Scalable Data Analytics,
Scalable Algorithms, Software Frameworks
and Visualization ICT-2013 4.2.a

Project **FP6-619435/SPEEDD**
Deliverable **D6.7**
Distribution **Public**

http://speedd-project.eu

D6.7

Final Integrated Prototype

Fabiana Fournier (IBM), Inna Skarbovsky (IBM), Aviad Sela (IBM) ,Natan Morar (UoB), Marius Schmitt (ETH), Vagelis Michelioudakis (NCSR), Rohit Singhal (CNRS)

Status: Submitted (Version 1)

October 2016

Project

Project Ref. no	FP7-619435
Project acronym	SPEEDD
Project full title	Scalable ProactivE Event-Driven Decision Making
Project site	http://speedd-project.eu/
Project start	February 2014
Project duration	3 years
EC Project Officer	Stefano Bertolo

Deliverable

Deliverable type	Report
Distribution level	Public
Deliverable Number	D6.7
Deliverable Title	Final Integrated Prototype
Contractual date of delivery	M33 (October 2016)
Actual date of delivery	October 2016
Relevant Task(s)	WP6/Tasks 6.7
Partner Responsible	IBM
Other contributors	NCSR, CNRS, Feedzai, ETH, UoB, Technion
Number of pages	47
Author(s)	Fabiana Fournier (IBM), Inna Skarbovsky (IBM), Aviad Sela (IBM) ,Natan Morar (UoB), Marius Schmitt (ETH), Vagelis Michelioudakis (NCSR), Rohit Singhal (CNRS)
Internal Reviewers	Alexander Artikis (NCSR), Ivo Correia (Feedzai)
Status & version	Submitted
Keywords	architecture design, scalability, cep, decision-making, proactive, prototype

Executive Summary

This document is the report part of Deliverable 6.7 “Final Integrated Prototype” and its purpose is to present the latest results of T6.3 (Architecture design of the SPEEDD prototype) and T6.4 (Implementation of the SPEEDD prototype), and the third and final version of integrated prototype implementation for both use cases (software).

The goals of WP6 (Scalability and System Integration) are to develop a highly scalable event processing infrastructure supporting real-time event delivery and to integrate the SPEEDD components into a prototype for proactive event-based decision support.

The purpose of this report is to summarize work in WP6 during months M24-M33. It describes the extensions made to the SPEEDD integrated prototype in the third and final version. It includes extensions to the use case implementations, an updated architecture to deal with performance challenges identified in the second prototype version, an extended DM module implementation which supports better uncertainty handling, and a final performance and scalability evaluation.

Document history

Version	Date	Author	Change Description
0.1	08/10/2016	Inna Skarbovsky (IBM)	First draft
0.2	26/10/16	Inna Skarbovsky (IBM)	Updates after internal review
1.0	30/10/16	Inna Skarbovsky (IBM)	Final version

Contents

List of Tables

List of Figures

1 Introduction

1. Purpose and Scope of the Document

This is the third and final version of the report on the design of the SPEEDD prototype architecture and final version of integrated prototype implementation. In essence, the SPEEDD architecture described in the D6.1 and D6.1(amended) versions has not changed. However, due to unsatisfactory performance evaluation results of last year, major emphasis in this year's work was placed in finding and eliminating the performance bottlenecks and improving the scalability of the provided architecture. Due to this work, the IBM Proton CEP engine principal implementation underwent some changes and Proton on STORM architecture was revised. Additional evaluation performance testing was undertaken, which has shown significant performance improvements from second year's results, as will be demonstrated later in the document. Some functional changes to the DM component to better integrate uncertainty aspects will be discussed. Additionally some extensions to the use cases implementation appear in the API reference section.

2. Relationship with Other Documents

The current document is based on D6.1 and D6.1 (amended) versions of the architecture design document. It is also an extension of D6.5 document describing the integrated prototype.

The current document refers to the system requirements for the Proactive Traffic Management use case described in D8.1 and for the Proactive Credit Card Fraud Management described in D7.1.

The complex event processing module is discussed in depth in both D3.2 (resubmitted) and D3.3 along with information on the architecture and the event patterns, and improvements to the CEP run-time engine implementation to eliminate performance bottlenecks in the stand-alone version of Proton.

The Decision Making module algorithms are described in D4.3.

For more details on the traffic simulation module please refer to D8.4.

The dashboard application design and the underlying approach are explained in D5.3.

The scalability component's approach and algorithms are described in D6.6.

3. Updates since the first and second versions

As mentioned in the previous section, the current document contains updates and extensions to the concepts introduced in the first and second design reports (D6.1 and its amended version from second year). Specifically, the chapters discussing the decision making and the complex event processing architecture new versions (v3) of these components can be found in sections 2 and 3.

The performance testing architecture and second performance evaluation analysis is provided in section 4.

The API reference for SPEEDD was updated with the new events and the changes to existing events are described in section 6.

2 Decision Making Uncertainty Aspects

The Decision Making (DM) module provides a host of proactive event-driven DM tools. Using the detected or forecasted events from the Complex Event Processing (CEP) module as inputs, it outputs appropriate decisions, possibly in the form of actions, to steer the system towards a desirable outcome.

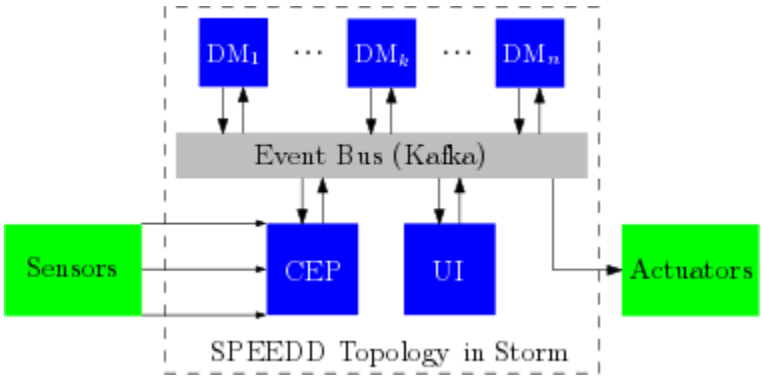

Figure 2-1 Implementation of DM in the SPEEDD topology

Within the SPEEDD topology, DM is implemented in the form of a number of bolts that receive and output events to the event bus, as illustrated in Figure 2-1. The bolts implement the DM algorithm at a local decision maker, or DM agent. The algorithm is typically driven by detected, derived or forecasted events from the CEP, as well as events from other DM agents. It outputs events that could convey decisions or actions, as well as initiate coordination with other DM agents. The architecture is distributed, event-driven and modular. All of these aspects are essential to the SPEEDD philosophy. The distributed and modular nature of DM permits new algorithms and methods to be added without an overhaul. In the following, we will summarize the interface of DM to other SPEEDD components, realized via events. Next, we will briefly describe the internal architecture of the DM component, in so far as it is relevant to understanding the DM functionality. Then, we provide an in-depth discussion of the way DM copes with the uncertainty inherent to traffic control problems. The comprehensive capabilities to mitigate uncertainty in the events reported to DM is the main feature added to DM in the final version of the integrated prototype.

1. DM Interface

The DM module interacts with other components of the integrated SPEEDD prototype via events as depicted in Figure 2-2. DM receives events from Complex Event Processing (CEP) and the User Interface (UI). Events from CEP include complex events such as predictions about freeway congestion. In addition to the high-level, derived events, DM also uses (time-averages of) low level sensor data.

For simplicity, we assume that all sensor data are passed through CEP, which allows us to perform the aggregation already in CEP. In practice, this means we compute averages over time and thereby reduce the number of events that need to be passed from CEP to DM. The different types of events consumed by the DM module are:

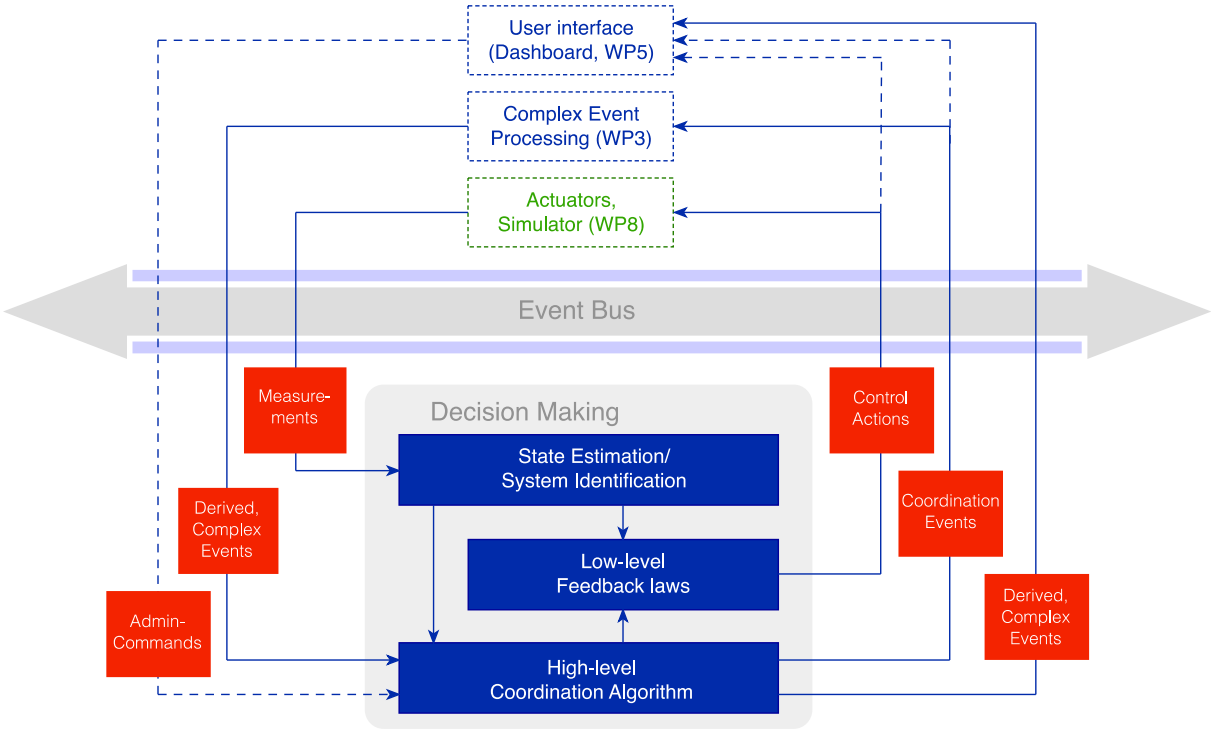

Figure 2-2 The DM module interacts with other modules of the integrated prototype via (complex) events, passed through the event bus. The internal structure of DM, which can be described in terms of the functionality it provides, will be explained in the following section.

1. **Measurements**: This includes all events that contain sensor readings from the micro-simulator or aggregates thereof, like the "**AverageDensityAndSpeedPerLocationOverInterval**". Usually, these events contain the attributes "average_occupancy", "average_speed" and "average_density" and potentially the empirical variances. DM expects that these quantities are given in the units used by the micro-simulator.
2. **Complex Events**: This category contains detected or predicted complex events like "**Congestion**" or "**PredictedCongestion**" that usually describe a binary state, i.e. some conditions like congestion that is either (expected to be) present or not. DM expects that forecasted events are equipped with an attribute "uncertainty" which provides (an estimate of) the probability that the respective event will eventually happen as predicted.
3. **Admin Commands**: Admin commands like "**setMeteringRateLimits**" are sent directly from the UI to overwrite the internal DM algorithms. Currently, no other admin commands are used.

Input events are assigned to an instance of the DM bolt based on the attribute dm_location. The events emitted by DM can be categorized as:

1. **Control actions**: Events like "**SetTrafficLightPhases**" set the behavior of actuators like traffic lights. Therefore, these events are sent most importantly to the real world actuators or the micro-simulator in experiments, but they usually are also of interest for the user-interface which presents them to potential human supervisors or operators. The format of these events depends on the respective actuator

2. **Coordination Events**: In accordance with the design philosophy of the SPEEDD project, the DM module is implemented in a distributed manner, as described in the next section. Not all decisions by DM can be made in a fully decentralized manner and some situations require (a limited amount) of communication between DM agents to achieve coordinate behavior. This is achieved via events like "**CoordinateRamps**", which are sent by one DM agent and processed by another one, via the general SPEEDD prototype event bus. They are not supposed to be used by any other component.
3. **Complex Events**: The DM module is used to estimate onramp queue lengths and emits corresponding estimates via the event "**AggregatedQueueLength**" for processing by CEP. This is an exception in so far as that aggregation is otherwise done by CEP and the aforementioned event is the only one in this category.

