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

Project	**FP6-619435/SPEEDD**
Deliverable	**D6.1 (amended) and D6.5**
Distribution	**Public**

http://speedd-project.eu

D6.1 (amended) and D6.5

The Architecture Design of the SPEEDD Prototype and Second Integrated Prototype

Alex Kofman (IBM), Fabiana Fournier (IBM), Inna Skarbovsky (IBM), Natan Morar (UoB), Marius Schmitt (ETH), Chithrupa Ramesh (ETH), Jason Filippou (NCSR), Elias Alevizos (NCSR), Rohit Singhal (CNRS), Jonathan Yom Tov (Technion)

Status: Final (Version 2.1)

February 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	Alina Lupu

Deliverable

Deliverable type	Report
Distribution level	Public
Deliverable Number	D6.1 (amended) and D6.5
Deliverable Title	The Architecture Design of the SPEEDD Prototype and Second Integrated Prototype
Contractual date of delivery	M24 (January 2016)
Actual date of delivery	January 2016
Relevant Task(s)	WP6/Tasks 6.3
Partner Responsible	IBM
Other contributors	NCSR, CNRS, Feedzai, ETH, UoB, Technion
Number of pages	90
Author(s)	Alex Kofman (IBM), Fabiana Fournier (IBM), Inna Skarbovsky (IBM), Natan Morar (UoB), Marius Schmitt (ETH), Chithrupa Ramesh (ETH), Jason Filippou (NCSR), Elias Alevizos (NCSR), Rohit Singhal (CNRS), Jonathan Yom Tov (Technion)
Internal Reviewers	Feedzai
Status & version	Final v2.1
Keywords	architecture design scalability cep decision-making proactive

Executive Summary

SPEEDD (Scalable ProactivE Event-Driven Decision making) will develop a system for proactive event-based decision-making: decisions will be triggered by forecasting events – whether they correspond to problems or opportunities – instead of reacting to them once they happen. The decisions and actions will be real-time, in the sense that they will be taken under tight time constraints, and require on-the-fly processing of "Big Data", i.e. extremely large amounts of noisy data storming from different geographical locations as well as historical data.

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

The purpose of this document is to describe the design of the SPEEDD prototype architecture. It discusses main architectural questions and decisions made to build a proactive decision support system that satisfies requirements of the two representative use cases. The document describes APIs provided by the prototype for integration, extensibility, and automation. We define the, main test cases for testing the prototype. Finally, the current version of the document describes our approach to performance and scalability evaluation of the prototype, discussing the challenges, the methodology used for performance evaluation, and the analysis of the initial results.

The work done so far on the architecture design includes analysis of the use case requirements, drafting the conceptual architecture and the corresponding technical architecture that should allow building the technology that satisfies the requirements. The following refinement of the high-level technical architecture involved evaluation of available technologies and selecting the appropriate stream processing and messaging platforms. Additionally, the event-driven architecture paradigm has been selected as the main architectural principle for SPEEDD integration. During the last year the architecture has been updated according to the enhancements done in different components.

Architectural decisions documented in the current deliverable guide the development of SPEEDD prototype. In the process of the development and according to the issues and questions identified, the architecture decisions are constantly revised, refined, and modified.

Contents

List of Tables

List of Figures

1 Introduction

1.1 History of the document

Version	Date	Author	Change Description
1.0	31/01/2015	Alexander Kofman (IBM)	First version of the architecture document
2.0 draft	24/01/2016	Alexander Kofman (IBM)	Draft of the architecture document v2.0
2.0	28/01/2016	Alexander Kofman (IBM)	Incorporate internal review comments
2.1	25/02/2016	Alexander Kofman (IBM)	Complete performance analysis results

1.2 Purpose and Scope of the Document

This is the second, amended version of the report on the design of the SPEEDD prototype architecture. It discusses main architectural questions and decisions made to build a proactive decision support system that satisfies the requirements of the two representative use cases. The document describes APIs provided by the prototypes for integration, extensibility, and automation.

1.3 Relationship with Other Documents

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 D3.2 where one can find more information on the architecture and the event patterns.

The Decision Making module algorithms are described in D4.2.

For more details on the traffic simulation module please refer to D8.2.

The dashboard application design and the underlying approach are explained in D5.2.

The scalability component's approach and algorithms are described in D6.4.

1.4 Updates since the first version

As mentioned in the previous section, the current document contains updates, corrections, and more detailed descriptions to the concepts introduced in the first design report (D6.1).

Specifically, the chapters discussing the complex event processing, the decision making, and the dashboard modules have been updated corresponding to the enhancements done in the new version (v2) of these components (sections 2.4.4 - 2.4.6 respectively).

A description of the architecture of the scalability module is now available in 2.4.7.

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

The setup guide has been updated with instructions for running SPEEDD runtime in docker[1] containers.

The details of the integration of AIMSUN traffic simulator and SPEEDD runtime are available in appendix 11.

Following the request from the last project review, we have added the appendix 12 that contains reference for all the events used in SPEEDD, thus serving as the API reference for SPEEDD.

2 SPEEDD Prototype Architecture

2.1 System Requirements

The requirements for the current prototype are derived from two problem domains – traffic management, and credit card fraud management. The detailed requirements for each domain are available in the deliverable D8.1 (Traffic) and D7.1 (Fraud) respectively.

The prototype should provide authoring tools that could be applied to the historic data in order to derive event pattern definitions and decision models to be deployed in runtime, as well as a scalable runtime system capable of detecting and predicting important business situations (traffic conditions, credit card fraud attempts) and issuing automatic actions aimed at preventing undesired situations.

To support credit card fraud detection scenario, it is required to provide continuous throughput of 1000 transactions per second, with latency less than 25 milliseconds. Availability is an important requirement for the fraud detection system, 99.9% is stated by the document. As the goal of the current project is implementing a prototype and not an operational system, we aim at building the architecture which could be further evolved and expanded to provide the required level of availability rather than achieving and testing the availability compliance of the prototype.

For the traffic management scenario, the projected throughput is 2000 sensor readings per second (computed based on the amount of sensors and the report frequency, assuming aggregated readings sent every 15 seconds by each of the 130 Sensys sensors installed along the Grenoble South Ring).

As we scale out the problem presented in the Grenoble traffic management scenario from the level of a single highway (or even a few ones) to managing traffic in the entire city, and further to managing traffic at the regional level and at the country-level, the requirements scale up correspondingly. Also, there are additional challenges:

- Large number of event sources – sensors and others
- Event rate grows correspondingly
- Network latency and partitioning

[1] https://www.docker.com/

In terms of integration with external systems the following is required:

- replay historic events from text files or a database (traffic, fraud)
- receive sensor reading messages generated by the micro-simulator (traffic)
- provide a mechanism to log output events and actions to a log for subsequent research
- provide a mechanism to connect to the traffic micro-simulator for updating the simulator configuration – action simulation

It is important to mention that many of the requirements learned for the traffic management and the credit card fraud management also applicable to many other domains; thus the current work should be also extensible to additional areas and use cases.

2.2 Approach

The design of the system architecture for a prototype like SPEEDD is an iterative process that starts with the beginning of the project and continuously evolves, as requirements of the different components are better understood and insights are gained. Therefore, a close iterative and collaborative process was carried out between the architecture team in WP6 "Scalability and System Integration" lead by IBM, and the technical teams of the SPEEDD prototype, specifically the teams of the real-time event recognition and forecasting (WP3), real-time decision making (WP4), real-time visual analytics (WP5), scalability (WP6), and the technical teams from the use cases (WP7 and WP8).

To this end we followed the steps below:

1. Iterative biweekly virtual meetings that included representatives of all partners involved. A very draft architecture presented at M3 of the project has been frequently updated and refined based on input and feedback to the current architecture (described in sections 2.3 - 2.8).
2. On a case-by-case basis, bilateral virtual meetings with a specific partner to elaborate on a specific issue (e.g., specific API).
3. Face-to-face meetings during the project meetings in May and September 2014. As part of every plenary project meeting, we're having a "Code Camp" day dedicated to architecture and integration technical discussions and work.

2.3 Conceptual Architecture

This section provides a high-level overview of SPEEDD prototype. The goal is to introduce the main concepts, high-level components and information flow without getting into implementation and technological details.

Figure 2.1 illustrates the conceptual architecture of SPEEDD prototype. We separate between the design time and the run time. The products of the design time activities are event processing definitions and decision making configurations that will be deployed and executed at the runtime.

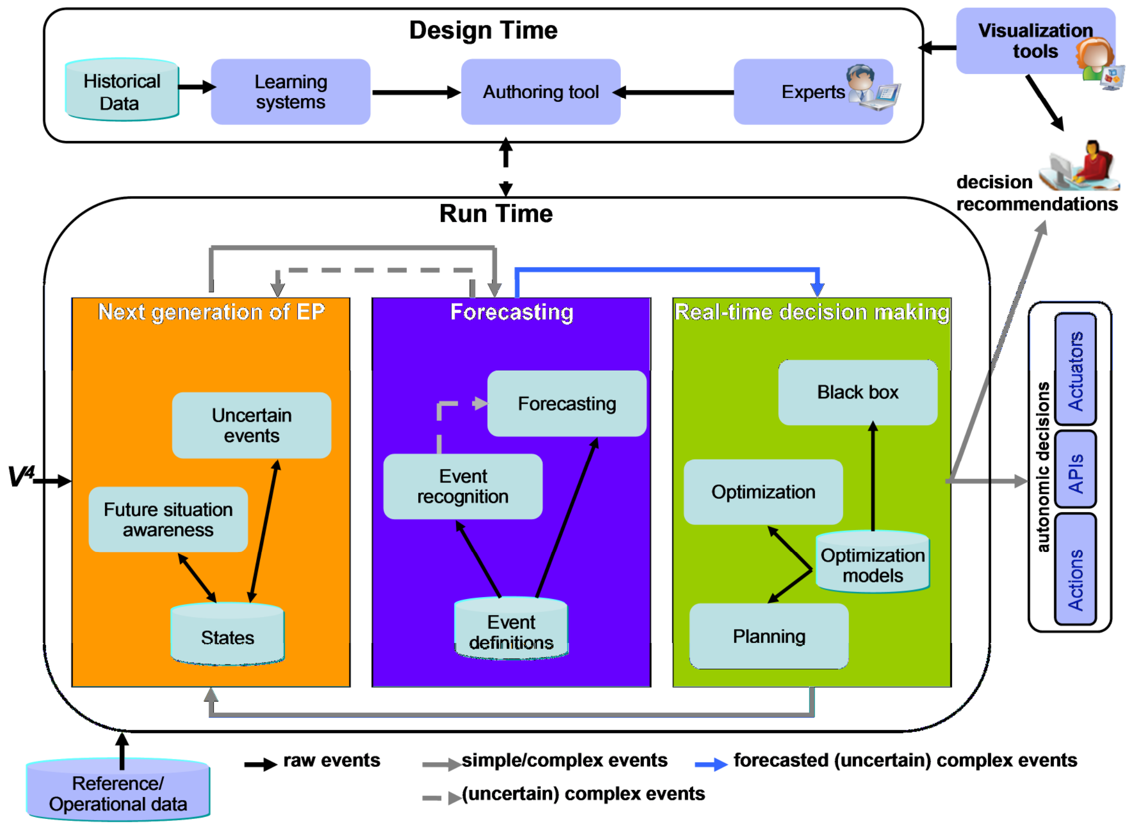


Figure 2.1 - Conceptual Architecture of SPEEDD Prototype

Historic data used at design time contains raw events reported during the observed period along with annotations provided by domain experts. These annotations mark important situations that have been observed in past and should be detected automatically in the future. Visualization tooling is used to sift through historic data to gain insights and create annotations. Domain experts apply tools and methodologies provided by SPEEDD authoring toolkit to extract derived event definitions from the annotated event history. This is a semi-automatic process involving applying machine learning tools to extract initial set of patterns which is further enhanced and translated with help of the domain experts into deployable CEP artefacts.

The runtime part is composed of the CEP component, the automatic decision making component, and visual decision support tooling. SPEEDD runtime receives raw events emitted by the various event sources (e.g. traffic sensors, transactional systems, etc., - depending on the use case) and emits actions that are consumed by the actuators connected to the operational systems or simulators.

The CEP component has been extended to cope with detecting and forecasting derived events under uncertainty. It processes raw as well as derived (detected and forecasted) events to detect and forecast higher-level events, or situations. These serve as triggers for the decision making component, which uses domain-specific algorithms to suggest the next best action to resolve or prevent an undesired situation.

The visualization component (further called the dashboard) facilitates decision making process for business users by providing easily comprehensible visualization of detected or forecasted situations along with output of the automatic decision making component – a list of suggested actions to deal with the situation. The SPEEDD system can be run in either open or closed loop mode. In case of the open loop, the user can approve, reject, or modify the action proposed by the automatic decision maker. The closed loop operation does not require user's approval, - the action is performed automatically[2]. A hybrid mode where some types of actions are taken automatically while other types require human attention is also supported; moreover, we believe that this mode is the most realistic one.

2.4 SPEEDD Runtime Architecture

The architecture of the runtime part of SPEEDD follows the Event-Driven Architecture paradigm[3]. This approach facilitates building loosely coupled highly composable systems as well as provides close alignment with the real world problems, including our representative use cases. Every component functions as an event consumer, or an event producer, or a combination of both. The event bus plays a central role in facilitating inter-component communication which is done via events. Figure 2.2 provides a refinement of the conceptual architecture described above where the runtime part is represented as a group of loosely-coupled components interacting through events. The event bus serves as the communication and integration platform for SPEEDD runtime.

[2] Actuators are out of scope of SPEEDD prototype. Under automatic action we mean that the message representing the action type and parameters is emitted by SPEEDD, so that the actual operational system listening to action events is supposed to execute it.
[3] G. Hohpe. Programming without a call stack – Event-driven Architecture. 2006, [Online]. At: http://www.eaipatterns.com/docs/EDA.pdf

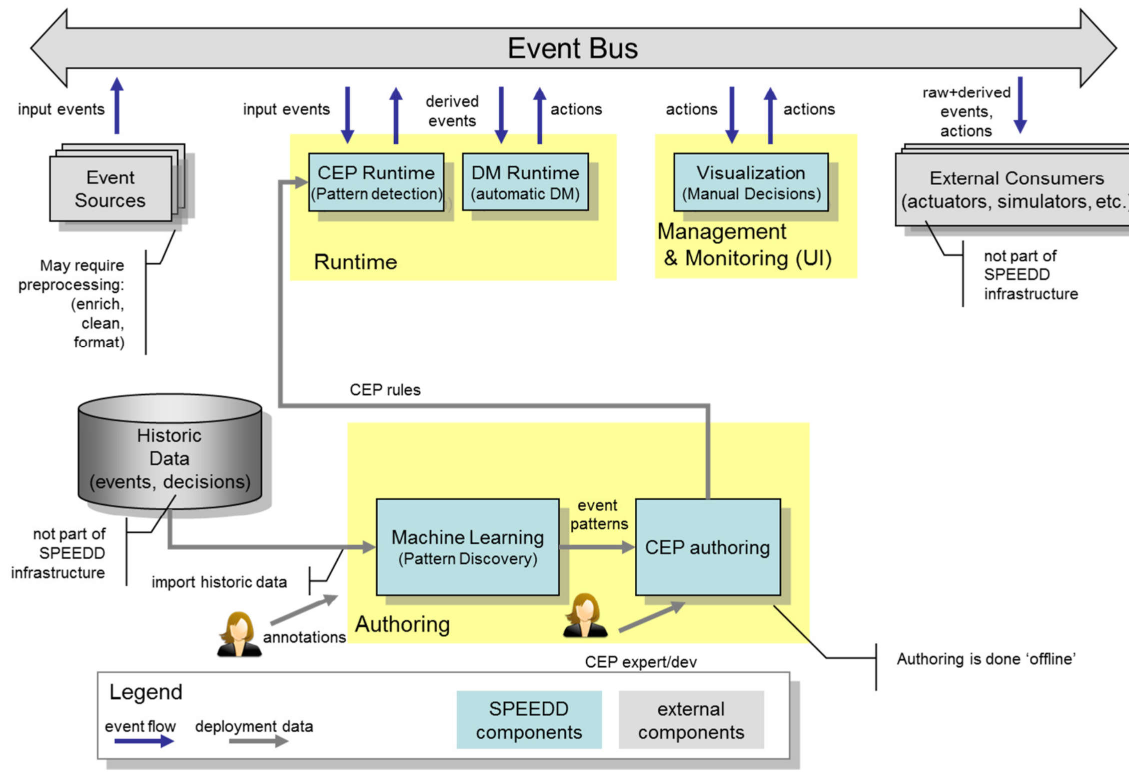

Event Bus
input events
input events
derived events
actions
actions
actions
raw+derived events, actions
Event Sources
CEP Runtime (Pattern detection)
DM Runtime (automatic DM)
Runtime
Visualization (Manual Decisions)
Management & Monitoring (UI)
External Consumers (actuators, simulators, etc.)
not part of SPEEDD infrastructure
May require preprocessing: (enrich, clean, format)
CEP rules
Historic Data (events, decisions)
not part of SPEEDD infrastructure
import historic data
Machine Learning (Pattern Discovery)
event patterns
CEP authoring
annotations
Authoring
CEP expert/dev
Authoring is done 'offline'
Legend
event flow
deployment data
SPEEDD components
external components

Figure 2.2 - SPEEDD - Event-Driven Architecture

Input from the operational systems (traffic sensor readings, credit card transactions) are represented as events and injected into the system by posting a new event message to the event bus. These events are consumed by the CEP runtime. The derived events representing detected or forecasted situations that CEP component outputs are posted to the event bus as well. The decision making module listens to these events so that the decision making procedure is triggered upon a new event representing a situation that requires a decision. The output of the decision making represents the action to be taken to mitigate or resolve the situation. These actions are posted as action events. The visualization component consumes events coming from two sources: the situations (detected as well as forecasted) and the corresponding actions suggested by the automatic decision components. Architecturally there is no difference between these two – both are events that the dashboard is 'subscribed to', although having different semantics and presented and handled differently. The user can accept the suggested action as is, modify the suggested action's parameters, or reject it (and even decide on a different action). In the case where an action is to be performed, the resulting action will be sent as a new event to the event bus so that the corresponding actuators are notified.

In the following subsections we describe the details of the runtime architecture including the design of each component and explain how the technology is being used to implement it.

Figure 2.3 and Figure 2.4 illustrate the SPEEDD runtime architecture for the traffic and credit card fraud use cases respectively. These diagrams include the technology platforms used to implement the architecture. We will use these illustrations as we discuss the details of each component.

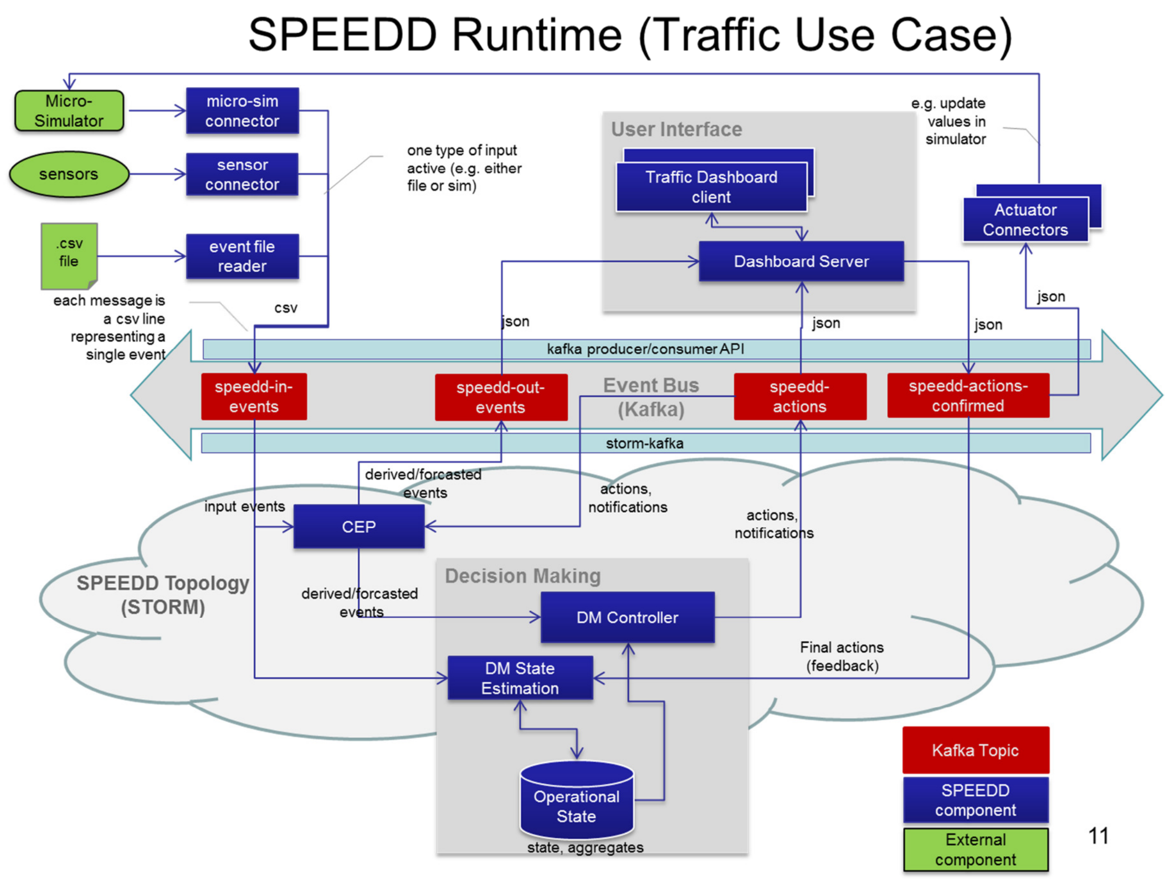


Figure 2.3 - SPEEDD Runtime - Event-Driven Architecture (Traffic Use Case)

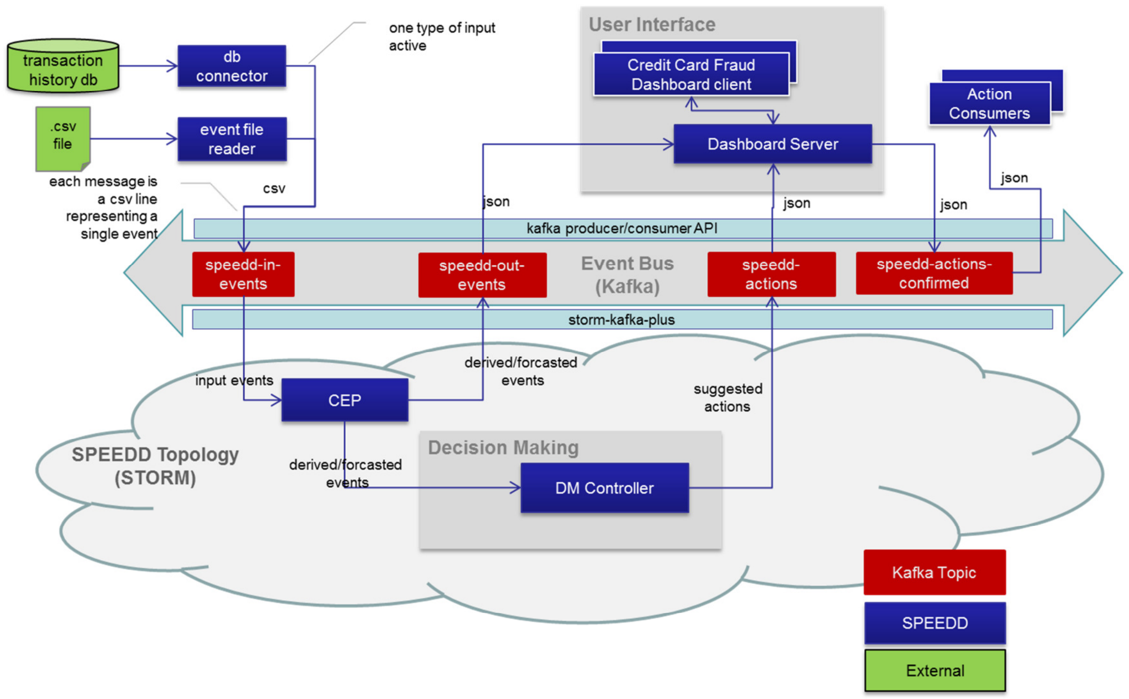


Figure 2.4 - SPEEDD Runtime - Event-Driven Architecture (Credit Card Fraud Use Case)

2.4.1 Event Bus

The technology chosen for the event bus component is Apache Kafka[4] (Kreps, Narkhede and Rao 2011). It provides a scalable, performant, and robust messaging platform that matches SPEEDD requirements (see 6.5 for our technology evaluation results). To implement routing of the events to event consumers we build upon the topic-based routing mechanism provided by Kafka. In Table 2.1 one can find the topics used by SPEEDD runtime along with the information about what components produce events or consume events for every topic.

