|  |  |  |
| --- | --- | --- |
|  | 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, Fabiana Fournier, …

Status: Draft (Version 0.1)

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 |  |
| Author(s) |  |
| Internal Reviewers | TBD |
| Status & version | Draft |
| Keywords | TBD |
|  |  |

|  |
| --- |
| **Executive Summary** |

In 1-2 pages give a

- 1-sentence punch line about the contribution of the deliverable

- 1-paragraph presentation of the project goals and the work package goals

- 1-paragraph description of the work presented in the deliverable

- 1-paragraph presentation of how the project (and the state-of-the-art if relevant) benefits from the work.

- 1-paragraph description of the main results/findings of the deliverable

- 1-paragraph presentation of the work to follow, based on the deliverable

|  |
| --- |
| **Contents** |

[1 Introduction 6](#_Toc400370822)

[1.1 History of the document 6](#_Toc400370823)

[1.2 Purpose and Scope of the Document 6](#_Toc400370824)

[1.3 Relationship with Other Documents 6](#_Toc400370825)

[2 Main Sections 7](#_Toc400370826)

[2.1 System Requirements 7](#_Toc400370827)

[2.2 Approach 7](#_Toc400370828)

[2.3 Conceptual Architecture 8](#_Toc400370829)

[2.4 SPEEDD Runtime Architecture 10](#_Toc400370830)

[2.4.1 Event Bus 12](#_Toc400370831)

[2.4.2 Event/Data Providers 14](#_Toc400370832)

[2.4.3 Action Consumption – Actuators/Connectors 15](#_Toc400370833)

[2.4.4 Complex Event Processor 15](#_Toc400370834)

[2.4.5 Decision Management 20](#_Toc400370835)

[2.4.6 Dashboard application 20](#_Toc400370836)

[2.5 Build-Time Architecture 20](#_Toc400370837)

[2.5.1 Event Pattern Mining 20](#_Toc400370838)

[2.5.2 Authoring of CEP Rules 20](#_Toc400370839)

[2.5.3 Decision Management – the Offline Part 20](#_Toc400370840)

[2.6 Integration – APIs and Data Formats 20](#_Toc400370841)

[2.7 Deployment Architecture 21](#_Toc400370842)

[2.8 Non-Functional Aspects 21](#_Toc400370843)

[2.8.1 Scalability 21](#_Toc400370844)

[2.8.2 Fault Tolerance 21](#_Toc400370845)

[2.8.3 Testability 21](#_Toc400370846)

[3 Conclusions 22](#_Toc400370847)

[4 Appendix – Technology Evaluation 22](#_Toc400370848)

[4.1 Stream Processing – requirements and evaluation criteria 22](#_Toc400370849)

[4.2 Storm 22](#_Toc400370850)

[4.3 Akka 22](#_Toc400370851)

[4.4 Spark Streaming 22](#_Toc400370852)

[4.5 Choice of the Messaging Platform 22](#_Toc400370853)

|  |
| --- |
| **List of Tables** |

# Introduction

## History of the document

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Change Description** |
| 0.1 | 1/2/2014 | Alexander Artikis (NCSR) | Set up the document |
| 1 | 1/2/2014 | Alexander Artikis (NCSR) | Content adjusted |

## Purpose and Scope of the Document

Briefly state the purpose and scope of the deliverable, and indicate the target readership

## Relationship with Other Documents

Describe how the deliverable relates to other deliverables and papers

# Main Sections

## System Requirements

This section lays out the main requirements for SPEEDD prototype. The requirements are based on the requirements documents provided by the use cases.

## 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 - ‎2.5).
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

## 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 management algorithms and 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 complex event pattern 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 management 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 complex event patterns 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 management 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]](#footnote-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.

## SPEEDD Runtime Architecture

The architecture of the runtime part of SPEEDD follows the “Event-Driven Architecture” paradigm. 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.

