Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D5.1 Situation Detection Mechanism

Deliverable Editor:	**Dimitris Apostolou (ICCS)**
Other contributors:	**Nikos Papageorgiou, Andreas Tsagkaropoulos, Yiannis Verginadis, Gregoris Mentzas (ICCS)**
Deliverable Reviewers:	**Salman Taherizadeh (JSI), Giannis Ledakis (Ubitech)**
Deliverable due date:	**30/04/2018**
Submission date:	**01/06/2018**
Distribution level:	**Public**
Version:	**1**

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

Change Log

Version	Date	Amended by	Changes
0.1	05/04/2018	Dimitris Apostolou (ICCS)	Table of Contents
0.2	20/04/2018	Dimitris Apostolou, Nikos Papageorgiou, Andreas Tsagkaropoulos, Gregoris Mentzas, Yiannis Verginadis (ICCS)	Draft version ready for internal review
0.3	30/05/2018	Dimitris Apostolou (ICCS)	Addressing reviewers' comments (JSI and Ubitech)
1.0	01/06/2018	Dimitris Apostolou (ICCS)	Final version

Table of Contents

List of Figures

List of Abbreviations

The following table presents the acronyms used in the deliverable.

Abbreviation	*Description*
AMQP	Advanced Message Queuing Protocol
CEP	Complex Event Processing
CPU	Central Processing Unit
CSV	Comma Separated Value
EP	Event Processing
HTTP	Hypertext Transfer Protocol
ICT	Information and Communication Technologies
IoT	Internet of Things
JSON	JavaScript Object Notation
MCA	Mobile Context Analyser
QoS	Quality of Service
SA	Situation Awareness
SDM	Situation Detection Mechanism
SML	Situation Modelling Language
SQL	Structured Query Language
UDP	User Datagram Protocol
UML	Unified Modelling Language
VM	Virtual Machine
XMI	XML Metadata Interchange

Executive Summary

This deliverable reports on the work performed under Task 5.1 "Situation awareness at the extreme edge of the network" with respect to the development of a situation detection mechanism. The mechanism is part of the Meta-Management layer of the PrEstoCloud architecture (D2.3), which mainly provides valuable input to the Control layer in order to perform adaptation of cloud resources. The situation detection mechanism is able to detect situations where the PrEstoCloud infrastructure requires an adaptation action which significantly influences the recommended processing topology, including the extreme edge layer.

We designed Situation Detection Mechanism so as it is modular and can be easily deployed as a Docker container or a set of Docker containers. Moreover, we designed Situation Detection Mechanism to be independent of Complex Event Processing libraries and we have shown that it can operate with both the Siddhi and Drools Complex Event Processing libraries. The deployment flexibility of Situation Detection Mechanism is quite important since it allows the PrEstoCloud adopter to use the Complex Event Processing engine of her choice based on the processing capabilities required in each case and the prior expertise regarding a certain Complex Event Processing engine. We also demonstrated a real-world scenario indicative of the usage of our component, and its capabilities.

The primary input for Situation Detection Mechanism is considered any health status event or application performance-related event transmitted to the Communication and Message Broker. Such events are exploited by SDM in order to reveal problematic situations with respect to the state of the cloud and edge resources used for hosting big data-intensive applications or to the performance state of the application itself. The Situation Detection Mechanism will provide input to the PrEstoCloud Adaptation Recommender (D5.5), which will consume the situations that are detected by Situation Detection Mechanism to initiate the most appropriate adaptation actions. The Adaptation Recommender will be fed also with the output of other WP5 components, namely, the Mobile Context Analyser and Workload Predictor and will generate as output specific recommendations for adaptations in the PrEstoCloud infrastructure resources.

1. Introduction

1.1 Scope

This deliverable reports on the baseline implementation of the Situation Detection Mechanism (SDM), a software system that will equip the PrEstoCloud platform with the necessary situation awareness for detecting situations requiring some kind of infrastructure or application adaptation or re-configuration. It will do so by processing and analysing data streams generated both by the PrEstoCloud platform as well as data-intensive applications and services deployed on PrEstoCloud, that is, on attached cloud resources or at computing resources at the extreme edge of the network.