Table 2.1 - Kafka topics in SPEEDD event bus

Topic Name	Description	Producers	Consumers
speedd-in-events	Input events	Event sources (e.g. traffic sensor readers, credit card transaction systems, file readers for replay etc.)	CEP runtime
speedd-out-events	Detected/Forecasted events	CEP, Decision making	Decision making, Dashboard
speedd-actions	Suggested decisions	Decision making	Dashboard, CEP
speedd-actions-confirmed	Actions confirmed for execution	Dashboard (in open loop mode), Decision making (in closed loop mode)	Dashboard, Actuators, CEP

[4] http://kafka.apache.org/

Topic Name	Description	Producers	Consumers
speed-admin	Administrative actions for configuring SPEEDD runtime	Dashboard	Decision Making

To allow scalable processing of massive stream of messages at high throughput Kafka provides the partitioning mechanism. Every topic can be partitioned into multiple streams that can be processed in parallel, while every partition can be managed in a separate machine. There may be more than one replica for every partition, thus providing resilience in case of failures.

In SPEEDD we exploit Kafka partitioning to build a scalable and fault-tolerant event bus. The topic that receives the biggest incoming traffic is speedd-in-events where all the input events are sent. The decision about the partitioning mechanism to use is use-case specific as we want to achieve nearly uniform distribution of load over different partitions. Below we describe the partitioning approach for each use case, providing the rationale for the design decisions. It is important to mention, though, that we may change the final partitioning mechanism based on the performance experiments on real and simulated data. We will be able to do that at any stage of the project development, thanks to the highly extensible and customizable partitioning framework that Kafka provides.

2.4.1.1 Partitioning for the Traffic Use Case

Assuming that we get relatively equal amount of events produced by every sensor, we could partition sensor reading events based on the sensor id. This should result in uniform distribution of the messages to partitions, which provides horizontal scalability of the topic. In v2 of the prototype we slightly enhance the partitioning strategy by using an additional attribute, dm_location, which identifies a road section which may include multiple sensors. This strategy is beneficial because it takes into account geographical location and proximity of the sensors. Given that all partitions contain similar amount of sensors we still get uniform distribution of the messages to partitions.

2.4.1.2 Partitioning for the Credit Card Fraud Use Case

For the credit card fraud use case, the card Primary Account Number (PAN) uniquely identifies the card. It is questionable though if we can assume uniform distribution of transactions among all card owners. Therefore the most suitable partitioning seems to be 'random' partitioning, that should guarantee uniform partitioning of the messages in the topic.

2.4.1.3 Ordering of events

Kafka guarantees that the order of events submitted to a topic's partition is preserved within same partition – the consumers will receive them in the same order. However, the order is not guaranteed across partitions. In our case this should not be an issue because the CEP component takes care of the out-of-order events as long as the delay between the event and its preceding event that arrives after that event is not too long – this assumption should be valid with Kafka.

2.4.1.4 Storm-Kafka Integration

SPEEDD event processing and decision making components run on top of Apache Storm (Toshniwal, et al. 2014), - a distributed scalable stream processing infrastructure (see 2.4.4.6, 6.2 for details).

Integration between Storm streaming platform and our Kafka-based event bus is done based on the storm-kafka integration module which has become a part of the storm project since v0.9.3[5]. The integration module provides two building blocks. KafkaSpout listens on a Kafka topic and creates a stream of the tuples. KafkaBolt posts incoming tuples to a configured topic. There is an extensible mechanism for serialization and deserialization of tuples to messages and vice versa.

The diagram on Figure 2.5 illustrates the way this integration is done in SPEEDD. Raw events posted on the *speedd-in-events* topic in comma separated values (CSV) format are de-serialized using the use-case specific scheme (in the diagram *AggregatedReadingScheme* corresponds to the traffic sensor aggregated reading event format). The resulting stream contains tuples of form {eventName, timestamp, attributes}. Outbound events are serialized as JavaScript Object Notation (JSON) text-based messages using *JsonEncoder* class configured via serializer.class parameter of the KafkaBolt.

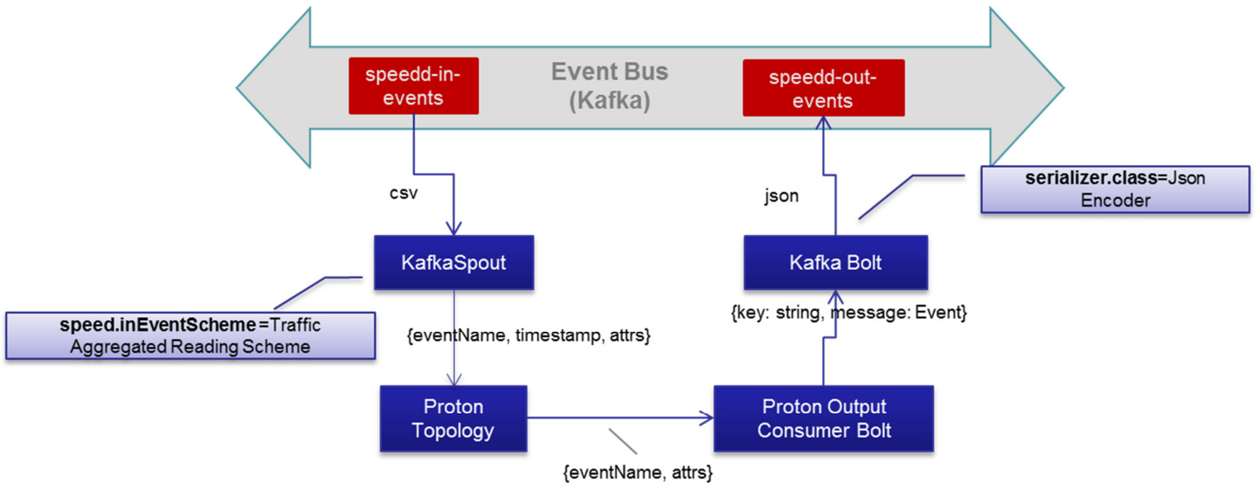

speedd-in-events
Event Bus (Kafka)
speedd-out-events
csv
json
serializer.class=Json Encoder
KafkaSpout
Kafka Bolt
speed.inEventScheme=Traffic Aggregated Reading Scheme
{eventName, timestamp, attrs}
{key: string, message: Event}
Proton Topology
Proton Output Consumer Bolt
{eventName, attrs}

Figure 2.5 - Storm-Kafka Integration

2.4.2 Event/Data Providers

Event providers provide the input interface of SPEEDD runtime with the external world. Every event that occurs in the external world that should be taken into account by SPEEDD to detect or predict an important business situation should be sent to the speedd-in-events topic on the event bus (see 2.4.1 above) as a message representing the event.

2.4.2.1 Event Providers for Traffic Use Case

As it is illustrated in Figure 2.3, events for the traffic use case come from the following sources:

- Traffic sensors – magnetic wireless Sensys sensors buried in the road
- Micro-Simulator – synthetic data generated by the micro-simulator
- Historic data – data from the sensors collected over some period of time that should be replayed to test or demonstrate the SPEEDD prototype

[5] http://storm.apache.org/documentation/storm-kafka.html

To enable processing of events generated by either of the above sources, a connector should be developed. The connector uses source-specific integration mechanism to read the data from the event sources and send them to SPEEDD event bus using Kafka producer API. The message data model and the format of the serialized representation are described in API and Integration part of this document (see 2.6). We define three connector types corresponding to the types of the event sources:

- Sensor connector[6]
- Micro-simulator connector
- File reader connector – replay past events from a file

Architecture and various aspects of AIMSUN micro-simulator connector are discussed in appendix 11.

2.4.2.2 Event Providers for Credit Card Fraud Use Case

The requirements for SPEEDD prototype in regard to the Credit Card Fraud use case only assume running SPEEDD in 'offline' mode by replaying historic events. Thus two types of connectors are considered in this design document (as shown in Figure 2.4):

- Database connector – replays events from Feedzai transaction database
- File reader connector – replays events from a file (with partially or fully anonymized data)

These connectors reuse the same design framework as described above. For instance, only a small portion of a connector code is use-case specific, where most of the functionality is reused between connectors. In case of the file reader connector the same connector can be used for either use case, while the parsing part is use-case specific.

The data model and the format of the messages are described in API and Integration part of this document (see 2.6).

2.4.3 Action Consumption – Actuators/Connectors

The outcomes of SPEEDD are actions that should be applied in the operational environment to resolve a problem or prevent a potential problem. According to the event-driven architecture principles, actions are represented as outbound events and are available to every interested party to receive and process them. The actuators connectors are interface points in SPEEDD architecture responsible for listening on the speedd-actions-confirmed topic for new actions and connect to operational systems to execute respective operations. The following provides details of the actuators for each use case.

2.4.3.1 Actions for the Traffic Use Case

As mentioned above, it is not planned to connect SPEEDD prototype to the traffic operational systems running in production mode. Instead, the detect→decide→act loop is implemented and tested using the AIMSUN micro-simulator developed as part of WP8. The traffic actuator connector listens on the outbound action events (speedd-actions topic on the event bus) and executes operations supported by the micro-simulator, e.g. update speed limits, set ramp metering rates, etc. (see appendix 11 for more

[6] Sensor connector is out of scope for SPEEDD prototype because connecting to the operational systems in production environment is not planned as a goal for the prototype

information on AIMSUN integration). The integration with the event bus for actuators is based on the Kafka consumer API.

2.4.3.2 *Actions for the Credit Card Use Case*

Per definition of the scope for the SPEEDD prototype, outbound events representing final decisions related to a suspected fraud situation represent the actions – the action information will be written to a log or recorded in a decision data store for further analysis and verification of the prototype functional correctness. No actual operation will be performed. The integration mechanism is the same as for the traffic use case – Kafka consumer API.

2.4.4 Complex Event Processor

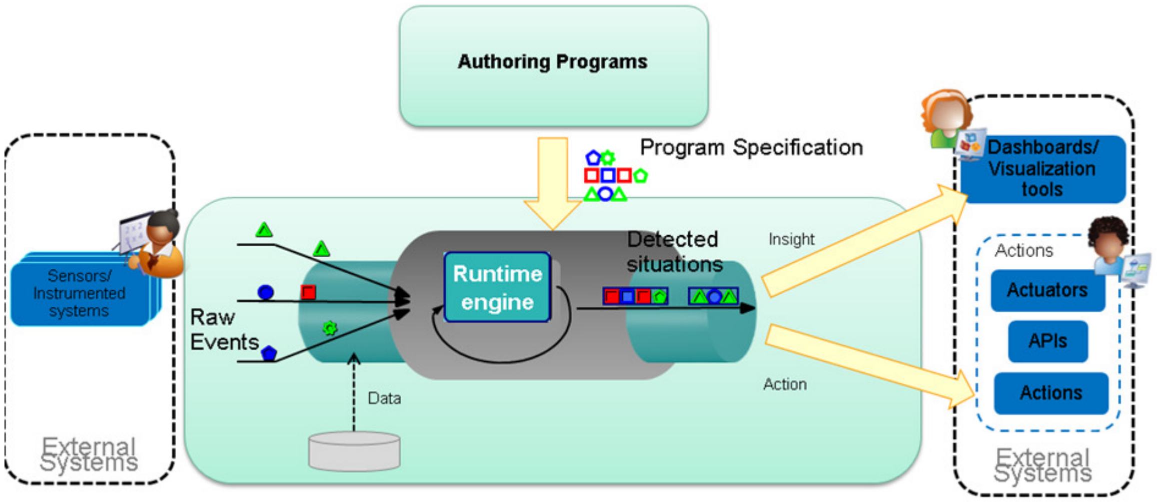


Figure 2.6 - Proton Authoring Tool and Runtime Engine

Proton—**IBM Proactive Technology On-Line**—is an open source IBM research asset for complex event processing extended to support the development, deployment, and maintenance of proactive event-driven applications. **Proactive event-driven computing** (Engel and Etzion 2011) is the ability to mitigate or eliminate undesired states, or capitalize on predicted opportunities—in advance. This is accomplished through the online forecasting of future events, the analysis of events coming from many sources, and the enabling of online decision-making processes.

Proton receives **raw** events, and by applying **patterns** defined within a **context** on those events, computes and emits **derived** events (seeFigure 2.6).

2.4.4.1 *Functional Highlights*

Proton's generic application development tool includes the following features:

- Enables fast development of proactive applications.
- Entails a simple, unified high-level programming model and tools for creating a proactive application.
- Resolves a major problem—the gap that exists between events reported by various channels and the reactive situations that are the cases to which the system should react. These situations are a

composition of events or other situations (e.g., "when at least four events of the same type occur"), or content filtering on events (e.g., "only events that relate to IBM stocks"), or both ("when at least four purchases of more than 50,000 shares were performed on IBM stocks in a single week").

- Enables an application to detect and react to customized situations without having to be aware of the occurrence of the basic events.
- Supports various types of contexts (and combinations of them): fixed-time context, event-based context, location-based context, and even detected situation-based context. In addition, more than one context may be available and relevant for a specific event-processing agent evaluation at the same time.
- Offers easy development using web-based user interface, point-and-click editors, list selections, etc. Rules can be written by non-programmer users.
- Receives events from various external sources entailing different types of incoming and reported (outgoing) events and actions.
- Offers a comprehensive event-processing operator set, including joining operators, absence operators, and aggregation operators.

2.4.4.2 Technical Highlights

- The Proton technology is platform-independent, uses Java throughout the system.
- Comes as a J2EE (Java to Enterprise Edition) application or as a J2SE (Java to Standard Edition) application.
- Based on a modular architecture.

2.4.4.3 Proton APIs

One of the main characteristics of CEP engines is the asynchronous way in which events are received and emitted to and out of the system. This is usually done through a publish/subscribe mechanism, with no Application Programming Interface (API) definition per-se.[7]

PROTON's JSON file that is created at build-time contains all EPN definitions, including definitions for event types, EPAs, contexts, producers, and consumers. The physical entities representing the logical entities of producers and consumers in PROTON are adapter instances. For each producer an input adapter is defined, which defines how to pull the data from the source resource and how to format the data into PROTON's object format before delivering it to the run-time engine. The adapter is environment-agnostic, but uses the environment-specific connector object, injected into the adapter during its creation, to connect to PROTON runtime.

At run-time, the standalone CEP engine receives incoming events through the input adapters, processes these incoming events according to the definitions, and sends derived events through the output adapters. At execution, the standalone run-time engine accesses the metadata file, loads and parses all the definitions, creates a thread per each input and output adapter, and starts listening for events incoming from the input adapters (producers) and forwards events to output adapters (consumers).

[7] This summary is taken from D6.1 - Second version of event recognition and forecasting technology

Note that for the distributed implementation on top of STORM, an input Bolt serves the same function as input adapter, and the derived events are passed as STORM tuples to the next stage in the SPEEDD topology processing (see Figure 2.3, Figure 2.4).

2.4.4.4 Proton High-level Architecture

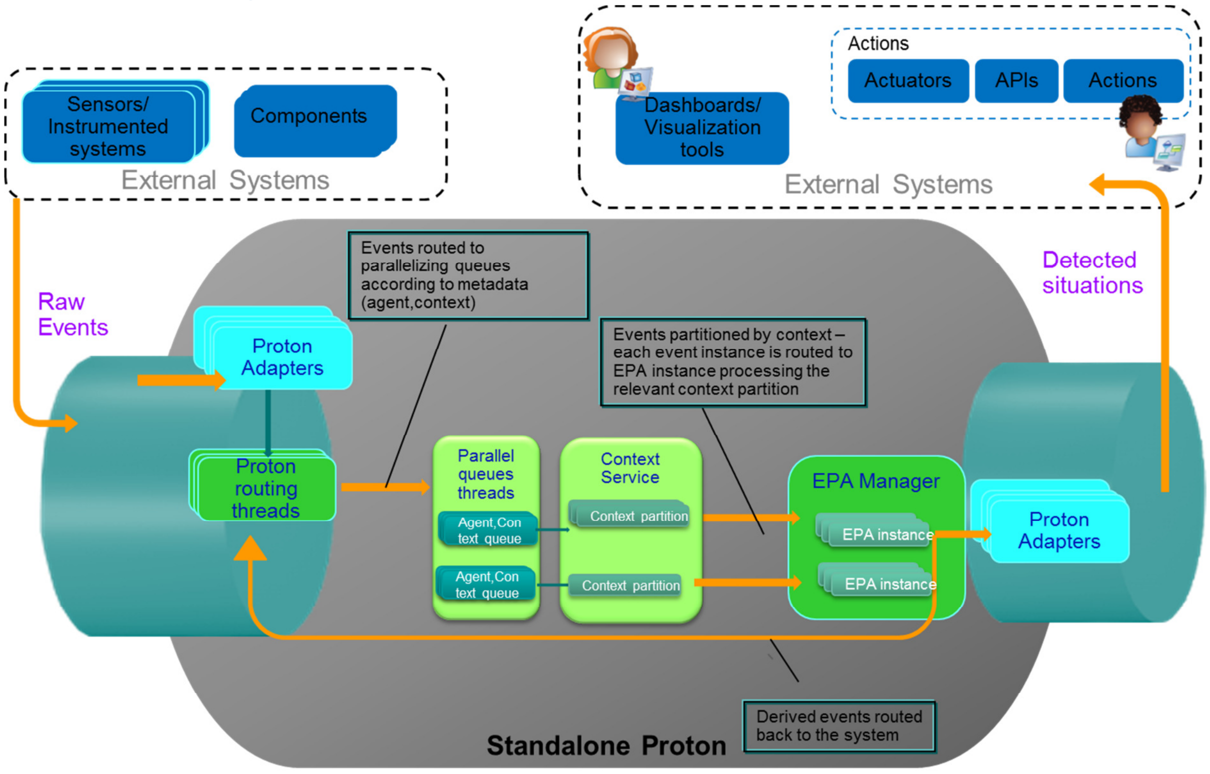


Figure 2.7 - Proton Runtime and external systems

Proton architecture consists of a number of functional components and interaction among them, the main of which are (see Figure 2.7):

- Adapters – communication of Proton with external systems
- Parallelizing agent-context queues – for parallelization of processing of single event instance, participating in multiple patterns/contexts, and parallelization of processing among multiple event instances
- Context service – for managing of context's lifecycle –initiation of new context partitions, termination of partitions based on events/timers, segmenting incoming events into context groups which should be processed together.
- EPA manager –for managing Event Processing Agent (EPA) instances per context partition, managing its state, pattern matching and event derivation based on that state.

In the context of the SPEEDD runtime, the traffic sensors, or the transaction terminals reporting input events correspond to the external systems producing raw events in the diagram. On the other hand, the dashboard, and the actuators (e.g. AIMSUN actuator component, or an actual traffic light controller) correspond to the external systems that are the consumers of the derived events (or, detected situations).

When receiving a raw event, the following actions are performed:

1. Look up within the **metadata**, to see which context effect this event might have (context initiator, context terminator) and which pattern this event might be a participant of
2. If the event can be processed in parallel within multiple contexts/patterns (based on the EPN definitions), the event is passed to **parallelization queues**. The purpose of the queues:
    a. Parallelize processing of the same event by multiple unrelated patterns/contexts at the same time keeping the order for events of the same context/pattern where order is important
    b. Solve out-of-order problems – can buffer for a specified amount of time
3. The event is passed to **context service**, where it is determined:
    a. If the context is an initiator or a terminator, new contexts might be initiated and or terminated, according to relevant policies.
    b. Which context partition/partitions this event should be grouped under
4. The event is passed to **EPA manager:**
    a. Where it is passed to the specific EPA instance for the relevant context partition,
    b. Added to state of the instance
    c. And invokes pattern processing
    d. If relevant, a derived event is created and emitted

2.4.4.5 Proton component architecture

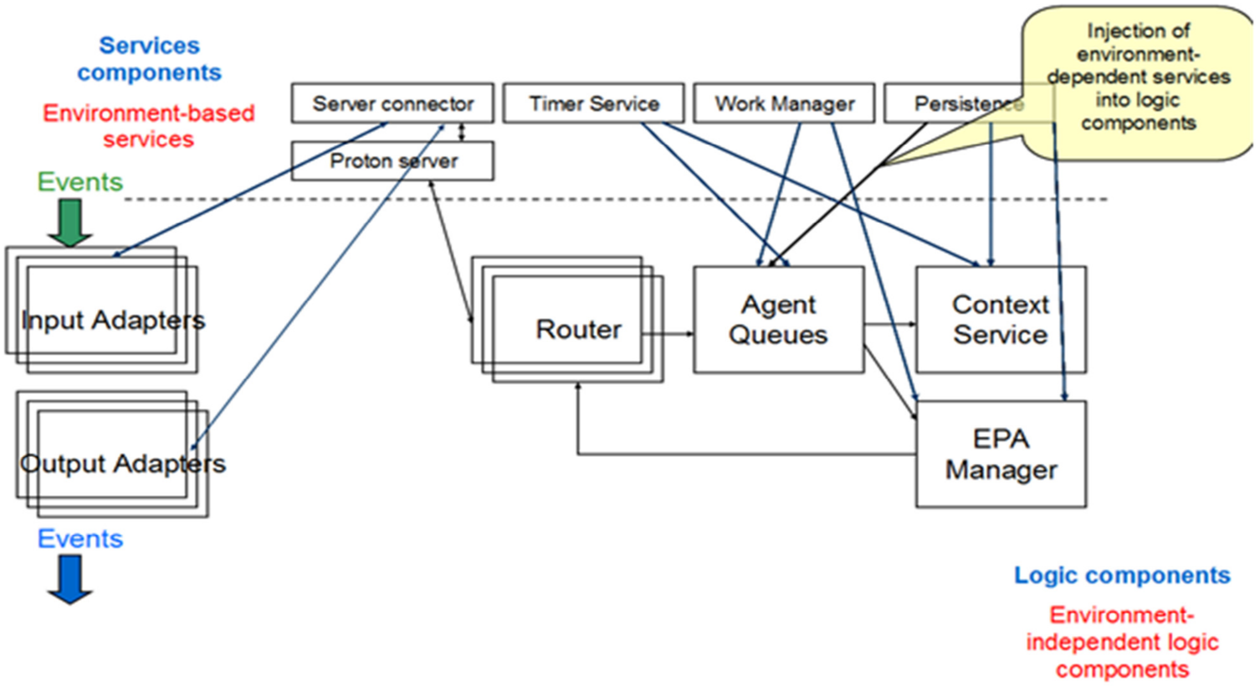


Figure 2.8 - Proton components

Proton's logical components are illustrated in Figure 2.8. The queues, the context service, the EPA manager are purely java-based. They utilize dependency injection to make use of the infrastructure services they require, e.g. work manager, timer services, communication services. These services are implemented differently for the J2SE and J2EE versions.

2.4.4.6 Distributed Architecture on top of STORM

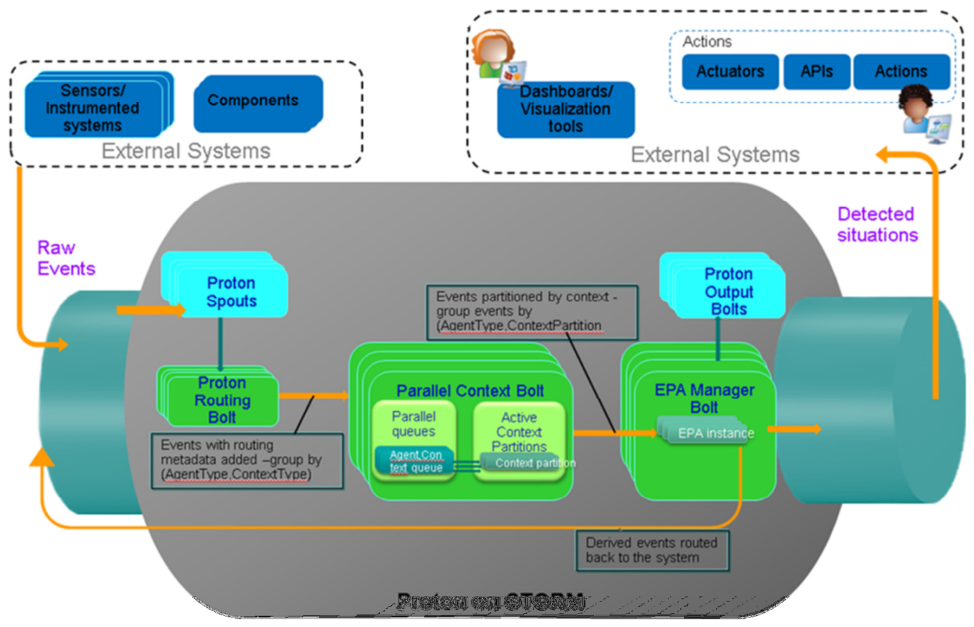

Sensors/ Instrumented systems
Components
External Systems
Dashboards/ Visualization tools
Actions
Actuators
APIs
Actions
External Systems
Raw Events
Detected situations
Proton Spouts
Proton Routing Bolt
Events with routing metadata added –group by (AgentType,ContextType)
Events partitioned by context - group events by (AgentType,ContextPartition
Parallel Context Bolt
Parallel queues
Agent,Context queue
Active Context Partitions
Context partition
Proton Output Bolts
EPA Manager Bolt
EPA instance
Derived events routed back to the system
Proton on STORM

Figure 2.9 - Architecture of Proton on Storm

The Proton architecture on top of Storm[8] (see Figure 2.9) preserves the same logical components as are present in the standalone architecture: the queues, the context service and the EPA manager, which constitutes the heart of the event processing system. However the orchestration of the flow between the components is a bit different, and utilizes existing Storm primitives for streaming the events to/from external systems, and for segmenting the event stream.