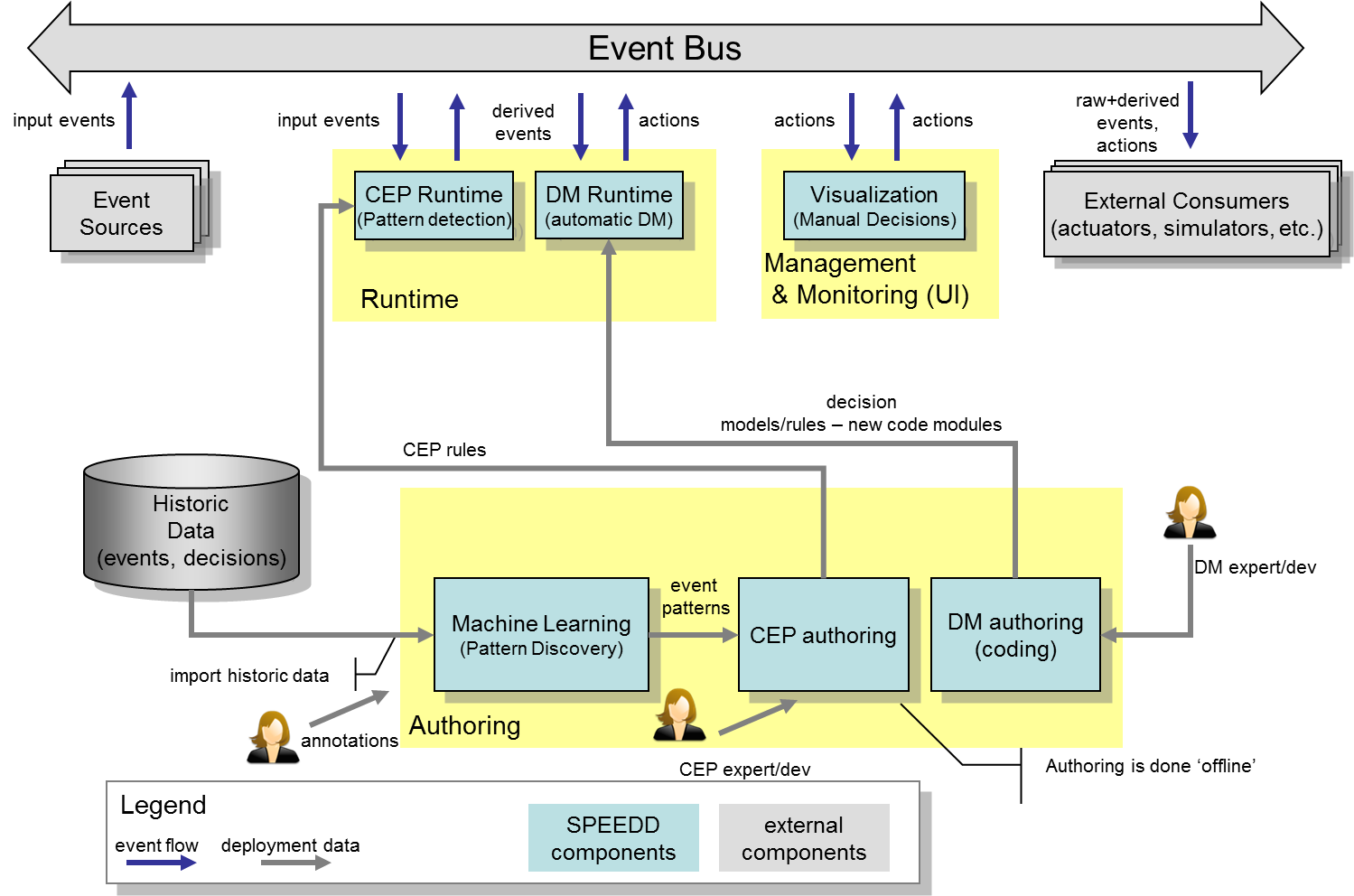


Figure ‎2.3 - SPEEDD - Event-Driven Architecture

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 management 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.

