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

Project **FP6-619435/SPEEDD**
Deliverable **D6.1**
Distribution **Public**

http://speedd-project.eu

The Architecture Design of the SPEEDD Prototype

Alex Kofman (IBM), Fabiana Fournier (IBM), Inna Skarbovsky (IBM), Natan Morar (UoB), Marius Schmitt (ETH), Anastasios Skarlatidis (NCSR)

Status: Final (Version 1.0)

October 2014

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	Aleksandra Wesolowska

Deliverable

Deliverable type	Report
Distribution level	Public
Deliverable Number	D6.1
Deliverable Title	The Architecture Design of the SPEEDD Prototype
Contractual date of delivery	M9 (October 2014)
Actual date of delivery	October 2014
Relevant Task(s)	WP6/Tasks 6.3
Partner Responsible	IBM
Other contributors	NCSR, CNRS, FeedZai, ETH, UoB, Technion
Number of pages	42
Author(s)	Alex Kofman (IBM), Fabiana Fournier (IBM), Inna Skarbovsky (IBM), Natan Morar (UoB), Marius Schmitt (ETH), Anastasios Skarlatidis (NCSR)
Internal Reviewers	FeedZai
Status & version	Draft
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 prototypes for integration, extensibility, and automation. Finally, main test cases are discussed for testing the prototype.

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. Main APIs have been defined and data formats discussed.

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

Contents

1 Introduction 7
1.1 History of the document 7
1.2 Purpose and Scope of the Document 7
1.3 Relationship with Other Documents 7
2 Main Sections 8
2.1 System Requirements 8
2.2 Approach 8
2.3 Conceptual Architecture 9
2.4 SPEEDD Runtime Architecture 11
2.4.1 Event Bus 13
2.4.2 Event/Data Providers 15
2.4.3 Action Consumption – Actuators/Connectors 16
2.4.4 Complex Event Processor 17
2.4.5 Decision Making 22
2.4.6 Dashboard application 24
2.5 Design-Time Architecture 26
2.5.1 Event Pattern Mining 27
2.5.2 Authoring of CEP Rules 27
2.6 Integration – APIs and Data Formats 28
2.7 Deployment Architecture 29
2.8 Non-Functional Aspects 30
2.8.1 Scalability 30
2.8.2 Fault Tolerance 30
2.8.3 Testability 30
3 Test Plan 32
4 Conclusions 33
5 Appendix – Technology Evaluation 34
5.1 Stream Processing – requirements and evaluation criteria 34
5.2 Storm 35
5.3 Akka 37

5.4 Spark Streaming 39
5.5 Streaming technologies – conclusion 40
5.6 Choice of the Messaging Platform 41

List of Tables

Table 2.1 - Kafka topics in SPEEDD event bus 13

List of Figures

Figure 2.1- SPEEDD design architecture approach 9
Figure 2.2 - Conceptual Architecture of SPEEDD Prototype 10
Figure 2.3 - SPEEDD - Event-Driven Architecture 11
Figure 2.4 - SPEEDD Runtime - Event-Driven Architecture (Traffic Use Case) 12
Figure 2.5 - SPEEDD Runtime - Event-Driven Architecture (Credit Card Fraud Use Case) 13
Figure 2.6 - Storm-Kafka Integration 15
Figure 2.7 - Proton Authoring Tool and Runtime Engine 17
Figure 2.8 - Proton Runtime and external systems 18
Figure 2.9 - Proton components 20
Figure 2.10 - Architecture of Proton on STORM 21
Figure 2.11 - SPEEDD Traffic Use Case, physical system 23
Figure 2.12 - SPEEDD Traffic Use Case, implementation at runtime 23
Figure 2.13 - Dashboard Architecture 24
Figure 2.14 - User Interface 26
Figure 2.15 - SPEEDD Design Time Architecture 27
Figure 2.16 - Deployment Architecture 29
Figure 5.1 - Storm Topology 36
Figure 5.2 - Storm Parallelization 36
Figure 5.3 - Akka Actors Hierarchy 38
Figure 5.4 - D-Stream is a major concept in Spark Streaming 39

1 Introduction

1.1 History of the document

Version	Date	Author	Change Description
0.1	15/10/2014	Alexander Kofman (IBM)	First draft
0.2	26/10/2014	Alexander Kofman (IBM)	Updates per internal review
1.0	30/10/2014	Alexander Kofman (IBM)	Final fixes and cleanup, sections on distributed counter removed as irrelevant

1.2 Purpose and Scope of the Document

This is 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 requirements of the two representative use cases. The document describes APIs provided by the prototypes for integration, extensibility, and automation. Finally, main test cases are discussed for testing the prototype.

It is important to mention that we anticipate some changes to the architecture during the project development, based on the lessons to be learned in the process. The amended version of this document describing the resulting design will be submitted on month 24.

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.

2 Main Sections

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 the 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 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).

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

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, as illustrated in Figure 2.1:

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.

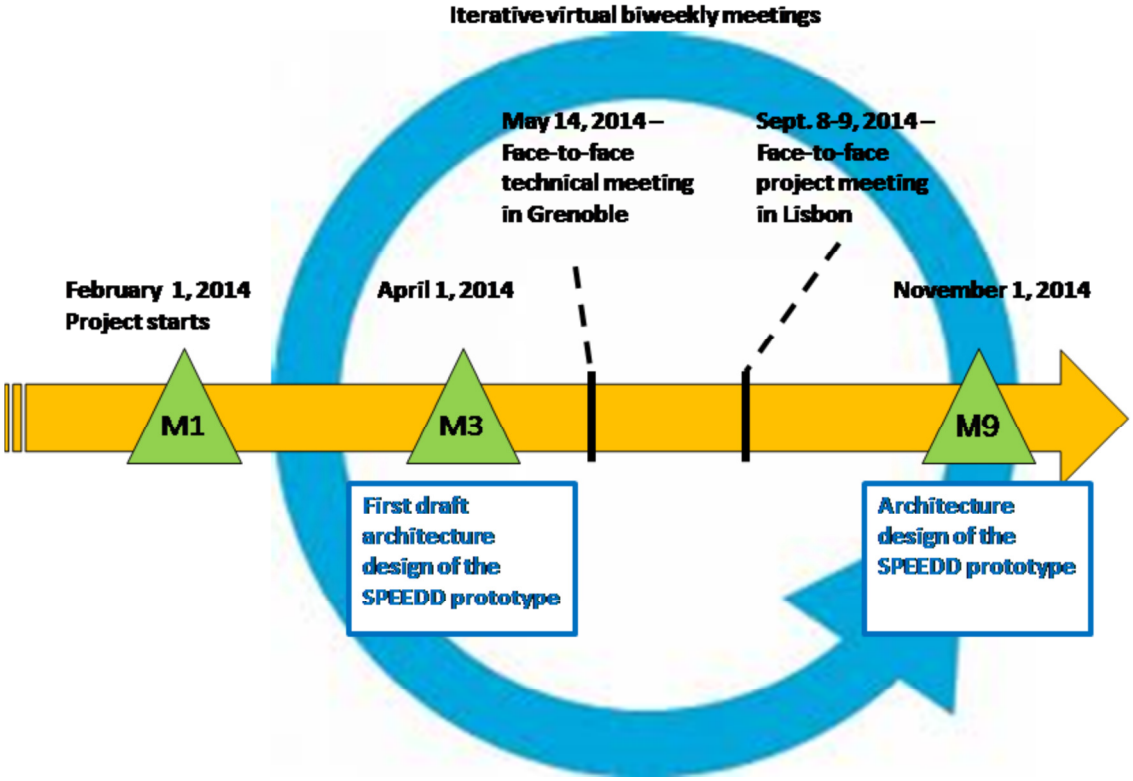

Figure 2.1- SPEEDD design architecture approach

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 2.2 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.