After the routing metadata of an incoming event is determined by the routing bolt (which has multiple independent parallel instances running), the metadata – the agent name and the context name – is added to the event tuple.

We use the Storm field grouping option on the metadata routing fields – the agent name and the context name – to route the information to the next Proton bolt – the context processing. Therefore all events which should be processed together – relating to the same context and agent – will be sent to the same instance of the bolt.

After queueing the event instance in the relevant queues (in order to solve out of order, if needed and parallelize event processing of the same instance where possible by different EPAs in the same EPN) and after processing by context service, the relevant context partition id is added to the tuple.

[8] Work on implementing Proton on Storm is being performed as part of the FERARI project (http://www.ferari-project.eu), and will become available as open source (https://bitbucket.org/sbothe-iais/ferari) in the next months. The architecture is described in the current document for completeness and clarity.

Here again we use the field grouping on context partition and agent name fields to route the event to specific instances of the relevant EPA, this way performing data segmentation – the event will be routed to the agent instance which manages the state for a specific agent on a specific partition.

If the pattern matching is done and we have a derived event, it will be routed back into the system, and passed through the same channels as the raw event.

2.4.5 Decision Making

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.

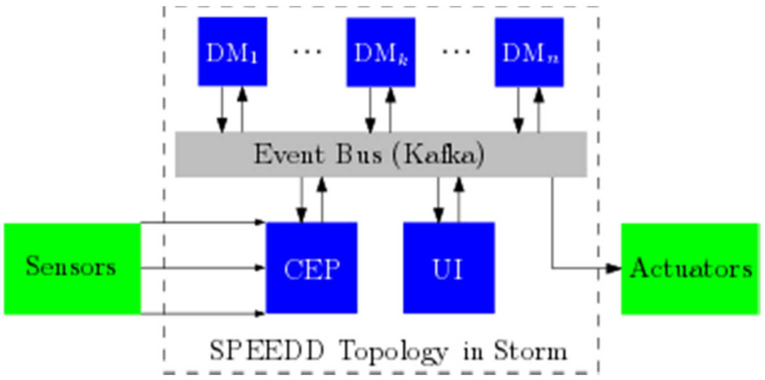


Figure 2.10 - Implementation of DM in SPEEDD topology

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 2.10. 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 DM algorithm may also require an external planning and optimization module for more complex DM tasks. This module is foreseen to be used sparingly. Currently, the mixed integer linear program solver LP Solve has been proposed for use in this module, and some initial progress has been made on integration with the SPEEDD runtime.

The architecture depicted in Figure 2.10 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. It also permits DM to tackle different use cases using the same architecture. We provide a detailed description of this architecture, using one of the use cases, namely traffic. At every stage of conception and design, considerable effort has been taken to provide a modular implementation. The main components in this architecture are the following:

- DM agents k = 1,…,n
- Input Events to the kth DM agent
- Output Events from the kth DM agent

We describe each of these components in the following subsections.

2.4.5.1 DM agent

The distributed nature of DM in traffic is illustrated in Figure 2.11. The DM agent or local controller controls a part of the complete traffic network, in the sense that it controls more than just one intersection or actuator in the system.

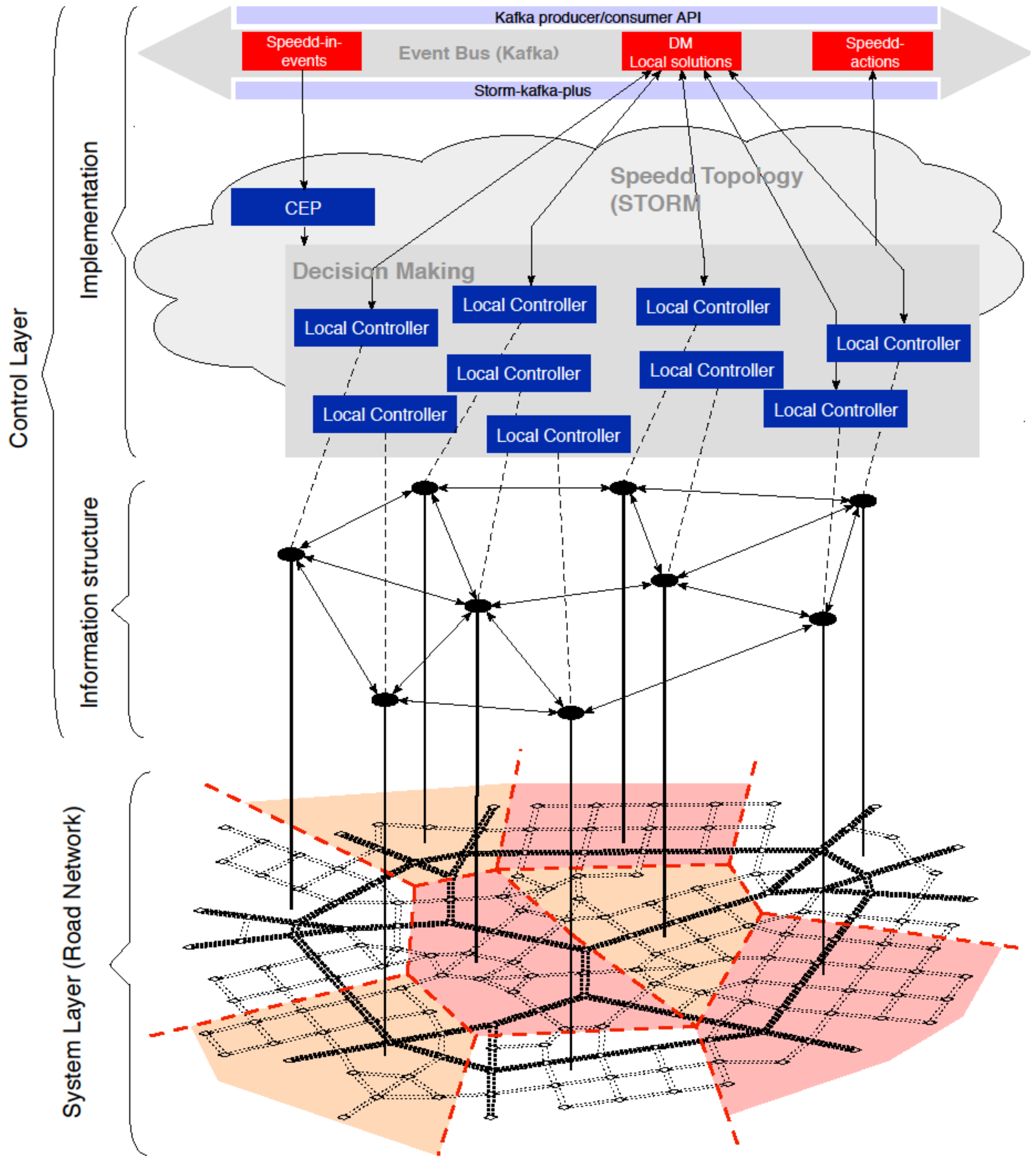


Figure 2.11. An overview of the distributed control approach applied to DM for the traffic scenario.

The complete traffic network is partitioned into smaller networks, as depicted in Figure 2.12 for the Rocade in Grenoble.

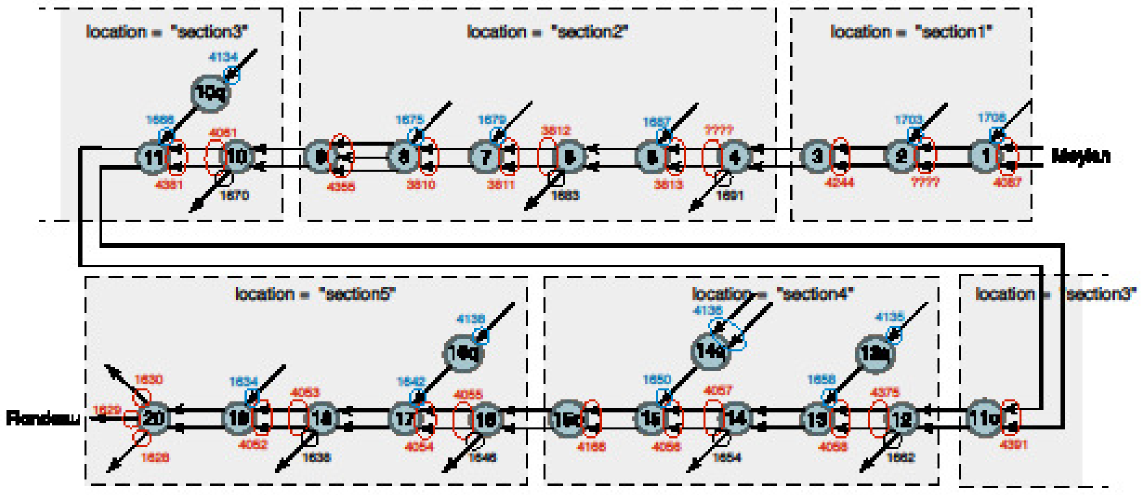


Figure 2.12 - Partitioning of the Rocade in Grenoble for the purpose of distributed DM

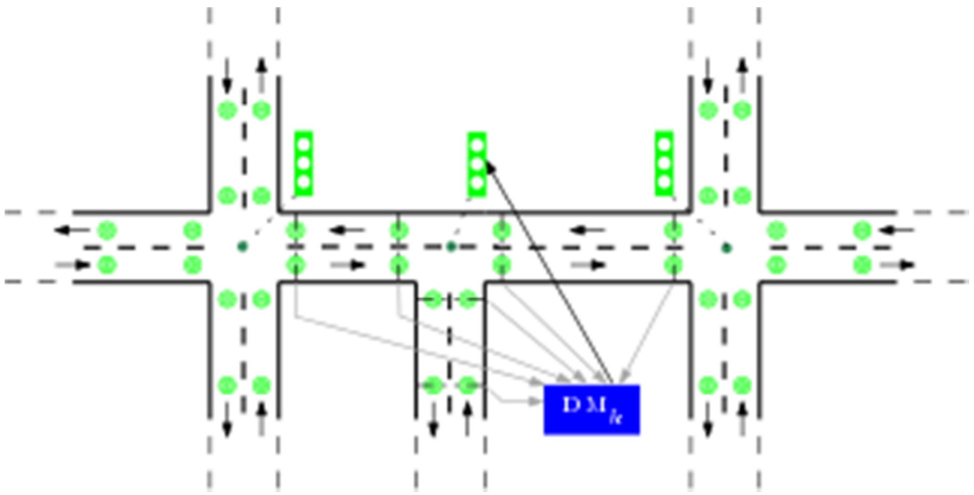


Figure 2.13 - Urban Traffic Network

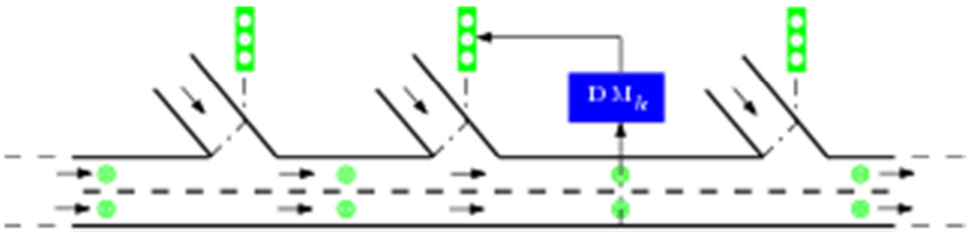
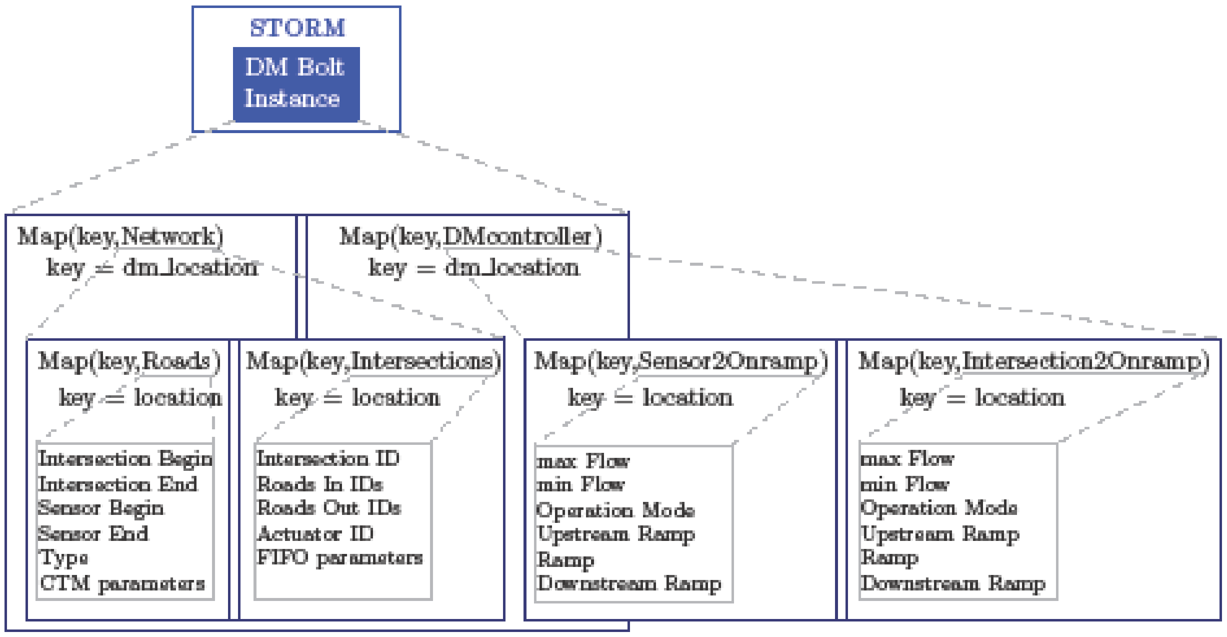


Figure 2.14 - Freeway Traffic Network

We begin by identifying a DM agent in the traffic scenario. Consider the freeway and urban traffic network illustrated in Figure 2.13 and Figure 2.14, respectively. In both these figures, the networks comprise of roads and intersections. Sensors placed along the roads provide measurements that drive the DM algorithm, in the form of processed events from the CEP. The DM algorithm generates control signals that are implemented through various actuators. One form of actuators, namely ramp meters, can be found at controlled or metered intersections, and they control the flow of traffic through the intersection. Other forms of actuators may also be present, such as variable speed limit indicators on certain roads, which control the flow of traffic out of these roads. These actuators are spatially distributed, and the information needed to drive them comes from sensors that are also spatially distributed. In this scenario, a DM agent implements the algorithm to drive a set of actuators.

Thus, in the traffic scenario, the set of DM agents k = 1,…,n can be thought to represent a partition of the complete traffic network into smaller networks. Naturally, spatial or geographical concerns determine an appropriate partition. However, the relevance and ease of coordination between the networks also plays a role in determining the partition. Furthermore, the networks provide a modular way to represent the complete traffic network within the SPEEDD topology. Modifying and accessing a modular representation may prove to be more practical, particularly in the case of an extensive traffic network, such as an urban traffic network.

It should be clear by now that to define the DM agents k = 1,…,n, we need to define the k = 1,…,n networks that constitute the complete traffic network. We use the following definition to structure our implementation in Storm as well, as shown below.



Each instance of the DM bolt in Storm is associated with a number of networks and their identifiers, given by the attribute dm_location. The network refers to the part of the complete traffic network that is associated with a DM agent. It is defined as follows.

- Network: A network comprises of a set of roads and intersections, which are defined below.
- Road: A road can be defined as a directed edge that begins at an intersection and ends at another. Thus, for the purpose of DM, we treat a two-way road as two separate roads. Typically, each road has two sets of sensors, one at the beginning and one at the end. In cases where only a single set of sensors are present, the measurements from the missing sensor are estimated. The density of cars on a road can be updated by keeping track of the number of cars that enter and exit the road. For appropriate DM, we classify roads into the following road types: 'freeway', 'metered onramp', 'unmetered onramp', 'offramp', 'main city' and 'minor city'. As the need arises, new road types and corresponding DM routines can be added. For future versions, actuators may also be associated with each road. The dynamic model for the road is given by the CTM model, which is used to simulate and predict flows, including the demand and supply, for the road.
- CTM model: For each road, the following parameters are used to define the CTM model: free-flow speed, congested wave speed, critical density, jam density, length of the road and capacity of the road. Using the available measurements, one can fit parameters to this model.
- Intersection: An intersection can be defined as a network node that merges traffic flow from a set of incoming roads and diverges the flow into a set of outgoing roads. If the intersection is metered, it will also contain an actuator. For ease of DM, the intersection and corresponding actuator are assigned IDs, which are associated with the IDs of its

incoming and outgoing roads. If the intersection is unmetered, its corresponding actuator ID is set to -1. A dynamical model must be assigned for each intersection as per the traffic rules of the network. A typical dynamical model, which we use currently, is the FIFO model. This model is used to simulate and predict flows, including demands and supplies, through the intersection.

- FIFO model: For each intersection, the following parameters are used to define the FIFO model: priorities, traffic light phases and the turning preference matrix. The traffic light phases are identified by a phase number or index, which is linked to the indices of the incoming and outgoing roads that are active during the phase. Note that multiple incoming roads and/or outgoing roads may be active during a single phase. The turning preference matrix contains the turning ratios corresponding to each pair of incoming and outgoing roads. If these values are not provided, measurements from the network can be used to fit parameters to this model. The priorities arise from traffic rules that give preference to certain traffic flows over others. Traffic flows through unmetered intersections are solely determined using priorities. However, priorities also play a role in determining flows through metered intersections. This is because, during traffic light phases that permit multiple flows simultaneously, certain traffic flows are prioritized.

The network class in the implementation provides functions that are called to process measurement events and provide data-filtering or observer functionalities.

Each instance of a DM bolt also contains a definition of the DM algorithm, called DMcontroller in the previous illustration, which is uniquely associated with a network through the dm_location attribute. This class maintains all the structures necessary for the control algorithm. For example, in the case of DM for freeways, it maintains the data structures Sensor2Onramp and Intersection2Onramp for every active location. It also provides functions that are called to process all incoming events, barring the recording of measurements.

2.4.5.2 Input Events

Input events are assigned to an instance of the DM bolt based on the attribute dm_location, as illustrated in Figure 2.15. This is to ensure that the variables associated with a network and its DM agent can be saved in the bolt instance and retrieved during each incoming event.

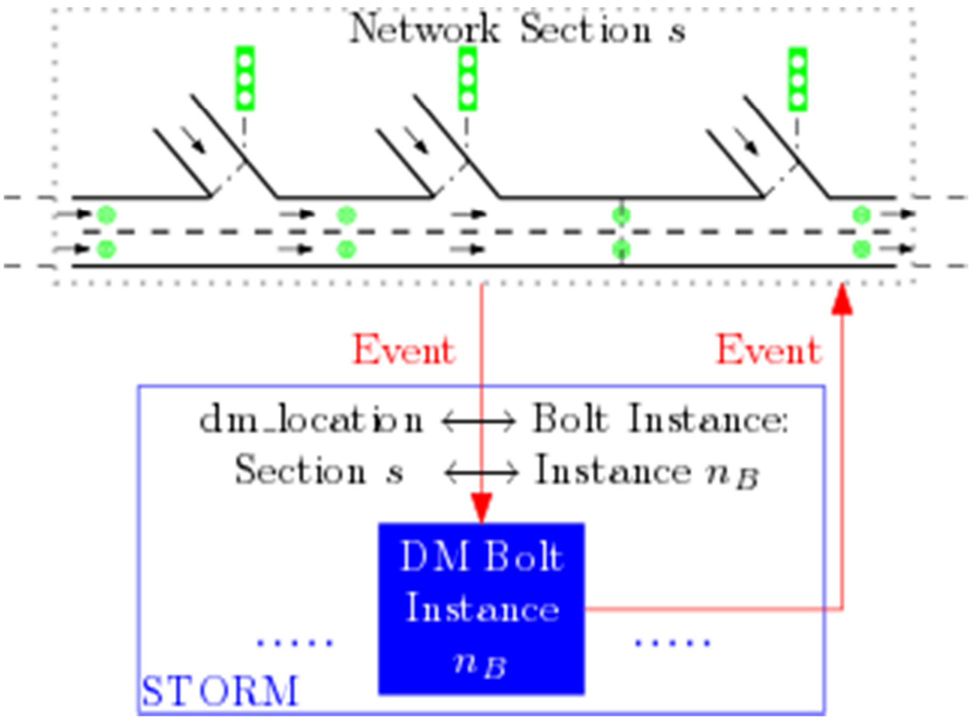


Figure 2.15 - Assignment of Events to Instances of the Bolt

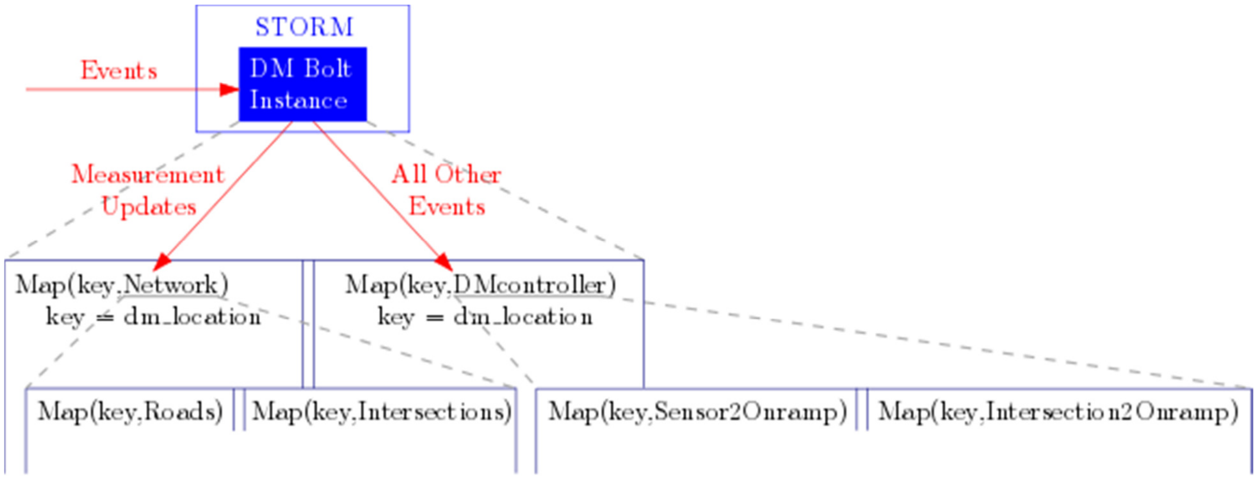


Figure 2.16 - Execution of Events within the Bolt

DM receives events from Complex Event Processing (CEP) and the User Interface (UI). Events from CEP include complex events such as advance predictions about freeway congestion. In addition to the high-level, derived events, DM also needs access to (time-averages of) low level sensor data, to obtain a complete description of the state of the traffic network. For simplicity, we assume that all sensor data are passed through CEP, which allows us to aggregate measurement data already at 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 input events can be categorized as measurement events and other events, and appropriate functions are called from the network and DMcontroller classes to process these events, respectively. This is illustrated in Figure 2.16.

Table 2.2 - Input Events (from CEP) for DM

Name	Attributes	Description	Usage
Predicted Congestion	{ string dm_location, integer location, double average_density, long timestamp }		(Freeway) Activates ramp metering algorithm
Congestion	{ string dm_location, integer location, double average_density, long timestamp }		(Freeway) Estimated density used by DM. (non-monotonic model: capacity drop is taken into account)
ClearCongestion	{ string dm_location, integer location, long timestamp }		(Freeway) Deactivates ramp metering algorithm
OnRampFlow	{string dm_location, integer location, double average_flow, double average_speed, double average_density, long timestamp }	Running average of onramp flow	(Freeway) Numerical state estimate for DM
2minsAverage Density AndSpeedFor Location	{string dm_location, integer location, double average_flow, double average_speed, double average_density, long timestamp }	Running average of mainline states	(Freeway, Inner City) Numerical state estimate for DM
PredictedRamp Overflow	{ string dm_location, integer location, double queueOccupancy, long timestamp }	Queue at metered onramp is almost full.	(Freeway) Initialize coordinated metering policy with upstream ramp.
ClearRampOver flow	{ string dm_location, integer location, (double queueOccupancy), long timestamp }	Queue at metered onramp has considerably reduced.	(Freeway) Deactivate coordinated metering policy with upstream ramp.