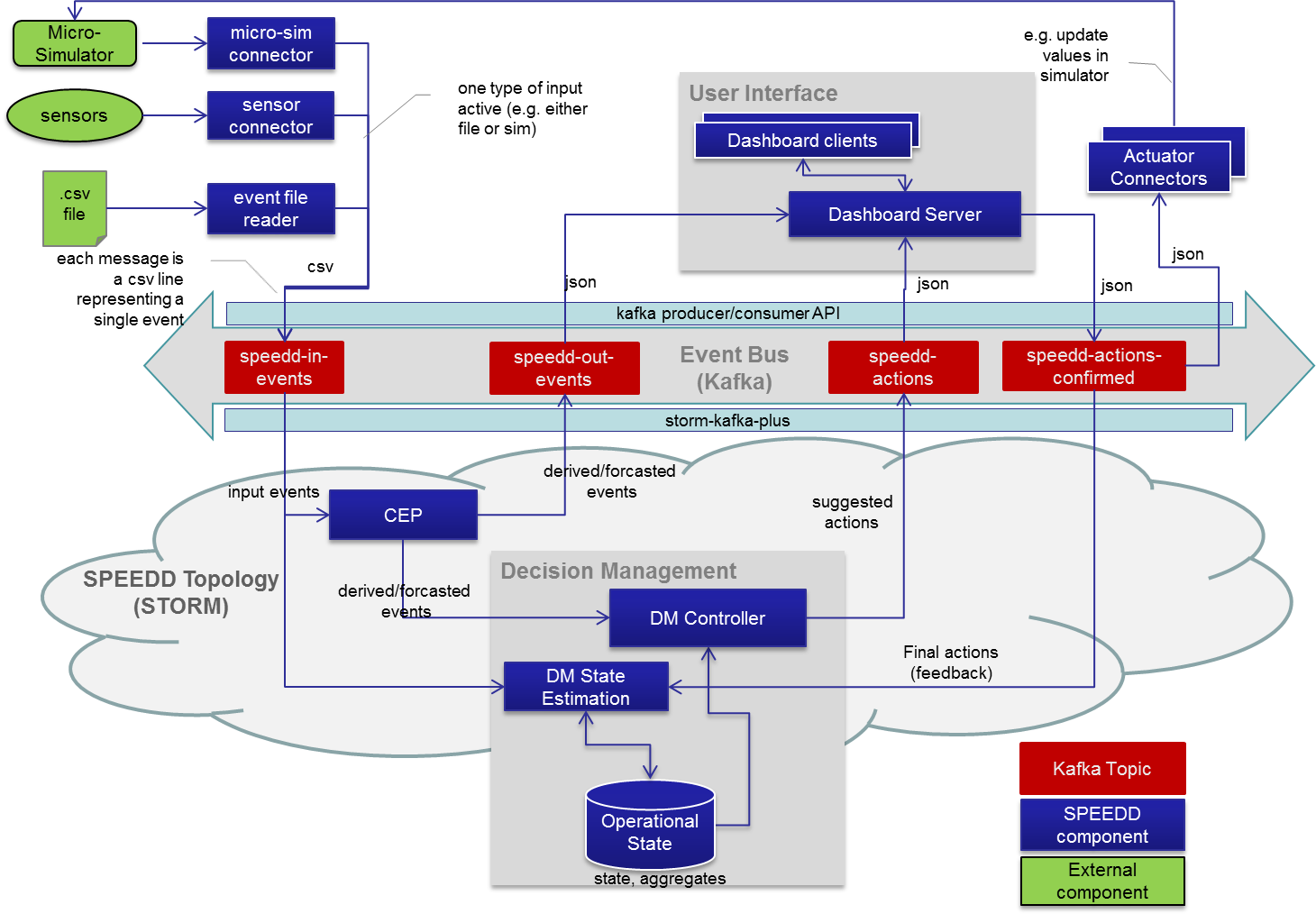


Figure ‎2.4 - SPEEDD Runtime - Event-Driven Architecture (Traffic Use Case)

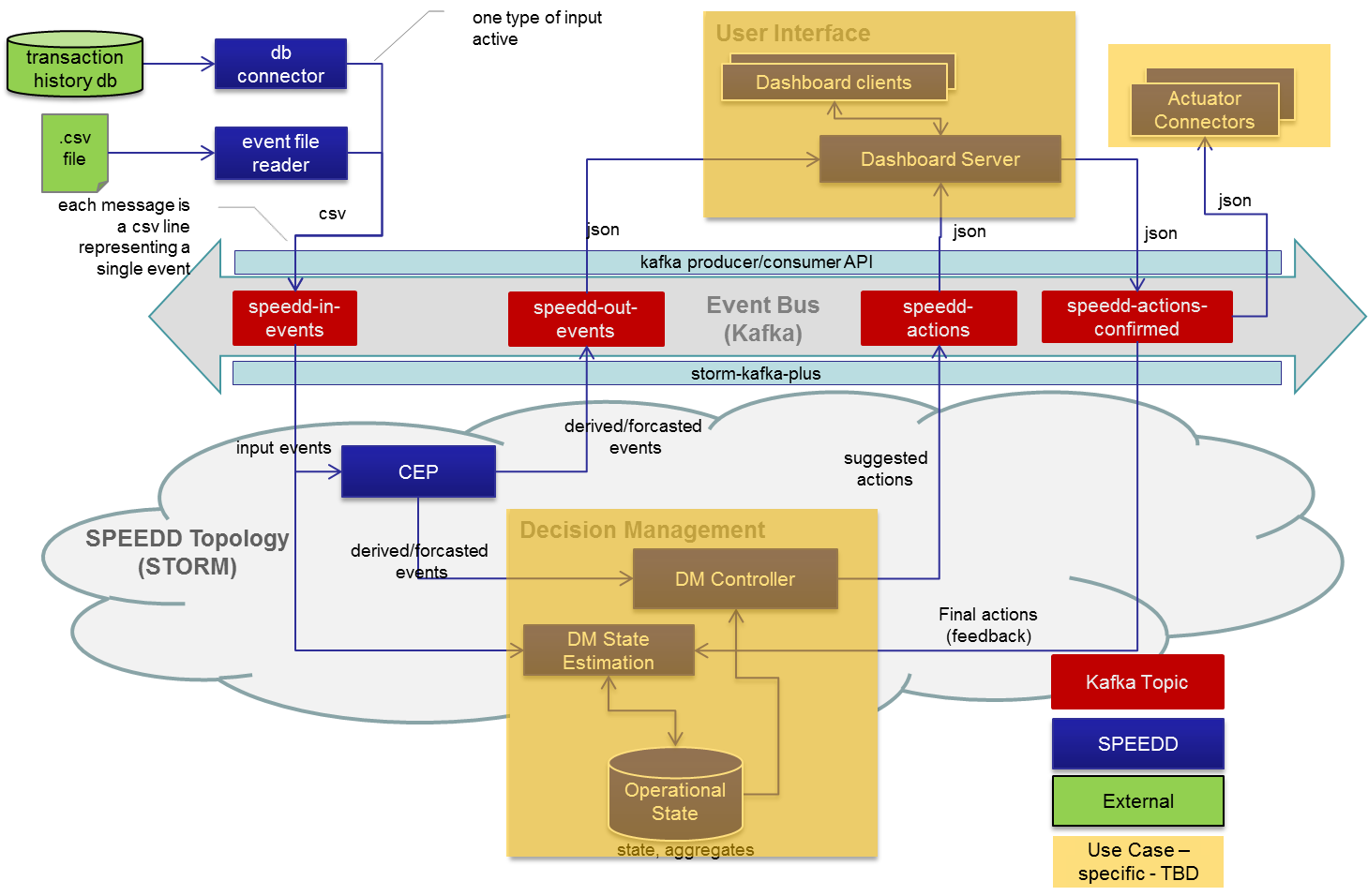


Figure ‎2.5 - SPEEDD Runtime - Event-Driven Architecture (Credit Card Fraud Use Case)

### Event Bus

The technology chosen for the event bus component is Apache Kafka[[2]](#footnote-2). It provides a scalable, performant, and robust messaging platform that matches SPEEDD requirements (see ‎4.5 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 Management,  Dashboard |
| speedd-actions | Suggested decisions | Decision Management | Dashboard |
| speedd-actions-confirmed | Actions confirmed for execution | Dashboard (in open loop mode), Decision Management (in closed loop mode) | Dashboard, Actuators |

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.