An overview of all event definitions is provided in the Appendix.

2. DM Architecture

A distributed implementation of the DM module was chosen in accordance with the overall SPEEDD goals of creating a scalable, modular, integrated prototype. Therefore, a hierarchical control approach has been chosen, with a low-level control layer that provides local feedback in an entirely decentralized, and therefore inherently scalable manner and a high-level coordination layer, which aims to coordinate the actions. This is achieved by appropriately adapting the control targets for the low-level controllers, such that the overall system optimizes a central control objective. In addition, the DM module contains algorithms for state estimation and system identification. The functionality for state estimation, system identification and (low-level) control can naturally be realized in a decentralized manner. Figure 2-2 depicts the internal structure of DM for freeway-ramp metering at runtime. A single bolt may manage several processes („MeteredOnramp n"), that control one metered onramp each. Therefore, each process runs a local „StateEstimator", „SystemIdentification" and „Controller" process. Coordination is taken care of via a „Coordination" algorithm which exchanges messages with other bolts via custom events. Note that such a controller structure necessitates prior information about the topology of the controlled network, i.e., information about the locations of sensors and actuators, the corresponding IDs and the structure of the connecting roads.

The main purpose of the following sections is to address the question of how the DM component, that is, the respectie sub-components as introduced in this section, deals with the uncertainty present in traffic control problems. Sources of uncertainty in traffic control fall into one of three categories:

(i) Sensor measurements are almost always corrupted by noise. For example, occupancy of a loop detector over an interval might not exactly correspond to the traffic density because of inhomogeneous traffic conditions, or measured velocities might differ from average velocities if cars are accelerating or slowing down. This type of uncertainty is usually modeled as additive noise. Then, estimates of the true values can be obtained via **State Estimation** using a Kalman filter, which we present in Section 2.3.

(ii) Traffic dynamics are inherently uncertain due to varying environmental conditions or arbitrary driver decisions. This type of uncertainty is hard to quantify a-priori, since it might even be time-varying in case of changing operation conditions. Therefore, we use a model-free, data-driven approach to estimate this uncertainty during operation based on the measured data. In Section 2.4, we state a suitable **System Identification** approach.

(ii) Forecasted, convex events are uncertain in the sense that there is no guarantee that they will actually occur, but instead, the confidence in there occurrence is quantified in the event attributes. The quantification of the uncertainty is carried out by the Complex Event Processing module. The usage of these events by the **Coordination** algorithms is described in Section 2.5

For completeness, it shall be mentioned that the **low-level control** algorithms do not operate directly on uncertain events, but on the outputs of state estimation, system identification and the coordination algorithms. They do not consider uncertainty directly.

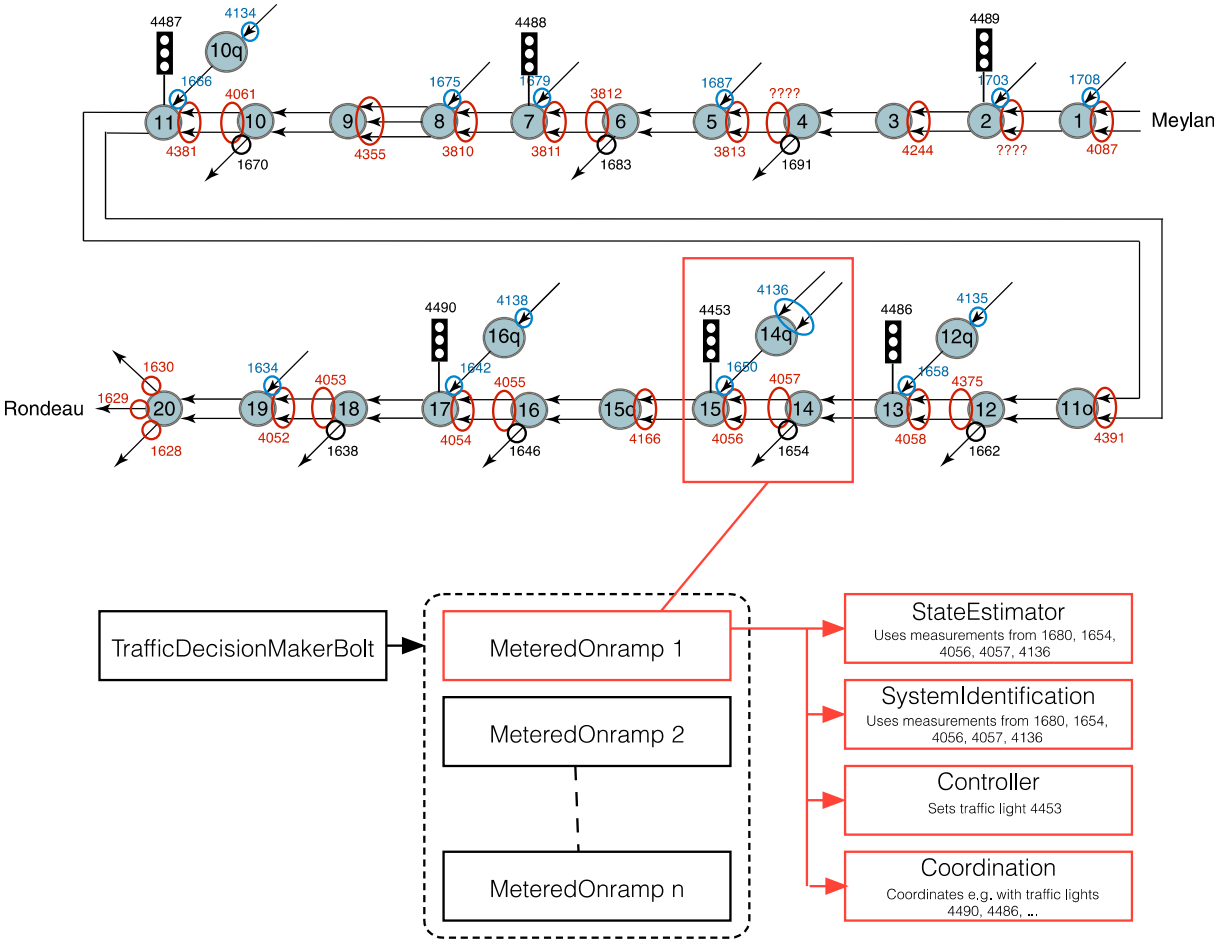

Figure 2-3 DM is implemented in a distributed manner, with separate processes fulfilling DM functionality for partitions of the road network. A single partition can be as small as a single metered onramp and the associated sensors, as depicted here.

3. Measurement Uncertainty

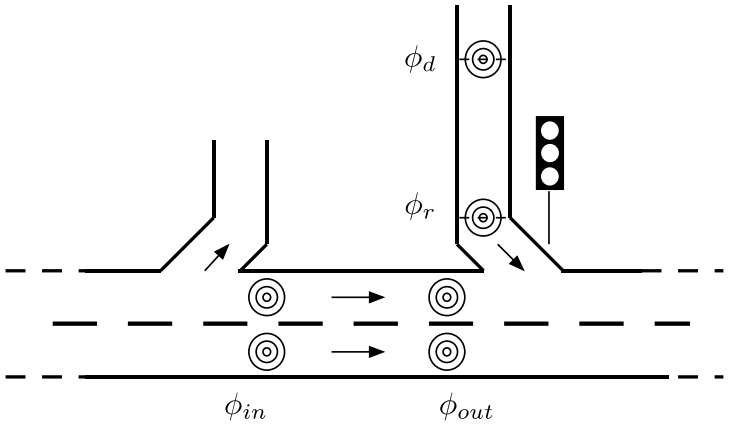

Figure 2-4 Sensor locations at a metered onramp.

In the SPEEDD freeway traffic control scenario, every metered onramp is equipped with loop detectors as depicted in Figure 2-4. At every sensor, the flow ϕ and the velocity v are measured. In addition, the occupancy is measured, which can be used to infer the traffic density ρ. The density evolves according to the conservation law

$$\rho(k+1) = \rho(k) + \frac{\Delta t}{l} \cdot (\phi_{in}(k) - \phi_{out}(k)) .$$

We assume that we obtain uncertain measurements of these quantities. The uncertainty may result from missed cars (e.g. because of a line change) in case of the flow measurements or errors in the density estimate inferred from the measured occupancy, due to inhomogeneous traffic conditions. We can express this uncertainty as additive noise. Then, we can obtain a better estimate of the true values using a Kalman Filter. The discrete-time Kalman Filter uses measurements , consisting of the measured means $\tilde{\rho}(k)$, $\tilde{\phi}_{in}(k)$. and $\tilde{\phi}_{out}(k)$. In addition, the corresponding variances $\tilde{\sigma}_{\rho}(k)$ $\tilde{\sigma}_{in}(k)$ and $\tilde{\sigma}_{out}(k)$ are reported. The Kalman Filter can be implemented in a recursive predictor-corrector form. That is, it only retains an internal estimate of the state $\hat{\rho}(k)$ and an estimate of the corresponding covariance $\hat{\Sigma}$, instead of the entire history of all measurements. Every time a new measurement arrives, a predictor step

$$\hat{\rho}(k+1|k) = \hat{\rho}(k) + \frac{\Delta t}{l} \cdot \left(\tilde{\phi}_{in}(k) - \tilde{\phi}_{out}(k)\right)$$
$$\hat{\Sigma}(k+1|k) = \hat{\Sigma}(k) + \left(\frac{\Delta t}{l}\right)^2 \cdot \left(\sigma^2_{\phi_{in}}(k) + \sigma^2_{\phi_{out}}(k)\right)$$

is performed first. Than, a corrector step

$$\hat{\rho}(k+1) = \hat{\rho}(k+1|k) + K(k) \cdot (\tilde{\rho}(k) - \hat{\rho}(k+1|k))$$
$$\hat{\Sigma}(k+1) = \hat{\Sigma}(k+1|k) - K(k) \cdot S(k) \cdot K(k)^{\top}$$

improves the predicted estimates using the most recent measurements, using the auxiliary quantities

$$S(k) = \hat{\Sigma}(k+1|k) + \sigma_\rho^2(k)$$
$$K(k) = \hat{\Sigma}(k+1|k) \cdot S_k^{-1}$$

It shall be noted that ramp metering seeks to control the traffic density immediately downstream of an onramp, whereas the Kalman Filter estimate corresponds to the section upstream of the ramp. Since downstream traffic includes the cars from the onramp in addition to the cars on the mainline, we can estimate the downstream density as

$$\hat{\rho}_{DS}(k) = \hat{\rho}(k) \cdot \frac{\tilde{\phi}_{out}(k) + \tilde{r}(k)}{\tilde{\phi}_{out}(k)} \cdot \gamma_m,$$

with γ_m a factor close to, but larger than one, that accounts for traffic distortions resulting from the merging process. The onramp queue-length evolves according to a similar conservation law

$$q(k+1) = q(k) + \Delta t \ \cdot (d_k(t) - r_k(t))$$

and similar equations for the Kalman Filter result. Note however, that occupancy measurements on the onramp can not be used to infer the density on the onramp as soon as a queue in stand-still is forming. To still retain a stable implementation, we use the fact that a minimal metering rate of r is mandatory. In addition, we use a priori bounds $0 \leq q(t) \leq \bar{q}$ on the queue length for correction of the estimates.

4. Traffic Dynamics Uncertainty

The importance of considering the effects of uncertainty in traffic dynamics can easily be exemplified with real-world data as depicted in Figure 2-5. These data clearly cannot be described by a single, deterministic function but require a stochastic model instead[1]. Traffic dynamics are inherently uncertain due to varying environmental conditions or arbitrary driver decisions. This type of uncertainty is hard to quantify a-priori, since it might even be time-varying in case of changing operation conditions. Therefore, we use a model-free, data-driven approach to estimate this uncertainty during operation based on the measured data, as described in detail in the corresponding section in the final Decision Making deliverable (D4.3). Here, it suffices to state that the System Identification component can estimate a stochastic model of the fundamental diagram based on data at runtime. The main interest is in identifying the density that maximizes the flow (in expectation), which is then provided to the low-level control algorithm as the control target. However, the estimation is computationally expensive and should not be performed every single time a new measurement arrives. It is performed periodically, that is, every N measurement events, if re-estimation is not triggered before. This might happen due to the

[1] Note that this figure does not rule out a deterministic model outright, since dependencies on other variables might exist that explain the variance in the data in a deterministic fashion However, even higher order road traffic models, or models considering additional independent variables, do not reduce the unexplained noise in real-world traffic data significantly.

arrival of certain complex events at DM, which indicate that an up-to-date, accurate model is necessary. Such events are described in the next section.