In Table 2.2, we provide an overview of the events from CEP that are currently implemented. We include a brief description of the attributes associated with the event and its use in DM. In contrast, events from the UI are mostly simple commands, by which a human operator can modify bounds and parameters in the models. An example of one such event is given in Table~\ref{tab:admin_events}, along with its attributes and use.

Table 2.3 - Input Events (from UI) for DM

Name	Attributes	Description	Usage

SetRampMetering Bounds	{ string dm_location, integer location, double lowerLimit, double upperLimit, long timestamp }	Upper- and lower bound for ramp metering algorithm.	(Freeway) Bounds will be strictly enforced by the ramp metering algorithm. Can be used to override DM entirely, if lower bound equals upper bound.

2.4.5.3 *Output Events*

The large majority of events created by DM are actuation commands for the actuators in a traffic network, that is traffic lights at intersections or at onramps for ramp metering purposes and variable speed limits. Currently, we expect that the majority of the actuation commands will also be of interest for UI, since the UI should be able to visualize the current operational state of the traffic network to the operators. The list of actuation commands is given in Table 2.4. Note that the format of the actuation commands is tailored to the micro-simulator AIMSUN, which serves as a validation platform in this use case. In addition to the actuation command, coordination commands are also outputed from DM, which serve as input events to other DM agents. Examples of such commands include predictions on ramp overflow and its clearing up.

Table 2.4 - Output Events from DM

Name	Attributes	Description	Usage
SetMeteringRate	{ integer actuatorID, double dutyCycle, long timestamp }	Change ramp metering duty cycle.	(Freeway) Immediately translated to red-green signal by AIMSUN
QueueLengthEstimate	{ string dm_location, integer location, double queueOccupancy, long timestamp }	Queue length updates at metered onramps.	(Freeway) Initialize coordinated metering policy with upstream ramp.

2.4.6 Dashboard application

Operators will interact with the outputs of the SPEEDD algorithms through a User Interface. The Dashboard Client communicates, via the Dashboard Server with the composite systems in the SPEEDD architecture. Operators can accept, respond to, or make suggestions and control actions, via the User Interface and these changes are fed back into the SPEEDD architecture, thus allowing for the seamless integration of expert knowledge and the outputs of complex algorithms. A diagram of the dashboard architecture can be seen in Figure 2.17.

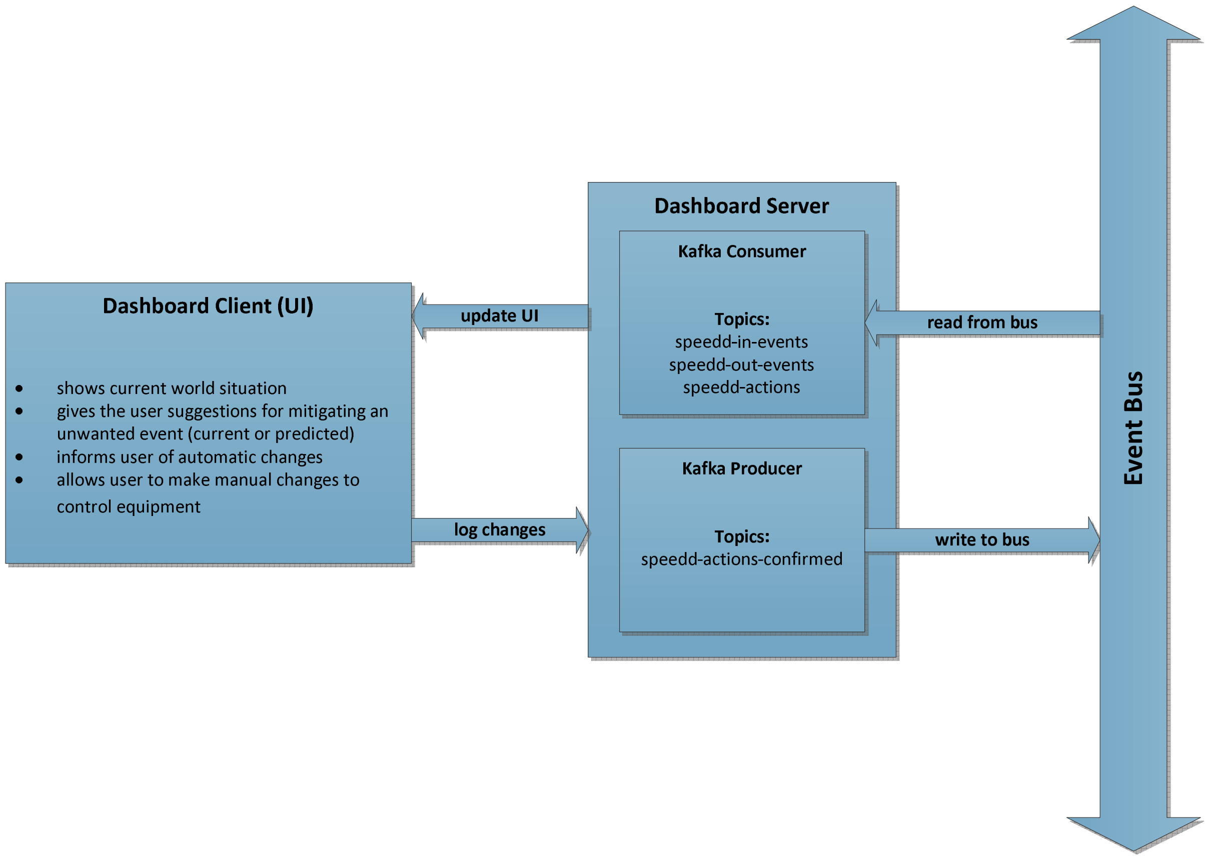


Figure 2.17 - Dashboard Architecture

The Dashboard Server is built using the Express[9] web application framework for Node.js[10] (Tilkov and Vinoski 2010). The server implements a Kafka consumer and producer (apart from hosting the files that generate the UI). The consumer listens for broadcasted messages in the Event Bus under the following topics: *speedd-out-events* and *speedd-actions*. The producer broadcasts messages under the topic *speedd-actions-confirmed*. For more details on the SPEEDD Kafka topics see section 2.4.1. Both the Kafka consumer and producer are implemented using the npm (node package manager) module ‘kafka-node’, a Node.js client with Zookeeper[11] (Hunt, et al. 2010) integration for apache Kafka.

The Dashboard Client is designed to provide the user with a clear picture of the current state of the world. The Dashboard Client achieves the picture of the current state by aggregating sensor readings (traffic management use-case) and transaction or account data (credit card fraud use-case) in human

[9] http://expressjs.com – Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications
[10] http://nodejs.org – Node.js is asynchronous event driven framework for building fast, scalable network applications
[11] https://zookeeper.apache.org – Zookeeper is an open-source distributed service for configuration, synchronization and naming registry for large distributed systems.
[10] https://angularjs.org/ – MVC framework for JavaScript
[11] http://d3js.org – D3.js is a JavaScript library for manipulating DOM based on data
[12] http://socket.io/– Open source implementation of web-sockets for real-time communication

readable form, current states of the control equipment available (e.g. speed limit signs, message signs, lanes, etc.) or accounts, current events identified by the Complex Event Processing (CEP) module and displays of the automated control events produced by the Decision making (DM) unit (e.g. ramp metering rates). Furthermore, the Dashboard Client aims to support the decision-maker by highlighting events which might require attention along with corresponding suggested mitigating strategies. Moreover, based on sensor readings, it helps the traffic managers get a better understanding to what degree the actions taken affect the drivers' behaviour and vice versa.

Figure 2.18 illustrates the current proposed design for a User Interface (UI) for the traffic management use-case along with a description of its components. It has been developed as a result of a thorough analysis of how the traffic managers in Grenoble operate (Deliverable 5.3.1) and study of previous research in human visual perception (see Deliverable 8.3 and 5.2).

Figure 2.19 shows the prototype version 2 of the Credit Card Fraud UI along with a description of its components. Its current form is a result of studies of state-of-the-art commercial fraud analysis systems and an outcome of several discussions with domain experts from Feedzai and FICO (see Deliverable 7.2).

Both UIs have been built in JavaScript with the use of some third-party open-source libraries. The web application framework AngularJS[10] (following the MVC design pattern) has been used in conjunction with D3js[11] (Bostock, Ogievetsky and Heer 2011) to produce the data driven displays. The client-server communication for the UI in the second prototype is realised through an open-source implementation of web-sockets called Socket.io[12].

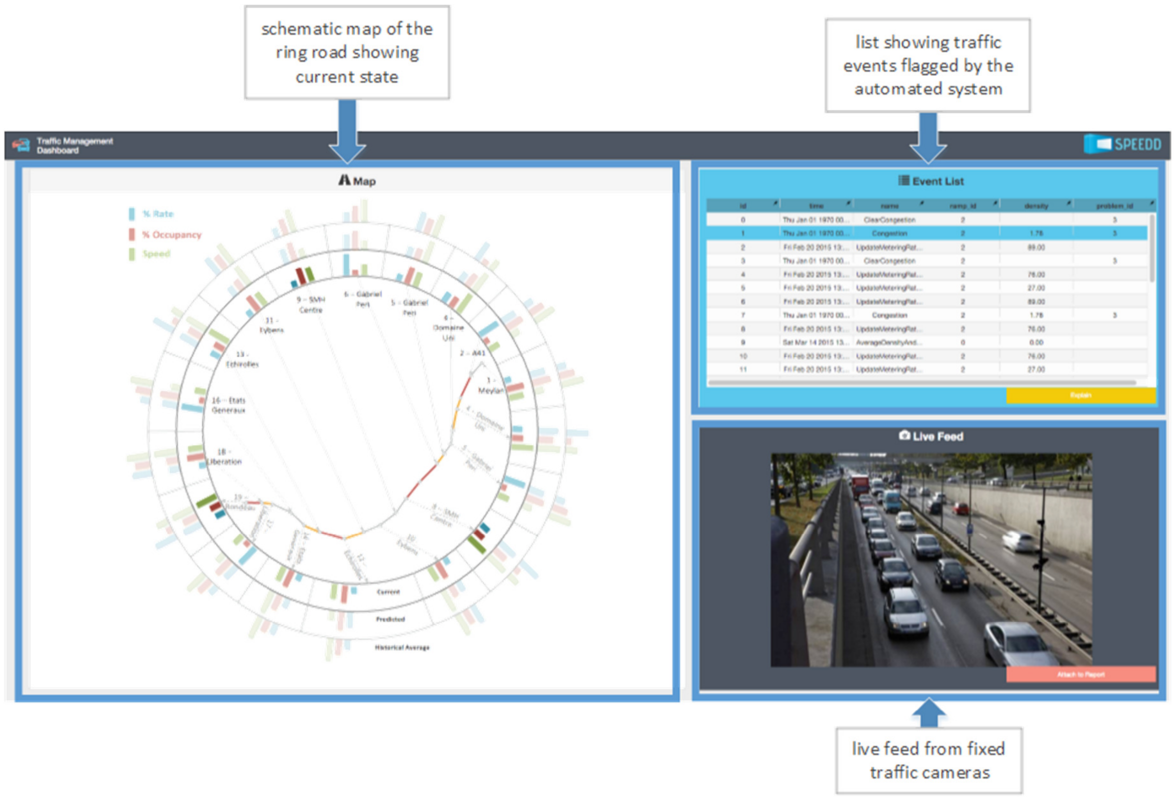

schematic map of the ring road showing current state
list showing traffic events flagged by the automated system
Traffic Management Dashboard
SPEEDD
Map
% Rate
% Occupancy
Speed
Current
Predicted
Historical Average
Event List
Explain
Live Feed
Attach to Report
live feed from fixed traffic cameras

Figure 2.18 – Prototype V2 User Interface for the Traffic Management use-case

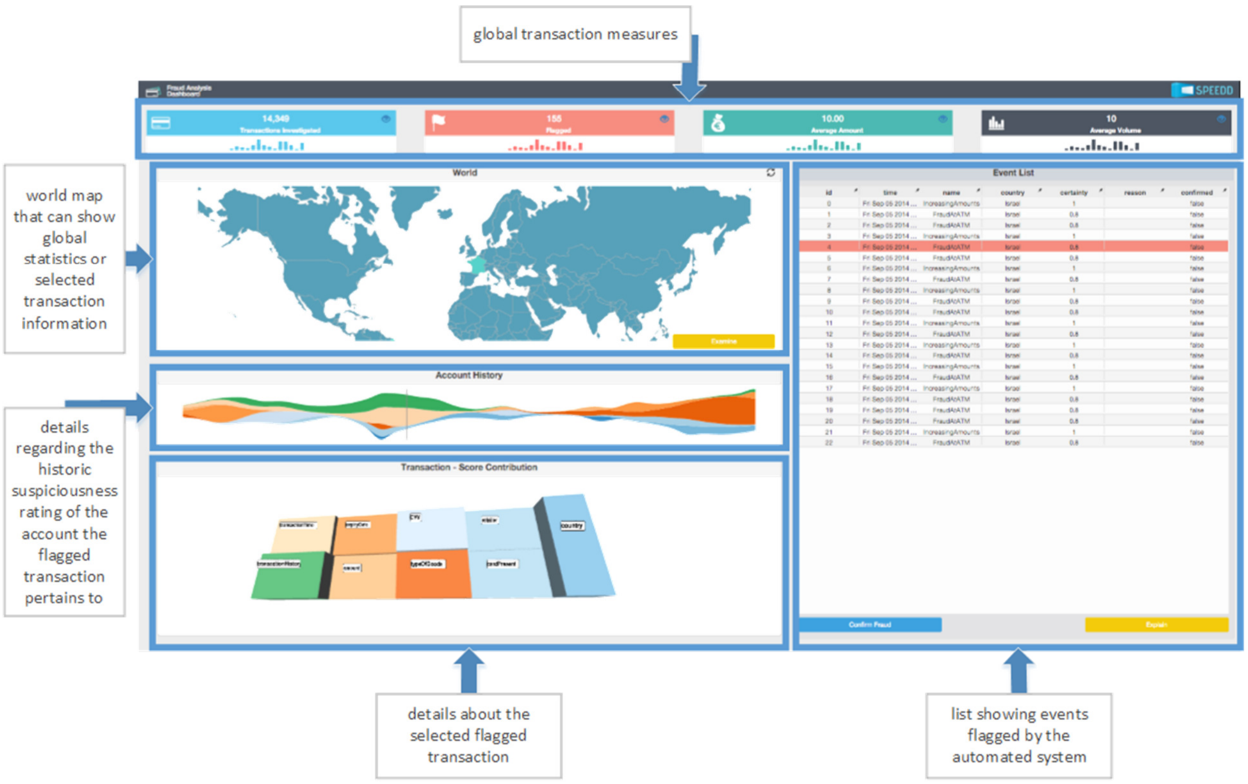

global transaction measures
Fraud Analysis Dashboard
SPEEDD
14,349
Transactions Investigated
155
Flagged
10.00
Average Amount
10
Average Volume
World
Examine
Event List
Account History
Transaction - Score Contribution
Confirm Fraud
Explain
world map that can show global statistics or selected transaction information
details regarding the historic suspiciousness rating of the account the flagged transaction pertains to
details about the selected flagged transaction
list showing events flagged by the automated system

Figure 2.19 – Prototype V2 User Interface for the Credit Card Fraud use-case

2.4.7 SPEEDD Scalability Component Architecture

The scalability component described below is targeted at the messaging scalability challenges that are present in the traffic management use case. Therefore this component is meant to be used in the context of the traffic management use case only.

2.4.7.1 Introduction

The purpose of the scalability component is to allow the system to scale up the number of messages it handles. Under normal circumstances in order to report a slowdown in traffic on the road each sensor must relay every speed measurement to the central processing location. However, the number of messages can be reduced by using the scalability component. The scalability component takes a threshold speed and reports whenever the average speed on the road decreases below it. It does this without requiring continuous reporting by the sensor nodes.

2.4.7.2 Approach

At the startup, a threshold is assigned to each sensor node. If, and only if, the measured speed decreases below the threshold the measurement is relayed to the central component called the coordinator. The coordinator then initiates a violation resolution procedure. It collects measurements from as many sensor nodes as needed in order to achieve an average speed that is higher than the threshold. Once this has been achieved the coordinator then modifies the local threshold given to each sensor node based on its measured speed and former threshold.

If measurements from all nodes have been collected and the speed is still below the threshold, the coordinator reports that a global violation has occurred and the average speed on the road has gone below the threshold.

2.4.7.3 Architecture

The system is comprised of (see Figure 2.20):

1. Sensor nodes which decide whether or not to relay the measurements to the coordinator.
2. A coordinator.
3. A transport mechanism which uses two Kafka topics, one to relay messages from the sensors to the coordinator and the other to relay messages in the other direction.

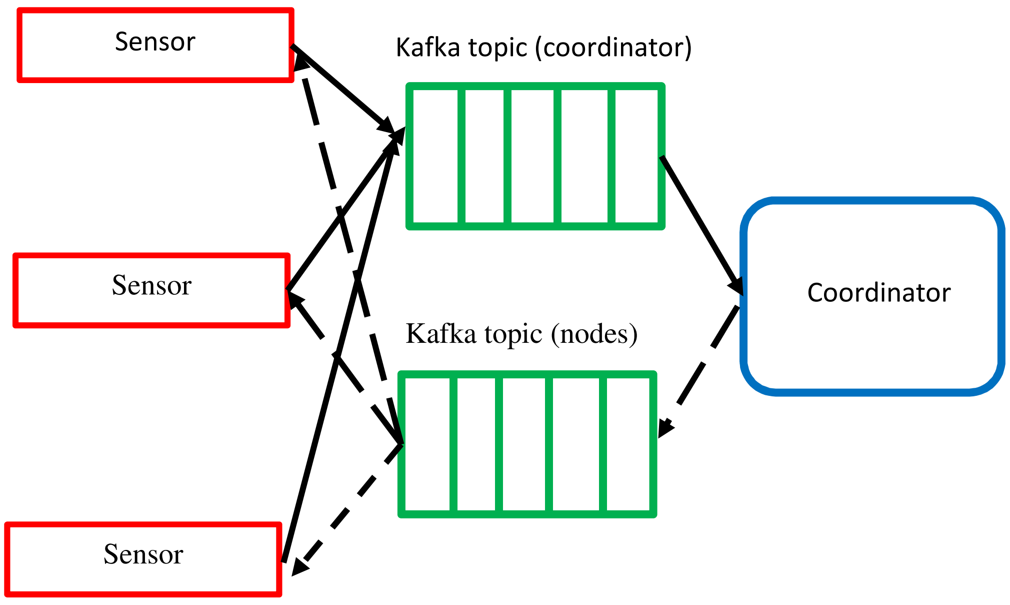


Figure 2.20 - Scalability Layer Architecture

Each sensor node is comprised of three components (see figure 2 below). The sensor receives speed readings from a physical traffic sensor/simulator. These readings are passed on to a buffer called Time Machine. Data is then passed on to the Gatekeeper. This component verifies that the new reading is over the minimal speed threshold. If the speed is below the threshold, the procedure of “violation resolution” is initiated. The coordinator is informed via the communicator that a violation has occurred. The coordinator then queries the other local nodes about their data. Upon receiving it calculates a new threshold for each node and transmits it. This procedure is repeated as necessary whenever a local violation occurs.

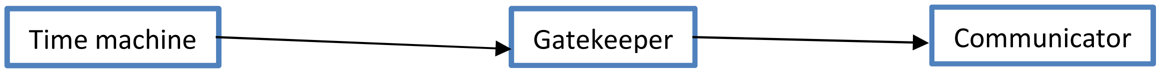


Figure 2.21 - Sensor node composition

The diagram in Figure 2.22 shows the SPEEDD runtime architecture with the communication scalability component.

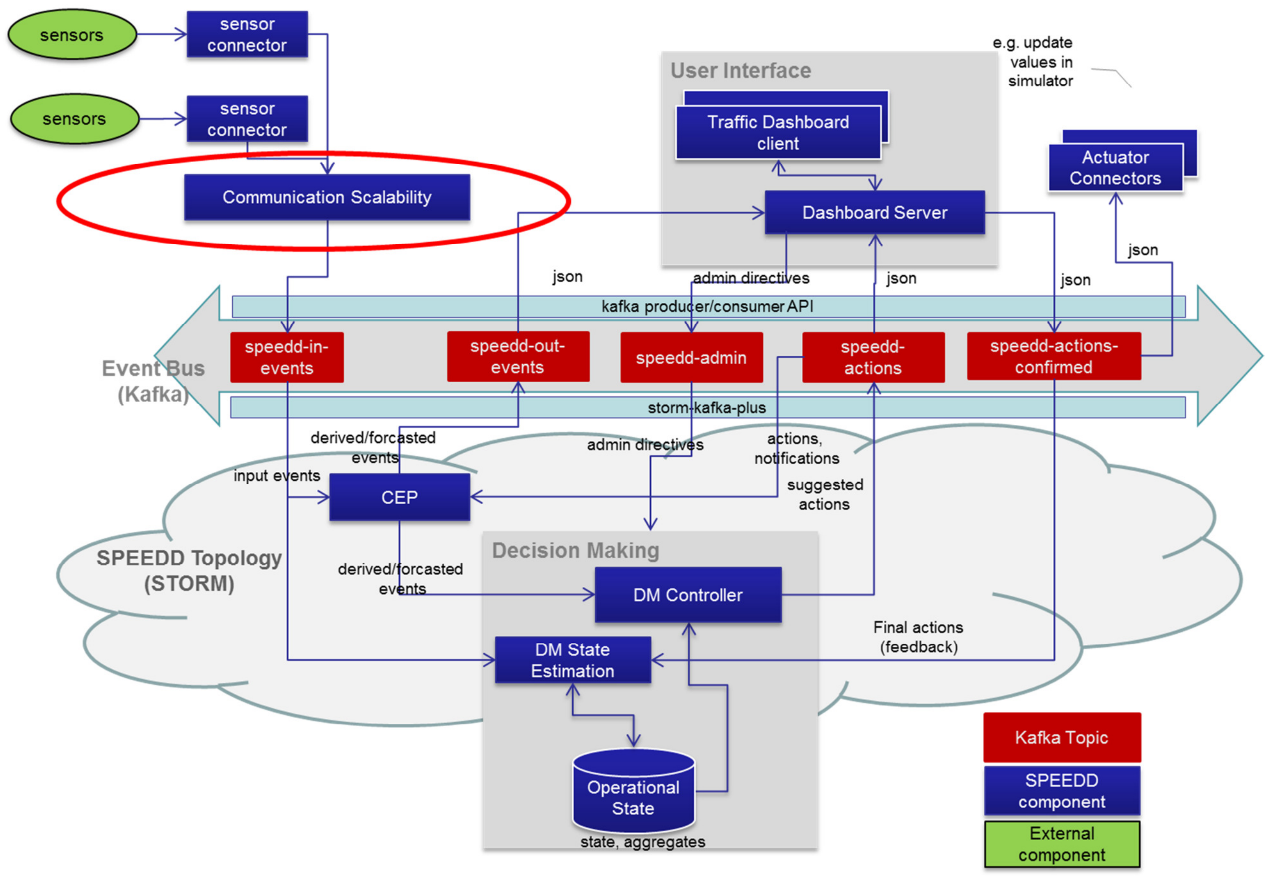


Figure 2.22 - Scalability Component in context of the SPEEDD architecture for Traffic Management

2.5 SPEEDD Design-Time Architecture

The conceptual view of the design time architecture for SPEEDD is presented in Figure 2.23. The goal of the design time is to create and/or update the Event Processing Network definition artifact that will be deployed in runtime and will be used by the Proton proactive event processing component to detect and predict situations. The details of the design time pipeline are described in the subsections below.