#### 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.

#### 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.

#### 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.

Persistence

Serialization format

Message schema

### 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.

#### 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. We define three connector types corresponding to the types of the event sources:

* Sensor connector[[3]](#footnote-3)
* Micro-sim connector
* File reader connector – replay past events from a file

#### 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.

### 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.

#### 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.

#### 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.

### Complex Event Processor

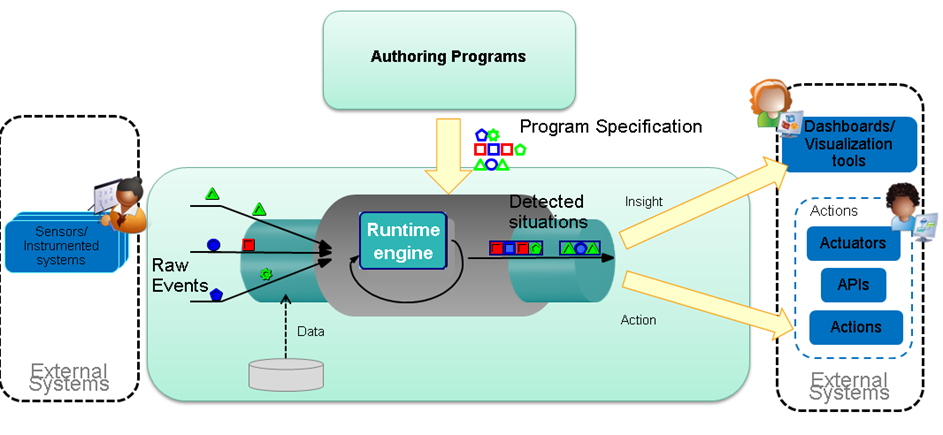
****

Figure ‎2.6 - 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, derives and emits **complex** events (see Figure ‎2.6).