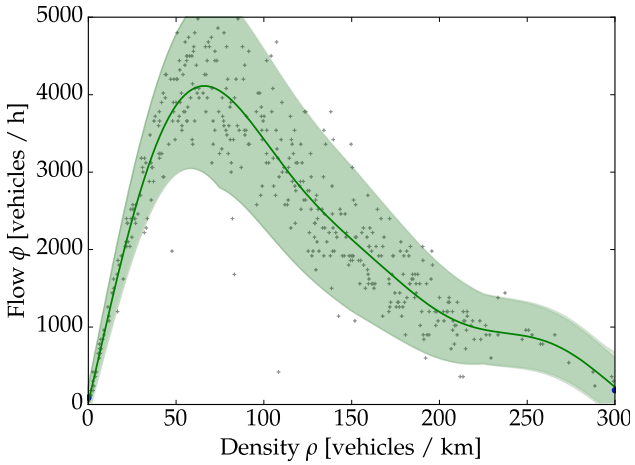

Figure 2-5 Data dample of real-world density-flow data pairs. These data can only poorly be represented by a single, deterministic function. However, a stochastic model, here depicted via its mean and 90% confidence interval, seems suitable.

5. Uncertainty in Complex Events

This section addresses the question on how complex events, and in particular uncertain complex events, are handled by DM. The handling of uncertain measurement events (**AverageDensityAndSpeed-PerLocationOverIntervall**, **AverageOnRampValuesOverInterval**, **AverageOffRampValuesOverInterval**) has been discussed in detail in Section 2.3. Furthermore, the usage of the events **setMeteringRateLimits** and **Coordinate Ramps** is straightforward, since these events dictate a specific behavior of DM, without any uncertainty. We provide an overview of the handling of the remaining events next:

Event name	Trigger Control	Re-estimation	Uncertainty
PredictedCongestion	Depending on uncertainty.	Depending on uncertainty.	Used for waiting-time trade-off as described in this section.
Congestion	Yes	Yes	Not present.
ClearCongestion	Yes	No	Not present.
PredictedRampOverflow	Depending on uncertainty.	Depending on uncertainty.	Used for waiting-time trade-off as described in this section.
ClearOnRampOverflow	Yes	No	Not present.
PossibleIncident	No	Yes	Used to determine if data are discarded before re-estimation.

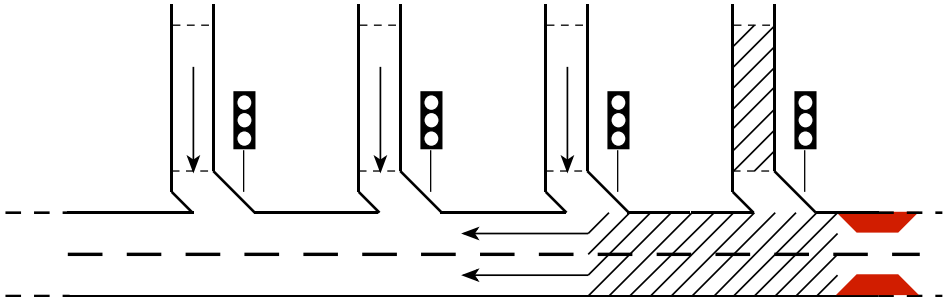

Figure 2-6 Freeway congestion caused by ramp overflow of bottleneck-adjacent ramp

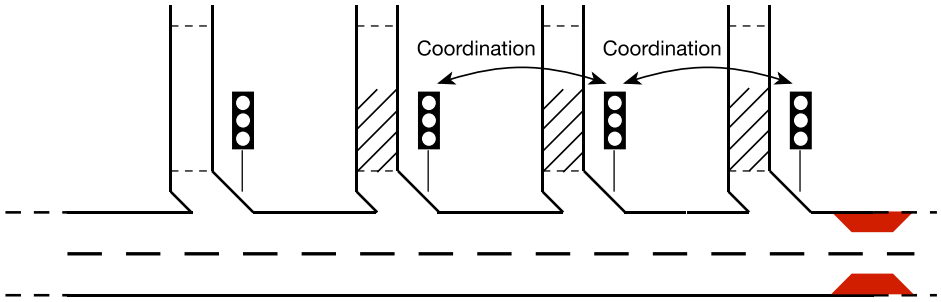

Figure 2-7 Prevention of ramp overflow and hence freeway congestion by ramp coordination

From the table, it can be seen that processing of the events **ClearCongestion** and **ClearOnRampOverflow** is straightforward, since they act as (certain) triggers for DM. Similarly, the event **Congestion** acts as a trigger but also initiates a re-estimation of the traffic dynamics as described in the previous section.

In case of a **PossibleIncident**, a re-estimation of the traffic dynamics is performed as well. In addition, we use a threshold on the uncertainty to determine if past traffic measurements are discarded before re-estimation. The rationale is that an incident as defined in the event definition will cause a change in the traffic dynamics. Therefore, past traffic measurement are considered outdated if the likelihood of an incident is sufficiently high.

To discuss the rationale DM is using to process **PredictedCongestion** and **PredictedRampOverflow** events, it is necessary to briefly review the idea of ramp coordination. Consider a freeway segment as depicted in Figures 2-5 and 2-6. Assume furthermore that, during a limited period of time, the traffic demand exceeds the bottleneck (in red) capacity such that the demand cannot be served completely. In case of entirely uncoordinated ramp metering, the onramp immediately upstream of the bottleneck will solely attempt to prevent a congestion forming at the bottleneck. In practice, this situation will usually lead to ramp overflow quickly (predicted by the **PredictedRampOverflow** event) and a congestion spreading upstream from the bottleneck is the result, as depicted in Figure 2-5. A **PredictedCongestion** event, while ramp metering at the closest ramp is already active (or a **PredictedRampOverflow** as discussed before), indicates that metering only the closest ramp might be insufficient to prevent the congestion. In such a case, one might consider using multiple upstream ramps that coordinate in holding traffic back on the onramps. Multiple ramps provide more storage space than a single one and therefore might prevent or at least delay the formation of a congestion queue, as depicted in Figure 2-6. However,

using multiple ramps comes at a cost. More cars are held back on the onramps, resulting in time lost for the drivers if the coordination was ultimately unnecessary to prevent congestion. In SPEEDD, the uncertainty is quantified in the aptly named attribute "uncertainty" and it is interpreted as the probability P[event_happens] that the event will happen within the time horizon T. We use the (un-) certainties P[congestion] (we will use the shorthand notation P[C] in the following) and P[ramp_overflow] to perform a trade-off between the (potential) benefits of preventing a congestion and the possibility of wasting driver's time on the onramps.

To do so, we first estimate the additional waiting time on the onramps. Let us consider a set of two onramps, the upstream (US) onramp and the downstream (DS) onramp. In case of a predicted congestion, the downstream onramp will always be active but the additional waiting time ΔTWT on the upstream onramp can be bounded as

$$\Delta TWT = \Delta t \cdot \sum_{t=0}^{T/\Delta t} q_{US}(t) \leq \Delta t \cdot \sum_{t=0}^{T/\Delta t} \frac{\bar{q}_{DS}}{\bar{q}_{DS} + \bar{q}_{US}} \bar{q}_{US},$$

for a situation in which coordinated ramp metering was ultimately unnecessary and therefore, the total amount of cars stored on both ramps is less than the space available on the downstream ramp

$$q_{US}(t) + q_{DS}(t) \leq \bar{q}_{DS}$$

Recall also that we use a coordination scheme which seeks to balance the occupancies on both onramps

$$\frac{q_{US}(t)}{\bar{q}_{US}} \approx \frac{q_{DS}(t)}{\bar{q}_{DS}}$$

Conversely, consider the case in which a congestion occurs, that could have been delayed if coordination was used. A congestion reduces the capacity F of a bottleneck by a (small) percentage p. Therefore, if the formation of congestion is delayed, more cars can pass the bottleneck. Note that a prediction of a congestion in T mean that during the time period [0, T] (we assume the event arrives at time t = 0 for ease of presentation) the traffic demand exceeds on average the bottleneck capacity and causes an overflow of the downstream ramp. Rarely, a congestion might arise despite the corresponding ramp not being full. We disregard this case in the computation of the decision on whether to coordinate. The average, surplus demand is equal to

$$\Delta d = \frac{l \cdot (\rho^c_{DS} - \rho_{DS}(0)) + \bar{q}_{DS} - q_{DS}(0)}{T}$$

In case coordination is used, the surplus demand can also been stored on the upstream onramp, which delays the congestion by

$$\Delta T = \frac{q_{US}}{\Delta d}$$

This delay until the occurrence of congestion allows a total, surplus number of cars equal to

$$\Delta cars = \Delta T \cdot F \cdot p = \frac{\bar{q}_{US} \cdot T}{l \cdot (\rho_{DS}^{c} - \rho_{DS}(0)) + \bar{q}_{DS} - q_{DS}(0)} \cdot F \cdot p$$

to pass the bottleneck in comparison to the case in which no coordination was used. The number of cars not passing the bottleneck will accumulate additional waiting time during the whole duration D of the congestion, which has to be estimated offline from data. The additional time spent in congestion therefore equals

$$\Delta TTT = \Delta cars \cdot D$$

The decision on whether to activate ramp coordination if a congestion is predicted depends on whether the expected costs of activating ramp coordination, equal to $E[cost\_cordination] = (1 - P[congestion]) \cdot \Delta TWT$ or the expected costs of not activating ramp coordination $E[cost\_no\_coordination] = P[congestion] \cdot \Delta TTT$ are higher. The metric used for predicted ramp overflow is very similar, since ramp metering decision will be disregarded if a ramp overlfow is imminent and the surplus cars are admitted to the freeway. Thus, ramp overflow ultimately also causes congestion on the mainline.

3 Scalability and Performance

1. Problems addressed

As part of the D6.5 deliverable a performance evaluation of the SPEEDD prototype was performed. The performance evaluation was done on a Mesos cluster of 4 machines, running Kafka and STORM frameworks. The evaluation was based on the fraud use case, since this is the use case with the strictest performance requirements in the project. The details for the testing configurations and the testing methodology can be found in the D.5 document, as well as detailed performance evaluation results.

As the result of the second version performance analyses some problems with regards to the architecture performance and scalability were identified (please refer to section 4 of D6.5 document for more information on the issues identified):

1. Exponential growth in processing latency of input events over time

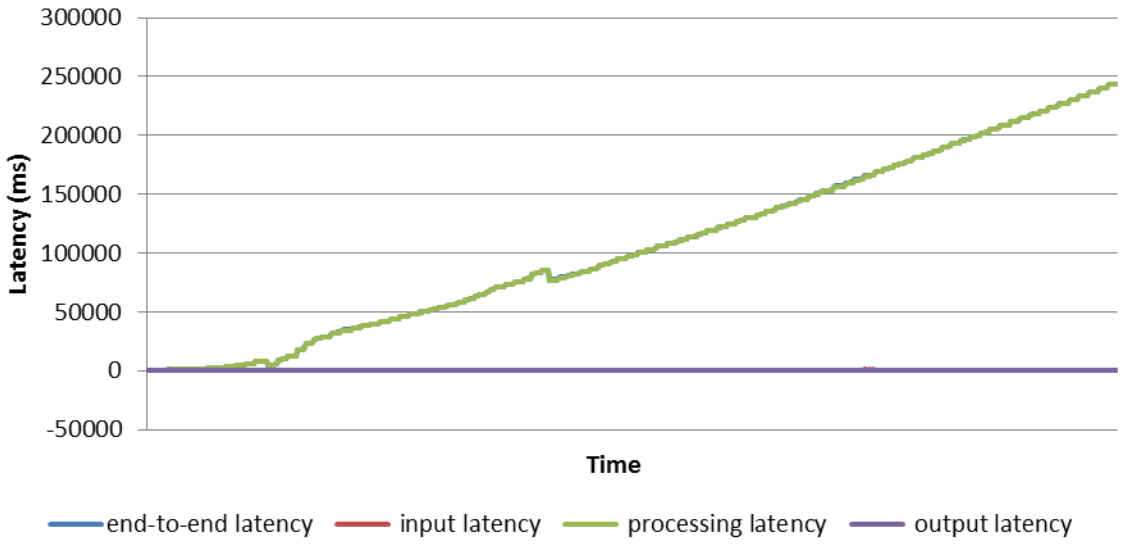

Figure 3-1 Increase in processing latency over time for 500 eps

   The reason for latency growth was not clear, and it was hypothesized that the latency might grow due to growing state and amount of stored data in the system over time.
2. Bottleneck in scalability due to limitation of distribution of data between instances of a context bolt in Proton on STORM architecture.
   Quoting from performance analyses section in D6.5 (section 4):
   "…learnt that all of the bolts composing Proton component's sub-tree in SPEEDD topology divide the load uniformly, except one bolt – the contextBolt. For contextBolt, at most three executors were actually processing tuples even when the topology had more. The reason for such behavior is the grouping strategy currently implemented in ProtonOnStorm: the combination of EPA name

and context type groups all the events that will be processed by the same contextBolt instance. There are three EPA agents in the EPN for the Credit Card use case, therefore there are exactly three
groups. This is one significant factor that limits the parallelism of the topology."