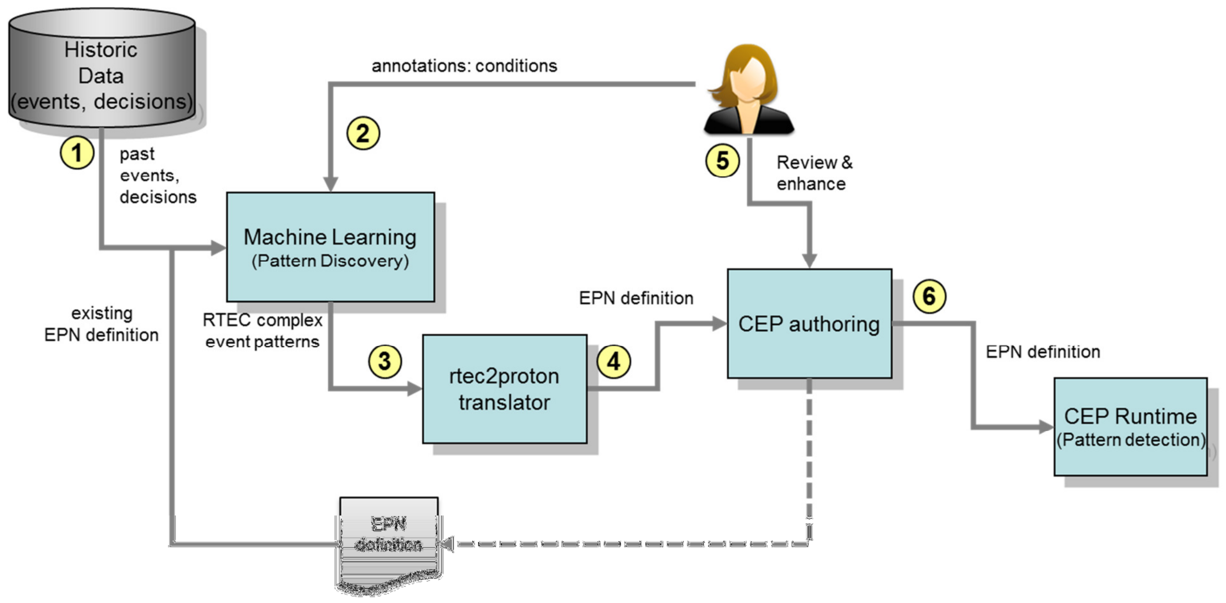


Figure 2.23 - SPEEDD Design Time Architecture

2.5.1 Machine Learning

Figure 2.23 presents the off-line architecture of the SPEEDD system. Past input events, recognized events, forecasted events and decisions are stored as historic data into a database (step 1). Domain experts analyze and annotate the historic data, in order to provide the golden standard for Machine Learning algorithms (step 2). Both historic data and annotation forms the input to the Machine Learning module. Specifically, the input is provided as a file in the form of comma separated values (CSV). The form of the CSV file is similar to the input format of the SPEEDD runtime, with additional columns for representing recognized events, forecasted events, decisions, and annotation. The CSVs are then loaded into a database in order for the Machine Learning module to be able to create chunks of data that provide input for the algorithms for performing the tasks of pattern mining and parameter learning. Optionally, the Machine Learning module can also accept domain / background knowledge and prior composite event pattern definitions in the form of logic-based rules. The module will combine the input data (i.e., historic data and rules) with the user-provided annotation, in order to (a) extract new event definitions, (b) refine the current event definitions and (c) associate each event definition with a degree of confidence (e.g., a weight or a probability). In step 3, the resulting output of the Machine Learning algorithms is a set of text-formatted files, using the logic-based representation of the RTEC system (Artikis, Sergot and Paliouras 2014). Thereafter, the resulting patterns are parsed by the "rtec2proton" translator and converted semi-automatically to JSON formatted Proton EPN definitions (step 4). All EPN definitions reviewed and manually refined by domain experts (step 5) using the Proton's CEP authoring tool. Finally, the refined EPN definitions are exported to the SPEEDD's CEP runtime system using Proton's JSON format.

2.5.2 Authoring of CEP Rules

Proton provides a web-based authoring application for creating and updating the event processing network definition. As mentioned above, the process of translation of the event pattern definitions produced by the machine learning component is semi-automatic: partial definition could be generated by the machine learning tool while a review and editing still might be required by human.

The output of this process is a JSON file containing the EPN definition.

More details about the Proton authoring tool along with the user manual can be found in the Proton user guide document (IBM Research 2015).

2.6 Integration – APIs and Data Formats

In the following, we summarize the integration details, APIs and data formats used for inter-component communication in SPEEDD runtime infrastructure.

2.6.1 Input Events

Input events for both Traffic Management and Credit Card Fraud Management use cases are formatted as comma-separated value tuples. Below we provide a brief summary of the event types and description of main data fields, along with some examples. The full description of the data formats is available in deliverables D7.1 and D8.1.

2.6.1.1 Traffic Management

For traffic management use case input events represent aggregated sensor readings. The format of sensor reading events produced by real physical sensors (provided to the project team as historic log files) differs from the format of the events generated by the AIMSUN simulator. Below we provide an excerpt from D8.1 that describes the sensor data formats for each case.

The current version of the SPEEDD prototype deals with aggregated sensor readings reported once per a predefined period of time (e.g. 15 seconds for the physical sensors). Individual sensor readings reported about every single vehicle will be addressed in the next version of the prototype.

2.6.1.1.1 Aggregated Sensor Readings

Aggregated sensor readings are reported every 15 seconds. Each reading has a csv representation that contains the following fields:

- **date**, format YYYY-MM-DD
- **time**, format hh:mm:ss GMT
- **location**, which is represented by the id of the access point collecting data
- **lane**, whose value match the kind of lane the sensor is installed in: slow (lane), fast (lane), on-ramp (entry), offramp (exit), etc.
- **occupancy**, that is the percentage of time the sensor had vehicle above itself
- **vehicles**, the number of vehicles that were counted by a sensor during the last 15 seconds
- speed, in kilometers per hour
- **histogram of speeds**: 20 bins of 10 kilometers per hour each (0-10, 10-20, …, 190-200)
- **histogram of lengths**: 100 bins of 0.5 meters each (0-0.5, 0.5-1, 1-1.5, …, 49.5-50)

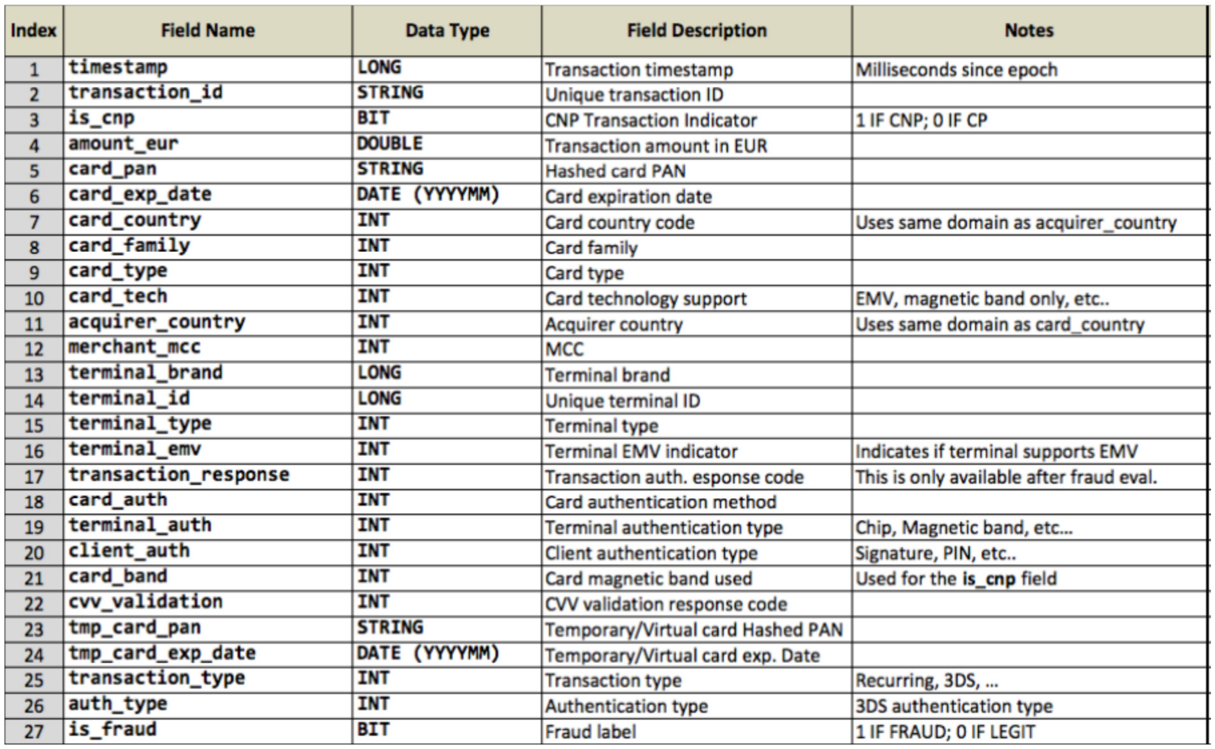
2.6.1.1.2 AIMSUN Simulator Sensor Readings

Every reading reported by AIMSUN every 15 seconds contains the following fields:

- **Simulation date and time**, in the format of YYYY-MM-DD HH:MM:SS
- **Detector ID** – sensor id placed on a section in AIMSUN
- **Vehicle Speed** – average vehicle speed (km/h)
- **Vehicle Count_car** – the number of cars passing through the sensor at each 15 seconds
- **Vehicle Count_truck** – the number of trucks passing through the sensor at each 15 seconds
- **Density_car** – density of cars over the last period of 15 seconds
- **Density_truck** – density of trucks over the last period of 15 seconds
- **Occupancy** – fraction of the last time period that there have been a vehicle over a sensor

2.6.1.2 Credit Card Fraud Management

Every transaction is reported as an event represented in a csv format. The table in the Figure 2.24 lists the fields present in each Transaction event.



Index	Field Name	Data Type	Field Description	Notes
1	timestamp	LONG	Transaction timestamp	Milliseconds since epoch
2	transaction_id	STRING	Unique transaction ID	
3	is_cnp	BIT	CNP Transaction Indicator	1 IF CNP; 0 IF CP
4	amount_eur	DOUBLE	Transaction amount in EUR	
5	card_pan	STRING	Hashed card PAN	
6	card_exp_date	DATE (YYYYMM)	Card expiration date	
7	card_country	INT	Card country code	Uses same domain as acquirer_country
8	card_family	INT	Card family	
9	card_type	INT	Card type	
10	card_tech	INT	Card technology support	EMV, magnetic band only, etc..
11	acquirer_country	INT	Acquirer country	Uses same domain as card_country
12	merchant_mcc	INT	MCC	
13	terminal_brand	LONG	Terminal brand	
14	terminal_id	LONG	Unique terminal ID	
15	terminal_type	INT	Terminal type	
16	terminal_emv	INT	Terminal EMV indicator	Indicates if terminal supports EMV
17	transaction_response	INT	Transaction auth. esponse code	This is only available after fraud eval.
18	card_auth	INT	Card authentication method	
19	terminal_auth	INT	Terminal authentication type	Chip, Magnetic band, etc...
20	client_auth	INT	Client authentication type	Signature, PIN, etc..
21	card_band	INT	Card magnetic band used	Used for the **is_cnp** field
22	cvv_validation	INT	CVV validation response code	
23	tmp_card_pan	STRING	Temporary/Virtual card Hashed PAN	
24	tmp_card_exp_date	DATE (YYYYMM)	Temporary/Virtual card exp. Date	
25	transaction_type	INT	Transaction type	Recurring, 3DS, ...
26	auth_type	INT	Authentication type	3DS authentication type
27	is_fraud	BIT	Fraud label	1 IF FRAUD; 0 IF LEGIT

Figure 2.24 - Fields in a Transaction event

2.6.2 Communication between SPEEDD components

There are two mechanisms of communication between SPEEDD components:

- Storm messaging infrastructure: events emitted by Proton are passed to the Decision Making component through Storm built-in messaging infrastructure. This channel of communication is 'static', in the sense that the wiring between the event source and the listener is set at the build time and cannot be changed without modifying the code

- SPEEDD Event bus: the kafka-based event bus serves as the main channel for both internal communication between SPEEDD components and external event producers and consumers

Independently of the communication channel, the communication is asynchronous, and event-based: both ‘input’ and ‘output’ objects represent events.

In 2.6.1 we described the structure of the input events for each use case. All the events emitted by SPEEDD components have uniform structure, defined by the org.speedd.data.Event interface:

```
public interface Event {
      public String getEventName();
      public long getTimestamp();
      public Map<String, Object> getAttributes();
}
```

Every event has an event name that identifies the type of the event in the system. For instance, “*AggregatedSensorRead*” is the value of the event name for aggregated sensor reading events for the traffic management use case.

Additionally, every event has a timestamp (number of milliseconds since January 1st, 1970). The timestamp represents the detection time, i.e. the time when it’s been recognized by the SPEEDD runtime. Specifically, for input events, the event timestamp is initialized by the code executed by Kafka spout to parse an input event. For derived events, the timestamp is initialized by the SPEEDD runtime component that has created it. Note, that for input events the occurrence time will be different from the detection time whereas for derived events and actions the occurrence and the detected times are equal.

Finally, every event object has a map of attributes, keyed by an attribute name (a string), where an attribute is an object, which allows supporting various attribute types.

Events posted on the event bus are serialized using JSON text-based format. One exception is for the input (raw) events which are comma-separated values, as stated above. Below is an example of a serialized event representing an action, - *“UpdateMeteringRateAction”*:

```
{
      "timestamp": 1409901066030,
      "Name": "UpdateMeteringRateAction",
      "attributes": {
            "density": 1.7333333333333334,
            "location": "0024a4dc00003354",
            "lane": "fast",
            "newMeteringRate": 2
      }
}
```

Components should use Kafka client API for posting and consuming events. The API is documented in the Kafka API documentation online[12].

The Kafka topics that serve as different event topics for SPEEDD event bus are described in 2.4.1, in Table 2.1.

The full list of all the event types used in SPEEDD is available in appendix 12.

2.7 Deployment Architecture

The diagram in Figure 2.25 shows the draft of SPEEDD runtime deployment architecture. The environment on the diagram corresponds to the performance testing setup for SPEEDD, where the goal is to generate input events at rates close or higher than the rates stated in the system requirements. Every box in the diagram represents a computing node (a physical or a virtual machine). Blue boxes represent SPEEDD runtime components. The red boxes correspond to the test agents that run on separate machines and generate input events. The dashed box represents an external planning and optimization module mentioned in 2.4.5. Kafka and Storm clusters are built according to the deployment pattern common to these systems: multiple Kafka broker machines handle different topic partitions (see 2.4.1.1 and 2.4.1.2 for discussion on partitioning of events). Multiple Storm supervisor instances run Proton event processing agents in parallel (see 2.4.4), sharing common nimbus instance for coordination. Both Storm and Kafka clusters use the same zookeeper cluster for distributed coordination and state management.

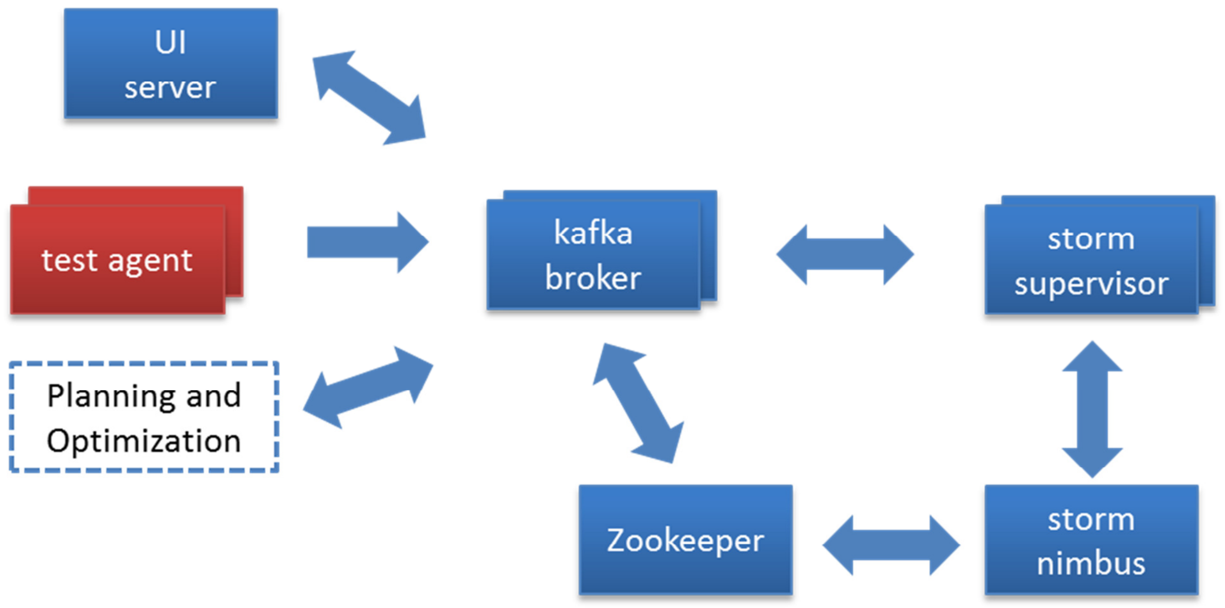


Figure 2.25 - Deployment Architecture

[12] http://kafka.apache.org/documentation.html#api

It is important to mention that for development, functional testing, and demonstration purposes the entire deployment of SPEEDD prototype can run on a single machine (e.g. a developer's laptop), in a single virtual machine instance, or in containers (see 7.2).

2.8 Non-Functional Aspects

2.8.1 Scalability

The proposed architecture is horizontally scalable to provide required throughput with limited latency. The event bus is based on Apache Kafka messaging infrastructure, which provides horizontal scalability via topic partitioning where every partition can be managed by a separate node. The partitioning strategy that should enable efficient scaling out is discussed in Sections 2.4.1.1 and 2.4.1.2.

As previously noted, the stream processing infrastructure is based on Apache Storm[13] which is also scalable as all the processing units (bolts and spouts) can be distributed over a large cluster of machines.

2.8.2 Fault Tolerance

Although the goal of the project is building a prototype and not an operational product, it should be robust enough to allow exploring and testing research ideas and algorithms implemented in the prototype. As the planned testing should involve replaying large amounts of historic events at rates close to reality, even errors and failures with low probability can become fairly common. Thus, our goal is to provide certain level of fault tolerance in the SPEEDD runtime to the level that would allow consistent and continuous running and testing.

Similar to as we do for scalability, to provide the required level of fault tolerance, we leverage the capabilities built into the middleware infrastructure SPEEDD is built upon. Every partition in Kafka messaging infrastructure can be configured to have one or more replica, thus allowing standing failure of K-1 broker servers when K is the number of replicas for a partition. In Storm, when workers die, they are automatically restarted. Workers can be restarted on a different node in case when their hosting node dies.[14]

2.8.3 Testability

Testability is an important aspect of every system, and SPEEDD is not an exception. We aim at providing testability on multiple granularity levels. We leverage various unit testing frameworks (e.g. JUnit[15], Mockito[16]) to implement unit tests for every component, e.g. a class implementing a Storm bolt, or a message serialization.

For component-level testing we run Storm and Kafka in embedded mode, i.e. in the same JVM that the unit test runs. This allows running a component (e.g. CEP topology) in its runtime environment, and verifying that the component's behavior complies with the designed API.

[13] https://storm.apache.org/
[14] https://storm.apache.org/about/fault-tolerant.html
[15] http://junit.org/
[16] https://code.google.com/p/mockito/

Event-driven architecture facilitates testing and debugging by easily adding/replacing event producers and consumers for generating test input or verifying output events. This provides a convenient platform for integration test automation.

Finally, all the tests can be run automatically as part of the automatic build process. We are using Maven build automation framework to build SPEEDD software.[17]

[17] http://maven.apache.org/

3 Test Plan

The test plan for SPEEDD prototype contains the following three types of test cases:

1. Unit
2. Functional
3. Performance

Unit tests are developed by component owners during the process of building the components. They come to verify component's behavior based on the contract declared in the API design.

Functional tests verify the correctness of the prototype functionality. For every test case, a limited sequence of raw events is injected using SPEEDD event submission API (Kafka producer API). The expected results include the list of detected or predicted events and the list of suggested automatic actions. The actual events and actions issued by SPEEDD runtime will be compared to the expected ones to verify the correctness. This part of functional testing can be fully automated using automated testing tools and frameworks (see 2.8.3). Testing of the dashboard component can be partially automated, while still requiring participation of a human tester to verify the visualization correctness.

Performance tests come to verify the performance and scalability characteristics of the prototype. In Section 2.7 we describe the deployment environment that should be used to run performance tests. In the course of performance testing we experiment with different cluster sizes and configurations to determine performance characteristics of the system (e.g. throughput, latency) as well as test the scalability of the prototype architecture.

4 Performance Analysis

4.1 Objectives

The objectives of the performance analysis process are as follows:

1) Assess the current system performance
2) Identify and investigate performance issues
3) Explore approaches to improve the performance and overall robustness of the system
4) Verify the ability of the system to match the performance requirements set by the use cases (see 2.1)

4.2 Approach

The main metric of the system that we are interested in is the latency as a function of the event rate. In order to learn the system behavior under load we stream the same set of events to SPEEDD at several different rates, analyze the latencies, and draw a graph of latency as the function of the event rate. We repeat the same experiment on different cluster sizes to learn about the scalability of SPEEDD runtime. The desired outcome is that the system performance improves with increase of the number of computing nodes (e.g. broker nodes and/or storm supervisor nodes).

The aggregated value for the latency that characterizes the system behavior at a given rate is computed as the 90% percentile of all the latency values observed during the run. This approach should eliminate the influence of the outliers while providing a good enough metric for the overall performance assessment of the system (Oaks 2014).

For synchronous systems the method of testing the latency (called 'response time') is fairly straightforward: a client calls the system under test (further SUT) continuously (i.e. next call starts immediately after receiving the response to the previous one), the latency is the time it took to receive the response.

The situation with event-driven systems is more complex. First, the semantics of latency differs from that of the response time. In SPEEDD we define the latency as the time period that it takes till the emission of an action (or an output event) starting from the latest input event that is required for derivation of the output event. For example: assuming that the event D is defined a sequence of events (E1, E2, E3) then the latency is the time period since an instance of E1 is reported till the emission of the corresponding instance of D. Of course, this definition does not work for all event patterns. For instance, it is not applicable for 'absence' event patterns, or any other event patterns triggered by expiration of a time window. Still, it is a good enough definition for the performance analysis task.

The conceptual view of the performance testing architecture is presented in Figure 4.1. 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.

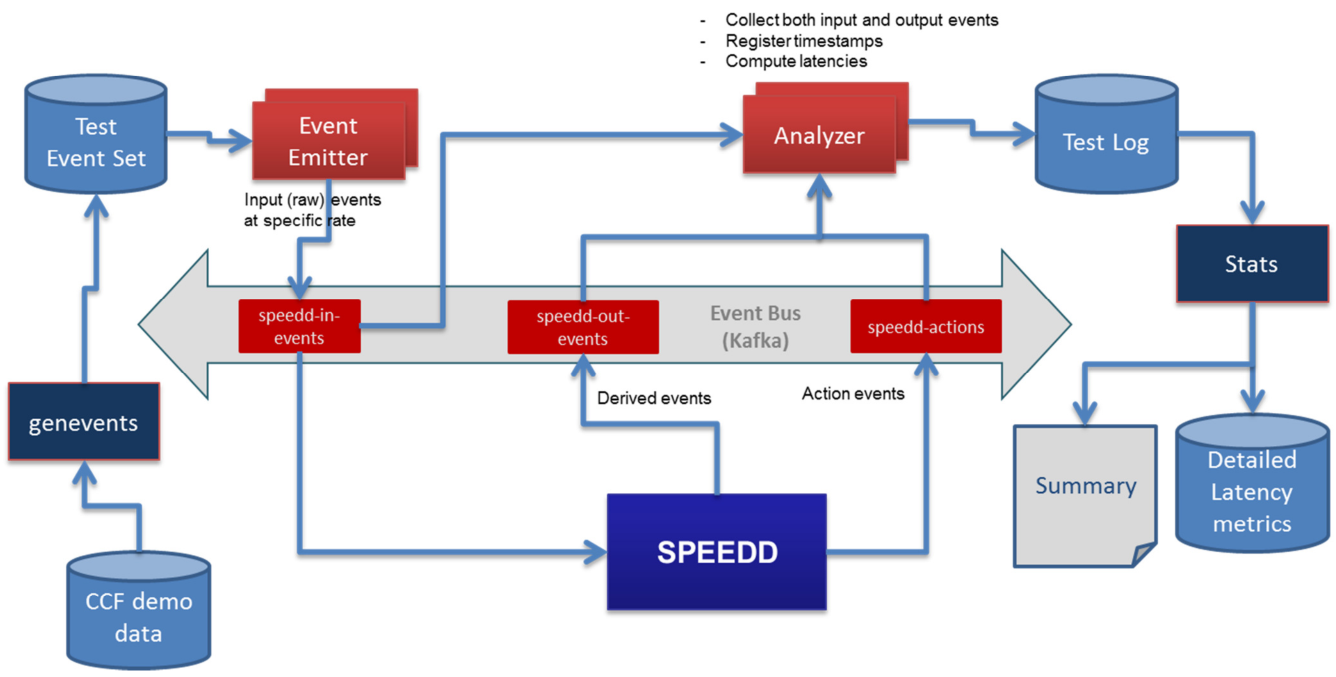


Figure 4.1 SPEEDD Performance Testing Architecture - Conceptual View

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.

4.3 Performance Testing Architecture and Configuration

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[18] 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[19] framework used to run storm cluster on Mesos, the kafka-mesos[20] framework used for running kafka cluster on Mesos.

There topology of the Mesos cluster is shown in the Figure 4.2.

