of CpGdiEventStream elements containing the nearby EventStreams

Throws:

java.sql.SQLException - If something went wrong

- ***getEventStreamsForRoute***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiEventStream[] getEv
entStreamsForRoute(java.lang.String wktGeometry, int epsg,
double buffer_m)
throws java.sql.SQLException
```

Get Event streams in a distance of a specific route

Parameters:

wktGeometry - WKT string to describe the route

epsg - Epsg code for projection

buffer_m - the distance between events and routes

Returns:

Array of CpGdiEventStream elements containing the nearby EventStreams

Throws:

java.sql.SQLException - If something went wrong

- ***getEventStreamsForRoute***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiEventStream[] getEv
entStreamsForRoute(org.postgis.PGgeometry pgeo, double buffer_m) throws
java.sql.SQLException
```

Get Event streams in a distance of a specific route

Parameters:

`pgeo - PGGeometry object to describe the route`

`buffer_m - the distance between events and routes`

Returns:

`Array of CpGdiEventStream elements containing the nearby EventStreams`

Throws:

`java.sql.SQLException - If something went wrong`

- ***getAllEventStreams***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiEventStream[] getAl
lEventStreams() throws java.sql.SQLException
```

Return all Event Streams as Array

Returns:

`CpGdiEventStream[]`

Throws:

`java.sql.SQLException`

- ***getSensorsForRoute***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiSensorStream[] getS
ensorsForRoute(eu.citypulse.uaso.gdiclient.routes.CpRoute cpr,
double buffer_m)
throws java.sql.SQLException
```

Get Sensor streams in a distance of a specific route

Parameters:

`cpr - Route that should be checked`

`buffer_m - the distance between events and routes`

Returns:

`Array of CpGdiSensorStream elements containing the nearby SensorStreams`

Throws:

`java.sql.SQLException - If something went wrong`

- ***getEventsForRoute***

```
public eu.citypulse.uaso.gdiclient.persistdata.CpGdiEvent[] getEventsFo
rRoute(eu.citypulse.uaso.gdiclient.routes.CpRoute cpr,
java.sql.Timestamp fromTime, int duration_s,  double buffer_m) throws
java.sql.SQLException
```

Get Events nearby the route which were created in duration_s before fromTime timestamp and have not been closed. So you can check e.g. for the interval of the last 500 seconds from the current time.

Parameters:

```
cpr - Route that should be checked

fromTime - start duration we are looking at

duration_s - duration of the interval

buffer_m - the distance between events and routes
```

Returns:

```
Array of CpGdiEvent elements containing the nearby Events in the
specified time
```

Throws:

```
java.sql.SQLException - If something went wrong
```

3.14.5 Component APIs

The whole component API documentation is available as a Javadoc at:

https://mobcom.ecs.hs-osnabrueck.de/svn/CityPulseDevelopment/CityPulseGDI/doc/index.html

Table 26: The end points of the Geo-spatial data base component

End point name	End point scope
`ws://locahost:7686`	Web-socket for non-Java usage uses textual communication