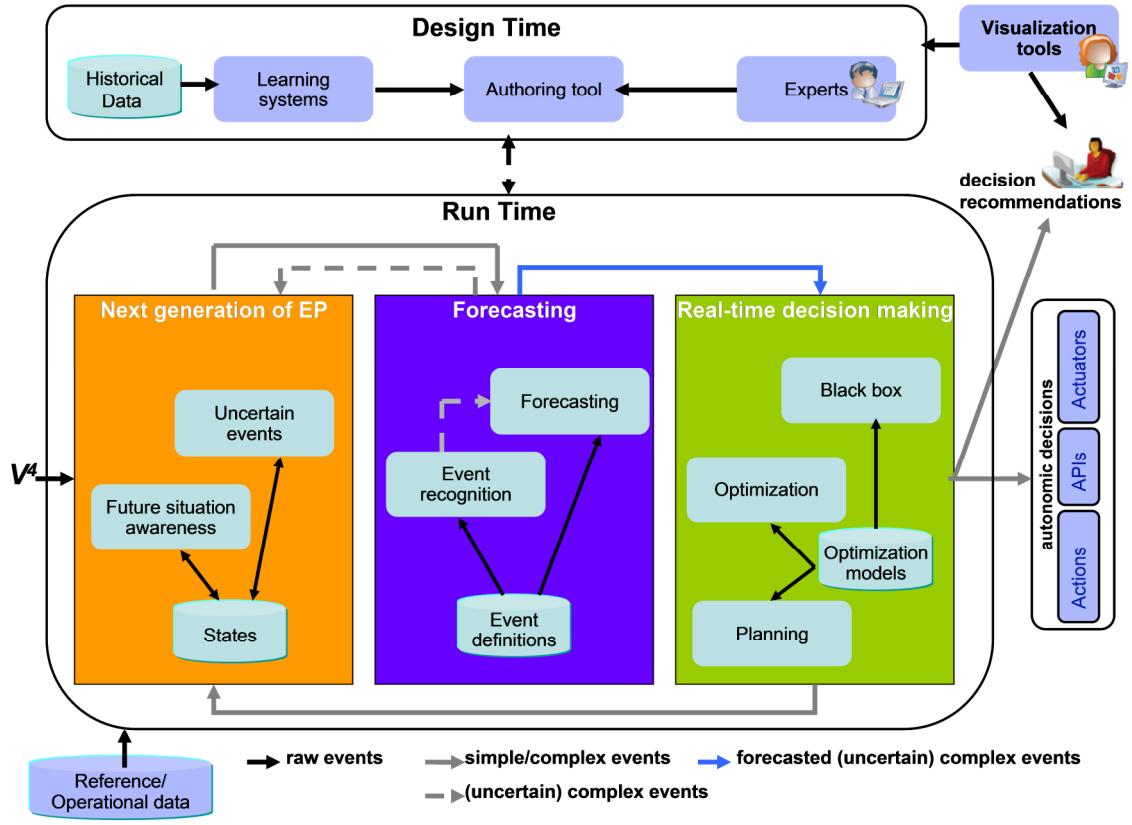

Figure 2.2 - Conceptual Architecture of SPEEDD Prototype

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 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 is capable of 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[1]. 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[2]. 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 2.3 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.

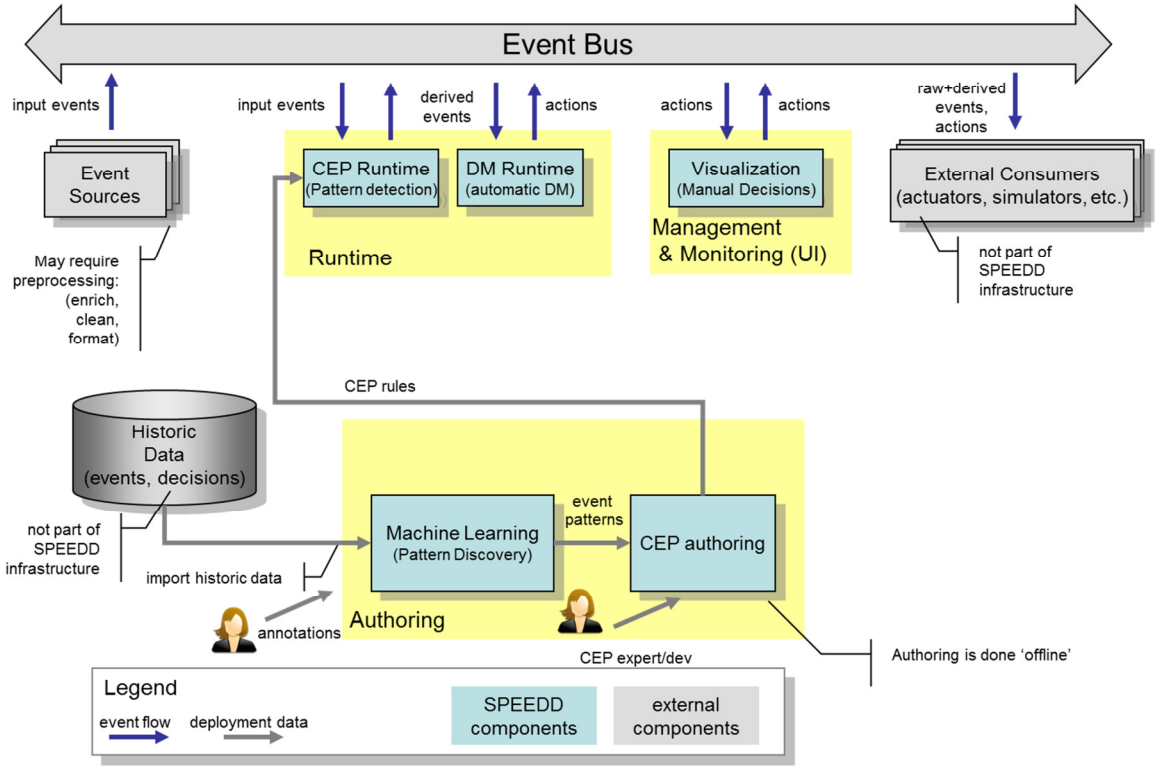

Figure 2.3 - SPEEDD - Event-Driven Architecture

[1] 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.

[2] G. Hohpe. Programming without a call stack – Event-driven Architecture. 2006, [Online]. At: http://www.eaipatterns.com/docs/EDA.pdf

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 on 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 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 are describe the details of the runtime architecture discussing design of each component and explain how the technology is being used to implement it.

Figure 2.4 and Figure 2.5 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.