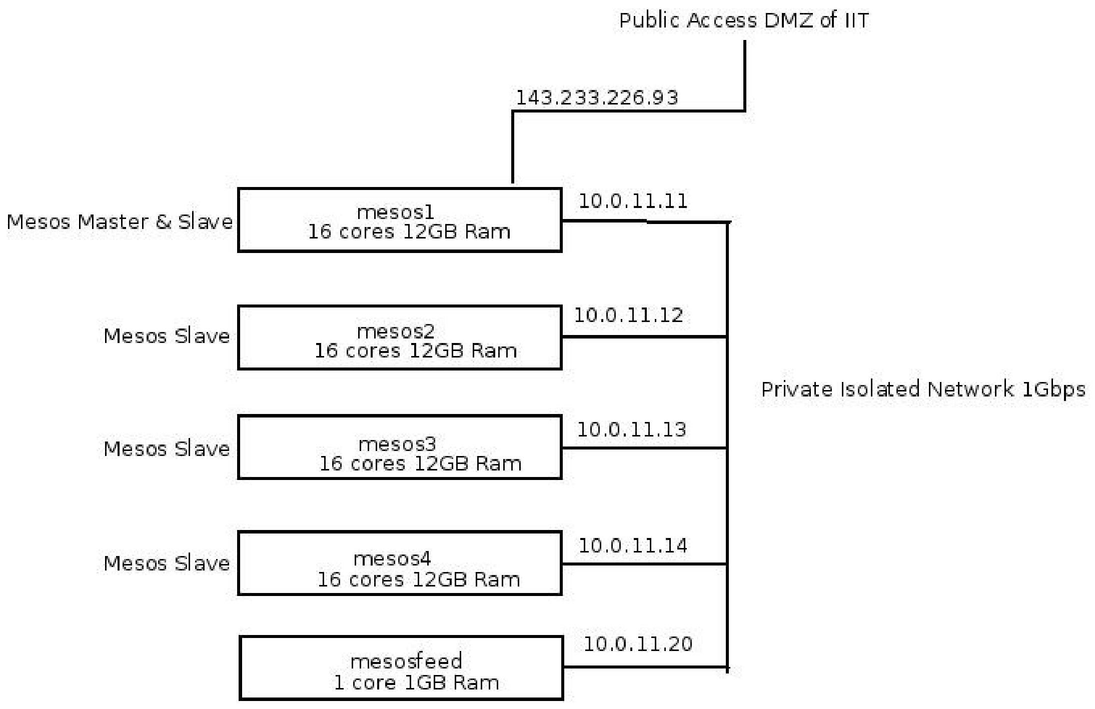


Figure 4.2 - Mesos cluster topology

The Table 4.1 lists the various configurations of SPEEDD runtime on which the performance tests performed. For definition of Storm components (e.g. worker, executor) please see Appendix 6.2. In addition, we introduce another deployment configuration parameter, - CEP parallelism hint. The CEP Parallelism Hint is a parameter provided by Proton CEP module for scaling the CEP topology over multiple threads and tasks. The value of the CEP Parallelism Hint determines the number of executors and tasks allocated for each bolt in Proton topology. That is, CEP Parallelism Hint = 2 means that 2 threads and two tasks will be created for each bolt in Proton runtime.

As one can see in the table, configurations that involve multiple kafka partitions and brokers are not documented 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

[18] http://mesos.apache.org/
[19] https://github.com/mesos/storm
[20] https://github.com/mesos/kafka

executors for kafka-bolt) was operating significantly below its capacity, and further increase of messaging power will 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

4.4 Performance Test Results

For each of the deployment configurations listed in Table 4.1, we performed tests for rates of 50 and 500 events per second. Based on the findings in these tests we decided to not test on higher rates in this version – as those would not provide any new information beyond the issues found during lower rate tests.

The scenario used for performance tests in this version was the Credit Card Fraud Management scenario. First, this use case implies the strictest performance requirements among the two use cases addressed in the project (see 2.1). Second, the derived event patterns for Credit Card are well suited for latency measurement technique described above, as the input events directly contribute to the detection of derived events. This is different in the Traffic Management use case, where the patterns are more complex and often the raw event does not directly trigger a derived one. Based on these two points, we decided to start the testing process with the Credit Card case, in order to obtain the initial performance results and gain insights into the overall system performance. As the next step we would proceed with more specific tests for the Traffic Management use case. However, the initial results and the performance issues encountered made us to decide that before proceeding to the next phase, the current issues have to be addressed. A more detailed description is provided below.

The performance results are summarized in Table 4.2. In section 4.5 below we provide a detailed analysis of the observed results and suggest the ways for improvement in future.

Table 4.2 - Performance Results Summary

Config	Number of workers	CEP parallelization factor	End-to-end latency (s) 50 events/sec	End-to-end latency (s) 500 events/sec
1	1	1	25	
2	1	2	25	215
3	2	4	21	148
4	4	8	13.5	121
5	4	16	15	129

6	8	16	7.7	75.4

4.5 Performance Analysis and Conclusions

The graph in the Figure 4.3 illustrates the current performance results. The graph shows the dependency of the end-to-end latency on the number of executors (threads). In the same figure there is a graph illustrating the number of workers. From looking at these graphs one can easily see that adding more executors to the topology does not improve the end-to-end latency. On the other hand, adding more workers (JVM processes) leads to lower latency values.

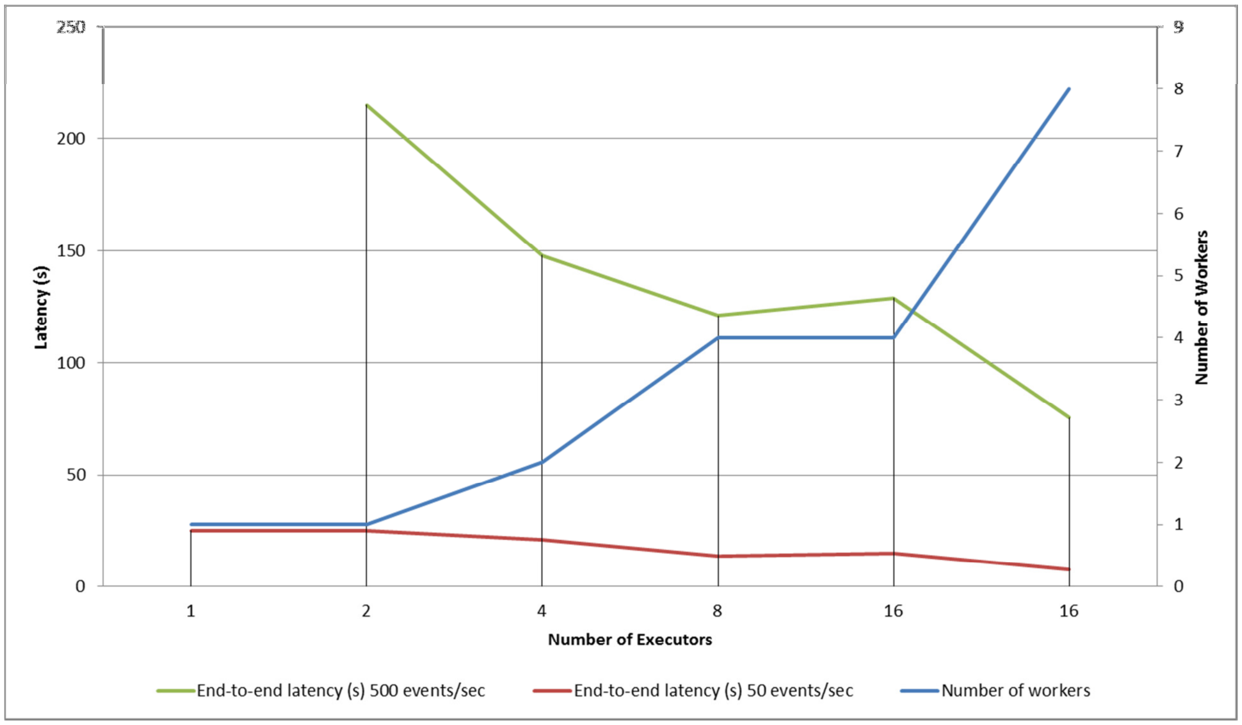


Figure 4.3 - Performance Results: Adding more workers improves latency, adding more executors – does not

To understand this behavior we inspected the metrics collected by STORM for all topology components. Investigating these metrics (presented by Storm UI web application) revealed two findings. First, we learnt that all of the bolts composing Proton component's sub-tree in SPEEDD topology divide the load uniformly, except one bolt – the contextBolt (see Figure 4.4 for a diagram of the SPEEDD topology for Credit Card use case). For contextBolt, at most three executors were actually processing tuples even when the topology had more (in Figure 4.5 one can see the statistics when running on topology with 16 executors for contextBolt). 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.

Another limiting factor identified in course of the performance testing is the use of Java threads inside the bolts. Specifically, contextBolt creates Java threads internally in order to carry context-specific computations. There are two problems in this approach: First, this limits the parallelization and distribution to a single JVM. Second, threads created "internally" are "invisible" to Storm metrics system, so it's hard to monitor the actual system performance using existing Storm mechanisms.

These issues in the current implementation make the contextBolt in Proton sub-tree a bottleneck of the system that is responsible for the poor performance results.

In the next version (v3) of SPEEDD prototype, the architecture of the ProtonOnStorm component should be revised and updated following the best practices of development for Storm platform.

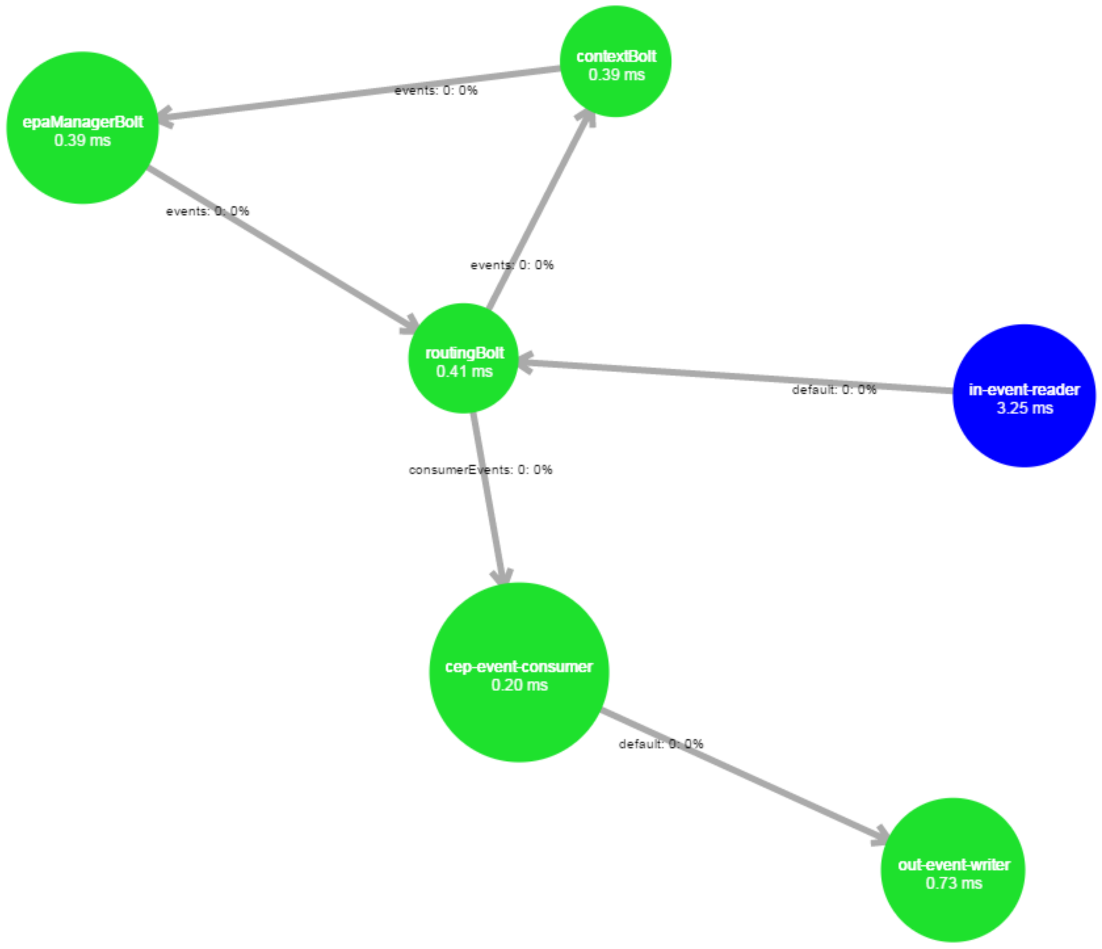
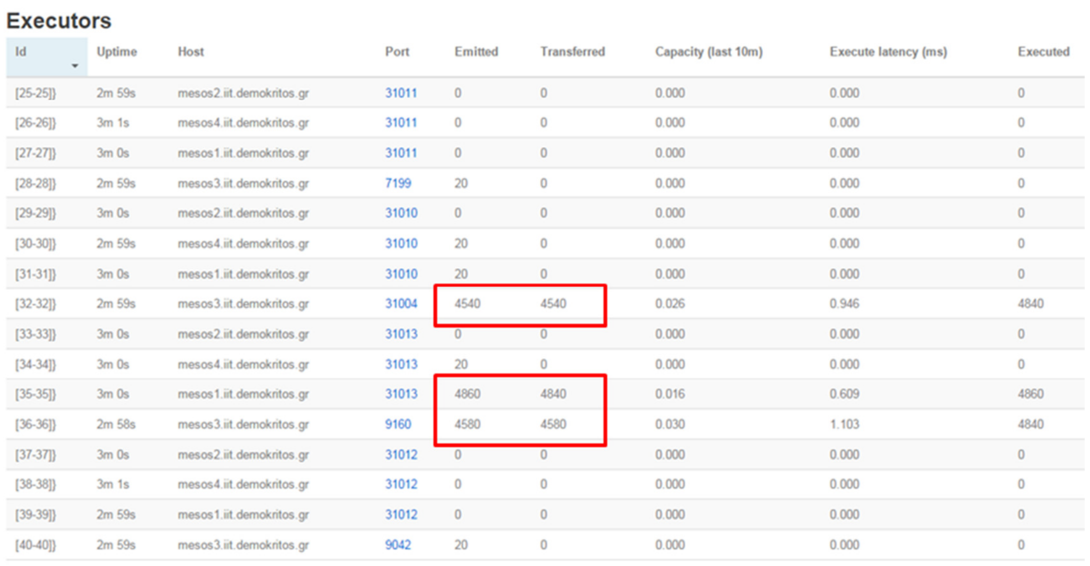


Figure 4.4 - SPEEDD Topology for Credit Card use case: a screenshot from Storm UI



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 4.5 - Storm UI: Proton Context Bolt runtime metrics: only 3 of 16 executors actually involved in processing

In addition to the performance-related architecture issues described above, we have also learnt that the clustered environment configuration should be revised for the further testing efforts. Although Mesos platform simplifies cluster management tasks, and improves resource utilization, for performance testing task it is sometimes required to control the deployment configuration more closely, e.g. bind processes to specific nodes. Depending on the framework (e.g. Kafka, storm) it might be hard to impossible to achieve that level of control.

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 Conclusions

In this document, we presented our proposed architecture for the SPEEDD prototype and drafted main architecture principles that should guide its development. The architecture follows the event-driven style enabling highly composable loosely coupled dynamic system that would be easy to build and test by a distributed team. Our implementation will be based on two mainstream technologies – Storm and Kafka, for stream processing and event bus implementation respectively. The document also defines the APIs and data formats for sending events to SPEEDD, and consuming events and actions produced by SPEEDD, as well as for inter-component communication within SPEEDD runtime.

It is important to mention that architecture is a continuously evolving artifact. In course of our development work we anticipate new findings, issues, and questions that will lead to revision of some architectural decisions. We believe that the architectural foundation documented here is agile enough to allow and facilitate these changes.

6 Appendix – Technology Evaluation

SPEEDD runtime architecture is built according to the event-driven paradigm and is based on two major technology platforms: stream processing and messaging technologies. In course of working on the architecture design we have evaluated several technologies and chose Apache Storm as our stream processing platform and Apache Kafka for the messaging. Below you can find some details regarding the evaluation process and the options explored.

6.1 Stream Processing – requirements and evaluation criteria

Our evaluation of the stream processing technologies was based on the following criteria:

1. Suitability for implementing SPEEDD components
    - The stream processing platform should allow building both CEP and DM functionality on top of it. Although DM component was going to be developed from scratch, the CEP component was planned to base on the Proton technology. The platform of choice should match Proton architectural principles and allow easy porting.
2. Scalability, Performance
    - Support 2K events/sec at <25 latency – derived from SPEEDD requirements (see 2.1)
3. Fault tolerance
    - Although the final product of the project is not a production-ready system but a prototype, fault tolerance is a highly desired feature when it comes to dealing with large volumes of events arriving at high rate.
4. Connectivity
    - The streaming platform should provide support for integration with high-throughput messaging systems
    - For managing operational data we are likely to need an in-memory data store. The streaming technology should provide a mechanism to connect to such a data store and use the data stored there as part of stream computation
5. Extensibility
    - Ability to override or customize behavior of computational nodes – required for implementing our functional components
    - Ability to develop integration components if not-available or do not match our needs
6. Programming model and language support
    - Java support is required (Proton is written in Java)
    - Support of other languages is advantageous
    - Programming model should be aligned easy porting of Proton code to it
7. Maturity
    - We are looking for a mature stable platform to build our features upon
8. Code reuse and cross-initiative collaboration

- We are working in collaboration with the FERARI[21] project which is also building a highly scalable event processing technology based on Proton as the complex event processing engine. Knowledge and code reuse are beneficial for both activities.

6.2 Storm

Apache Storm (Toshniwal, et al. 2014)[22] is a distributed real-time computational system that provides a set of general primitives for performing computations on streams of data at high speed. Among its key characteristics are:

- **Broad set of use cases:** stream processing (processing messages), continuous computation (continuous query on data streams), distributed RPC (parallelizing intensive computational problem)
- **Scalability:** just add nodes to scale out for each part of topology. One of initial applications processed 1,000,000 messages per second on a 10 node cluster
- **Fault tolerant:** automatic reassignment of tasks as needed
- **Programming language agnostic:** topologies and processing components can be defined in many languages
- **Configurable message delivery semantics:** At-most-once message delivery (default), At-least-once delivery (can be enforced), exact-once (using Trident[23] high-level abstraction)

Stream in Storm is an unbounded sequence of tuples, where a tuple is a named list of values. A field in a tuple can be an object of any type.

Two basic building blocks are available: spouts and bolts. A spout is a source of one or more streams. For example, a spout can connect to a message queue, read messages from the queue and emit them as a stream. A bolt is a processing node. A bolt consumes one or more streams, and may emit new streams as output. Bolts can perform various types of processing: filtering, aggregation, joining multiple streams, writing to databases, executing arbitrary code, etc.

Spouts and bolts are connected into networks called topologies. Each edge of the network represents a stream between a spout and a bolt or between two bolts. An example of Storm topology is presented in Figure 6.1. One can build an arbitrarily complex multi-stage stream computation using this model. A topology is a deployable unit for Storm cluster. Multiple topologies can run on a single cluster.

Storm ecosystem provides integration with a wide variety of the messaging systems and databases, among them are such messaging technologies as Kestrel, RabbitMQ, Kafka, JMS, and such databases as MongoDB, Cassandra and a variety of RDBMS's.

[21] http://www.ferari-project.eu/
[22] http://storm.apache.org/
[23] http://storm.incubator.apache.org/documentation/Trident-tutorial.html

Topologies can be defined and submitted both declaratively and programmatically, using many programming languages, including Java, Python, Ruby, and others.

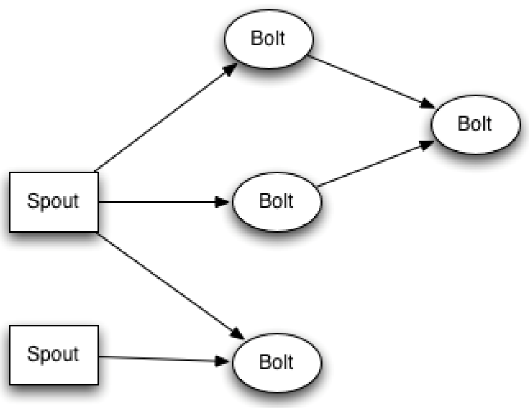


Figure 6.1 - Storm Topology[24]

A wide support for programming languages for developing spouts and bolts is provided as well. All JVM-based languages are supported. For non-JVM languages, a JSON-based protocol is available that allows non-JVM spouts and bolts to communicate with Storm. There are adapters for this protocol for Ruby, Python, Javascript, Perl and PHP.

Each computation node in a Storm topology executes in parallel. A developer can specify how much parallelism they want for a specific computational node; Storm will spawn that number of threads across the cluster to do the execution. This process is illustrated in Figure 6.2.

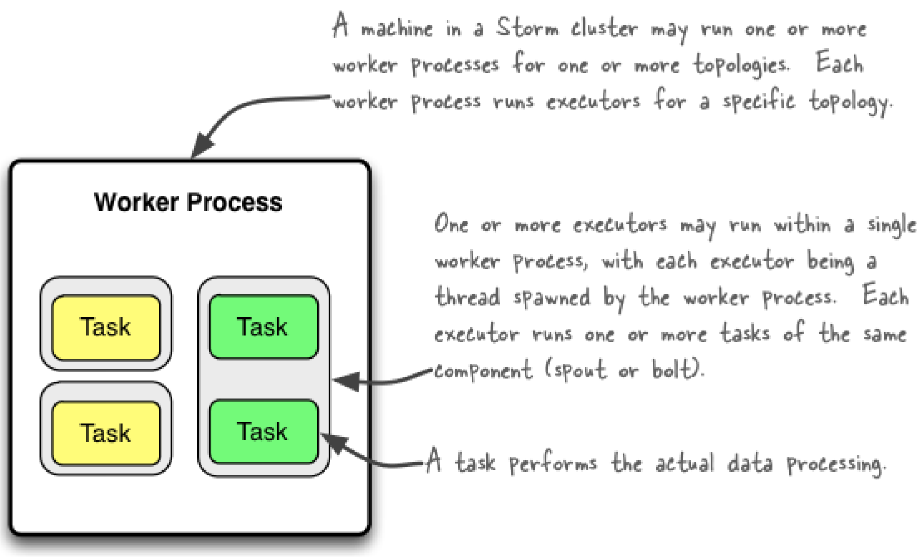

A machine in a Storm cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology.
Worker Process
Task
Task
Task
Task
One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component (spout or bolt).
A task performs the actual data processing.

Figure 6.2 - Storm Parallelization[25]

[24] The diagram is taken from http://storm.apache.org/documentation/Tutorial.html

A running topology consists of many worker processes running on many machines within a Storm cluster. Executors, which are threads in a worker process, run one or more tasks of same component (a bolt or a spout).

Storm provides a mechanism to guarantee that every tuple will be fully processed; if the processing of the tuple fails at some point, the source tuple will be automatically replayed. In many cases losing an event impacts the accuracy of the event processing logic; in such cases the guaranteed message processing provided by Storm is critical.

The performance of Storm highly depends on the application. According to the information on the project site[26], Storm has been benchmarked at processing of 1M of 100 byte messages per second per node on the hardware with dual Intel E5645@2.4Ghz CPU with 24GB RAM, however the details of the benchmark were not available to the authors. A recent performance analysis[27] comparing performance characteristics between IBM Infosphere Streams product and Storm reported ~50K emails per second on a 4-node cluster for an email processing application. While that is much more modest result than 1M per second mentioned earlier, still the reported throughput matches our needs. The final conclusion about Storm performance for SPEEDD requires dedicated performance testing on our workloads and environment.

Storm has recently graduated to become a top-level Apache project.

6.3 Akka

Akka[28] is a toolkit for building scalable distributed concurrent systems. Being written in Scala, it also provides Java API. Due to its modular structure, it can be run as a standalone microkernel, or used as a library in another application.

Akka provides an actor-based programming model (Hewitt, Bishop and Steiger 1973). The functional units are implemented as loosely coupled actors communicating between them via immutable messages. The actors can run on the same or separate machines; the location of an actor is transparent to the developer.

An actor is a container for state, behavior, and the mailbox. All actors compose hierarchies, where an actor is the supervisor for all its children thus having control over children's lifecycle. This provides a convenient mechanism for dealing with failures: a supervisor strategy determines the behavior in case of a child's failure. Akka champions the "let it crash" semantics: instead of dealing with preventing

[25] The diagram taken from http://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html
[26] https://storm.incubator.apache.org/about/scalable.html
[27] https://developer.ibm.com/streamsdev/wp-content/uploads/sites/15/2014/04/Streams-and-Storm-April-2014-Final.pdf
[28] http://akka.io/

failures assume that actors are supposed to fail and crash frequently, and provide simple and robust mechanisms for recovery.

When an actor terminates – all its children are terminated automatically. Actor hierarchy is illustrated in Figure 6.3.

Various and extensible policies are available for managing the actor's mailbox; for example, FIFO, or priority-based.