#### 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.

#### 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.

#### High-level Architecture

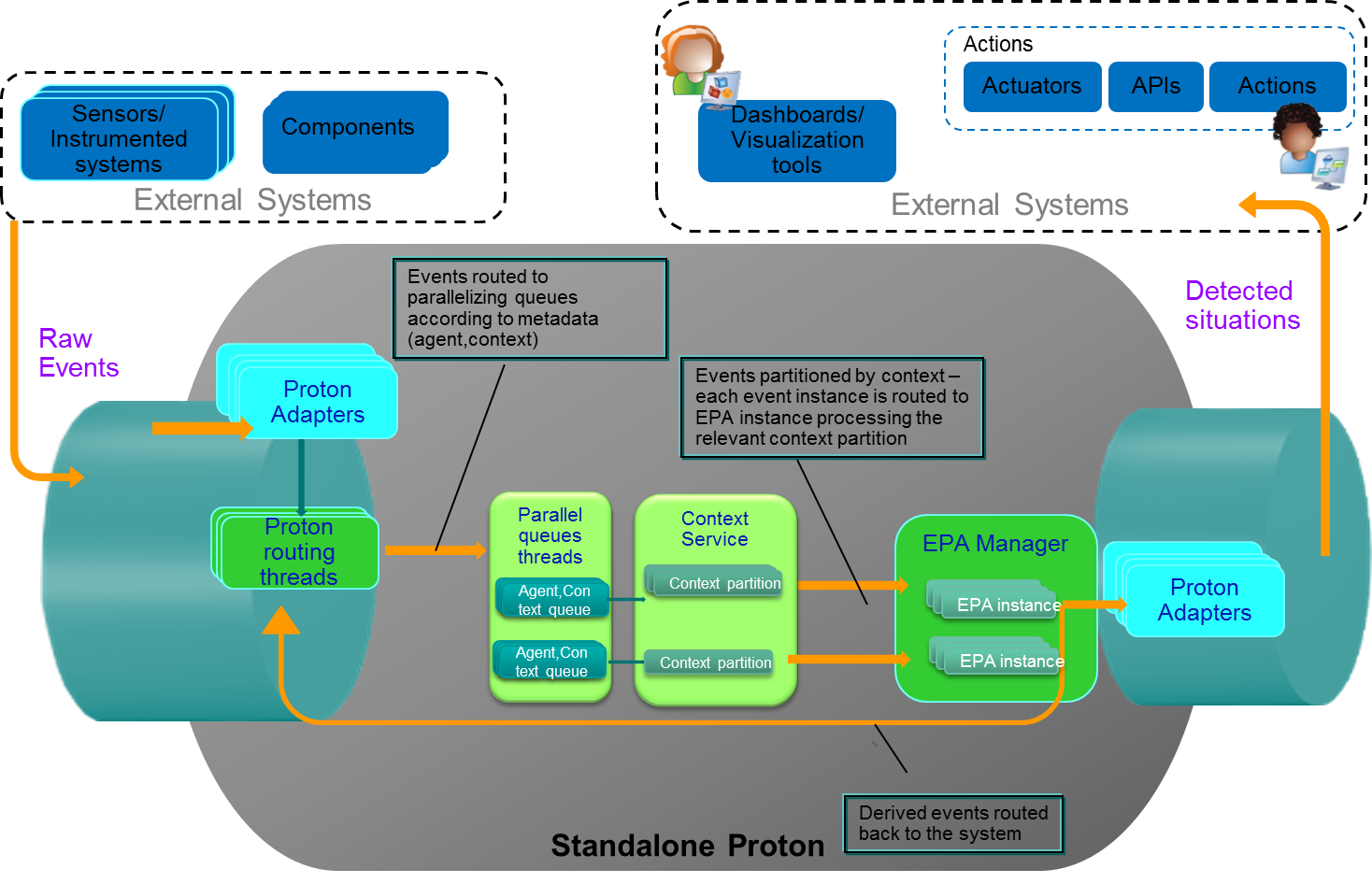


Figure ‎2.7 - Proton Runtime and external systems

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 complex 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:
   1. 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
   2. Solve out-of-order problems – can buffer for a specified amount of time
   3. Solve correctness problems
3. The event is passed to **context service**, where it is determined:
   1. If the context is an initiator or a terminator, new contexts might be initiated and or terminated, according to relevant policies.
   2. Which context partition/partitions this event should be grouped under
4. The event is passed to **EPA manager:**
   1. Where it is passed to the specific EPA instance for the relevant context partition,
   2. Added to state of the instance
   3. And invokes pattern processing
   4. If relevant, a derived event is created and emitted

#### Proton component architecture

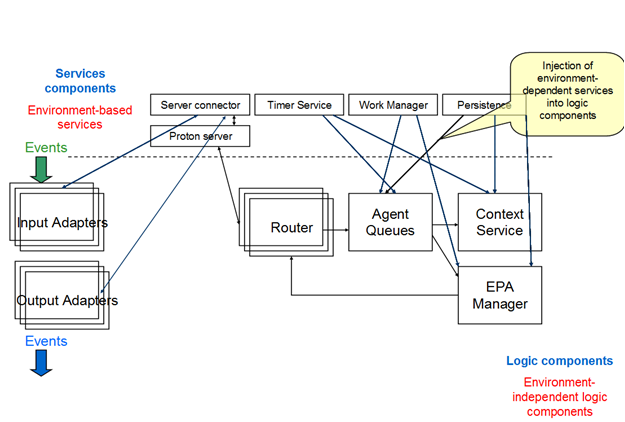


Figure ‎2.8 - Proton components

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.

#### Distributed Architecture on top of STORM

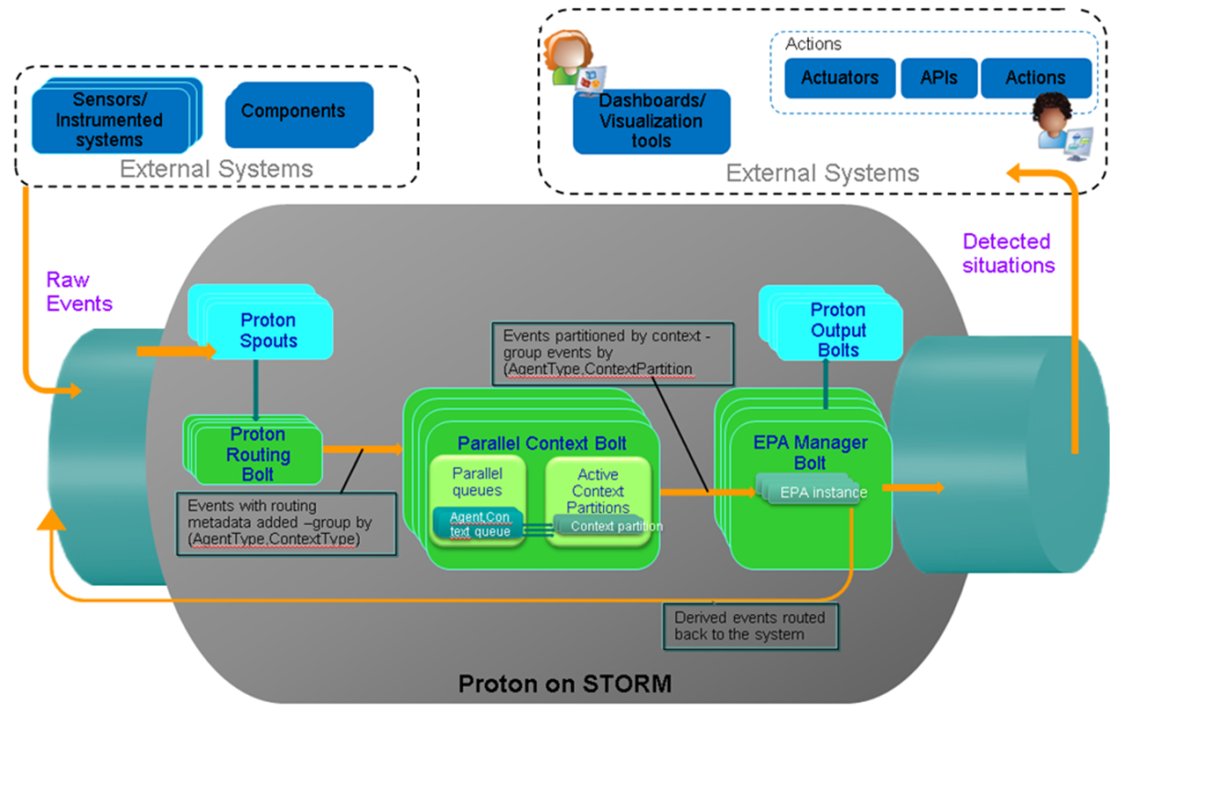


Figure ‎2.9 - Architecture of Proton on STORM

The Proton architecture on top of STORM (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.