The scope of this deliverable includes the description of the Situation model that abstractly depicts entities and relationships between entities that model the SDM capabilities and enable the detection of situations with respect to the status of the services and the deployment infrastructure. Secondly, this deliverable describes the research and development work towards the SDM whose goal is to identify interesting situations that might lead to resources adaptation recommendations or data-intensive application reconfiguration or redeployments.

SDM is designed so that it can use the following inputs: (i) Big Data streams; (ii) event data; (iii) contextual data; (iv) QoS variations (e.g., due to low bandwidth) and (v) monitoring data related to the real-time processing networks. The overarching goal for SDM is to enhance PrEstoCloud with reliability with respect to the stream processing topology as it will be able to detect possible failures and trigger corrective actions through issuing new adaptation recommendations. Moreover, SDM aims to serve at the same time as a feedback management mechanism with respect to the analysis of the outcomes of resources adaptations and data-intensive redeployments, thus it will allow for improvements of future adaptations.

1.2 Relation to PrEstoCloud Tasks

The SDM component has been defined in the description of work of the PrEstoCloud project as part of Task 5.1. It materialises a software system able to sense the PrEstoCloud Infrastructure and deployed applications and services, and detect situations in the way the latter was defined in Deliverable D2.1 (Scientific and Technological State-of-the-Art analysis) and formulated as a functional requirement in Deliverable D2.2 (High-level requirements analysis for the PrEstoCloud platform). The system also adheres to the foundations set for the entire PrEstoCloud topology in deliverable D2.3 (PrEstoCloud Conceptual Architecture). The SDM receives input from the Communication and Message broker for real-time data streams and processes it taking into account the specifications of the communication format developed in Deliverable D2.4 (Format and procedures for plugging in real-time data streams). The SDM is itself a primary input for the operation of the Resources Adaptation Recommender (Task T5.2).

1.3 Document Structure

The deliverable is structured as follows: Section 2 presents the related work already performed on situation awareness, modelling and detection in the area of ubiquitous computing. Section 3 describes the situation metamodel and our approach for the development of the PrEstoCloud situation detection mechanism. Section 4 presents the architecture and the technologies used to implement our situation detection approach. Section 5 includes an analysis of the performance of the situation detection mechanism. Last, we present our conclusions and future work in Section 6.

2. Related Works

This section discusses works that pertain situation awareness, modelling and detection, with an emphasis on ubiquitous computing systems and applications.

2.1 Situation Awareness

Situation Awareness (SA) refers to the "perception of the elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future" (Endsley 2016), (Franke and Brynielsson 2014). To realize systems for Situation Awareness, individual pieces of raw information (e.g. sensor data) should be interpreted into a higher, domain-relevant concept called situation, which is an abstract state of affairs interesting to specific applications. The power of using "situations" lies in their ability to provide a simple, human-understandable representation of, for instance, sensor data (Loia et al. 2016). In the context of dynamic computing systems, situation is defined as an event occurrence that might require a reaction (Adi and Etzion 2004).

In pervasive computing, SA is the capability of the entities of the computing environment to be aware of situation changes and automatically adapt themselves to such changes to satisfy user requirements, including security and privacy (Yau & Liu 2006). SA is one of the most fundamental features to support dynamic adaptation of entities in pervasive computing environments. A situation is a set of contexts in the application over a period of time that affects future system behaviour. A context is an instantaneous, detectable, and relevant property of the environment, system, or users, such as location, available bandwidth and a user's schedule (Yau et al. 2002a, b).

A pervasive computing environment involves a set of cooperative entities, each of which has related context data. In order to support situation-aware adaptation of the entities in pervasive computing environments, it is necessary to model and specify context and situation in a way such that multiple entities can easily exchange, share and reuse their knowledge on context and situation (Yau et al. 2006).

2.2 Situation Modelling

A situation is a subjective concept, whose definition depends on sensors in a current system, which decide available contexts used in a specification; on the environment where the system works, which determines the domain knowledge to be applied (e.g., a spatial map); and on the requirement of applications, which determines what states of affairs are interesting. The same sensor data can be interpreted to different situations according to the requirements of applications (Ye et al .2012). Situations are composite entities whose constituents are other entities, their properties and the relations in which they are involved (Costa et al. 2006). Situations support us in conceptualizing certain "parts of reality that can be comprehended as a whole" (Hoehndorf 2005).