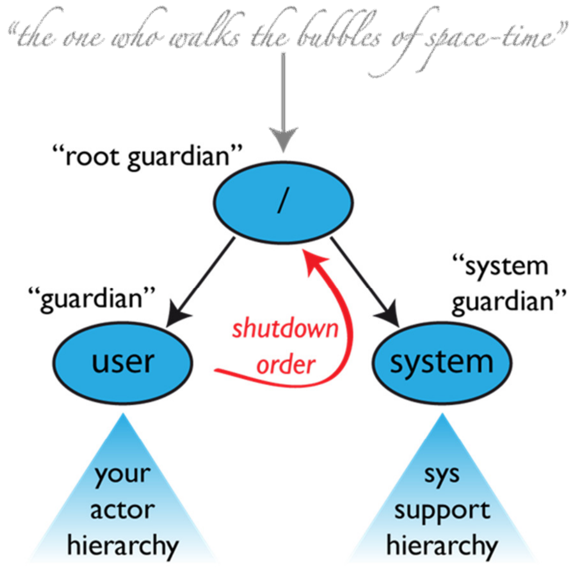


Figure 6.3 - Akka Actors Hierarchy[29]

The communication between actors is asynchronous. The decision about the location of each actor is configurable, and can be done programmatically. Akka provides at-most-once and at-least-once delivery mechanisms. The ordering is guaranteed in scope of the same sender-receiver pair.

Scalability is provided via transparent remoting and powerful and extensible routing mechanism. Multiple instances of the same actor can be created to handle incoming messages in parallel. The router agent receives a new message and routes it to the processing actors according to a routing strategy (e.g. round-robin, random, custom, etc.).

There are several connectivity modules for Akka. Among them are ZeroMQ module and akka-camel module, which supports a variety of protocols (HTTP, SOAP, JMS, and others).

Akka provides akka.extensions mechanism which allows extending the toolkit with new capabilities (typed actors, serializations are examples of extensions).

According to user reports, Akka is very stable since v2.0 (current version is 2.3.6). The toolkit is developed by Typesafe Inc[30]. The project has been open sourced, Typesafe is the major contributor.

[29] Picture source: http://doc.akka.io/docs/akka/2.3.6/general/supervision.html

Akka is an important part of their software stack, which provides some confidence in project's continuity. There is a vivid development community around Akka, including Scala development community but not limited to it.

According to the information on the web[31], Akka has been tested to support up to 50 million messages per second on a single machine. The memory footprint is very small: ~2.5 million actors per GB of heap. Typesafe is known to operate a 2400 nodes Akka cluster.

6.4 Spark Streaming

Apache Spark (Zaharia, Chowdhury, et al. 2012) is a general purpose cluster computing system designed to process large volumes of data in distributed and parallel manner. Spark Streaming[32] (Zaharia, Das, et al. 2013)[33] is an extension of Spark API that enables processing of live data streams.

The programming model of Spark Streaming extends the Spark programming model and follows the functional programming paradigm. The programming model is data-centric, in sense that its focus is on the operations of data items. D-Stream (or, discretized stream) is a basic abstraction that represents input data stream divided into batches (see Figure 6.4). The D-stream is represented as a sequence of RDDs where every RDD contains data from a certain time interval. Programming logic is defined in the form of operations on D-streams where input of an operation is a d-stream and the output is another d-stream. A useful feature of Spark Streaming is window operations that are available "out of the box".

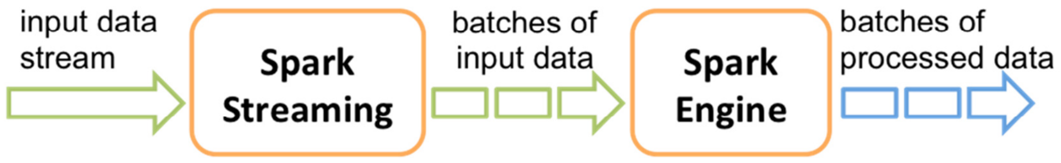


Figure 6.4 - D-Stream is a major concept in Spark Streaming

Spark Streaming deals with failures by providing mechanisms for responding to a worker or a driver node failure. The replication mechanism allows recomputing the RDD from the original data set. In contrast the other frameworks that support at-least-once message delivery, Spark streaming provides stateful exactly-once delivery semantics[34].

Scala and Java APIs are available.

[30] http://typesafe.com
[31] http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine
[32] https://spark.apache.org/
[33] https://spark.apache.org/streaming/
[34] https://spark.apache.org/docs/latest/streaming-programming-guide.html#failure-of-a-worker-node

Spark Streaming supports variety of data sources and sinks. Among them are various messaging systems (e.g. Kafka, ZeroMQ), HDFS, Twitter, databases, dashboards, etc.

Among other nice features, there are modules implementing some machine learning and graph analysis algorithms.

Performance benchmarks available on the web report throughput of 670K records per second. The same benchmark run on Storm gave 115K records per second. That said, as we mentioned above, performance benchmarks are highly application dependent and it is hard to rely on a specific benchmark result in regard to our project.

Spark Streaming is part of the Apache Spark project which is a top-level project in Apache organization.

6.5 Streaming technologies – conclusion

All the evaluated technologies match our requirements for the stream processing platform. The performance results look very promising even though need more thorough testing on SPEEDD workload. All provide good rich support for connectivity and extensibility.

In terms of maturity of the technology, Spark Streaming and Storm seem to be more widely used and tested than Akka. Storm seems to be most popular based on the amount of information and reference available on the Web.

From the programming model perspective we tend to prefer Akka and Storm over Spark Streaming, because the functional data-centric programming model implemented in Spark Streaming is less aligned with Proton implementation architecture. Akka provides the strongest alignment and flexibility. However the capabilities of Storm are good enough for our needs. Also, Storm is stronger than the two other candidates in terms of supported languages (Akka and Spark are Scala/Java only).

Another important criterion mentioned above is the code reuse and collaboration with other projects. Storm is the stream platform chosen by the FERARI project, and at the moment of current evaluation, there has been already work in progress on porting Proton on Storm. Building SPEEDD on Storm can provide better reuse opportunity and knowledge sharing than other platforms.

Weighing all the considered results we decided to choose Storm as our stream processing infrastructure.

6.6 Choice of the Messaging Platform

Our requirements to the messaging platform for building the event bus infrastructure for SPEEDD include:

- Publish and subscribe capabilities
- Scalability and performance – at least 10 K/s

- Ordering of messages – order of events is important. Although there are some capabilities in Proton to deal with out-of-order events, these might not be present or be hard to implement for other components of SPEEDD
- Ease of use in prototype – we need a light-weight simple technology that could run on a developer's laptop for development and testing purposes and still allow large deployment to stand real-world scale message rates
- Fault-tolerance – as mentioned in 2.8.2, we need a robust messaging platform that would allow continuous running of SPEEDD prototype on large amount of data coming at high rate in face of occasional local failures

The technologies we considered as potential candidates for SPEEDD were RabbitMQ, Kafka, ActiveMQ, and ZeroMQ – all are highly popular and widely used.

RabbitMQ is the leading implementation of AMQP protocol (Vinoski 2006). Implementing a standard is a strong benefit as it allows for better integration with external systems (esp. in finance domain). RabbitMQ is more mature than Kafka, and provides rich routing capabilities. However, there is no guarantee on order delivery. Among the strengths – RabbitMQ outperforms ActiveMQ by factor of 3.

ZeroMQ is a library of messaging capabilities. It provides the best performance comparing to RabbitMQ and ActiveMQ but is too low level and would require a significant development effort to build our custom messaging solution using ZeroMQ-provided building blocks.

Among the strong sides of ActiveMQ is high configurability; however its reportedly poor performance (22 messages per second in persistent mode[35]) does not match our needs.

Kafka provides partitioning of a fire hose of events into durable brokers with cursors – a very scalable approach. It supports both online and batch consumers and producers. Designed from the beginning to deal with large volumes of messages, Kafka provides a very simple routing approach – topic based only, which is sufficient for SPEEDD messaging needs. Kafka guarantees ordered delivery within same partition – good enough for SPEEDD. Performance-wise, Kafka outperforms RabbitMQ when durable ordered message delivery is required (Kreps, Narkhede and Rao 2011) (publish: 500K messages per second, consume: 22K messages per second).

Kafka is written in Scala (Java API is available). There are client libraries in all common languages.

Based on the above, our Kafka seems to be the best choice for the event bus infrastructure.

[35] http://bhavin.directi.com/rabbitmq-vs-apache-activemq-vs-apache-qpid

7 Appendix – Setup Guide

7.1 Overview

There are two options for running the first version of the SPEEDD prototype:

1) running SPEEDD runtime in Docker containers provided as part of this deliverable
2) building the prototype from source code and running on any machine that satisfies the system and software prerequisites (see https://github.com/speedd-project/speedd/wiki/Setting-Up-Development-Environment#setup-development-environment-on-your-machine)

In the following we provide instructions for each option.

7.2 Running SPEEDD runtime in Docker containers

The easiest way to setup an instance of SPEEDD runtime cluster for development and testing purposes is to use the container-based deployment based on the Docker[36] technology. The docker image source code is shared on github: https://github.com/speedd-project/speedd-docker

The following instructions are copied from the README file in the git repository. Note that the master copy of the instructions is in the source repository.

This project configures the SPEEDD multi-node dockerized environment. It's comprised of the following components:

- Zookeeper
- Storm cluster:
  - Storm nimbus
  - Storm supervisor
  - Storm UI
- Kafka broker
- SPEEDD UI
- SPEEDD client

7.2.1 Pre-requisites

1. Install docker toolkit:
   1.1. Windows
   1.2. Mac OS X

   Note: Because of an issue with default docker machine you have to create a new machine with overlay storage driver using the following command:

   ```
   docker-machine create -d virtualbox --engine-storage-driver overlay overlay
   ```

[36] https://www.docker.com/

After creating the overlay machine please update the start.sh file under your Docker Toolbox installation folder to have the following line:
Instead of: `VM=default`
Should be: `VM=overlay`

2. Start docker-machine and open the terminal window (e.g. Docker Quickstart Terminal)
3. Build prerequisite images (this step is required because we use our own forked version of the storm and kafka images)
   3.1. Build storm images
      3.1.1. clone https://github.com/speedd-project/storm-docker.git
      3.1.2. cd storm-docker
      3.1.3. ./rebuild.sh
   3.2. Build kafka image
      3.2.1.clone https://github.com/speedd-project/kafka-docker.git
      3.2.2.cd kafka-docker
      3.2.3.docker build -t wurstmeister/kafka-docker --rm=true .
4. Create the 'projects' folder in your home directory and check out the speedd project's source code into it. Then build the speedd-runtime project (`mvn clean install -DskipTests assembly:assembly` - run it from the speedd-runtime folder)

7.2.2 Usage

Start a SPEEDD cluster for traffic management use case:

```
docker-compose -f docker-compose.yml -f docker-compose-tm.yml up -d
```

Note: The client and UI containers will wait for 2 minutes to have the kafka brokers up and topics initialized. Also, it might take a few minutes to storm cluster to fully initialize. Please take this into account before accessing the UI or starting SPEEDD topology.

Start a SPEEDD cluster for credit card fraud management use case:

```
docker-compose -f docker-compose.yml -f docker-compose-ccf.yml up -d
```

Destroy the SPEEDD cluster:

```
docker-compose -f docker-compose.yml -f docker-compose-tm.yml stop
```

Open SSH to the client:

```
ssh root@<docker-machine-ip> -p 49022
```

Note: *The password is initialized to 'speedd'*

Deploy SPEEDD topology (example for the traffic management use case):

1. Open ssh to the client container
2. `cd /opt/src/speedd/speedd-runtime/scripts/traffic`
3. `./start-speedd-runtime-docker`

Stream events into SPEEDD (example for the traffic management use case):

1. Open ssh to the client container
2. `cd /opt/src/speedd/speedd-runtime/scripts/traffic`
3. `./playevents-traffic-docker`

Open SPEEDD UI:

open http://<docker-machine-ip>:43000 in your browser

Open Storm UI:

open http://<docker-machine-ip>:49080 in your browser

7.3 Building from sources

Although we provide a virtual machine image for convenience, it is possible to run the software on any machine that satisfies the system and software requirements. For detailed instructions see the README files as follows:

- SPEEDD runtime: https://github.com/speedd-project/speedd
- Traffic Management Dashboard installation instructions: https://github.com/speedd-project/speedd/tree/master/speedd-ui
- Credit Card Fraud Management installation instructions: https://github.com/speedd-project/speedd/tree/master/speedd_ui_bf

7.4 Offline pattern mining – setup and usage instructions

We outline the build instructions, runtime requirements and module execution procedure for the Offline Machine Learning component of SPEEDD (SPEEDD-ML).

Instructions to build from source

(a) Dependencies

In order to build SPEEDD Machine Learning Module from source, you need to have Java SE Development Kit (e.g., OpenJDK) version 7 or higher and SBT (v0.13.x) installed in your system. All library dependencies are defined inside the build.sbt file. The module requires the following projects to be locally build and published:

1. Clone and publish locally the auxlib project:

```
$ git clone -b v0.1 --depth 1 https://github.com/anskarl/auxlib.git
$ cd auxlib
$ sbt ++2.11.7 publishLocal
```

2. Clone and publish locally the Optimus project (further instructions can be found here)

```
$ git clone -b v1.2.1 --depth 1 https://github.com/vagm/Optimus.git
$ cd Optimus
$ sbt publishLocal
```

3. Clone and publish locally the LoMRF project (further instructions can be found here).

```
$ git clone -b v0.4.2 --depth 1 https://github.com/anskarl/LoMRF.git
$ cd LoMRF
$ sbt publishLocal
```

Once you have successfully built and published the auxlib, Optimus and LoMRF projects, you can either build a standalone version or a "cluster" version.

(b) LPSolve Installation Instructions

Weight learning of SPEEDD ML requires LPSolve to be installed in your OS. We provide instructions for Windows, Linux and Mac OS X systems.

i. Linux distributions

For example, on a ***Debian-based*** system, type the following command:

```
$ sudo apt-get install lp-solve
```

To install Java Native Interface support for LPSolve v5.5.x you need follow the instructions below:

- Download LPSolve dev, 64bit *lp_solve_5.5.2.x_dev_ux64.zip* or for 32bit *lp_solve_5.5.2.x_dev_ux32.zip*, from LPSolve official repository.
    - Extract the file
    - We only need the `lpsolve55.so` file.
- Download LPSolve java bindings (lp_solve_5.5.2.x_java.zip) from LPSolve official repository.
    - Extract the file
    - We only need the `lpsolve55j.so` files

- Create a directory containing the `lpsolve55.so` and `lpsolve55j.so` files, e.g., `$HOME/lib/lpsolve55`
- Add this directory to `LD_LIBRARY_PATH` in your profile file:
    - For BASH shell e.g., inside .profile, .bashrc or .bash_profile file in your home directory:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib/lpsolve55
```

    - For CSH/TCSH e.g. inside ~/.login file in your home directory:

```
$ set LD_LIBRARY_PATH = ($LD_LIBRARY_PATH $HOME/lib/lpsolve55
.)
```

or in ~/.cshrc file in your home directory

```
$ setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$HOME/lib/lpsolve55:.
```

ii. Apple MacOS X

Either download and install from the LPSolve website or from your favorite package manager.

For example, from macports:

```
$ sudo port install lp_solve
```

or from homebrew:

```
$ brew tap homebrew/science
$ brew install lp_solve
```

To install the Java Native Interface support for LPSolve v5.5.x you need follow the instructions below:

- Download LPSolve dev, 64bit *lp_solve_5.5.2.x_dev_ux64.zip* or for 32bit *lp_solve_5.5.2.x_dev_ux32.zip*, from LPSolve official repository.
    - Extract the file
    - We only need the lpsolve55.dylib file.

- Download LPSolve java bindings (lp_solve_5.5.2.x_java.zip) from LPSolve official repository.
  - Extract the file
  - We only need the `lpsolve55j.jnilib` files
- Create a directory containing the `lpsolve55.dylib` and `lpsolve55j.jnilib` files, e.g., `$HOME/lib/lpsolve55`
- Add this directory to LD_LIBRARY_PATH inside .profile file in your home directory:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib/lpsolve55
```

iii. Microsoft Windows

To install LPSolve v5.5.x in your system, follow the instructions below:

- Download LPSolve dev, 64bit *lp_solve_5.5.2.x_dev_win64.zip* or for 32bit *lp_solve_5.5.2.x_dev_win64.zip*, from LPSolve official repository.
  - Extract the file
  - We only need the lpsolve55.dll file.
- Download LPSolve java bindings (lp_solve_5.5.2.x_java.zip) from LPSolve official repository.
  - Extract the file
  - We only need the lpsolve55j.jar and lpsolve55j.dll files
- Create a directory containing the lpsolve55.dll, lpsolve55j.jar and lpsolve55j.dll files, e.g., C:\path\to\lpsolve55
- Add this directory to the PATH environment variable in your system environment variables

(c) Build SPEEDD-ML Module

To build the SPEEDD Machine Learning Module, give the following commands:

```
$ cd path/to/speedd/speedd-ml
$ sbt clean dist
```

After a successful compilation, the SPEEDD Machine Learning Module is located inside the ./target/universal/speedd-ml-<version>.zip file. You can extract this file and add the path/to/speedd-ml-/bin in your PATH, in order to execute the SPEEDD Machine Learning Module scripts from terminal.

(d) Initialize database schema

In the `schema` directory there are CQL files that define the schema for each use case. For example, to initialize the schema in Cassandra DB for Traffic Management use case write the following command:

```
$ cqlsh -f path/to/speedd/speedd-ml/schema/cnrs.cql
```

Runtime Requirements

SPEEDD-ML module requires Apache Cassandra 2.1.x and Apache Spark 1.5.1 (for Scala 2.11) installed in your system.

- **Apache Cassandra**: To install Apache Cassandra follow the instructions available here.
- **Apache Spark for Scala 2.11*:*** By default Apache Spark 1.5.1 is provided only for Scala 2.10. In order to compile Apache Spark 1.5.1 for Scala 2.11, download Spark 1.5.1 source file spark-1.5.1.tgz and give the following commands:

```
$ tar xf spark-1.5.1.tgz
$ cd spark
$ ./dev/change-scala-version.sh 2.11
$ ./make-distribution.sh --name scala_2.11 --tgz --skip-java-test -Phadoop-
 2.4 -Pyarn -Dscala-2.11
```

The resulting Spark distribution will be packed in file spark-1.5.1-bin-scala_2.11.tgz.

SPEEDD-ML Execution and Configuration

(a) Quick notes for running a local instance of the module

To start a local instance of the Cassandra DB:

```
$ /path/to/cassandra/bin/cassandra -f
```

To stop Cassandra, press CTRL+C.

To start a local Spark instance, in standalone mode:

```
$ cd /path/to/spark/
$ ./sbin/start-all.sh
```

To stop a Spark local instance:

```
$ cd /path/to/spark
$ ./sbin/stop-all.sh
```

(b) SPEEDD-ML Module Configuration

All configuration files of the SPEEDD ML module are located in the path/to/speedd-ml-v0.1/etc. Depending on your installation and OS configuration you may need to adjust the parameters of speedd-ml-0.1/etc/spark-defaults.conf file. The spark-defaults.conf file is a standard Spark Configuration file (see Available Properties from Spark Documentation)

For example, you may need to change the spark master URL:

```
spark.master            spark://localhost:7077
```

(c) SPEEDD-ML executable script files

- Data loading
  - `cnrs-loader.sh` : loads data from CSV files into the Cassandra DB (Traffic Management use case)
- Machine Learning
  - speedd-wlearn.sh : performs weight leaning

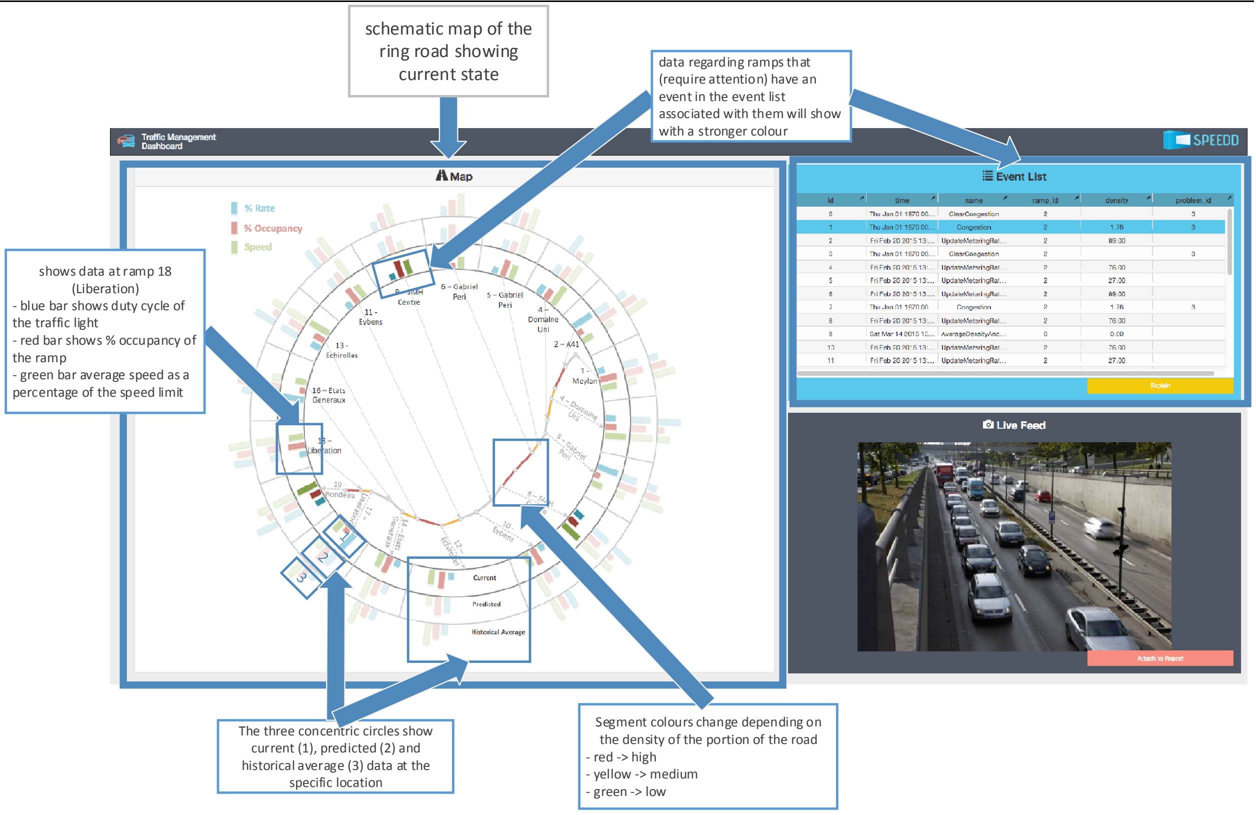
Example parameters for loading raw input data in:

```
$ cnrs-loader.sh \
    -d /path/to/CSV/ \  # path to directory containing the CSV files
    -t input \          # load and transform for raw input data
    -i suffx:csv.gz \   # load files that match to *.csv.gz (the files can be gzipped)
```

Example parameters for weight learning:

```
$ speedd-wlearn.sh \
    -t CNRS \                      # perform learning for the traffic management use case
    -in /path/to/cnrs.mln \        # path to the initial MLN file
    -out /tmp/cnrs-out.mln \       # path to the resulting MLN file
    -i 1397041487,1397042487 \     # the temporal interval read input and annotation data from the DB
    -bs 20                         # perform online learning for micro-batched with 20 time-points duration
```