Listing 29 presents the signature of the end points used inside the web service. It accepts text input with the following methods and parameters:

```
# resetAllOsmIdCost()
- Action:     Reset all cost multiplicators to 1.0
- Returns:    number of updated multiplicators

To operate quickly on a graph we need to get the startpoint end endpoint nodeIDs in our graph.
# getNearestId(x-coordinate,y-coordinate)
- Action:     Search for a node id near the coordinate
- Returns:    node id

# updateOsmIdCost(osm_id, cost_multiplicator)
```

```
- Action:      Update cost-multiplicator for one specific osm-id
- Returns:    Count of updated multiplicators (edge based)

# updateCostArea(con,min_x,min_y,max_x,max_y,costMultiplicator)
- Action:      Updates cost multplicator for an area intersecting this bounding box
- Returns:    Number of updated multiplicators

# getCityRoute(srcXlon,srcYlat,trgXlon,trgxlat,costMode,optional:numberOfRoutes)
- Parameters:
  costMode = distance, time, pollution, combined
- Action:      Find path with minimum cost between two nodes AND resets the cost multiplicator
for alternate routes automatically
- Returns:  CSV (Semicolon separated) with header containing the following columns (if you need
other format, just ask):
      geometry: linestring or multilinestring
  length_m: length(meters)
  time_s:   estimated time(seconds) - Has do be adapted, currently on top speed
  total_cost: Total cost of this path
examples:
getCityRoute(10.1580607288361,56.1477407419276,10.1165896654129,56.2257947825602,dis
tance)
example:getCityRoute(10.1580607288361,56.1477407419276,10.1165896654129,56.22579478
25602,distance,30)

# resetAllMultiplicators(optional:costMode)
- Parameters:
  costMode = distance, time, pollution, combined
           if not defined or 'combined' is used, all multiplicators are reset to 1.0
- Action: Resets cost multiplicators to 1.0
- Returns: number of updated multiplicators
- examples: resetAllMultiplicators(combined)
           resetAllMultiplicators(pollution)
           resetAllMultiplicators(distance)
           resetAllMultiplicators(time)

#updateCostCircular(xLon,xLat,costMode,radius_m,costMultiplicator)
- Action: updates cost multiplicators
- Parameters:
  xLon: Longitude in WGS84
  xLat: Latitude in WGS84
  costMode: distance, time, pollution (combined would not make any sense here)
  radius_m: radius surrounding coordinate in meters (all edges that touch the circle are affected)
  costMultiplicator: double value, which defines the cost multiplication (dependent on the
costMode during route selection)
- Returns:   number of updated multiplicators
example:  updateCostCircular(10.158,56.21,pollution,400,5.0)
          updateCostCircular(10.156,56.20,distance,250,4.0)
          updateCostCircular(10.155,56.18,time,100,3.0)
```

Listing 29: the signature of the end point.

3.15 Technical adaptation

3.15.1 Component description

The technical adaptation component evaluates the criticality of the technical issues resulting in QoS decrease. If the QoS update creates a critical situation that violates the QoS constraints defined by the user at design time, an adaptation action is triggered, trying to recover the QoS performance for the user query by replacing the data streams used. The adaptation results in redeployments of updated composition plans. The adaptation may happen on different granularity levels, according to different adaptation strategies as specified in [CityPulse-D5.1], it may reuse some functionalities of the data federation component to re-discover and re-compose data streams, as specified in [CityPulse-D3.2]. The technical adaptation component is closely integrated with the data federation component.

It can be used for the following types of scenarios:

- Scenario 1: The user deploys a query over traffic data streams and asks for the accuracy of the query to be above 90%, however, at run-time, one of the wireless traffic sensors has lowered its sampling rate because of low battery level, and thus resulting in less accurate observation values. The QoS change is observed by the data quality monitoring component and reported to the QoS channels on the message bus. The technical adaptation component captures the QoS change and determines this is a critical QoS update. It triggers the adaptation process and re-deploys the query with a neighboring traffic sensor, which has sufficient battery energy and a higher accuracy.

It cannot be used in the situations where no functional equivalent data streams can be found, i.e., no stream replacements are possible.

3.15.2 Known limitations:

- Trade-offs between efficiency and effectiveness: Different adaptation strategies have different adaptation scopes, resulting in a trade-off between adaptation efficiency and effectiveness, as discussed in [Gao16].
- Message loss: The adaptation action may result in message loss. More details in [Gao16].
- Crisp QoS Constraints: Crisp constraints are defined when evaluating the criticality. If the QoS updates are too fluctuated, some adaptations maybe unnecessary. [Joshi14]

3.15.3 Component location on GitHub

This component can be found at the following GitHub location: https://github.com/CityPulse/ACEIS

3.15.4 Component prerequisite

- See prerequisite for the data federation component [insert cross-ref for data federation].

3.15.5 CityPulse framework dependencies

- The technical adaptation component is deployed together with the data federation.
- See dependencies for the data federation component [insert cross-ref for data federation].

3.15.6 Component deployment procedure

- See deployment procedure for the data federation component [insert cross-ref for data federation].

3.15.7 Component programmatic APIs

Table 27 contains that list of classes which can be used during the development of the technical adaptation component.

Table 27: The classes of the technical adaptation component

Package name	Class name	Class scope
org.insight_centre.aceis.observations	QosUpdate	Modelling the QoS updates.
org.insight_centre.aceis.subscriptions.streams	QosUpdateStream	Modelling the QoS update streams.
org.insight_centre.aceis.subscriptions	QosUpdateListener	Listens to QoS updates.
org.insight_centre.aceis.subscriptions	TechnicalAdaptationManager	Handles the QoS adaptation actions.
org.insight_centre.citypulse.server	QosServerEndpoint	QoS update server, responsible for creating QoS update streams, which contains QoS listeners for adaptation manager instances.