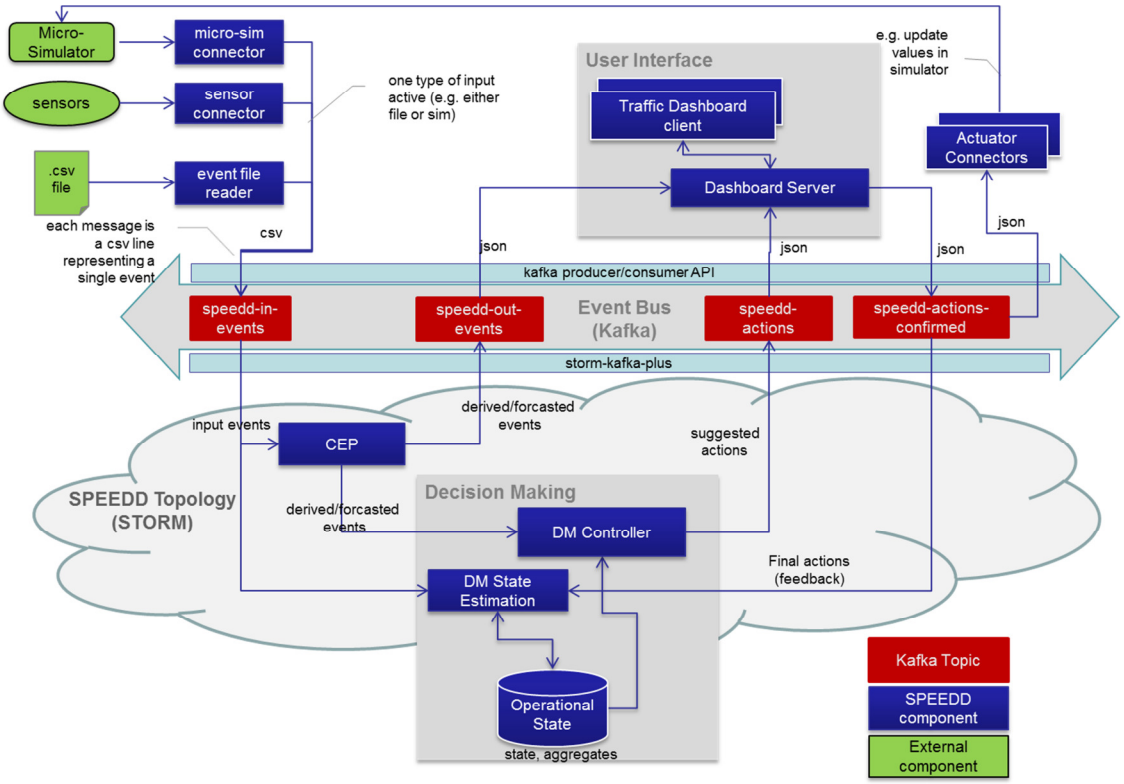

Figure 2.4 - SPEEDD Runtime - Event-Driven Architecture (Traffic Use Case)

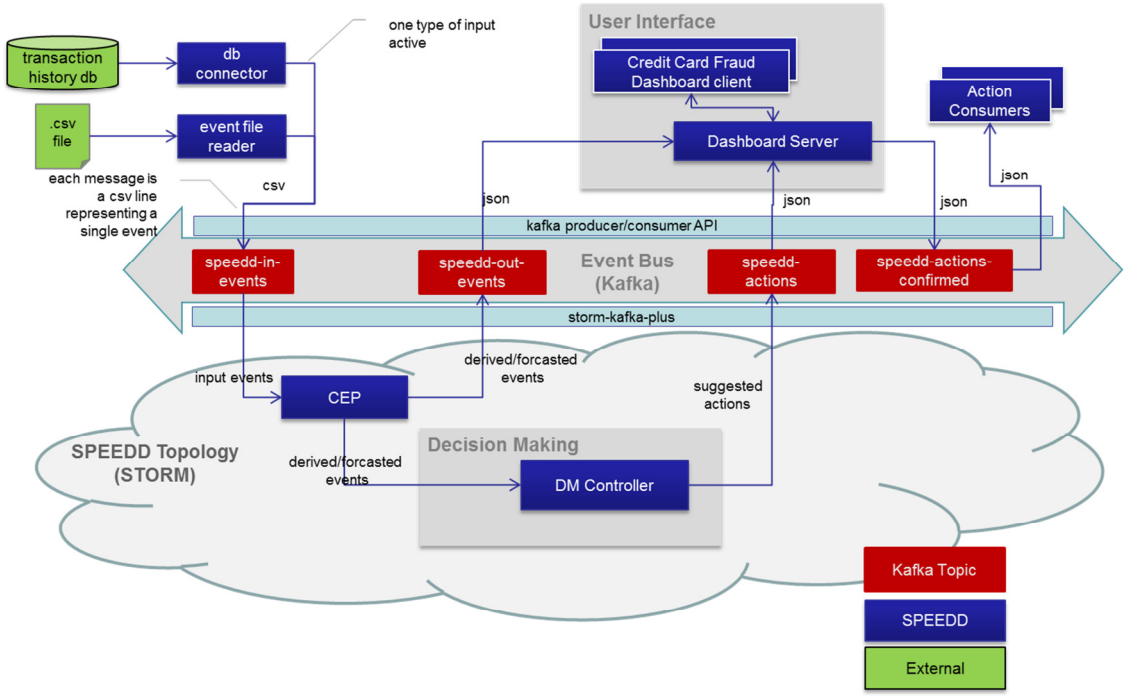

Figure 2.5 - SPEEDD Runtime - Event-Driven Architecture (Credit Card Fraud Use Case)

2.4.1 Event Bus

The technology chosen for the event bus component is Apache Kafka[3]. It provides a scalable, performant, and robust messaging platform that matches SPEEDD requirements (see 5.5 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, Dashboard
speedd-actions	Suggested decisions	Decision making	Dashboard
speedd-actions-confirmed	Actions confirmed for execution	Dashboard (in open loop mode), Decision making (in closed loop mode)	Dashboard, Actuators

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

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 on 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.

2.4.1.2 Partitioning for the Credit Card Fraud Use Case

For the credit card fraud use case, the card 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

Integration between Storm streaming platform and our Kafka-based event bus is done based on the Storm-Kafka-Plus project[4]. Storm-Kafka-Plus 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 2.6 illustrates the way this integration is done in SPEEDD. Raw events posted on the *speedd-in-events* topic in 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

[4] https://github.com/wurstmeister/storm-kafka-0.8-plus

serialized as JSON text-based messages using *JsonEncoder* class configured via serializer.class parameter of the KafkaBolt.