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.

### Decision Management

Architecture of the decision management component(s), approaches, issues, etc. Possibly discuss separately the design for every use case.

### 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 ‎2.10.



Figure ‎2.10 - Dashboard Architecture

The Dashboard Server is built using the Express web application framework for Node.js. 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-in-events*, *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 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 Management (DM) unit (e.g. ramp metering rates). Furthermore, the Dashboard Client aims to support the decision-maker by predicting 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 ‎2.11 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, 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 ‎2.11 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.



Figure ‎2.11 - User Interface

## Build-Time Architecture

The conceptual view of the build time architecture for SPEEDD is presented in Figure ‎2.6. The goal of the build time is to provide

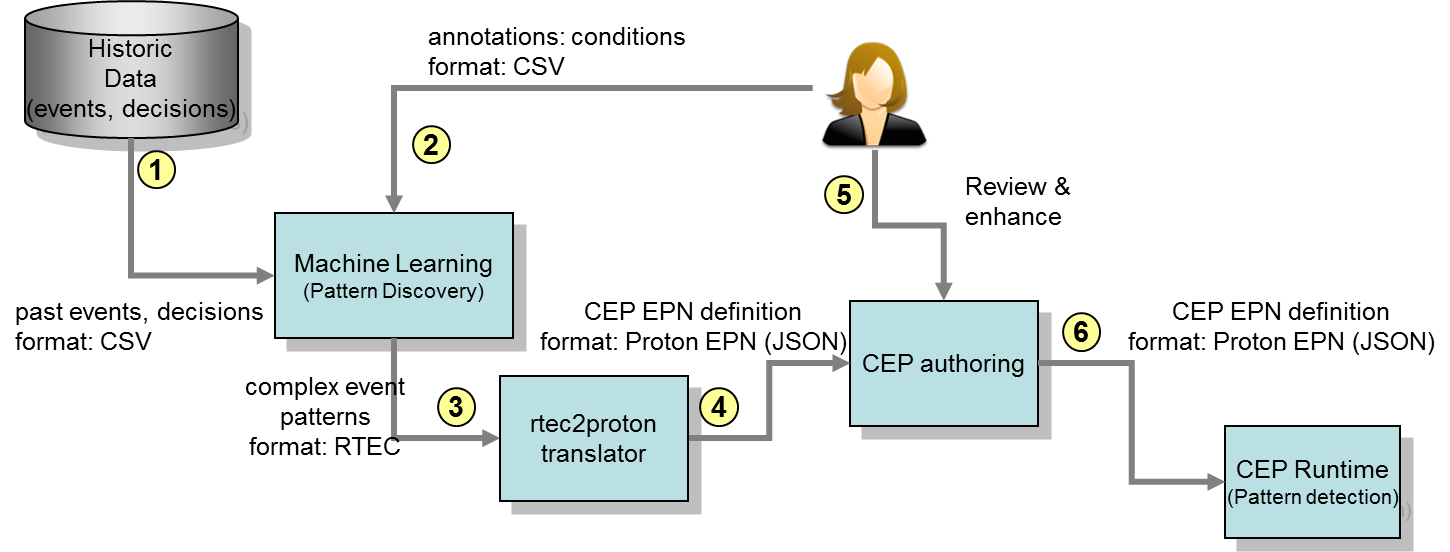


Figure ‎2.12 - SPEEDD Build Time Architecture

### Event Pattern Mining

Figure 2 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 architecture system, 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[[4]](#footnote-4). 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.

### Authoring of CEP Rules

Describe the process, challenges, and approach to translation of the CEP patterns discovered in using machine learning into Proton EPN definition.

### Decision Management – the Offline Part

Should see if this is relevant. If it is, describe how decision management component is configured or adjusted based on the exploration of the historic data.

## Integration – APIs and Data Formats

Describe the integration mechanisms between different components and between the system and the outside world. List and describe the APIs and data formats in use.

## Deployment Architecture

TBD – describe proposed deployment architecture

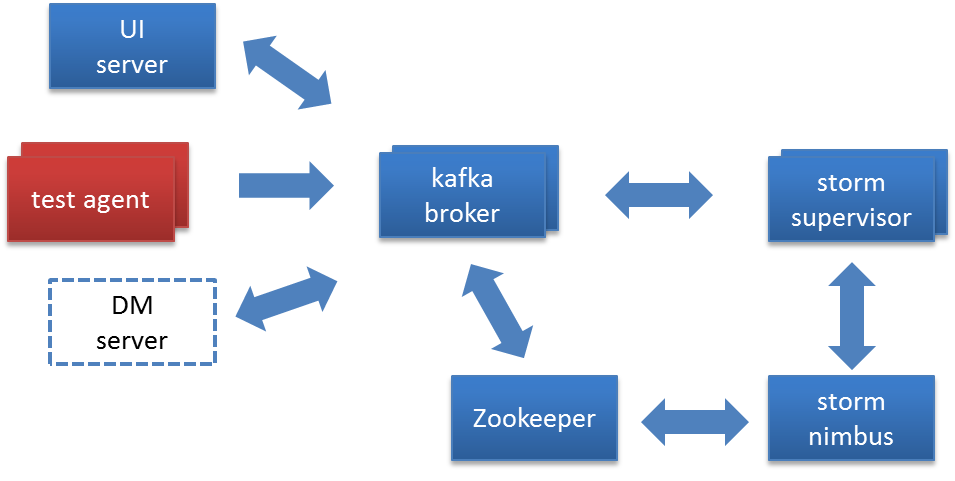


Figure ‎2.13 - Deployment Architecture

## Non-Functional Aspects

### Scalability

Explain why is the proposed architecture is scalable. Describe how the system will scale up and out to match the load.