The class QoSUpdate has the following methods:

QosUpdate

```
public QosUpdate(java.util.Date obTimestamp, java.lang.String id,
java.lang.String correspondingServiceId,  QosVector qos)
```

Creates a QoS update for a stream

Parameters:

`obTimestamp` - update timestamp

`id` - update id

`correspondingServiceId` - stream(event service) id

`qos` - QoS vector

compareTo

```
public int compareTo(QosUpdate other)
```

Specified by:

`compareTo` in interface `java.lang.Comparable<QosUpdate>`

The class QoSUpdateStream has the following methods:

QosUpdateStream

```
public QosUpdateStream(java.lang.String txtFile,
java.lang.String sid, java.util.Date start,
java.util.Date end)
throws java.io.IOException
```

Creates a qos update stream using local qos update record file

Parameters:

`txtFile` - path to the qos record file

`sid` - sensor service id

`start` - start time

`end` - end time

Throws:

`java.io.IOException`

addListener

```
public void addListener(QosUpdateListener qul)
```

Adds a QoS listener for this stream

Parameters:

`qul` - QoS update listener to add

removeListenter

```
public void removeListenter(java.lang.String sessionId)
```

Removes a QoS listener for this stream

Parameters:

`sessionId` - session of the listener to be removed

sendUpdate

`public void sendUpdate(QosUpdate qu)`
Sends qos updates to the session in the listener

Parameters:

qu - QoS update

run

`public void run()`
Starts the QoS update stream

Specified by:

`run` in interface `java.lang.Runnable`

The class QoSUpdateListener has the following methods:

QosUpdateListener

`public QosUpdateListener(javax.websocket.Session session)`
Creates an qos update listener for a session

Parameters:

`session` - session to send updates

sendUpdate

`public void sendUpdate(QosUpdate qu)                    throws java.io.IOException`
Sends qos updates via session

Parameters:

qu - qos update

Throws:

`java.io.IOException`

The class TechnicalAdaptationManager has the following methods:

TechnicalAdaptationManager

`public TechnicalAdaptationManager(SubscriptionManager sub, EventDeclaration currentCompositionPlan, QosVector constraint, WeightVector weight, java.lang.Double freq, TechnicalAdaptationManager.AdaptationMode mode, java.util.Date start, java.util.Date end)`
Creates technical adaptation manager

Parameters:

`sub` - subscription manager

`currentCompositionPlan` - currently monitored composition plan

`constraint` - qos constraints

`weight` - qos preferences

`freq` - sensor frequency

`mode` - adaptation mode

`start` - start time

`end` - end time

adaptation

```
public EventDeclaration adaptation(java.lang.String serviceID)
throws java.lang.Exception
```

Starts qos adaptation

Parameters:

`serviceID` - the critical service triggering the adaptation

Returns:

new composition plan conforming with user's constraints

Throws:

`java.lang.Exception`

onMessage

```
public void onMessage(java.lang.String message,
javax.websocket.Session session)
throws java.lang.Exception
```

Receives qos update message, evaluates the criticality of the udpate and triggers adaptation if needed.

Parameters:

`message` - qos udpate message

`session` - websocket session used

Throws:

`java.lang.Exception`

onOpen

```
public void onOpen(javax.websocket.Session session)
```

Upon opening a new session, relevant subscriptions to qos updates are made

Parameters:

`session` - websocket session

setConstraint

`public void setConstraint(QosVector constraint)`
Sets the QoS constraints for the adaptation manager.

Parameters:

`constraint`- QoS constraint vector

setMode

`public void setMode(TechnicalAdaptationManager.AdaptationMode mode)`
Sets the QoS adaptation mode for the adaptation manager.

Parameters:

`model`- QoS adaptation mode, could be global, local or incremental, if mode==null no adaptation is allowed.

The class QosServerEndpoint has the following methods:

QosServerEndpoint

`public QosServerEndpoint()`
Creates an instance for the QosServerEndpoint, this endpoint is used internally for the technical adaptation manager and is not intended to interact with outside programs/users.

onMessage

```
public java.lang.String onMessage(java.lang.String message,
javax.websocket.Session session)
throws java.lang.Exception
```
Upon receiving qos subscriptions, create qos subscriptions on relevant qos streams

Parameters:

`message` - QoS subscription message wrapped as Json

`session` - websocket session

Returns:

response message

Throws:

`java.lang.Exception`

3.15.8 Error handling

The log file of the component can be found at EC-log.log

Table 28 present the list of errors which might appear during the deployment or the execution of the component.