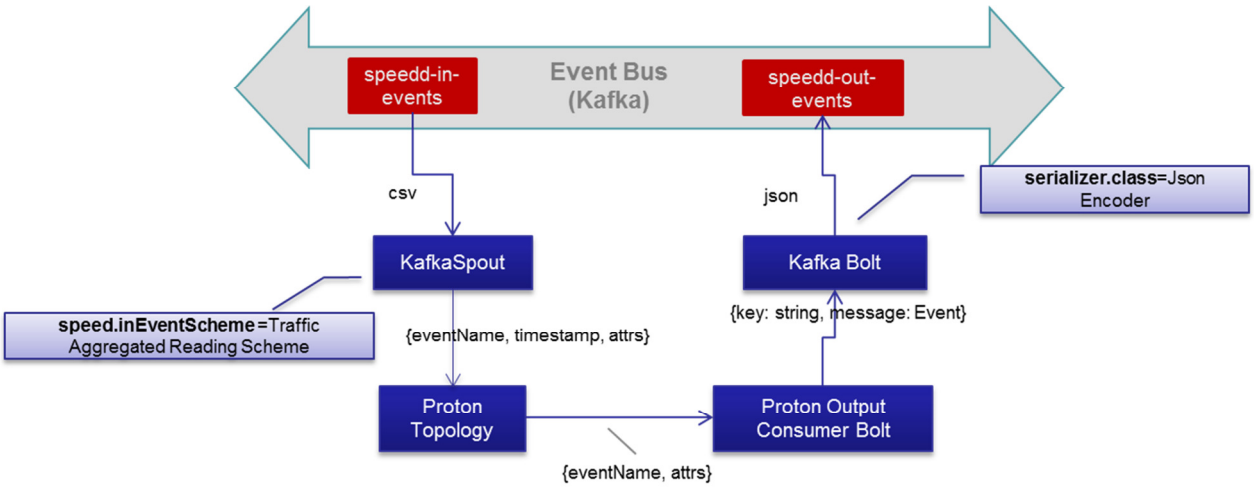

Figure 2.6 - Storm-Kafka Integration

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 2.4, 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

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[5]
- Micro-simulator connector
- File reader connector – replay past events from a file

[5] 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

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 2.5):

- 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 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 will be implemented and tested using the AIMSUN micro-simulator developed as part of WP8. The traffic actuator connector will listen on the outbound action events (speedd-actions-confirmed topic on the event bus) and execute operations supported by the micro-simulator, e.g. update speed limits, set ramp metering rates, etc. 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

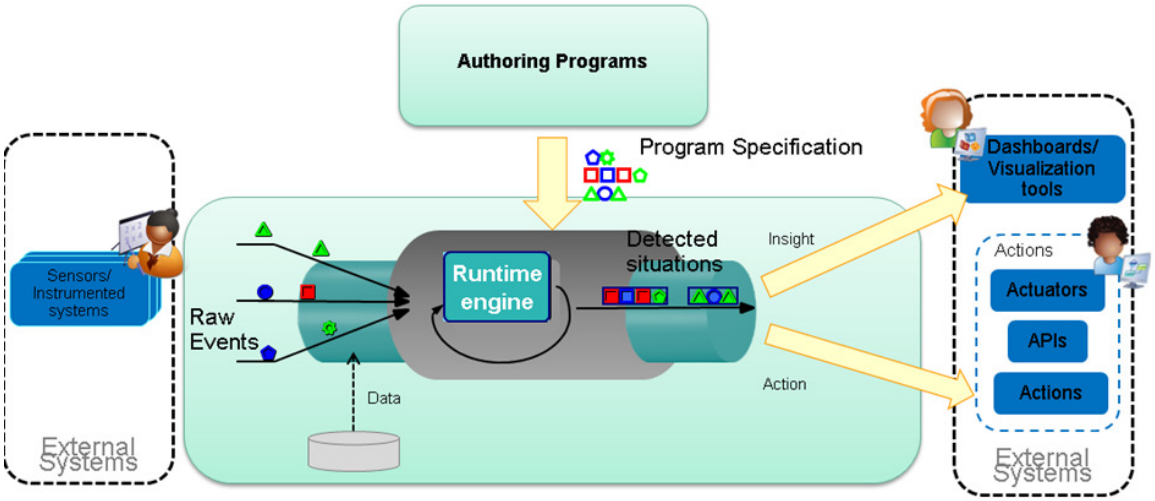

Figure 2.7 - Proton Authoring Tool and Runtime Engine

Proton—IBM Proactive Technology On-Line—is a scalable integrated platform to support the development, deployment, and maintenance of proactive event-driven applications. **Proactive event-driven computing** 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 (see Figure 2.7).

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

- 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 High-level Architecture

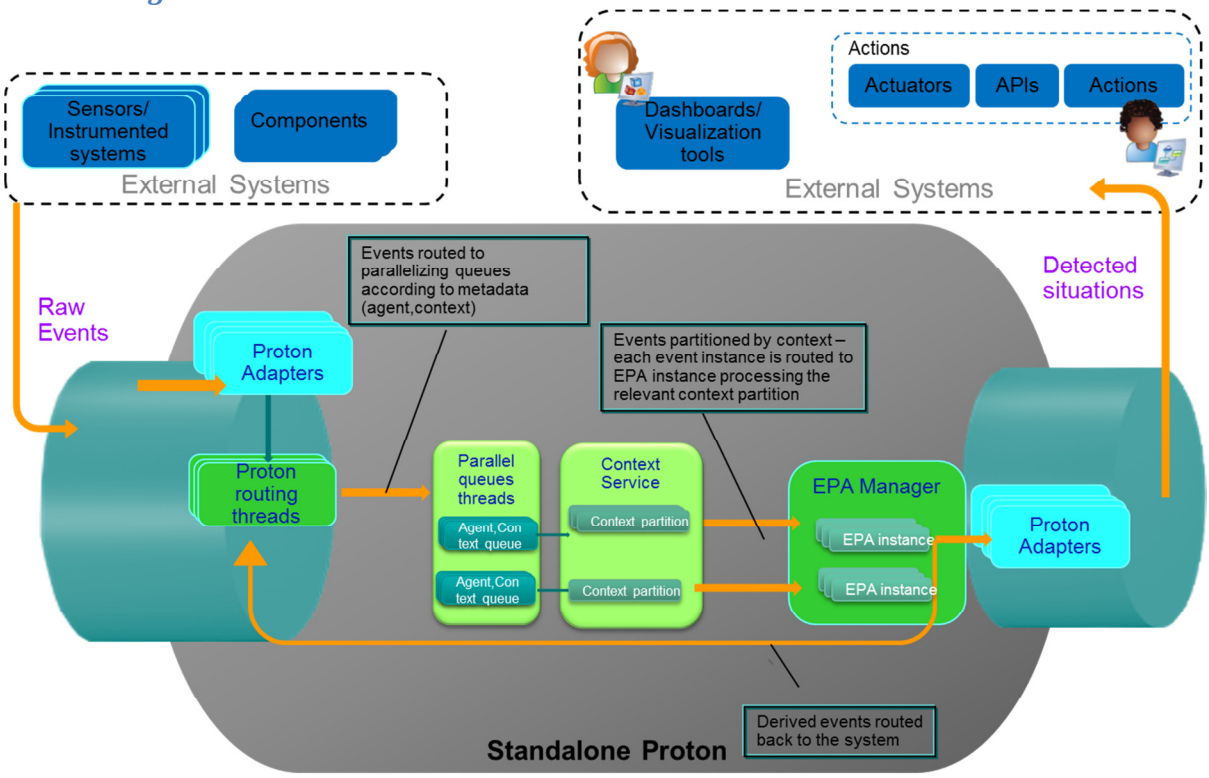

Figure 2.8 - Proton Runtime and external systems

Proton architecture consists of a number of functional components and interaction among them, the main of which are (see Figure 2.8):

- 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.