Figure 3-2 - Active context bolt instances out of all existing instances demonstrates the limitation in distribution of tuples among instances of the context bolt – even though that in the topology configuration 16 instances of context bolt are created, only 3 of them are actively receiving and processing event tuples (the active bolts receiving and transmitting events are marked in red in the figure).

Executors

Id	Uptime	Host	Port	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed
[25-25]	2m 59s	mesos2.iit.demokritos.gr	31011	0	0	0.000	0.000	0
[26-26]	3m 1s	mesos4.iit.demokritos.gr	31011	0	0	0.000	0.000	0
[27-27]	3m 0s	mesos1.iit.demokritos.gr	31011	0	0	0.000	0.000	0
[28-28]	2m 59s	mesos3.iit.demokritos.gr	7199	20	0	0.000	0.000	0
[29-29]	3m 0s	mesos2.iit.demokritos.gr	31010	0	0	0.000	0.000	0
[30-30]	2m 59s	mesos4.iit.demokritos.gr	31010	20	0	0.000	0.000	0
[31-31]	3m 0s	mesos1.iit.demokritos.gr	31010	20	0	0.000	0.000	0
[32-32]	2m 59s	mesos3.iit.demokritos.gr	31004	4540	4540	0.026	0.946	4840
[33-33]	3m 0s	mesos2.iit.demokritos.gr	31013	0	0	0.000	0.000	0
[34-34]	3m 0s	mesos4.iit.demokritos.gr	31013	20	0	0.000	0.000	0
[35-35]	3m 0s	mesos1.iit.demokritos.gr	31013	4860	4840	0.016	0.609	4860
[36-36]	2m 58s	mesos3.iit.demokritos.gr	9160	4580	4580	0.030	1.103	4840
[37-37]	3m 0s	mesos2.iit.demokritos.gr	31012	0	0	0.000	0.000	0
[38-38]	3m 1s	mesos4.iit.demokritos.gr	31012	0	0	0.000	0.000	0
[39-39]	2m 59s	mesos1.iit.demokritos.gr	31012	0	0	0.000	0.000	0
[40-40]	2m 59s	mesos3.iit.demokritos.gr	9042	20	0	0.000	0.000	0

Figure 3-2 - Active context bolt instances out of all existing instances

2. Approach

Two parallel approaches were adopted for minimizing processing latency on the one hand, and improving system scalability on the other, in SPEEDD prototype:

1. Proton run-time engine investigation for bottlenecks - Since the main component contributing to processing latency in the analyzed prototype was the CEP component profiling tools were used to identify the performance bottlenecks in Proton's core engine implementation.

   As a result of this, bottlenecks due to waiting locks and thread-pool exhaustion issues were discovered, and fixed. Please refer to section 6 of D3.2 for detailed problem description, corrective actions taken, and the updated results.

2. Proton on STORM architecture was revised to remove the bottleneck in context bolt scalability. The following section describes in details the changes made to the Proton on STORM implementation.

3. Proton on STORM architecture enhancements

As a result of non-functional analyses of application execution, including performance analyses, we have reached the conclusion that Proton's architecture on top of STORM can be modified and enhanced to better utilize the distribution infrastructure provided by STORM.

Specifically, a bottleneck was identified in tuple-distribution mechanism of tuples sent from routing bolt to context-service bolt.

In our previous implementation of Proton on STORM, we used the Storm field grouping option on the metadata routing fields – the agent name and the context name – to route the incoming events between the routing bolt to the context processing bolt. (refer to **Figure 3-3 - Proton on STORM general architecture with the old grouping scheme** for zoom in into Proton on STORM component architecture and old distribution scheme)

However, this makes the distribution of input events very application-dependent and limited to the actual number of agent-context pairings existing in the application. For example, as demonstrated in **Figure 3-6 Runtime distribution of Transaction instances**, for an application of 3 EPAs "sitting" on three different contexts, all input events at this stage would be partitioned into 3 groups and sent to three instances of context bolt, even if the number of tasks specified for this bolt in STORM's topology is higher than this number.

As opposed to that, the distribution of tuples outgoing from context bolt to EPA manager bolt instances is pretty efficient, since it is based on segmentation context values, which is assumed to be more or less uniformly distributed between all input events. The utilization of EPA manager bolt's instance is application-independent and can be further tuned by specifying the amount of tasks for this bolt. Addition of such segmentation partition information to field grouping directive between routing and context bolts should achieve the same level of distribution as in epaManagerBolt. As an example, consider a segmentation context based on customerId. By adding this information to fieldGrouping, instead of partitioning on 3 agents+context pairs, and therefore distributing all input tuples only between 3 context bolt instances, the distribution would be based on the number of available customerIds in the input data.

Therefore, as an enhancement of Proton on STORM architecture, we changed the mechanism of distribution of input tuples between the routing bolt and context bolt, to make it segmentation-dependent, the same way as EPA manager bolt distribution scheme is segmentation-dependent, instead of application-dependent.

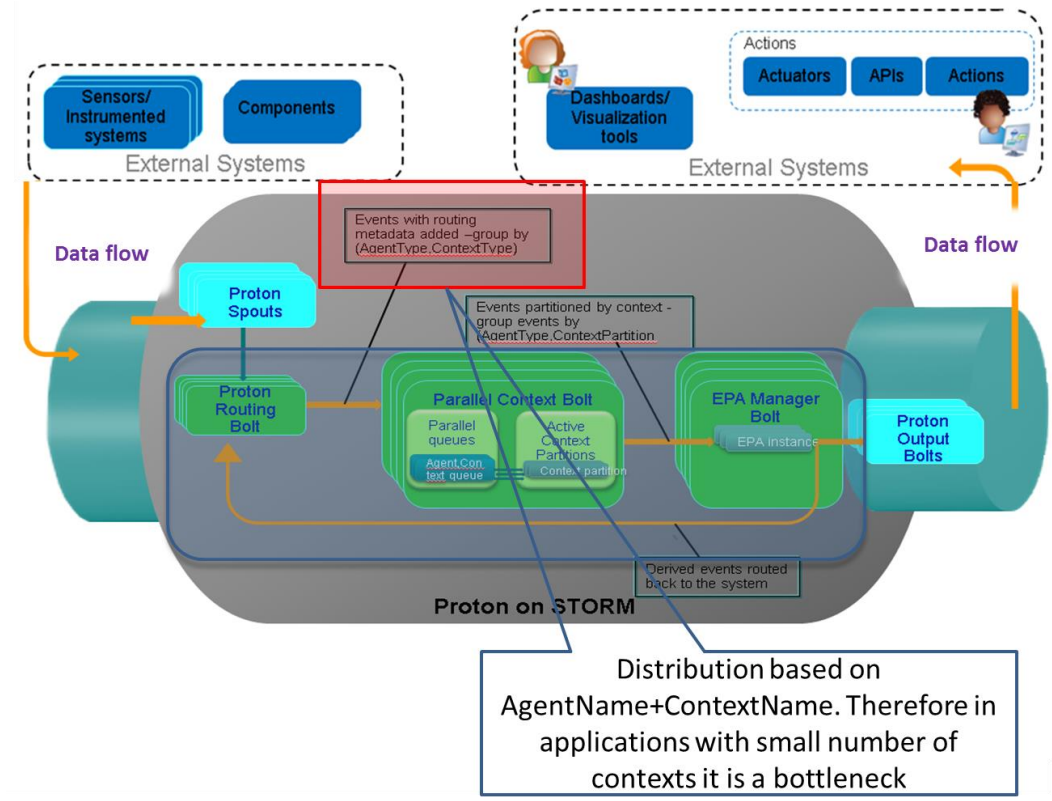

Figure 3-3 - Proton on STORM general architecture with the old grouping scheme

Focusing on the highlighted part of the topology shown in **Figure 3-3 - Proton on STORM general architecture with the old grouping scheme**:

Zooming in into the context bolt implementation, the previous and the updated versions can be seen in **Figure 3-4 Field grouping approach based on metadata** and **Figure 3-7 Field grouping approach based on segmentation partitioning** correspondingly:

In **Figure 3-4 Field grouping approach based on metadata** we can see that in v2 of Proton on STORM architecture the grouping scheme used was based on AgentName+ContextName.

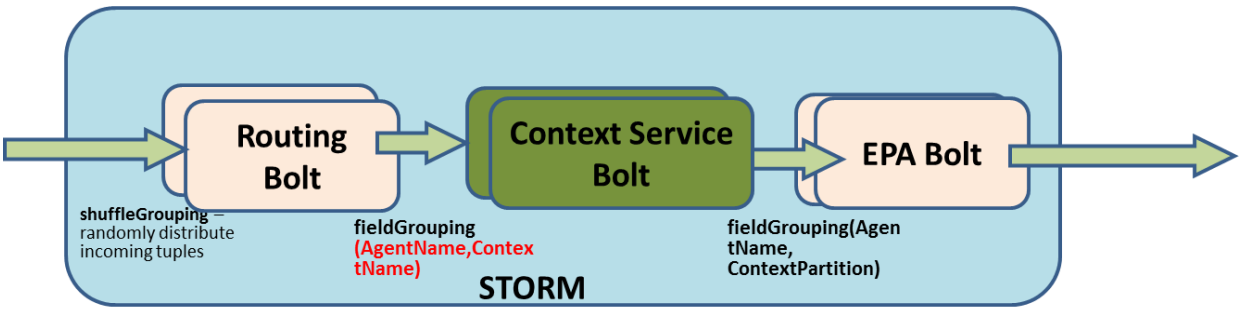

Figure 3-4 Field grouping approach based on metadata

If we take for example the fraud application, and focus on 3 EPAs (see **Figure 3-5 Fraud application's EPAs**

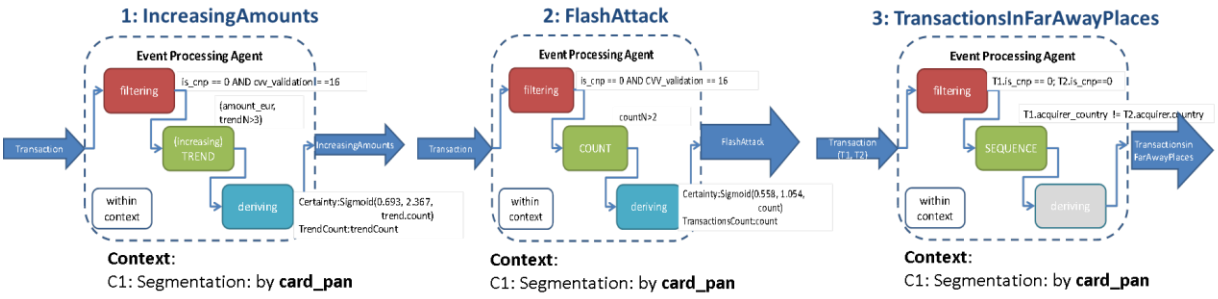

Figure 3-5 Fraud application's EPAs (refer to D3.2 for the EPAs details)

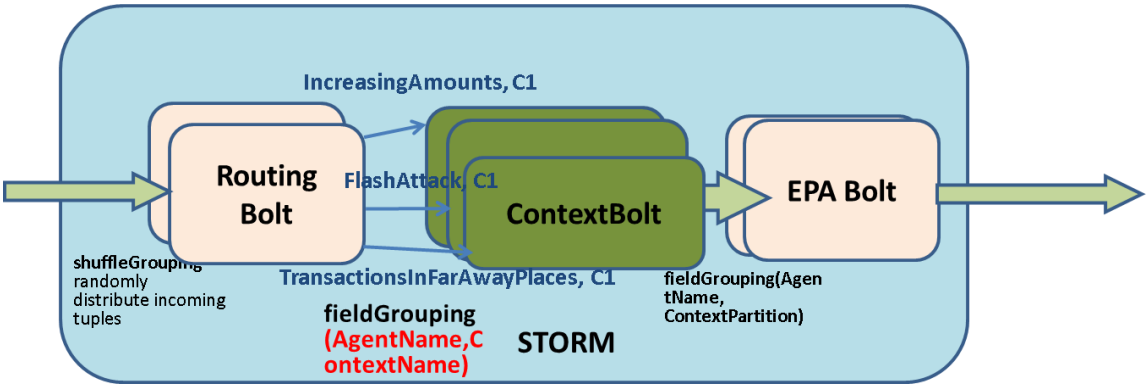

Figure 3-6 Runtime distribution of Transaction instances

In the new proposed architecture, the distribution of tuples between the routing bolt and context service bolts is not only be based on application metadata, which is limited in applications with small number of EPAs and contexts, but, as described previously, on segmentation context partitions as shown in **Figure 3-7 Field grouping approach based on segmentation partitioning**:

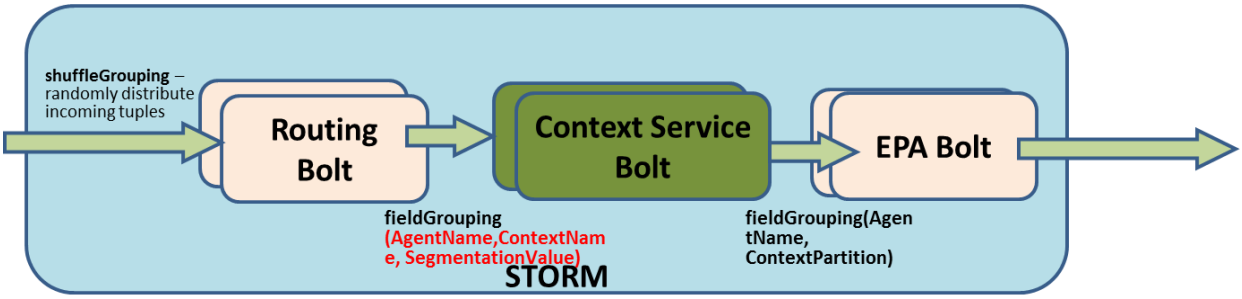

Figure 3-7 Field grouping approach based on segmentation partitioning

Therefore, for the previous example, the distribution of tuples would be unlimited since it is dependent on runtime segmentation values (see **Figure 3-8 Runtime distribution of Transaction instances in the new approach**)

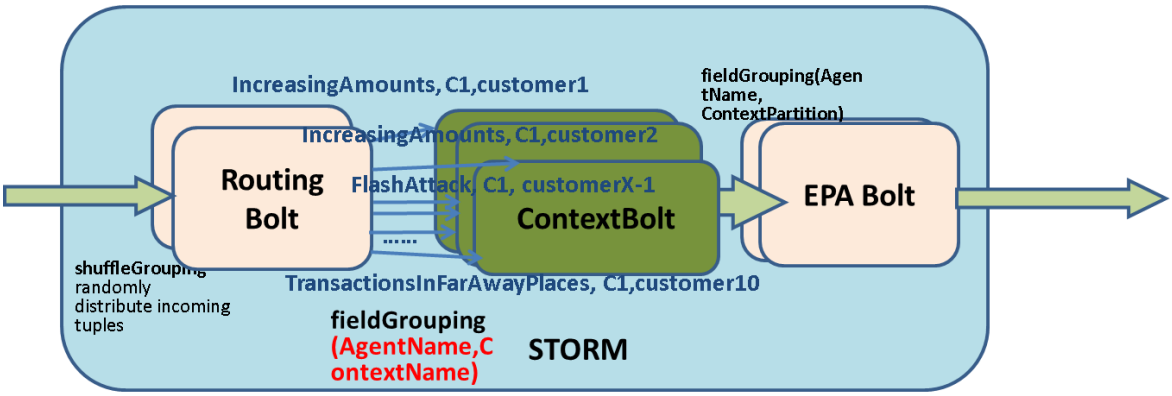

Figure 3-8 Runtime distribution of Transaction instances in the new approach

To allow for segmentation-based partitioning of tuples in the RoutingBolt, the following necessary changes needed to be made in the Proton on STORM architecture and implementation:

- Allow routing bolt's inspection of the relevant context information metadata to get the composed segmentation context expression definitions
- Allow access from routing bolt to expression evaluator for expression evaluation
- Evaluation of composed segmentation expression for each tuple, and addition of this information to field grouping expression

4 Second Performance Analysis

1. Objectives

The objectives of the second performance analysis are as follows:

1) Assess the current system performance
2) Explore processing latency improvements, if any
3) Explore scalability improvements, if any
4) Verify the ability of the system to match performance requirements set by the use cases

2. Approach

The approach for testing the performance of the integrated prototype was the same as described in the section 4-2 of D6.5. We summarize the general testing architecture here; please refer to document D6.5 for more details.

The conceptual view of the performance testing architecture is presented in **Figure 4-1 SPEEDD Performance Testing Architecture - Conceptual View**.The event emitter streams events from the test data set to speedd-in-events topic at the required rate. A module called “analyzer” records all the events – both input and emitted ones. For every event the analyzer registers the time it’s been encountered. The output of the analyzer is a test log which is processed later by the “stats” utility which computes latencies for every output event.

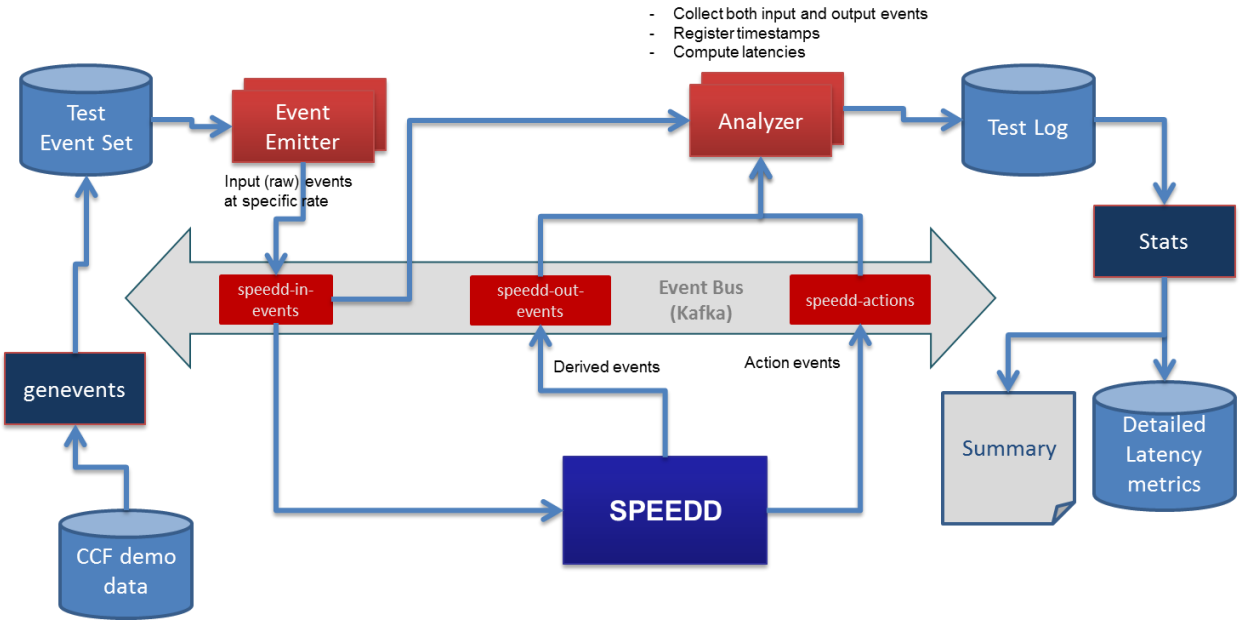

Figure 4-1 SPEEDD Performance Testing Architecture - Conceptual View

For every output event the “stats” utility computes the following values:

1) End-to-end latency – time period between the input event and the derived event from the Kafka consumer perspective
2) Input latency – time period between posting input event to Kafka and its detection by SPEEDD storm topology
3) Processing latency – time taken by derivation process in SPEEDD STORM topology
4) Output latency – time taken to deliver the derived event to the event consumer on Kafka

In order to correlate the derived event with the latest input event that triggered the derivation, we leverage the feature of Proton that allows attaching to a derived event the matching set of contributing input events. Thus, given in instance of the derived event one can easily obtain the list of the events that has contributed to the event pattern, along with their timestamps – so it’s straightforward to compute the latency as defined above.

The performance tests were executed on a cluster comprised of four physical machines of the following configuration:

- CPU: 2 x Intel Xeon E5520 @ 2.27GHz -- Cores : 16threads (8 cores)
- RAM: 12GB ECC
- Disks: 2x1TB (RAID1)
- NIC : 4 x 1Gbps
- OS: Debian 8

The cluster has the Mesos[2] framework installed that manages the computational resources thus simplifying the task of cluster configuration and resource allocation. A single virtual machine runs on every physical machine (to simplify maintenance and management), with exception of one machine where another small VM runs that functions as a mesos gateway server.

The storm-mesos[3] framework used to run STORM cluster on Mesos, the kafka-mesos[4] framework used for running Kafka cluster on Mesos.

The topology of the Mesos cluster is shown in the **Figure 4-2 Mesos cluster topology**

[2] http://mesos.apache.org/
[3] https://github.com/mesos/storm
[4] https://github.com/mesos/kafka

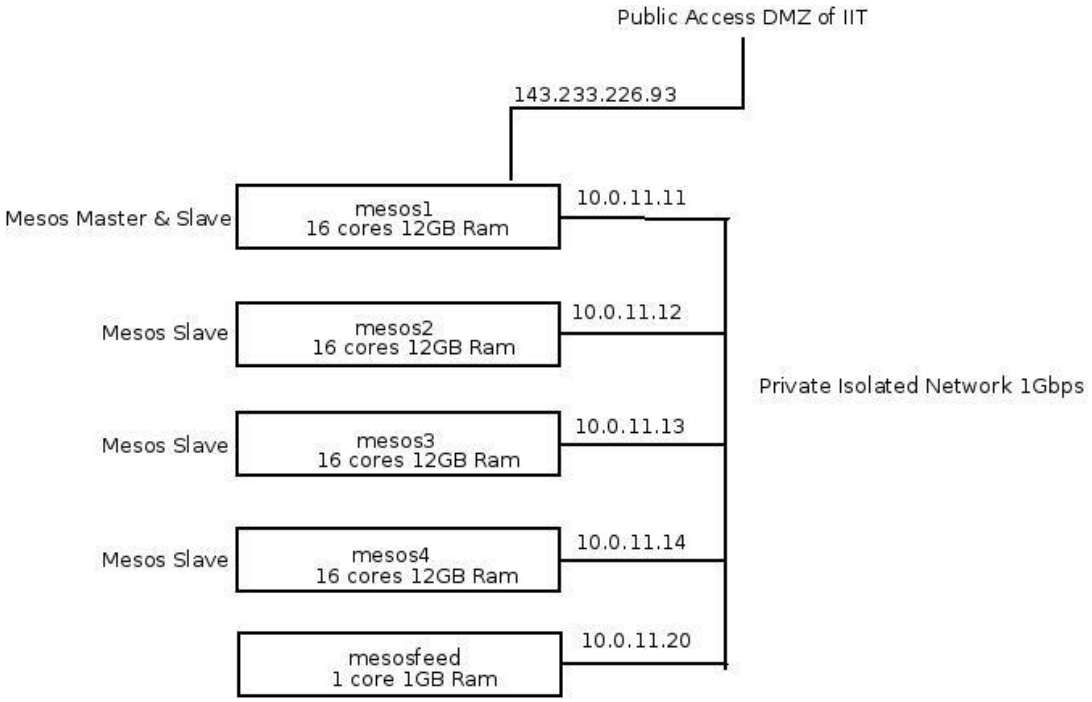

Figure 4-2 Mesos cluster topology

Table 4.1 - Performance Test Configurations states the different configurations tested (for information regarding STORM terminology and CEP parallelism hint please refer to section 4 of D6.5)

As one can see in the table, configurations that involve multiple Kafka partitions and brokers were not tested in this version. The reason for that is that initial test results have demonstrated that the messaging layer in its minimal configuration (single broker, single executor for kafka-storm spout, 1-2 executors for kafka-bolt) was operating significantly below its capacity, and further increase of messaging power would not improve the performance of the entire system.