Situations are often reified (such as in Barwise 1989, Costa et al. 2006), or ascribed an "object" status (Kokar et al. 2009), which enables one not only to identify situations in facts but also to refer to the properties of situations themselves. For example, we could refer to the duration of a particular situation or whether a situation is current or past, which would enable us to say that the situation "Drone out of range" occurred yesterday and lasted two hours. The temporal aspect of situations also enables us to refer to change in time, thus we could say that "VM1's CPU utilisation is rising" or that "VM2 memory has been overdrawn for the last 15 days".

Costa, et al. 2012 developed the Situation Modeling Language (SML) which is a graphical language for situation modeling, allowing the expression of primitive situation types and complex situation types (with temporal constraints when required) SML allows composition of situations using the temporal formal relations defined by Allen (1983), namely before, meets, overlaps, starts, during, finishes, coincides and their relations (after, met by, overlapped by, started by, includes, and finished by).

A situation type definition in SML is a composition of two kinds of models (Sobral et al. 2015): a context model and a situation type model. The context model is a structural model that defines the classes of entities and relationships that exist in the modelled domain, which in turn are referred by the situation type

model entities. In order to define context models, SML employs an ontologically well-founded UML class diagram profile called OntoUML (Guizzardi 2005).

In Context Toolkit (Dey 2000), situation is modelled on a system level as the aggregation of context, but there is no language-level situation modelling. Situation calculus and its extensions (McCarthy 2000) model situation based on the effects of actions and events, and consider situation as a complete state of the world. A core SAW ontology (Matheus et al. 2003) models situation as a collection of Goals, SituationObjects and Relations using UML, and can be converted to OWL representation. A conceptual model for context and situation for service-based systems and a situation specification example based on the conceptual model using F-logic are presented in (Yau et al. 2006).

Kalyan et al. (2005) introduced a multi-level situation theory, where an intermediate level micro situation is introduced between infons and situations. Infons and situations are two of the major conceptual notions in situation theory. An infon embodies a discrete unit of information for a single entity (e.g., a resource node), while a situation makes certain infons factual and thus support facts. Situations are considered as a hierarchical aggregation of micro situations and situations. This work aims to assist information reuse and support ease of retrieving the right kind of information by providing appropriate abstraction of information. Using the concept of micro situations, the authors address how to handle complex user queries by creating simpler entity-specific micro situations and further combining them to arrive at users' current situation and as well as to enable efficient reasoning. We follow a similar approach in which we allow for different abstraction levels for situations, as described in the Section 3.

2.3 Situation Detection in Ubiquitous Environments

Situations in computing infrastructures are highly related to sensor data, domain knowledge on environments and individual users, and applications. Sensor data occur in large volumes, in different modalities, and are highly inter-dependent, dynamic and uncertain. Situations are in a rich structural and temporal relationship, and they evolve in diffuse boundaries. In addition, the complexity in domain knowledge and applications makes studying situations a very challenging task. (Ye et al. 2012).

Situation detection has been studied extensively in ubiquitous computing, Figure 1 shows an overview of existing approaches.