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
    c. Solve correctness problems
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

2.4.4.4 Proton component architecture

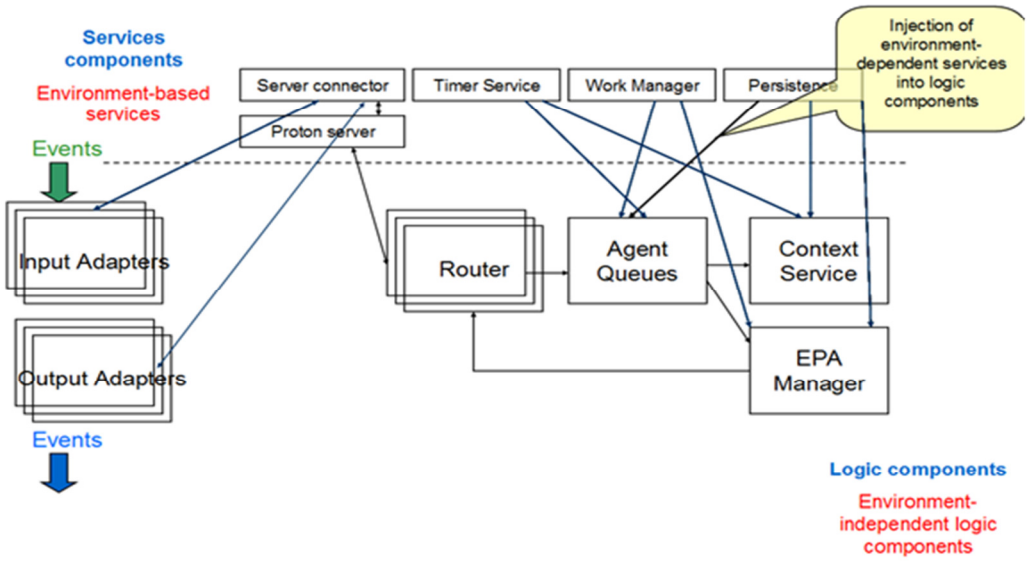

Figure 2.9 - Proton components

Proton's logical components are illustrated in Figure 2.9. 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.

2.4.4.5 Distributed Architecture on top of STORM

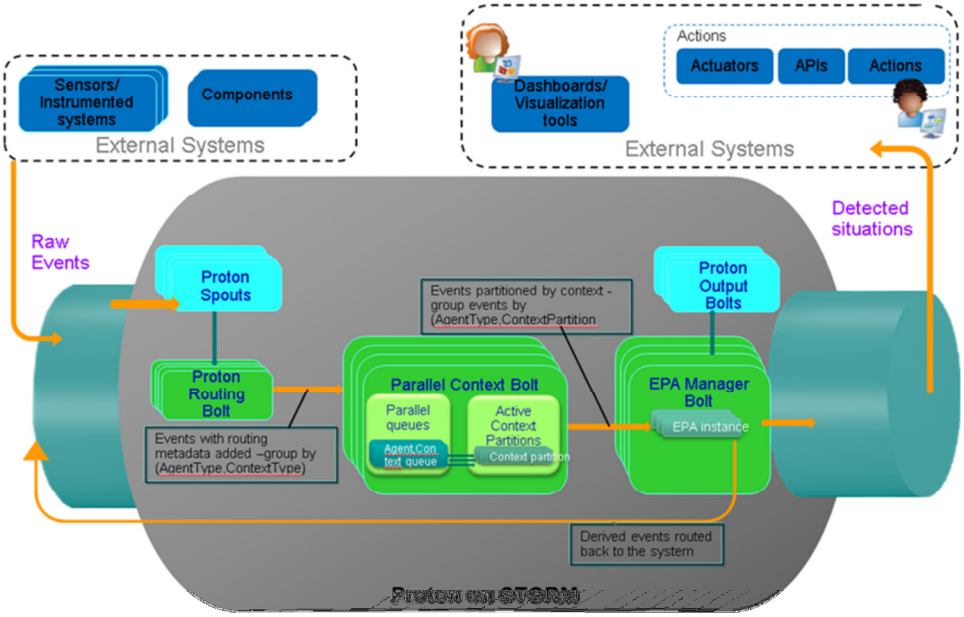

Figure 2.10 - Architecture of Proton on STORM

The Proton architecture on top of STORM[6] (see Figure 2.10) 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.

[6] Work on implementing Proton on Storm is being performed as part of FERARI project (http://www.ferari-project.eu), and will be available as open source (https://bitbucket.org/sbothe-iais/ferari). 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 aim of Decision Making is to provide a body of proactive event-driven decision-making tools, which exploit the detected or forecasted events of complex event processing. The Decision Making module receives as inputs the detected, derived and forecasted events and emits control actions or appropriate suggestions. Therefore, it functions both as an event consumer and as an event producer at the same time.

In this sense, decision making is the task of finding the optimal response to a specific situation, which is described by the detected or forecasted events. It is naturally represented as a parametric optimization problem. The main task of decision making is to solve this optimization problem, which can be accomplished in two conceptually different ways:

- The parametric optimization problem is solved offline such that an explicit solution is obtained. Note that this is a “difficult" task, since an optimal answer to any situation that might arise during operation needs to be computed. If such an explicit solution can be obtained, it takes the form of a feedback rule, e.g. a linear controller K(s) or state feedback -K*x. Therefore, it can be efficiently implemented in a unified architecture using the existing SPEEDD components (e.g. as a STORM Bolt).
- The construction of an explicit solution may be computationally intractable for certain problems. In such a case, the solution to multiple distinct instances of the optimization problem needs to be computed at runtime. In contrast to the first case, in which only the evaluation of a feedback rule is required, the algorithmic solution of an optimization problem is not trivial and it is not tractable to solve such a problem within the stream processing environment adopted in SPEEDD (STORM). We therefore assume the existence of a use-case specific “optimization black-box” outside the actual SPEEDD framework, which can be queried whenever such a decision is required.

In the following, we will briefly sketch the resulting DM architecture using the problem of freeway ramp-metering (regulating the traffic inflow on a freeway in order to maximize throughput) from the traffic use case as an example: A low-level ramp metering controller receives measurements of the local traffic density and the local traffic flows, as well as notifications about detected or predicted congestion queues. It will then emit a recommendation to change the ramp metering rates accordingly (Figure 2.11). For a network of interaction freeways, a network-wide planning algorithm can be used for coordination purposes, implemented as an external oracle that can be queried.

Since a road network is naturally a spatially distributed system, the architecture of the decision-making module reflects this structure. The Kalman filters and the local feedback controllers are explicit decision

rules which can be efficiently implemented as STORM bolts in a distributed manner (Figure 2.12). Preliminary theoretical results suggest that such controllers may perform asymptotically optimal with regard to flow maximization for a single freeway; however, coordination is required to achieve optimal operation of more complex road networks. Network-wide planning can be superimposed by querying an external black-box.