### Fault Tolerance

Explain what types of failures the system is designed to stand. Describe the designed behavior of the system in case of such failures.

### Testability

Describe the approach to testing the system. Address the functional testing as well as performance testing approach as designed.

# Conclusions

TBD

# Appendix – Technology Evaluation

This could be one or more appendix parts. Here we’ll explain the approach, criteria, and the final choice of the technology stack that was made.

## Stream Processing – requirements and evaluation criteria

## Storm

## Akka

## Spark Streaming

## Choice of the Messaging Platform

# Appendix – Distributed Thresholded Counter

This document gives a high level overview of the architecture of the distributed thresholded counter, and describes its components. This architecture will serve as the foundation for future distributed stream monitoring.

## Objective

The objective is to provide an alert whenever the number of data records for a specific client over a specific time window exceeds a global threshold, in an environment where the events are registered at a number of distributed nodes. The solution aims to minimize communication between the different nodes.

## Method

We assume that for every event registered at one of the nodes we receive an event**,** which contains the details of the originating device and the event start time.

The global threshold will be split into local thresholds, one for each node, such that their sum is equal to the global threshold. As long as the local count of events for a single client does not exceed the local threshold, no action is required. In case the count for a given client exceeds the local threshold further action is required to determine whether the global count has exceeded the global threshold. In order to determine this the node interacts with a coordinator, which in turn interacts with the other nodes. This interaction is referred to as a **violation resolution process**. During this process, the coordinator determines if the global counter for this client has crossed the global threshold. At the end of this process each center receives a new threshold value for this client. In case the global threshold for the client has not been crossed, the node will verify that the local counter for the client remains below the threshold. If the global threshold has been crossed, the center will verify that the local counter remains above the local threshold.

### Distributed Algorithm

We assume that all clocks at all the nodes are synchronized. We first describe the distributed algorithm assuming that there are no time delays of any sort (no network or processing delays). We then proceed to describe how time delays are handled.

As mentioned above, each node is assigned a default local threshold such that their sum is equal to the global threshold. In addition, after a violation resolution procedure has been performed for a certain client, it may be assigned a local threshold that is different from the default. Let us denote the number of nodes by , the global threshold by , and the local default threshold at each node by (we can assume that ). If the client has been assigned a specific threshold at node , let us denote this threshold by .

When the node detects that the counter for the counter exceeds its threshold (either a specific threshold assigned to it, or the default one if no specific threshold has been assigned to it), it notifies the coordinator, and reports the relevant client and counter value. The coordinator requests the counter value for this client from the rest of the nodes. Upon receipt of the current counters from all nodes, the coordinator can determine the global counter for the client. Let us denote the counter for the client at node by , and the global counter for by (note that ). Let us define . The coordinator sends node the value as the new threshold value for . One can easily verify that the sum of the new local thresholds is equal to . Note that is negative if and only if the global counter for has crossed the global threshold. If is positive, the nodes will alert the coordinator when the local counter for *exceeds* the new threshold. If is negative, the nodes will alert the coordinator when the local counter *drops below* the new threshold.

#### Handling Time Delays

In this section we describe modifications to the algorithm that enable handling time delays that may occur as a result of processing or communications load, and how race conditions that may occur due to these delays are resolved. As mentioned above, we assume that the clocks at all the nodes are synchronized. In addition, we assume that maximum time a message will be delayed is bound from above. Let Δ denote that upper bound on the time delay in seconds.

Note that our goal is to detect threshold crossings as if all the events were time-stamped and processed at a central location. In order to do so, when a node notifies the coordinator that a counter crossed a local threshold, it also reports the time at which this event has happened. In addition, when the coordinator requests the current value of the counter for this client from the other nodes, it will specify the time for which this value is required (note that the request for the counter value can be for a time earlier by at most 2Δ seconds from the current time). In order to provide this functionality, each node needs to store past counter values for the last 2Δ seconds.

An additional issue that needs to be addressed by the nodes is as follows: assume a violation resolution process has been performed due to a threshold crossing that occurred for the phone number at time . After the violation has been resolved, all the nodes need to re-process the counter updates for from onwards (since the local threshold value for may have changed.) We refer to this functionality as **Update Replay**.

Finally, we define a mechanism for resolving race conditions. A race condition may occur if one node initiates a violation resolution process on a threshold crossing for the phone number at time , and before this process has been completed, a second node initiates a violation resolution process on a threshold crossing for the same phone number at time . To avoid this, we assume the following:

1. When the coordinator requests a node for a counter value at time , it will reply with the counter value only after it has verified that there are no earlier local threshold crossings for the requested client. If the node determines that an earlier local threshold crossing has occurred for the client it will initiate a violation recovery process with the earlier time, and ignore the count request.
2. Consider a node that initiates a violation recovery process for a threshold crossing that occurred at time , or a node that has received from the coordinator a request for the counter for a phone number at time . In case there are no race conditions, the coordinator is expected to provide a new threshold value. If, however, a different node has initiated a violation recovery process for a threshold crossing that occurred for the same phone number at an earlier time, , then the node will receive a counter request for the time (instead of a new threshold value). In this case, the node should reply to the counter request, and all previous violation resolution processes for this phone number are canceled.