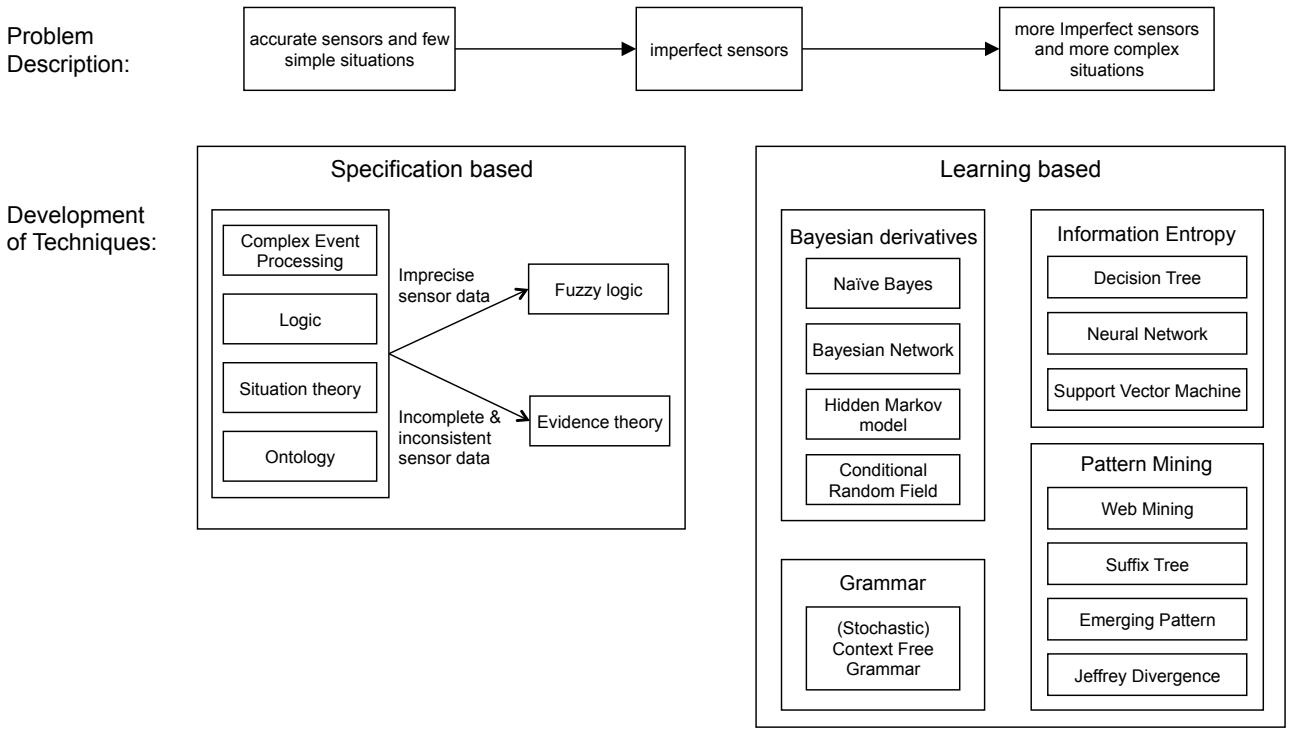

Figure 1. Overview of situation detection approaches (adapted from Ye et al. 2012)

Specification-based approaches represent expert knowledge in the form of logic rules based on event and sensor data, and apply reasoning engines to infer proper situations from current sensor input (Ye et al. 2012). Existing approaches range from earlier attempts in first-order logic (Ranganathan et al. 2004) to complex event processing and more advanced logic models that aims to support efficient reasoning while keeping expressive power, see, e.g., Loke 2010. With their powerful representation and reasoning capabilities, ontologies have been widely applied, see Chen et al. 2004. As more and more sensors are deployed in real-world environments for a long-term experiment, the uncertainty of sensor data starts gaining attention. To deal with the uncertainty, traditional logic-based techniques need to be incorporated with other probabilistic techniques (Delir Haghighi et al. 2008). Specification-based approaches have introduced uncertainty metrics to describe sensor data, including incompleteness, accuracy, timeliness, and reliability (Gray and Salber 2001; Lei et al. 2002; Cohen et al. 2002). The concept hierarchy in ontologies are typically used to evaluate the precision of sensor data against the conditions of rules.

Learning-based techniques have been widely applied to learning complex associations between situations and sensor data (Ye et al. 2012). Most of the examined learning techniques are supervised learning, such as naïve Bayes, Bayesian networks, HMMs, CRFs, and so on. These techniques learn the models and parameters from training data that usually requires a human to label a situation to sensor data that are observed during the occurrence of this situation. When there exists a large number of situations to be identified, manual labelling of training data may place a significant burden on developers involved in the data collection. Therefore, supervised learning techniques may have limitations in real-life deployment where scalability, applicability, and adaptability are highly concerned (Gu et al. 2010). To tackle this issue, researchers have employed unsupervised learning approaches. Among them, suffix-tree and Jeffrey divergence can extract features from sensor observations, which are distinguishable from one situation to another (Ye et al. 2012). A neural network is classified as unsupervised, although parameters of neural networks can sometimes be estimated using supervised learning. Web mining techniques are not strictly unsupervised learning in that they perform the learning on web documents, rather than on the collected sensor data. Decision trees and Support Vector Machines, which are built on information entropy, have also been used to classify sensor data into situations based on features extracted from sensor data.