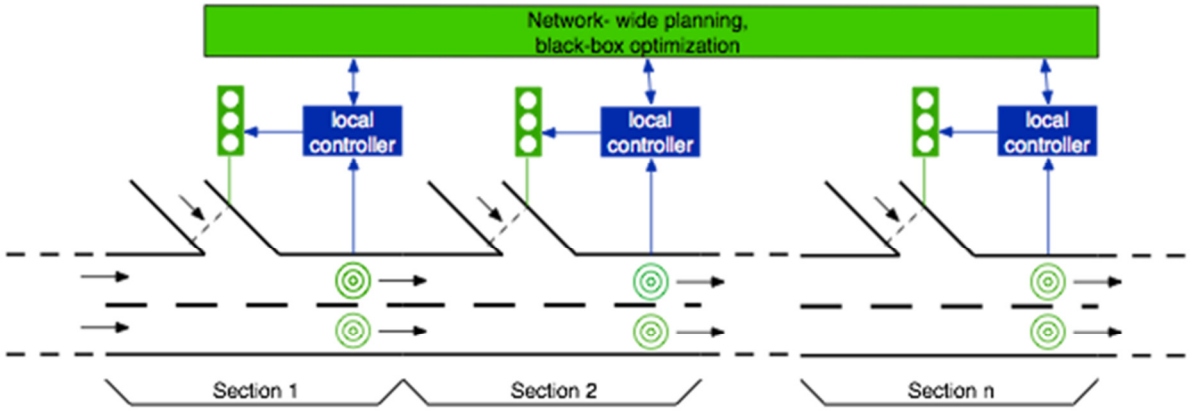

Figure 2.11 - SPEEDD Traffic Use Case, physical system

The decision making architecture for the credit-card fraud detection use case is conceptually identical, although it does not require separate state estimation. The credit card use case can naturally be described in form of distinct events.

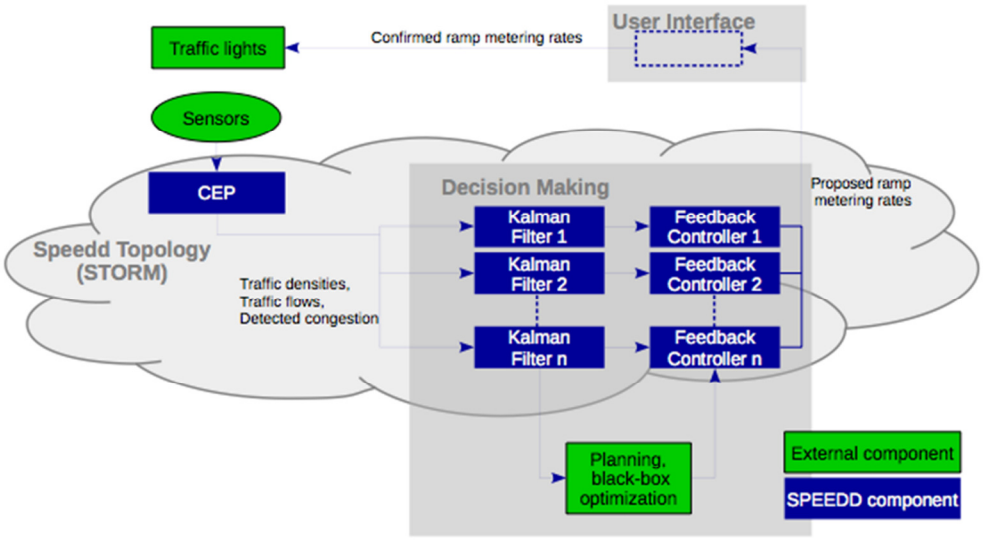

Figure 2.12 - SPEEDD Traffic Use Case, implementation at runtime

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 2.13.

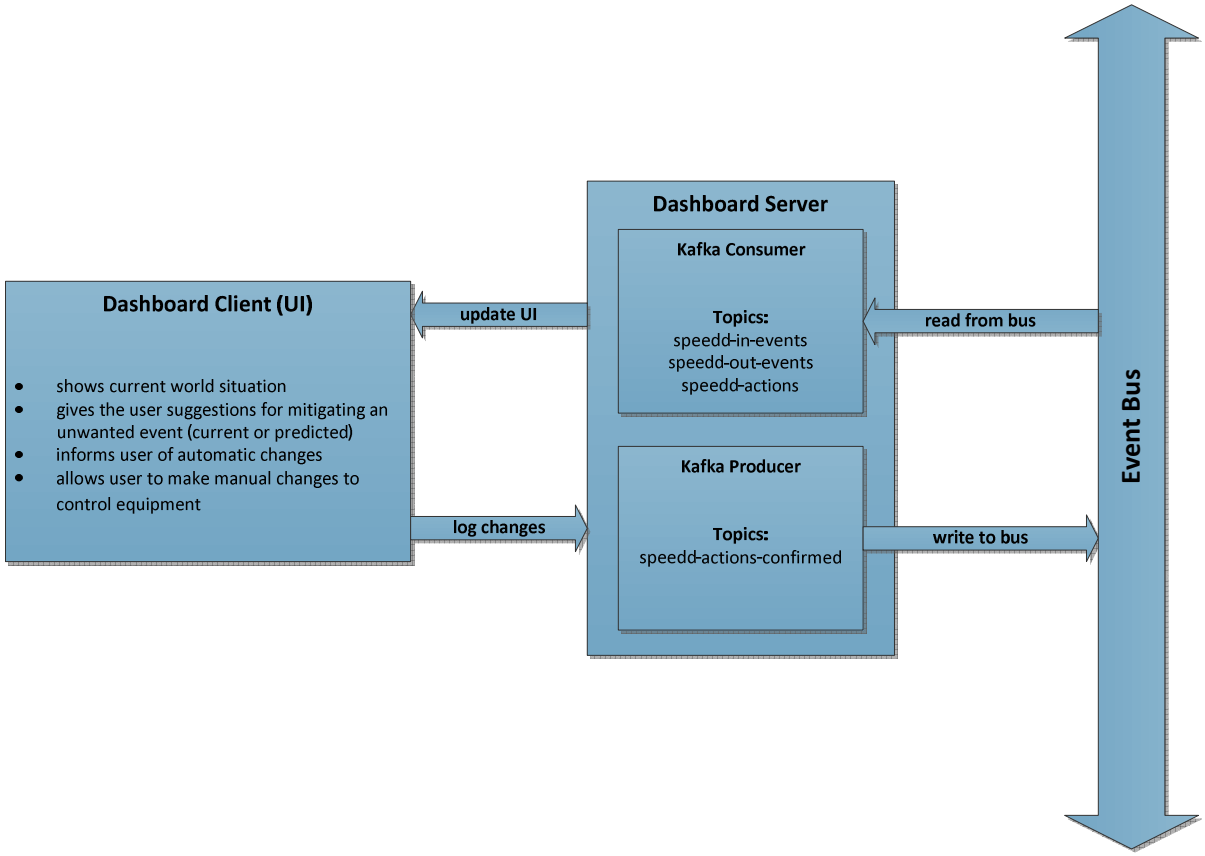

Figure 2.13 - Dashboard Architecture

The Dashboard Server is built using the Express[7] web application framework for Node.js[8]. 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

[7] 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

[8] http://nodejs.org – Node.js is asynchronous event driven framework for building fast, scalable network applications

producer are implemented using the npm (node package manager) module 'kafka-node', a Node.js client with Zookeeper[9] 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 in human readable form, current states of the control equipment available (e.g. speed limit signs, message signs, lanes, etc.), 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 2.14 illustrates a 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, by employing task decomposition and eye tracking techniques for determining the sources of information polled during a simulated exercise. For more details please refer to (Deliverable 5.3.1 Initial Report on User Interface Design for Traffic Management: using Cognitive Work Analysis to develop Ecological Interfaces for Traffic Management). The UI is generated using D3js[10], a Javascript library that enables the update of DOM elements as soon as corresponding data are received from the server. The server-client communication is implemented through SSE (server-sent events), standardized as part of HTML5 while confirmed user actions are sent to the server through an XMLhttpRequest.