Table 4.1 - Performance Test Configurations

Config	# Brokers	# Workers	CEP Par Hint	# Other Executors	# Other Tasks
1	1	1	1	1	1
2	1	1	2	2	2
3	1	2	4	4	4
4	1	4	8	4	4
5	1	4	16	8	8
6	1	16	16	8	8

3. Performance Test Results

We have used the Credit Card Fraud Management scenario for the performance evaluation. This use case imposes the strictest performance requirements (latency<25 milisec). We had several purposes in

the performance evaluation – to compare the performance of current prototype to the 2nd year version to see if the advancements in the implementation and architecture have managed to achieve their purpose and demonstrate a significant improvement in performance, to validate the performance of the prototype versus the use case requirements, and to demonstrate the ability to scale. Therefore we have built our test suite around the test suite of 2nd year to allow comparability of performance results, since this test suite was enough to achieve our other purposes.

With regards to processing latency improvements and scalability improvements our results demonstrate that:

- In 3rd year prototype we have managed to eliminate the bottlenecks leading to waiting locks and thread pool exhaustion, and which in turn led to exponential growth in processing latency over time. **Figure 4-3 Latency for 50 eps rate, 2nd year** , Figure 4-4 Latency for 50 eps, 3rd year **prototype**, **Figure 4-5 Latency for 500 eps, 2nd year** and **Figure 4-6 Latency for 500 eps,3rd year prototype** show the processing latencies measured in first performance evaluation (v2 in second year) and second performance evaluation (v3 in third year) for both 50 and 500 eps injection rates.

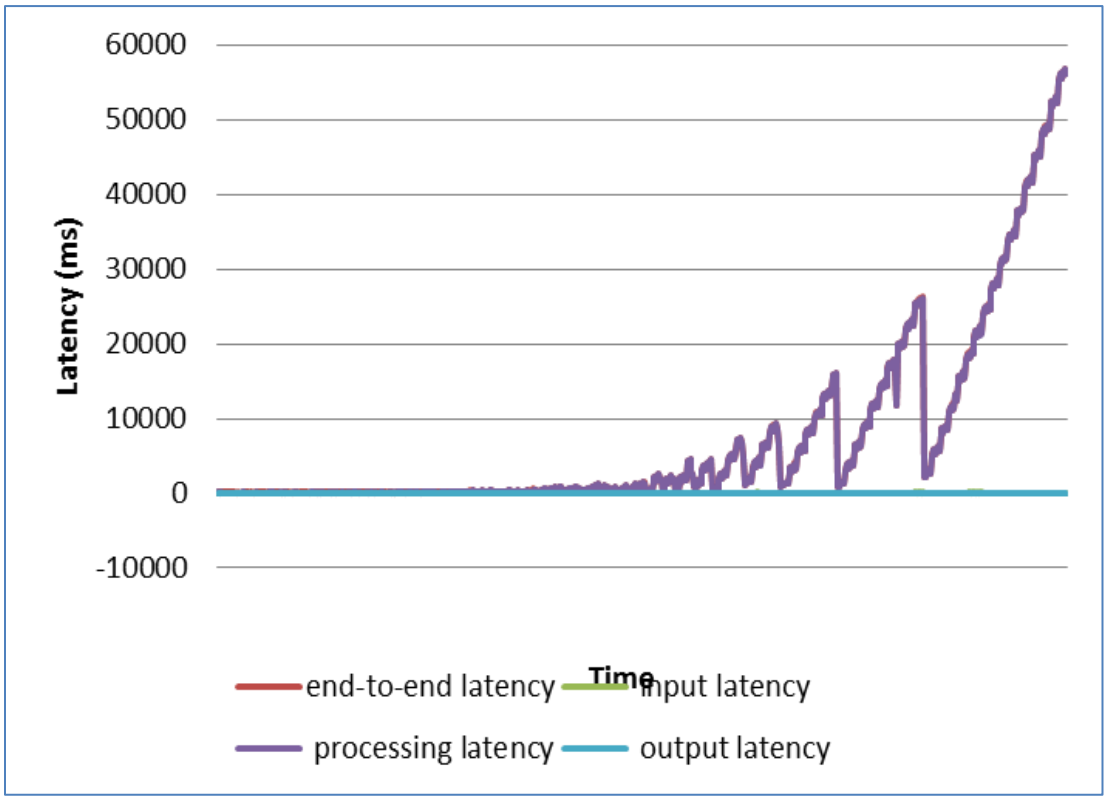

Figure 4-3 Latency for 50 eps rate, 2nd year

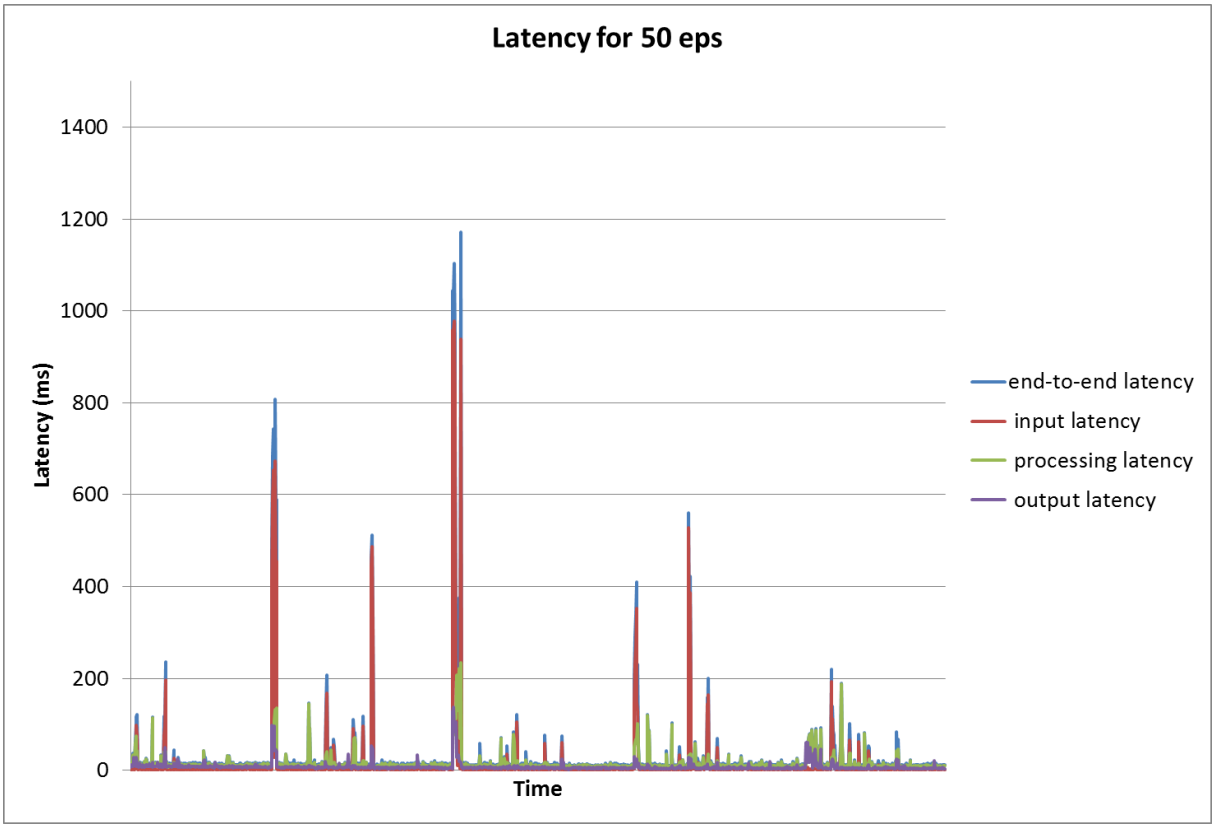

Figure 4-4 Latency for 50 eps, 3rd year prototype

Figure 4-5 Latency for 500 eps, 2nd year

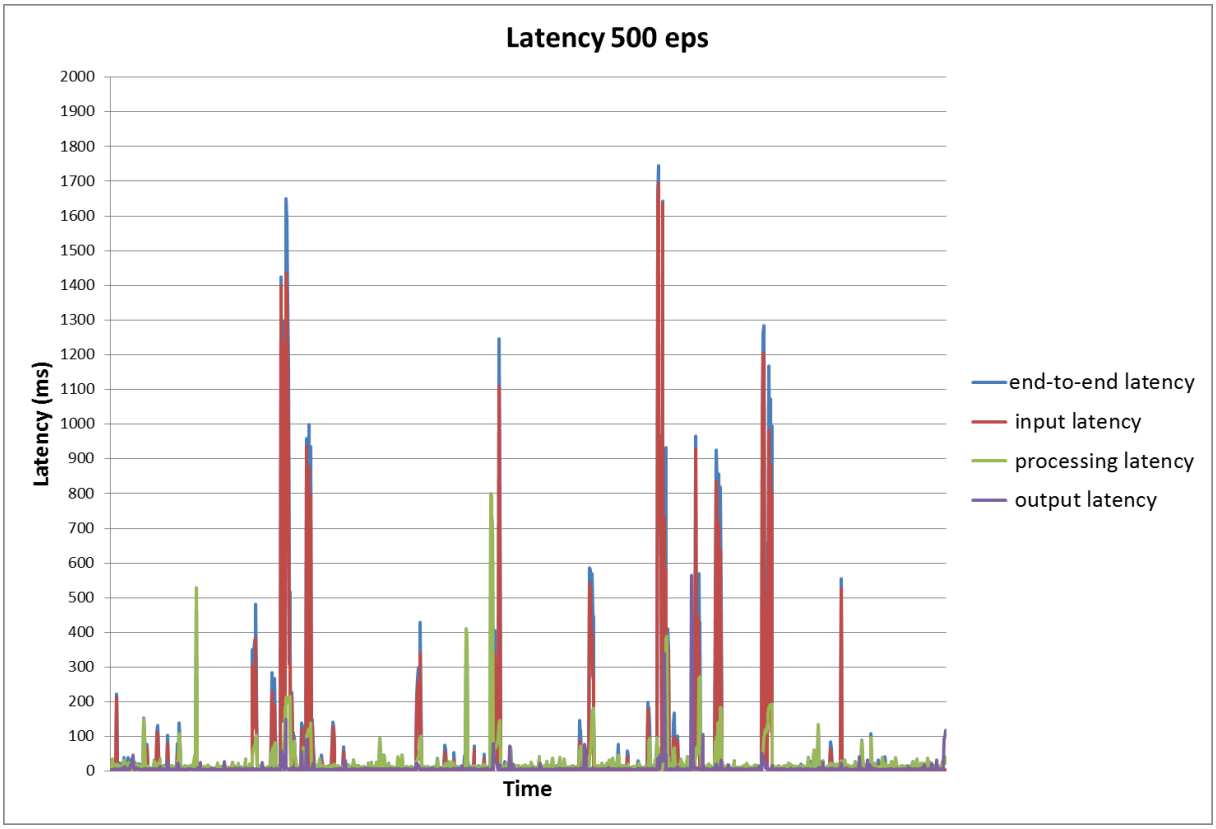

Figure 4-6 Latency for 500 eps,3rd year prototype

As demonstrated in the figures above, we can see that the processing latency was increasing exponentially over time in the second year prototype. For event rate of 50 eps the maximal processing latency reached was around 60,000 ms, while in the 3rd year prototype processing latency at peak reached 1,200 ms. For injection rate of 500 eps the change is even more dramatic – in 2nd year prototype the maximal latency observed was around 250,000 ms , while in 3rd year prototype peak latency is 1,600 ms.

The figures also demonstrate the lack of general trend of latency increase over time in 3rd year prototype – while there might be some peaks of latency (due to reasons which will be specified later in the section), the average latency stays the same over time, around 14 ms for injection rate of 50 eps and 45 ms for injection rate of 500 eps for this particular configuration which is depicted in the Figures (4 workers and parallelization factor 8).

- With regards to scalability, as described previously in section 3, and seen in **Figure 3-2 - Active context bolt instances out of all existing instances**, the scalability of the context bolt component was limited by the number of EPAs-context pairs in the application, impairing application's scalability . After the changes to the distribution scheme, the context bolt

scalability is no more limited to the application's metadata, as can be seen from **Figure 4-7 Scalability of context bolt, 3rd year prototype** (the active context bolts instances receiving and emitting tuples are marked in red, we can see that all of instances are active). We can see that in the 3rd year prototype, all existing context bolt instances process data, as opposed to **Figure 3-2 - Active context bolt instances out of all existing instances**, where out of all existing context bolt instances, tuples were routed only to 3 instances.