8 Appendix – User Reference: Traffic Management Dashboard

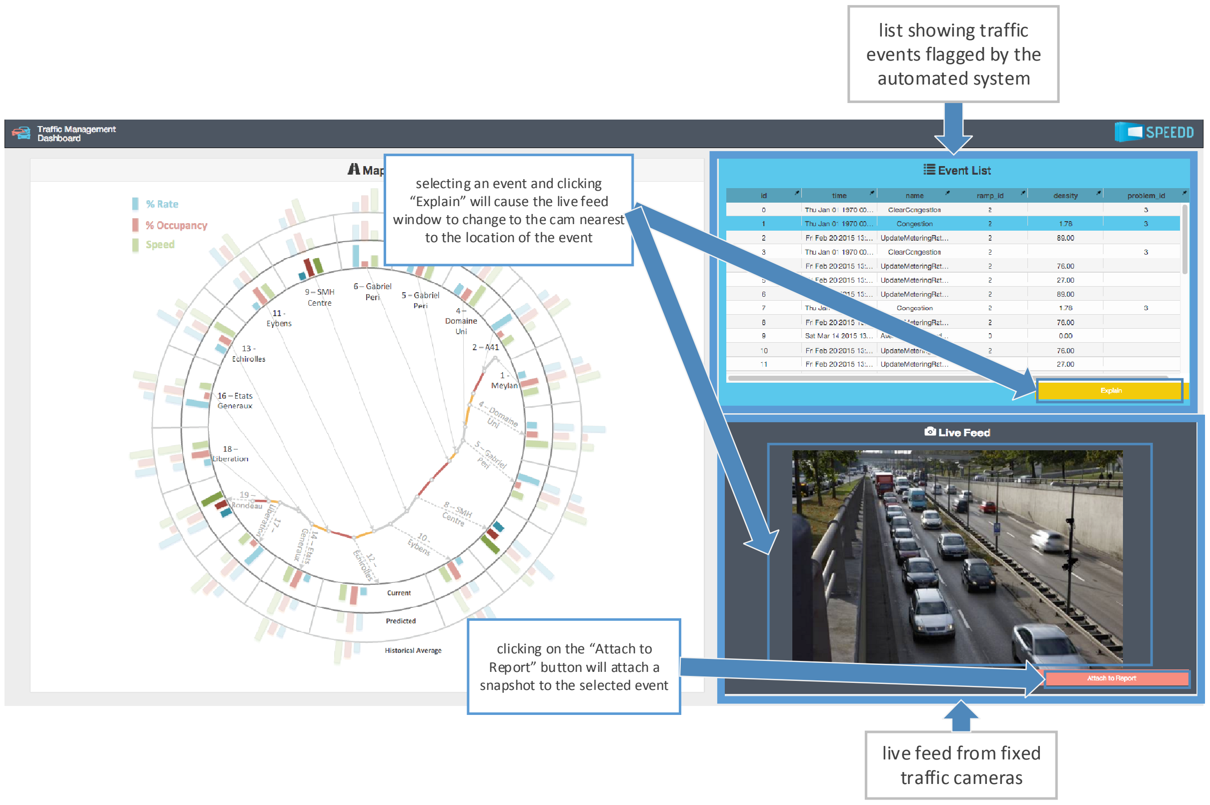

schematic map of the ring road showing current state
data regarding ramps that (require attention) have an event in the event list associated with them will show with a stronger colour
Traffic Management Dashboard
SPEEDD
Map
% Rate
% Occupancy
Speed
shows data at ramp 18 (Liberation)
- blue bar shows duty cycle of the traffic light
- red bar shows % occupancy of the ramp
- green bar average speed as a percentage of the speed limit
Current
Predicted
Historical Average
Live Feed
Attach to Report
The three concentric circles show current (1), predicted (2) and historical average (3) data at the specific location
Segment colours change depending on the density of the portion of the road
- red -> high
- yellow -> medium
- green -> low

Event List

id	time	name	ramp_id	density	problem_id
0	Thu Jan 01 1970 00...	ClearCongestion	2		3
1	Thu Jan 01 1970 00...	Congestion	2	1.78	3
2	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	89.00	
3	Thu Jan 01 1970 00...	ClearCongestion	2		3
4	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	76.00	
5	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	27.00	
6	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	89.00	
7	Thu Jan 01 1970 00...	Congestion	2	1.78	3
8	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	76.00	
9	Sat Mar 14 2015 13:...	AverageDensityAnc...	0	0.00	
10	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	76.00	
11	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	27.00	

Explain

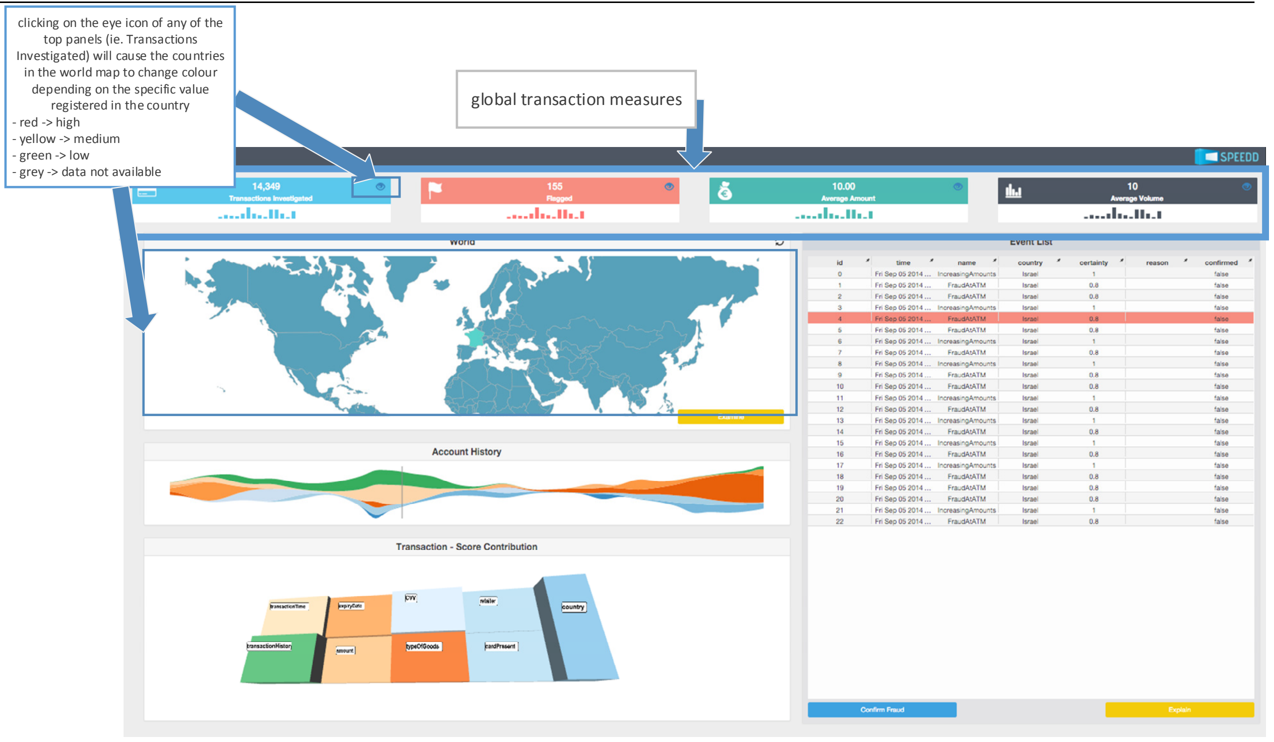
Figure 8.1 - Traffic Management Dashboard – Reference (1/2)


list showing traffic events flagged by the automated system
Traffic Management Dashboard
SPEEDD
Map
% Rate
% Occupancy
Speed
Current
Predicted
Historical Average
selecting an event and clicking "Explain" will cause the live feed window to change to the cam nearest to the location of the event
Event List
Explain
Live Feed
Attach to Report
clicking on the "Attach to Report" button will attach a snapshot to the selected event
live feed from fixed traffic cameras

id	time	name	ramp_id	density	problem_id
0	Thu Jan 01 1970 00...	ClearCongestion	2		3
1	Thu Jan 01 1970 00...	Congestion	2	1.78	3
2	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	89.00	
3	Thu Jan 01 1970 00...	ClearCongestion	2		3
[illegible]	Fri Feb 20 2015 13:...	UpdateMeteringRat...	2	76.00	
5	[illegible] Feb 20 2015 13:...	UpdateMeteringRat...	2	27.00	
6	[illegible] 2015 13:...	UpdateMeteringRat...	2	89.00	
7	Thu Jan [illegible]	Congestion	2	1.78	3
8	Fri Feb 20 2015 [illegible]	[illegible]eteringRat...	2	76.00	
9	Sat Mar 14 2015 13...	Ave[illegible]	0	0.00	
10	Fri Feb 20 2015 13:...	UpdateMeteri[illegible]	2	76.00	
11	Fri Feb 20 2015 13:...	UpdateMeteringRat...	[illegible]	27.00	

Figure 8.2 - Traffic Management Dashboard – Reference (2/2)

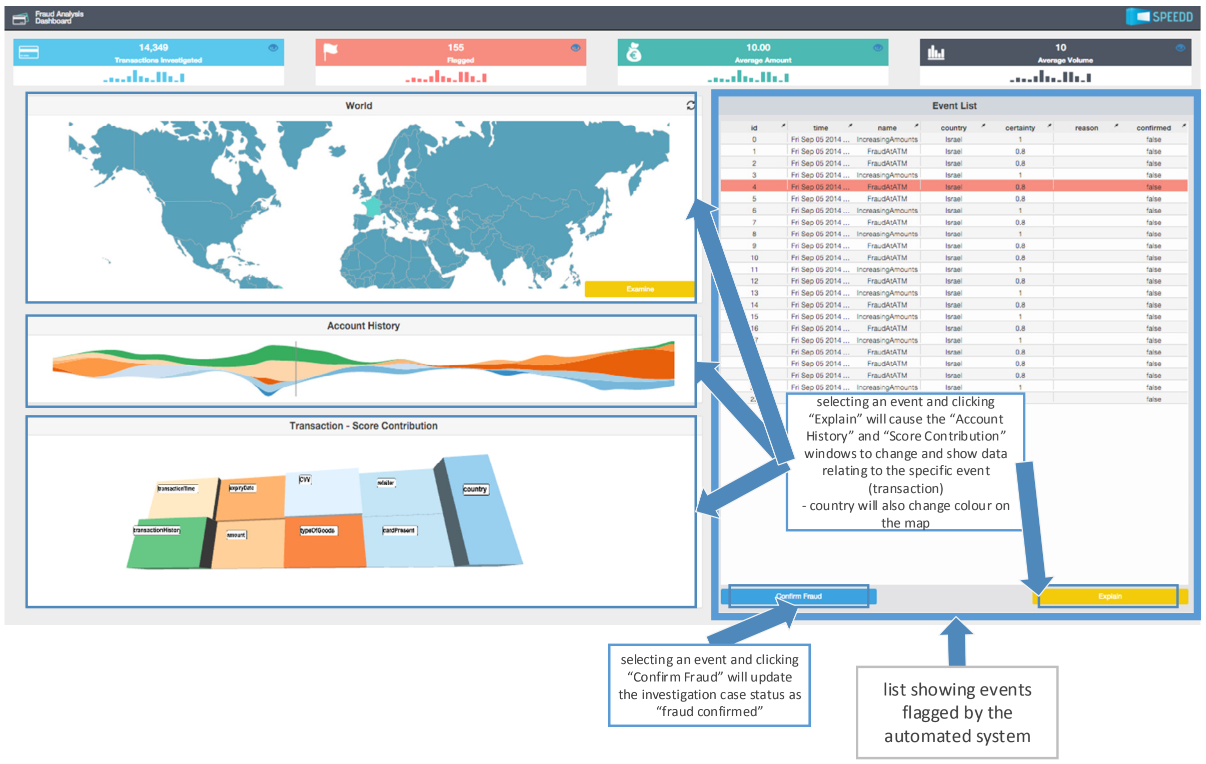
9 Appendix – User Reference: Credit Card Fraud Management Dashboard


clicking on the eye icon of any of the top panels (ie. Transactions Investigated) will cause the countries in the world map to change colour depending on the specific value registered in the country
- red -> high
- yellow -> medium
- green -> low
- grey -> data not available

global transaction measures

id	time	name	country	certainty	reason	confirmed
0	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
1	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
2	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
3	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
4	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
5	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
6	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
7	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
8	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
9	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
10	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
11	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
12	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
13	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
14	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
15	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
16	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
17	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
18	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
19	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
20	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
21	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
22	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false

Figure 9.1 - Credit Card Fraud Management Dashboard – Reference (1/3)



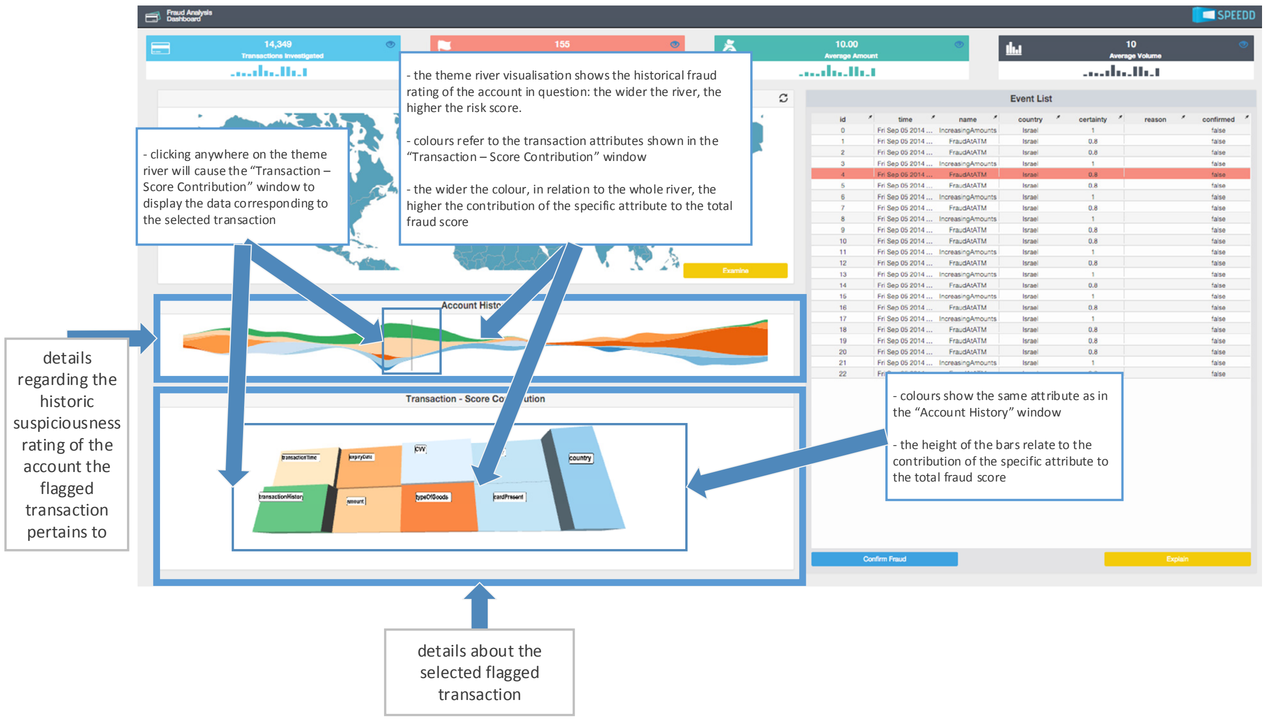
id	time	name	country	certainty	reason	confirmed
0	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
1	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
2	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
3	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
4	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
5	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
6	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
7	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
8	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
9	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
10	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
11	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
12	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
13	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
14	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
15	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
16	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
[illegible]	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
[illegible]	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
[illegible]	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
[illegible]	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
[illegible]	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
[illegible]						false

selecting an event and clicking "Explain" will cause the "Account History" and "Score Contribution" windows to change and show data relating to the specific event (transaction)
- country will also change colour on the map

selecting an event and clicking "Confirm Fraud" will update the investigation case status as "fraud confirmed"

list showing events flagged by the automated system

Figure 9.2 - Credit Card Management Dashboard – Reference (2/3)



id	time	name	country	certainty	reason	confirmed
0	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
1	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
2	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
3	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
4	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
5	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
6	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
7	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
8	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
9	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
10	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
11	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
12	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
13	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
14	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
15	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
16	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
17	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
18	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
19	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
20	Fri Sep 05 2014 ...	FraudAtATM	Israel	0.8		false
21	Fri Sep 05 2014 ...	IncreasingAmounts	Israel	1		false
22	Fri					false

- clicking anywhere on the theme river will cause the "Transaction – Score Contribution" window to display the data corresponding to the selected transaction

- the theme river visualisation shows the historical fraud rating of the account in question: the wider the river, the higher the risk score.

- colours refer to the transaction attributes shown in the "Transaction – Score Contribution" window

- the wider the colour, in relation to the whole river, the higher the contribution of the specific attribute to the total fraud score

details regarding the historic suspiciousness rating of the account the flagged transaction pertains to

- colours show the same attribute as in the "Account History" window

- the height of the bars relate to the contribution of the specific attribute to the total fraud score

details about the selected flagged transaction

Figure 9.3 – Credit Card Management Dashboard – Reference (3/3)

10 Appendix – Demonstrating SPEEDD Prototype

10.1 Demo Scenarios

For demonstration purposes, demo scenarios were prepared for the traffic management and for the credit card fraud detection use cases. The scenarios are available in a form of storyboards here: http://www.speedd-project.eu/sites/default/files/credit_card_fraud_demo_v2.pdf, http://www.speedd-project.eu/sites/default/files/traffic_management_demo.pdf.

10.2 Running demo scenarios

In the following sections we assume that you run the demo in the virtual machine created from the provided image. In case you run your own environment, the paths have to be adjusted.

10.2.1 Traffic Management

1. cd ~/speedd/speedd-runtime/scripts/traffic
2. sudo ./start-speedd-runtime
3. start sending input events by running ./playevents-traffic
4. open a new terminal shell
5. cd ~/speedd/speedd-ui/bin
6. Run the dashboard by starting ./run.sh
7. Open the following URL in your browser to work with the dashboard UI: http://localhost:3000.

10.2.2 Credit Card Fraud Detection

1. cd ~/speedd/speedd-runtime/scripts/ccf
2. sudo ./start-speedd-runtime
3. start sending input events by running ./playevents-fraud
4. open a new terminal shell
5. cd ~/speedd/speedd_ui_bf/bin
6. Run the dashboard by starting ./run.sh
7. Open the following URL in your browser to work with the dashboard UI: http://localhost:3000.

10.2.3 Stopping SPEEDD prototype

1. cd ~/speedd/speedd-runtime/scripts
2. Kill the topology by running 'sudo ./kill-speedd-runtime'
3. Stop the UI by killing the process (or Ctrl-C in the terminal shell where the UI process has been started)

Note: It is important to stop the prototype running current demo scenario before running another demo scenario (for example, if the traffic management demo is running, it is important to stop it before running the credit card fraud demo)[37].

[37] This is a temporal limitation caused by inability to run two topologies with same name on a single STORM cluster. This issue will be fixed in the future versions of SPEEDD software.

11 Appendix – Aimsun Integration

11.1 Aimsun Integration Architecture

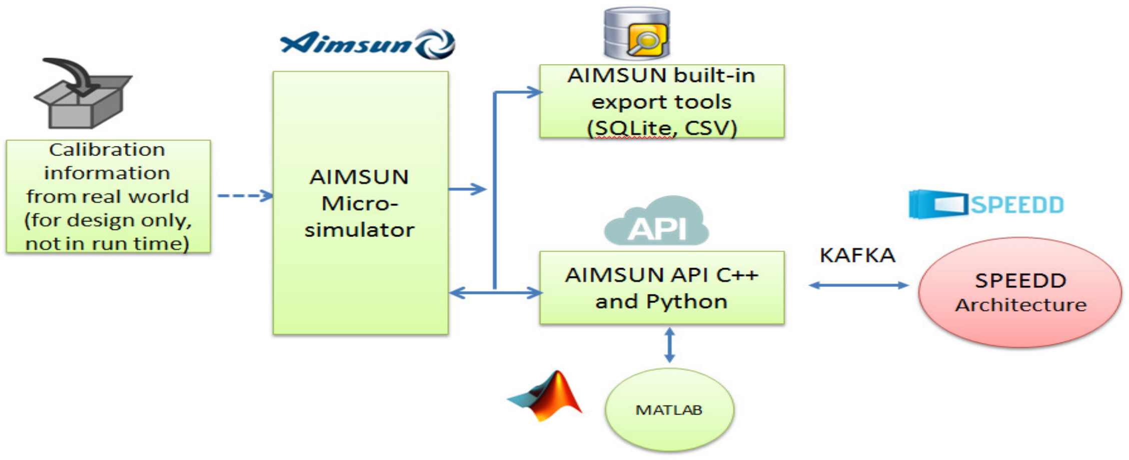

Aimsun
Calibration information from real world (for design only, not in run time)
AIMSUN Micro-simulator
AIMSUN built-in export tools (SQLite, CSV)
API
AIMSUN API C++ and Python
KAFKA
SPEEDD
SPEEDD Architecture
MATLAB

Figure 11.1 - AIMSUN Integration Architecture

AIMSUN integration comes to enable implementing closed traffic control loop where SPEEDD runtime receives simulated sensor readings in real time and applies control actions to prevent or mitigate undesired traffic conditions. Another aspect of the integration is the ability to control the simulator remotely which is important given the distributed nature of the SPEEDD project team.

Because the AIMSUN product is targeted at the single user scenario running on a Windows desktop machine, integration with a distributed system has proved challenging, and has been only partially accomplished. Some of the commands can only be executed locally on the AIMSUN machine.

The diagram in Figure 11.1 illustrates the integration architecture between the Aimsun simulator and the SPEEDD runtime. The configuration and the calibration of the simulation are done as part of the simulation design process and are not available at runtime. The integration uses AIMSUN API for C++ and Python. Following the SPEEDD event-driven architecture, the interaction between AIMSUN and SPEEDD runtime is event-based, using SPEEDD kafka-based event bus.

Simulated traffic sensor readings are sent to speedd-traffic-in-events topic as input events. Actions emitted by the Decision Making module are posted to the speedd-traffic-actions topic where they are consumed by the AIMSUN integration module, and applied to adjust simulation parameters thus mimicking the traffic control action.

Currently the following two traffic control actions implemented:

1. Control the traffic light phases

2. Control the Speed limit of each section

In addition to the traffic control actions, it is possible to start the pre-configured simulation remotely. Stopping or pausing of the simulation is not available in the remote mode yet.

The following lists the parameters for each AIMSUN command.

11.1.1 Control traffic light phases

1. Junction ID (Intersection ID)
2. Phase ID
3. New Phase time (seconds)

11.1.2 Control section speed limit

1. Section ID
2. New Speed limit (km/h)

12 Appendix – SPEEDD Event Reference

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

12.1 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

12.2 Credit Card Fraud Management Use Case

12.2.1 Transaction

Attribute Name	Attribute Type	Description
card_pan	String	Hashed card PAN
terminal_id	String	Unique terminal ID
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

12.2.2 SuddenCardUseNearExpirationDate

Attribute Name	Attribute Type	Description
card_pan	String	Hashed card PAN
TransactionsCount	Integer	Number of transactions in the pattern
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

12.2.3 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

12.2.4 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

12.3 Traffic Management Use Case

12.3.1 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

12.3.2 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

12.3.3 Congestion

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

12.3.4 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

12.3.5 ClearCongestion

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
problem_id	String	Identifies the problem detected by SPEEDD

12.3.6 OnRampFlow

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

12.3.7 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

12.3.8 2minsAverageDensityAndSpeedPerLocation

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

12.3.9 PredictedTrend

Attribute Name	Attribute Type	Description
location	String	Id of the sensor location (collection point)
problem_id	Double	Identifies the problem detected by SPEEDD

12.4 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.

12.4.1 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

12.4.2 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

12.5 AIMSUN Simulation Control Commands

12.5.1 setTrafficLightPhaseTime

Attribute Name	Attribute Type	Description

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

12.5.2 setSpeeddLimit

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

13 Bibliography

Artikis, Alexander, Marek Sergot, and Georgios Paliouras. "An Event Calculus for Event Recognition." *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2014.

Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer. "D³ data-driven documents." *IEEE Transactions on Visualization and Computer Graphics, vol. 17. no. 12*, 2011: 2301-2309.

Engel, Yagil, and Opher Etzion. "Towards proactive event-driven computing." *In Proceedings of the 5th ACM international conference on Distributed event-based system (DEBS '11).* New York, NY, USA: ACM, 2011. 125-136.

Etzion, O. "Towards an Event-Driven Architecture: An Infrastructure for Event Processing." *RuleML.* 2005.

FIWARE Project. *NGSI-9/NGSI-10 information model.* 5 12, 2014. https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/NGSI-9/NGSI-10_information_model (accessed 01 20, 2016).

Hewitt, Carl, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." *In Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI '73).* Stanford, CA, USA: AAAI Press, 1973. 235-245.

Hunt, Patrick, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." *USENIX Annual Technical Conference (USENIX ATC '10).* Boston, MA, USA, 2010. 145-158.

IBM Research. *IBM Proactive Technology Online User Guide.* Haifa, 2015.

Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." *In Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB).* Athens, Greece, 2011.

Oaks, Scott. *Java Performance: The Definitive Guide.* O'Reilly Media, Inc.,, 2014.

Tilkov, Stefan, and Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs." *IEEE Internet Computing, vol. 14, no. 6*, 2010: 80-83.

Toshniwal, Ankit, et al. "Storm@twitter." *In Proceedings of SIGMOD/PODS'14 International Conference on Management of Data.* Snowbird, UT, USA: ACM, 2014. 147-156.

Vinoski, Steve. "Advanced Message Queuing Protocol." *IEEE Internet Computing*, 2006: 87-89.

Zaharia, Matei, et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." *In Proceedings of 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12).* San Jose, CA, USA: ACM, 2012. 15-28.

Zaharia, Matei, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: fault-tolerant streaming computation at scale." *In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13).* Farmington, PA, USA: ACM, 2013. 423-438.