The scope of the UI is detailed in deliverable 5.3.1. However, Figure 2.14 illustrates how the UI is partitioned into sections which relate to Understanding Traffic Behaviour (in the top left of the screen); Understanding Road User Behaviour (in the top right of the screen); Understanding the Current State of the World (in the centre and bottom right of the screen); Understanding Control Status (i.e., ramp metering on the bottom left of the screen); Understanding and Making Control Actions (in the bottom centre of the screen). The aim is to provide a display through which the Traffic Manager in not only able to monitor the current state of the world, traffic and road user behaviour, but also to understand which control actions are available (or plausible) at a given point in time. Further development will be directed towards ways in which we are able to manage, through the UI, the dialogue between Traffic Managers and the SPEEDD decision modules, e.g., in terms of agreeing with or disputing suggestions.

At the time of working on the current document the design work on the dashboard client for the credit card fraud management use case is still in progress. The overall architecture of the component should remain the same as described above. The specifics of the problem domain will be embodied in the design of the user interface and visualization techniques.

[9] https://zookeeper.apache.org – Zookeeper is an open-source distributed service for configuration, synchronization and naming registry for large distributed systems

[10] http://d3js.org – D3.js is a JavaScript library for manipulating DOM (Document Object Model) based on data

Figure 2.14 - User Interface

2.5 Design-Time Architecture

The conceptual view of the design time architecture for SPEEDD is presented in Figure 2.15. 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.

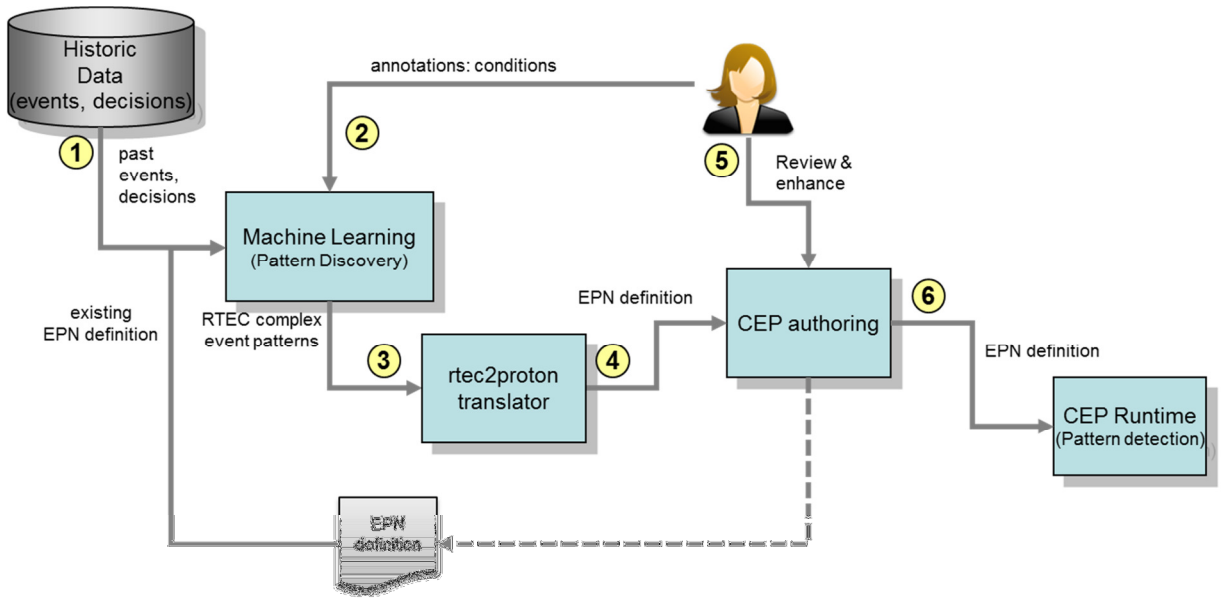

Figure 2.15 - SPEEDD Design Time Architecture

2.5.1 Event Pattern Mining

Figure 2.15 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. 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[11]. 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 the 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

[11] Artikis A., Sergot M. and Paliouras G. An Event Calculus for Event Recognition. IEEE Transactions on Knowledge and Data Engineering (TKDE), to appear.

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.

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.

All the communication between SPEEDD components is done by means of posting events on the event bus (Apache Kafka) and listening/reading events as they're posted by others.

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).

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 by the corresponded requirement documents. 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.

2.7 Deployment Architecture

The diagram in Figure 2.16 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 on 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 optional decision making external (black-box) component 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.

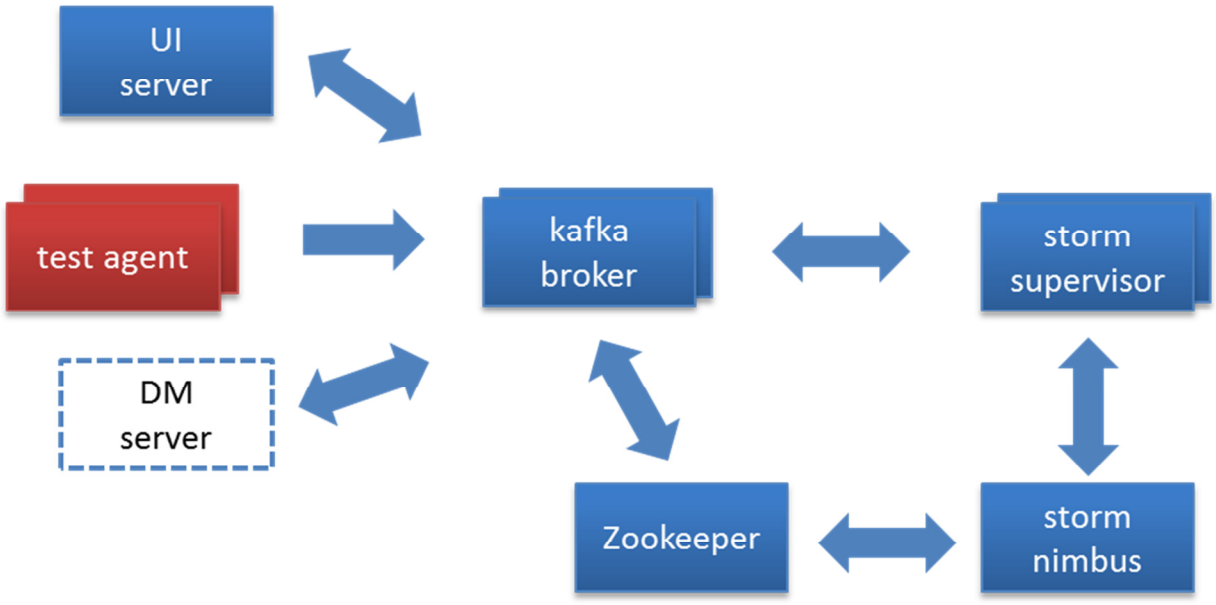

Figure 2.16 - Deployment Architecture

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), or in a single virtual machine instance.

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

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 2.4.1.1 and 2.4.1.2.

The stream processing infrastructure is based on Apache Storm 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.[13]

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[14], Mockito[15]) 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.