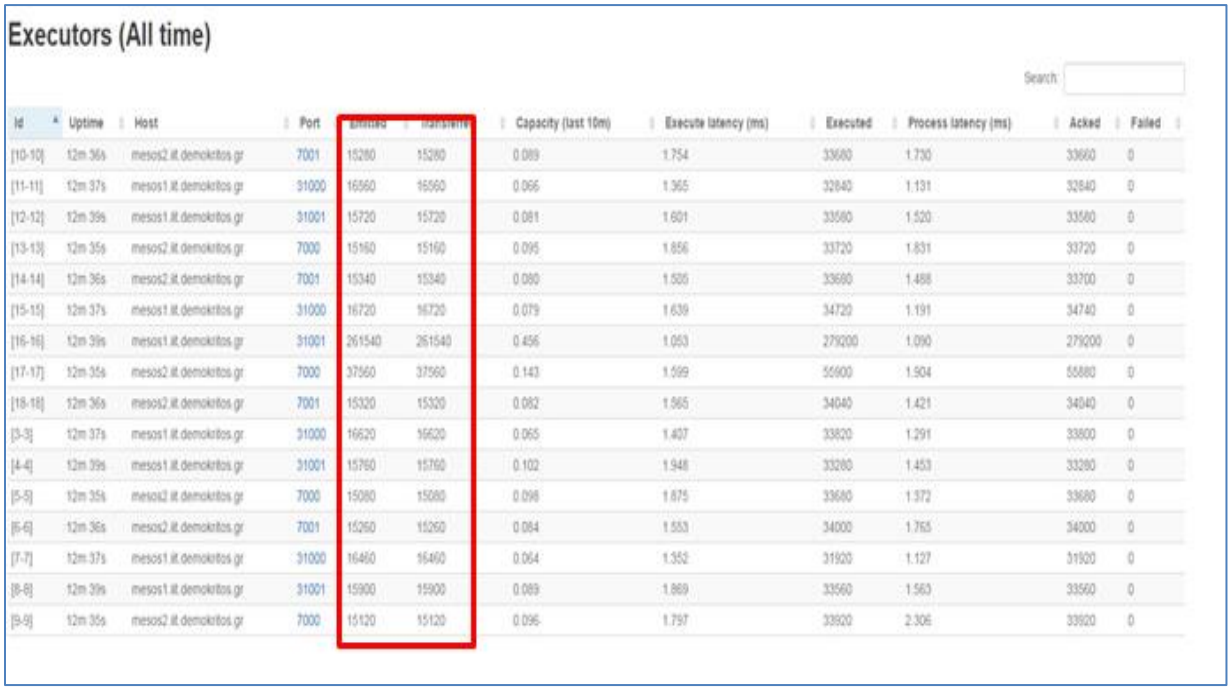

Figure 4-7 Scalability of context bolt, 3rd year prototype

The performance results are summarized in Table 4.2. In the section below we provide a detailed analysis of the observed results and our conclusions.

Table 4.2 - Performance Results Summary (90% percentile values)

Config	Number of workers	CEP parallelization factor	End-to-end latency (ms) 50 events/sec	End-to-end latency (ms) 500 events/sec
1	1	1	31	189
2	1	2	86	253
3	2	4	25	111
4	4	8	14	44.7
5	4	16	16	87
6	8	16	11	16

4. Performance Analysis and Conclusions

Current performance results can be seen in **Table 4.2 - Performance Results Summary (90% percentile values)**. The **Figure 4-8 Performance Results: Adding more workers improves latency** summarizes those

results. It can be seen from the trend in the figures that adding more executors to a worker doesn't help to achieve better latency results. On the contrary, it reduces the performance by a number of milliseconds. This is logical, as CEP already uses thread parallelization to distribute the data load between multiple threads within the same JVM, therefore adding more tasks to an existing JVM (worker) only adds management overhead on those additional tasks.
Adding more workers to the STORM topology on the other hand, significantly reduces the latency, since the data load is distributed between more JVMs, each one with its own separate resources. The influence of addition of workers to the higher injection rate topology is more dramatic, as in the case of 50 eps less workers are sufficient to handle the load and give satisfactory performance results, while as the injection rate increases a better data distribution can be achieved between numerous processing units, the less backlog is created per separate processing instances and the better the performance results.

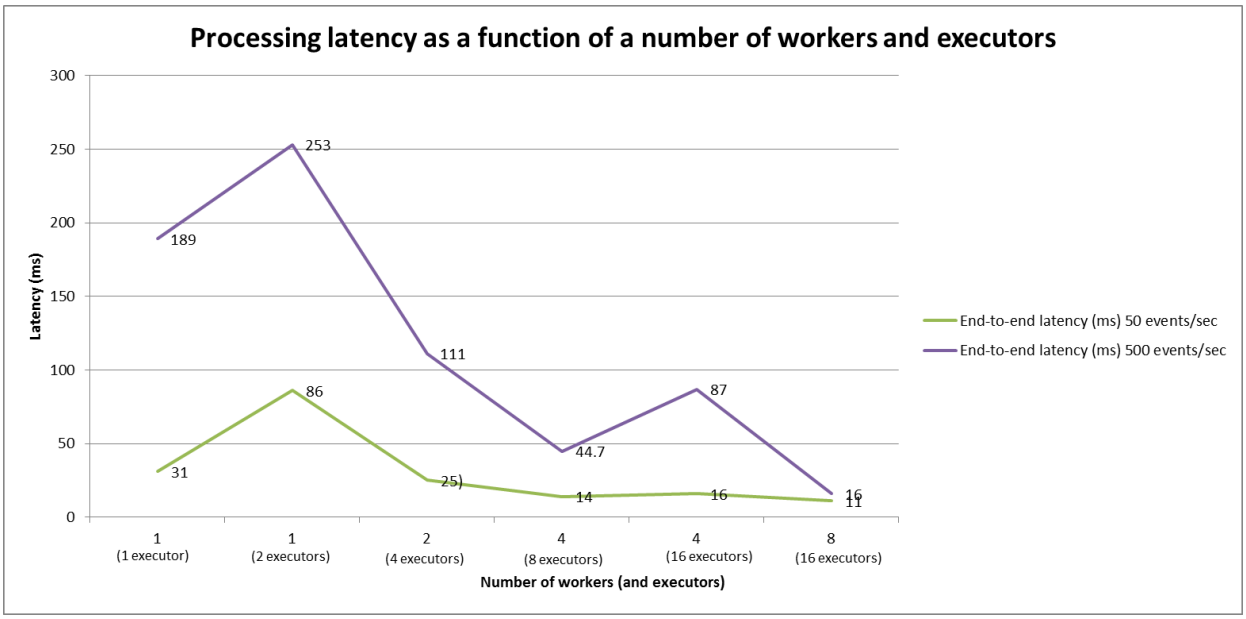

Figure 4-8 Performance Results: Adding more workers improves latency

The cluster on which the testing was performed is limited in resources and therefore cannot be configured to support large topologies consisting of hundreds of workers, however we can extrapolate by the given results that the system can be easily scaled out by adding sufficient amount of workers to the topology to allow for faster injection rates and providing the required performance for those rates.

Some additional points to consider when analyzing the performance results:

1. The provided values for end-to-end latency are the 90% percentile of the transactions. The average values are much lower (for example, for configuration 1 with 50 eps injection rate, the percentile latency is 31 ms, while the average latency is 20 ms). This is due to the peaks in latency which occur from time to time. The peaks usually occur across all types of latency (in-latency, processing latency and out-latency) at the same time, which leads us to the conclusion that at this moment the system is executing some additional work on the background, like JVM

garbage collection. In any case, we should keep in mind that on the average, the latency is even lower than summarized in the results Table 4.2 - Performance Results Summary (90% percentile values)

2. The placement of the STORM workers on the cluster nodes is managed by the storm-on-mesos framework. We have seen that the placement is far from optimized – in many configurations all the workers in the topology were placed on a single cluster node, leaving all the workload of the topology to this single node. The results of performance tests in such cases were probably suboptimal, but nevertheless they were included in the evaluation. With optimal placement of workers to allow for optimal cluster resource consumption we predict the performance will be further improved.

The detailed results of the performance testing are available as an excel spreadsheet in the project's document repository: http://speedd-project.eu/deliverables

5 Use case extensions

In this section we will summarize the additions of new events and patterns and changes to existing events and patterns which were done to demonstrate some UI features, to support uncertainty extensions in DM component and to farther promote the use case implementation. Further information concerning the changes can be found in the 7Appendix – SPEEDD Event Reference and in D3.3 Third version of event recognition and forecasting technology.

1. Fraud use case

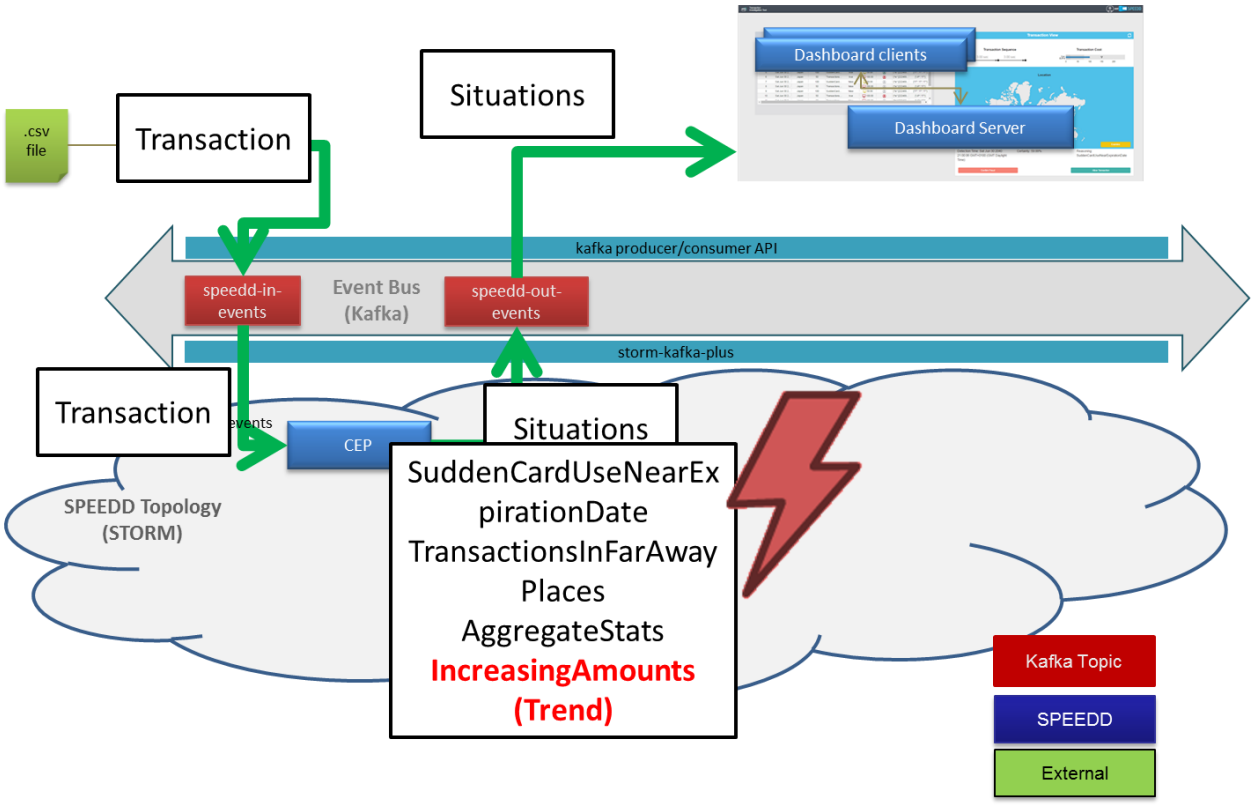

Figure 5-1 Fraud use case , 3rd year prototype

As can be seen in **Figure 5-1 Fraud use case , 3rd year prototype**, an IncreasingAmounts event has been added to the fraud use case implementation in the prototype. This event is a result of a Trend pattern detection, and visualized in a different manner than the rest of fraud events in the UI. For further information, please see the IncreasingAmounts for the event description, the D3.3 for the pattern description and D5.3 for visualization description.