Note that the race condition prevention and the update replay mechanisms described above guarantee the correctness of the algorithm, since they guarantee that the earliest threshold crossings are handled first, and that all subsequent updates are processed according to the newly determined thresholds.

## Architecture

The system is comprised of a local agent installed at each node, and an additional coordinator node. The local agent at each node is comprised of the following components:

1. Stream Aggregator:Receives the raw CDR events and emits a count per phone number and time window;

2. Time Machine: Stores historical data from the stream aggregator and under normal circumstances emits only the latest count;

3.Gatekeeper: Responsible for passing on counts which exceeded the threshold and initiating local violation resolution procedures;

4.Communicator: Responsible for communicating with the coordinator.

The stream aggregator receives the events and maintains a counter for each phone number representing the number of events for that client in the time window. We refer to these counters as the **counters**. The stream aggregator sends a **Counter Update Message** to the time machine, which saves it to a buffer and passes it on to the gatekeeper. The gatekeeper passes on to the communicator a Threshold Pass Message for each client which has exceeded its threshold. When this message is sent it also sends a Pause Message to the time machine to indicate that no further Counter Update Messages for this client should be sent until the violation recovery procedure has completed. The communicator is responsible for sending Threshold Pass Messages and Counter Reply Messages to the coordinator and for receiving messages from the coordinator and sending them backwards.

In case a violation recovery procedure is initiated by the coordinator it passes a Counter Request Message through the communicator to the time machine. The time machine replies with a Counter Reply Message.

When the violation recovery is complete the coordinator will pass a Play Message to all nodes in order for the time machine to replay all data from time t.

### Messages

There are six types of messages passed between the components:

1. **Counter Update Message** which contains a client (string), timestamp of start of time window (the size of the time window is fixed) and number of events (int). This is passed between the stream aggregator and the time machine.
2. **Threshold Pass Message** which contains a client (string), timestamp, count (int) and threshold (int). This is passed between the communicator and the coordinator.
3. **Counter Request Message** which contains a client (string) and a timestamp. This is passed from the coordinator to the communicator and from there to the stream aggregator.
4. **Counter Reply Message** which contains a client (string) and a timestamp.
5. **New Threshold Message** which contains a client (string), the new threshold for that number (int) and whether the phone number should be reported if it passes over or under the threshold (boolean). This is passed from the coordinator to the communicator.
6. **Pause/Play message, which contains a pause/play instruction (boolean), phone number (string) and timestamp.**

## Diagrams

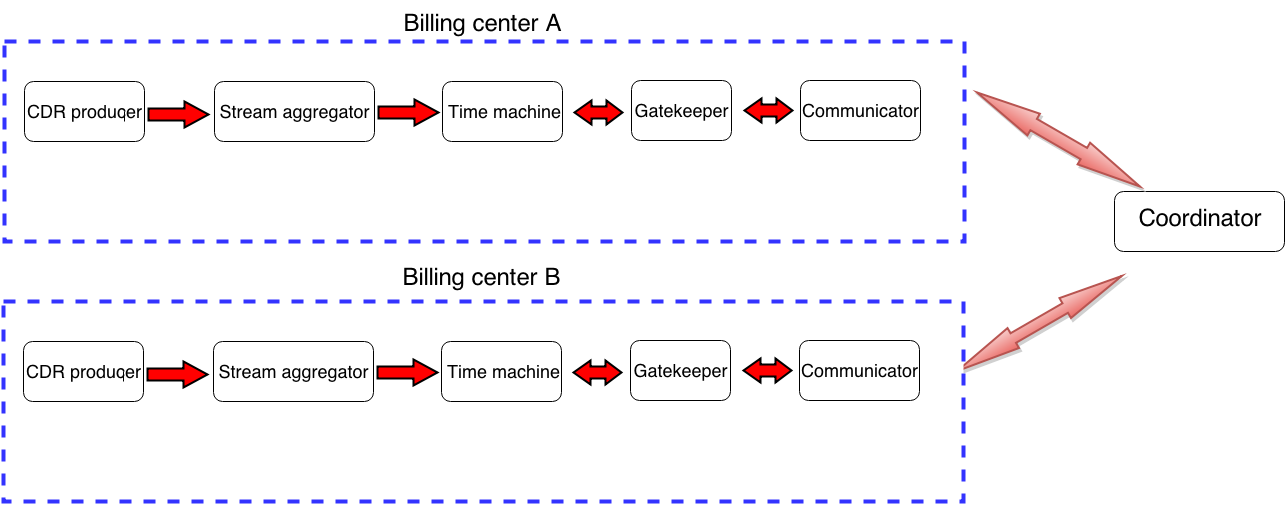


Figure 1 – components when events are CDRs arrival

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. [↑](#footnote-ref-1)
2. <http://kafka.apache.org/> [↑](#footnote-ref-2)
3. 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 [↑](#footnote-ref-3)
4. Artikis A., Sergot M. and Paliouras G. An Event Calculus for Event Recognition. IEEE Transactions on Knowledge and Data Engineering (TKDE), to appear. [↑](#footnote-ref-4)