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.

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

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.[16]

[16] 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 will be 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 will verify the correctness of the prototype functionality. For every test case, a limited sequence of raw events will be 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 2.7 we described the deployment environment that should be used to run performance tests. In course of performance testing we will 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 Conclusions

In this document, we drafted main architecture principles that should guide the development of SPEEDD prototype. 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.

5 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.

5.1 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 interested to collaborate with FERARI[17] project which is also building a highly scalable event processing technology based on Proton. Knowledge and code reuse would be beneficial for both activities.

5.2 Storm

Apache Storm[18] 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[19] 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 5.1. 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.

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

[17] http://www.ferari-project.eu/
[18] http://storm.apache.org/
[19] http://storm.incubator.apache.org/documentation/Trident-tutorial.html

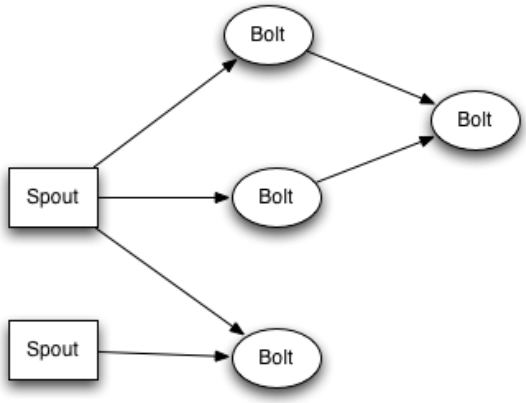

Figure 5.1 - Storm Topology[20]

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 5.2.

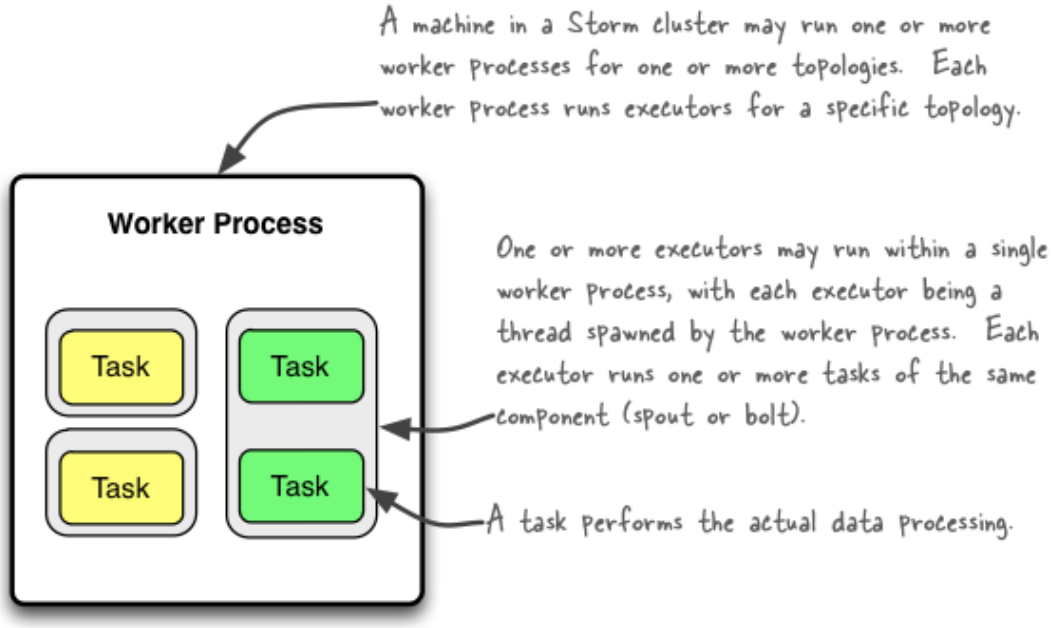

Figure 5.2 - Storm Parallelization[21]

[20] 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 storm 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[22], storm has been benchmarked at processing of 1M of 100 byte messages per second per node on the hardware with 2xIntel E5645@2.4Ghz CPU with 24GB RAM, however the details of the benchmark were not available to the authors. A recent performance analysis[23] comparing performance characteristics between IBM Infosphere Streams product and Storm reported ~50K emails /sec on a 4-node cluster for an email processing application. While that is much more modest result than 1M /s 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.

5.3 Akka

Akka[24] 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. 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 failures assume that actors are supposed to fail and crash frequently, and provide simple and robust mechanisms for recovery.

---

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

When an actor terminates – all its children are terminated automatically. Actor hierarchy is illustrated in Figure 5.3.

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

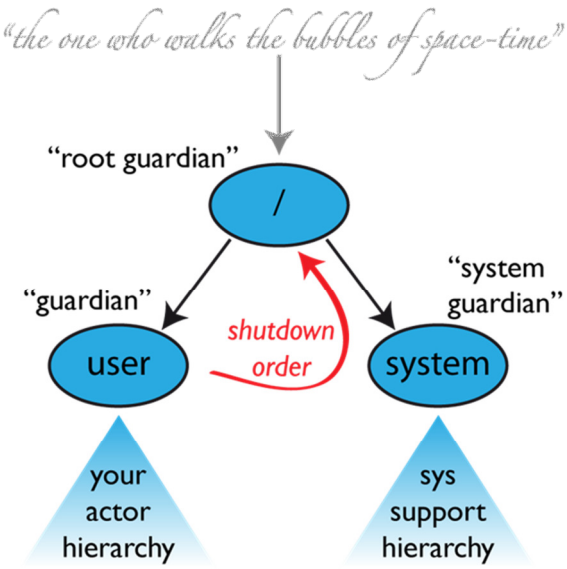

Figure 5.3 - Akka Actors Hierarchy[25]

The communication between actors is asynchronous. The decision about the location of each actor is configurable, and can be done programmatically. Messages can get lost – 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[26]. The project has been open sourced, Typesafe is the major contributor.

[25] 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[27], 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.

5.4 Spark Streaming

Apache Spark[28] is a general purpose cluster computing system designed to process large volumes of data in distributed and parallel manner. Spark Streaming[29] is an extension of Spark API that enables processing of live data streams[30].

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 5.4). 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 5.4 - 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

[26] http://typesafe.com
[27] http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine
[28] https://spark.apache.org/
[29] https://spark.apache.org/streaming/
[30] Spark: Cluster Computing with Working Sets. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010, Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. HotCloud 2012. June 2012.

contrast the other frameworks that support at-least-once message delivery, Spark streaming provides statefull exactly-once delivery semantics[31].

Scala and Java APIs are available.

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 670 K records per second. The same benchmark run on Storm gave 115 K records /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.

5.5 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 FERARI, and at the moment of current evaluation there was work in progress on porting Proton on Storm. Building SPEEDD on Storm would provide better reuse opportunity than other platforms.

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

[31] https://spark.apache.org/docs/latest/streaming-programming-guide.html#failure-of-a-worker-node

5.6 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. 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 msgs/sec in persistent mode[32]) 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[33] (publish: 500K msgs/sec, consume: 22K msgs/sec).

Kafka is an early Apache incubator project, less mature than RabbitMQ, written in Scala (Java API is available). There are client libraries in all common languages.

[32] http://bhavin.directi.com/rabbitmq-vs-apache-activemq-vs-apache-qpid
[33] http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf

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