Table 28: The error handling of the data federation component

Error name	Error description	Possible solution
Adaptation failed	The technical adaptation component failed to create an adaptation that fulfills the QoS constraints.	Try defining more relaxed QoS constraints.
Failed to de-register queries	Unable to deregister CQELS/CSPARQL queries	Restart component.
IOException during websocket session	Triggered by unexpected client connection loss etc.	Restart component.
Invalid input format	Cannot parse the input message, identified by the a warning log starts with " unexpected msg ..."	Check the input format and content.
Cannot create composition plan	The data federation server cannot create composition plans that satisfy the event request. Identified by the message "FAULT: Cannot create composition plans under specified constraints."	Check QoS constraints, try defining more relaxed constraints.
Cannot register event requests	The data federation server cannot register the event request over message bus data streams. Identified by the message "FAULT: Failed to register event request."	Check connectivity to the message bus.

4. Conclusions

The cities have started to deploy sensor and actor devices in their environment, e.g. intelligent lighting, and observation and monitoring devices to collect traffic, air quality and water/waste data. However, the current focus is mainly on collection, storage and visualisation of the datasets with an emphasis on high performance computing and visual computing solutions. While the recent efforts in this area have enabled emerging technologies and solutions to develop novel techniques for smart city applications and use-cases scenarios, there is however a gap in providing efficient and scalable methods that enable (near)real-time processing and interpretation of streamed sensory and social media data in smart city environments.

The CityPulse project proposes a framework for large-scale data analytics to provide information in (near-)real-time, transform raw data into actionable information, and to enable creating "up-to-date" smart city applications.

The CityPulse middleware components are also reusable in different application domains and are provided as open-source (https://github.com/CityPulse).

In order to reduce complexity and time for developing new applications a set of APIs is provided by each of the CityPulse components. This way the developers of services are able to abstract the complexity of the CityPulse middleware and are not bound to use specific technologies for the implementation.

This report presents on one hand the CityPulse framework out of the box visualization components, which can be used for monitoring in real-time the state of the city or to assess the performance of the CityPulse backend server.

On the other hand the report contains the user manual (with the appropriate API description) of the CityPulse framework components.

5. References

Barbieri Et al. 2009	Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., & Grossniklaus, M. (2009, April). C-SPARQL: SPARQL for continuous querying. In Proceedings of the 18th international conference on World wide web (pp. 1061-1062). ACM.
CityPulse-D2.2	Tsiatsis, V. et al. (2014, August). CityPulse D2.2 – Smart City Framework.
CityPulse-D3.1	Kolozali, S. et al. (2014, August). CityPulse D3.1 – Semantic Data Stream Annotation for Automated Processing.
CityPulse-D3.2	Gao, F. Et al. (2014, August). CityPulse D3.2 – Data Federation and Aggregation in Large-Scale Urban Data Streams.
CityPulse-D4.2	Kuemper, D. Et al. (2016, March) CityPulse D4.2 – Testing and Fault Recovery for Reliable Information Processing.
CityPulse-D5.1	Alessandra Mileo et al. (2014, July). CityPulse D5.1- Real-time Adaptive Urban Reasoning.
CityPulse-D5.2	Alessandra Mileo et al. (2015, August). CityPulse D5.2- User-Centric Decision Support in Dynamic Environments.
Ganz Et al. 2016	F. Ganz, P. Barnaghi, and F. Carrez, “Real world internet data,”Sensors Journal, IEEE, vol. 13, no. 10, pp. 3793–3805, 2013.
Gao Et al. 2016	Feng Gao, Muhammad Intizar Ali, Edward Curry and Alessandra Mileo, QoS-aware Adaptation for Complex Event Service, ACM Symposium on Applied Computing, Apr., 2016.
Hwang Et al.. 2003	S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, “Performance evaluation of main-memory R-tree variants,” in Advances in Spatial and Temporal Databases , Springer, 2003, pp. 10–27.
Puiu Et al. 2016	D. Puiu, P. Barnaghi, R. Tönjes, D. Kümper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T. Pham, C. Nechifor, D. Puschmann, J. Fernandes, “CityPulse: Large Scale Data Analytics Framework for Smart Cities”, IEEE Access, Vol. 4, 2016, pp. 1086 – 1108.