2. Traffic use case

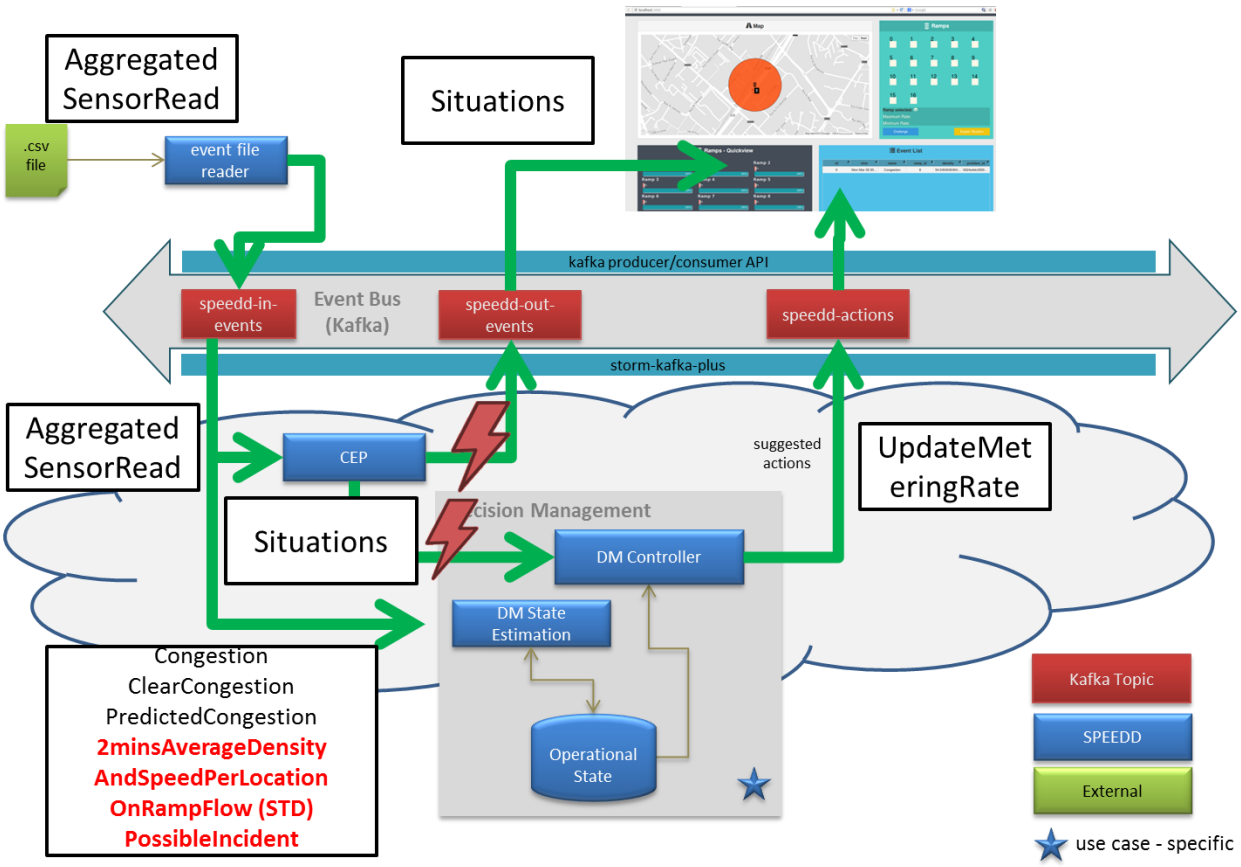

Figure 5-2 Traffic use case, 3rd year prototype

For the traffic use case, a new derived event was introduced in 3rd year prototype – PossibleIncident. This event depicts an occurrence of sudden buildup of traffic in the monitored section of the road, and is used by decision making in its traffic management algorithms.

Additionally the average calculation events were updated with standard deviation calculations for density and flow.

Additional information regarding the additions and updates to the patterns and events, and their usage in the DM module, can be found in IncreasingAmounts, the D3.3 deliverable and the D 4.3 deliverable.

6 Conclusions

In this document, we presented the updates to SPEEDD architecture performed in order to improve the performance and increase the scalability of the system. We have described the changes done to the Complex Event Processing engine to achieve those improvements. The DM component was extended with support for uncertainty in the input and derived events to allow for better and faster decision making. Additionally we have outlined the updates performed to the API events to support additional functionality added to the UI, DM and CEP modules as part of use cases implementation extension in year 3rd prototype (specifics can be found in the 3rd year deliverables of each module). Last but not least, a second performance evaluation was performed with the same tests/configurations as first year performance to allow comparability of results.

To sum up our findings in the second performance evaluation:

- We have achieved drastic improvement in the latency results in the 3rd year prototype as opposed to the 2nd year prototype
- The scalability bottleneck in the prototype was eliminated; the data load can be distributed and processed optimally between all the instances of processing tasks in the cluster.
- We have demonstrated the scalability potential of SPEEDD architecture. The BigData systems comprise of clusters of hundreds of nodes running thousands of processing instances. Given such infrastructure and based on the scalability trend we have seen in the performance evaluation, we have shown that the SPEEDD architecture can be scaled out horizontally to achieve required goals.

7 Appendix – SPEEDD Event Reference

This section contains reference information about the event types used in SPEEDD along with the structure of the event objects.

1. Common attributes for all events emitted by Proton

Attribute Name	Attribute Type	Description
Certainty	Double	The certainty that this event happen (value between 0 to 1)
OccurrenceTime	Date	No value means it equals the event detection time, other option is to use one of the defined distribution functions with parameters
ExpirationTime	Date	Only till this time the cost and certainty parameters of the event are valid, and only till this time a proactive action is considered
Cost	Double	The cost of this event occurrence. Negative if this is an opportunity
Duration	Double	Used in case the this event occur within an interval

2. Credit Card Fraud Management Use Case

7.1.1 Transaction

Attribute Name	Attribute Type	Description
card_pan	String	Hashed card PAN
terminal_id	String	Unique terminal ID
transaction_id	String	Unique identified of the transaction
cvv_validation	Integer	CVV validation response code
amount_eur	Double	Transaction amount in EUR
acquirer_country	Integer	Acquirer country code
card_country	Integer	Card country code
is_cnp	Integer	CNP (“Card Not Present”) transaction indicator: 1 if CNP, 0 if CP
card_exp_date	Date	Card expiration date

7.1.2 SuddenCardUseNearExpirationDate

Attribute Name	Attribute Type	Description
card_pan	String	Hashed card PAN
TransactionsCount	Integer	Number of transactions in the pattern
is_cnp	Integer	CNP (“Card Not Present”) transaction indicator: 1 if CNP, 0 if CP
transaction_ids	String[]	ids of transactions that contributed to the pattern
timestamps	Long[]	Timestamps of the transactions that

		contribute to the pattern
acquirer_country	Integer	Acquirer country code
card_country	Integer	Card country code

7.1.3 TransactionsInFarAwayPlaces

Attribute Name	Attribute Type	Description
card_pan	String	Hashed card PAN
transaction_ids	String[]	ids of transactions that contributed to the pattern
timestamps	Long[]	Timestamps of the transactions that contribute to the pattern
acquirer_country	Integer	Acquirer country code
card_country	Integer	Card country code

7.1.4 IncreasingAmounts

Attribute Name	Attribute Type	Description
card_pan	String	Hashed card PAN
TrendCount	Integer	Number of transactions in the participant set
is_cnp	Integer	CNP ("Card Not Present") transaction indicator: 1 if CNP, 0 if CP
transaction_ids	String[]	ids of transactions that contributed to the pattern
amounts	Double[]	Amounts of all transactions that contribute to the pattern
timestamps	Long[]	Timestamps of the transactions that contribute to the pattern
acquirer_country	Integer	Acquirer country code
card_country	Integer	Card country code

7.1.5 TransactionStats

Attribute Name	Attribute Type	Description
Country	Integer	Country code
average_transaction_amount_eur	Double	Average transaction amount over the measured period
transaction_volume	Double	Total transaction volume
transaction_count	Double	Number of transactions counted

3. Traffic Management Use Case

7.1.6 AggregatedSensorRead

Attribute Name	Attribute Type	Description

location	String	Id of the sensor location (collection point)
lane	String	Lane (e.g. slow, fast, onramp, offramp)
occupancy	Double	Fraction of time that the cross-section of the sensor is occupied (%)
vehicles	Integer	Number of vehicles passed over the sensor
average_speed	Double	Average speed of the vehicles over the reported period

7.1.7 SimulatedSensorReadingEvent

Attribute Name	Attribute Type	Description
detectorId	String	Id of the simulation detector (imitating sensor)
dm_location	String	Location-based partition id
vehicle_speed	Double	Average speed
vehicle_count_car	Integer	Number of cars passed over the sensor
vehicle_count_truck	Integer	Number of trucks passed over the sensor
density_car	Double	Average density of cars
density_truck	Double	Average density of trucks
occupancy	Double	Average occupancy of the section

7.1.8 Congestion

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
dmPartition	String	Location-based partition id
sensorId	String	Identifies the simulated detector ID
average_density	Double	Average density of the vehicles over the reported period
problem_id	String	Identifies the problem detected by SPEEDD

7.1.9 PredictedCongestion

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
average_density	Double	Average density of the vehicles over the reported period
problem_id	String	Identifies the problem detected by SPEEDD

7.1.10 ClearCongestion

Attribute Name	Attribute Type	Description

location	String	Id of the sensor location (collection point)
problem_id	String	Identifies the problem detected by SPEEDD
dmPartition	String	Location-based partition id
sensorId	String	Detector id of the simulated sensor

7.1.11 CoordinateRamps

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector id of the simulated sensor
targetOccupancy	Double	Calculated target occupancy
problem_id	String	Identifies the problem detected by SPEEDD
dmPartition	String	Location-based partition id

7.1.12 AggregatedQueueLength

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector id of the simulated sensor
queueLength	Double	Calculated queue length for the location
maxQueueLength	Double	Maximum possible queue length for the given location
dmPartition	String	Location-based partition id

7.1.13 PredictedRampOverflow

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector id of the simulated sensor
dmPartition	String	Location-based partition id

7.1.14 PossibleIncident

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector id of the simulated sensor
dmPartition	String	Location-based partition id
problem_id	String	Identifies the problem detected by SPEEDD

7.1.15 ClearOnRampOverflow

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector id of the simulated sensor
dmPartition	String	Location-based partition id

7.1.16 AverageOnRampValuesOverInterval

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector Id
average_flow	Double	Average flow of the traffic over the reported period
average_speed	Double	Average speed of the vehicles over the reported period
average_occupancy	Double	Average density of the traffic over the reported period
dmPartition	String	Location-based partition id
standard_dev_flow	Double	Standard deviation in average_flow among the onramp events falling within time window
standard_dev_density	Double	Standard deviation in average_density among the onramp events falling within time window

7.1.17 AverageOffRampValuesOverInterval

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector Id
average_flow	Double	Average flow of the traffic over the reported period
average_speed	Double	Average speed of the vehicles over the reported period
average_occupancy	Double	Average density of the traffic over the reported period
dmPartition	String	Location-based partition id
standard_dev_flow	Double	Standard deviation in average_flow among the onramp events falling within time window

standard_dev_density	Double	Standard deviation in average_density among the onramp events falling within time window

7.1.18 AverageDensityAndSpeedPerLocationOverIntervall

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
sensorId	String	Detector Id
average_flow	Double	Average flow of the traffic over the reported period
average_speed	Double	Average speed of the vehicles over the reported period
average_occupancy	Double	Average density of the traffic over the reported period
dmPartition	String	Location-based partition id
standard_dev_flow	Double	Standard deviation in average_flow among the onramp events falling within time window
standard_dev_density	Double	Standard deviation in average_density among the onramp events falling within time window

7.1.19 AverageDensityAndSpeedPerLocation

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
average_flow	Double	Average flow of the traffic over the reported period
average_speed	Double	Average speed of the vehicles over the reported period
average_density	Double	Average density of the traffic over the reported period

7.1.20 PredictedTrend

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
problem_id	String	Identifies the problem detected by SPEEDD
dmPartition	String	Location-based partition id
sensorId	String	Detector id of the simulated sensor

average_density	Double	The value of "average_density" (the normalized density) attribute in the last participant in the trend
count	Integer	Number of participants in the detected trend

4. Traffic Control Actions

The following events are emitted by the Decision Making module in order to mitigate or prevent congestion via controlling the metering rates on the ramp, i.e. the fraction of the green light.

7.1.21 UpdateMeteringRateAction

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
lane	String	Lane (e.g. slow, fast, onramp, offramp)
density	Double	Current density at the controlled section
newMeteringRate	Double	New value of the metering rate (fraction of the green light)
controlType	String	auto | partial | full

7.1.22 setMeteringRateLimits

This event is emitted by the dashboard application in response to the operator's command to limit the automatic metering rates to the specified range.

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
upperLimit	Double	Maximal value of the metering rate
lowerLimit	Double	Minimal value of the metering rate

5. AIMSUN Simulation Control Commands

7.1.23 setTrafficLightPhaseTime

Attribute Name	Attribute Type	Description
junctionID	Integer	Intersection id
phaseID	Integer	Phase of the traffic light
phaseTime	Iinteger	New phase time (seconds)

7.1.24 setSpeeddLimit

Attribute Name	Attribute Type	Description
sectionID	Integer	Controlled section of the road
speedLimit	Integer	New speed limit