Compared to specification-based approaches, a distinguishable features of the learning-based approaches is their ability in uncovering a pattern or correlation underlying data. Learning-based approaches can be used to extract categorical features from numerical sensor data; for example, learning network surges or abnormal VM power consumption from sensor data. They can learn correlations between a combination of relevant categorical or numerical sensor data and situations; for example, learning the pattern of how services consume memory resources when they perform a certain activity or run a specific method.

Learning-based approaches have a stronger capability to resolve uncertainty by training with the real-world data that involves noise. These approaches not only learn associations between sensor data and situations, but also the effect that the uncertainty of sensor data has on the associations. For example, the conditional probabilities learned in naïve Bayes includes the reliability of sensor data as well as the contribution of the characterised sensor data in identifying a situation (Ye et al. 2012).

2.4 Event Processing in Ubiquitous Environments

The first version of SDM focuses on providing detection capabilities for situations that are few and can be modelled manually. Hence, we follow a specification-based approach. Specifically, we will follow a complex event processing approach for modelling and detecting situations. The reason for this decision is the maturity of the approach as well as the availability of highly capable event processing engines that can cope with the veracity and volume and events that the PrEstoCloud infrastructure and associated applications will generate.

Dayarathna and Srinath (2018) surveyed recent advancements in event processing in depth by classifying the broad area of EP into three main areas of focus: EP use cases, EP system architectures, and EP open research topics. In another recent survey, Flouris et al. (2015) analysed adaptive CEP (i.e., alter query

execution plan at runtime), CEP in distributed settings, and CEP with imprecisions in the data. Moreover, the survey made by Gaber et al. (2014) concerned stream processing on ubiquitous environments.

A few notable surveys were conducted on the algorithmic aspects of stream processing. Stream clustering has been an area of great importance. The work done by Silva et al. (2013) is one of the earliest examples of this category of surveys. A second example is the work done by Aggarwal (2013). The work by Amini et al. (2014) is the third example of such surveys. The work by Hirzel et al. (2014) is an example of a survey conducted on the performance optimization techniques of data stream processing.

Dayarathna and Srinath (2018) categorize software tools for EP under three subtypes: Event Processing Platforms, Distributed Stream Computing Platforms, and CEP libraries (i.e., CEP engines). Event Processing Platforms are types of ESP software that provide high-level programming models such as expressive event processing languages (EPLs) and built-in functions for event filtering, correlation, and abstraction. Distributed Stream Computing are platforms that provide explicit support for distribution of computation across multiple nodes in a computer cluster. The CEP library (we also use the term CEP engine interchangeably) in general is a software component that especially focuses on detection of complex events. In the following sub-sections, we briefly present recent advances in CEP engines as well as distributed stream computing platforms.

Distributed Stream Computing Platforms

Dayarathna and Srinath (2018) distinguish two main variations of Distributed Stream Computing Platforms: per-event processing and microbatching. Per-event processing EP systems treat each and every event that they receive individually. They provide very low latencies while low throughput compared to their counterpart (microbatching). Example systems that implement per-event processing include S4, Storm, and Flink. Batched stream processing systems provide high throughput but introduce relatively high latency. Spark Streaming and Storm Trident are examples for systems that follow microbatching. Spark Streaming, Apache Flink and Apache Apex are examples of Distributed Stream Computing Platforms that have both batch and stream processing capabilities. A feature comparison of such platforms is provided by Dayarathna and Srinath (2018). Notable, only Spark Streaming, Storm, and Flink are equipped with an SQL-like query language.

Complex Event Processing Systems

The CEP system should be able to identify meaningful patterns, relationships, and data abstractions among apparently unrelated events, and fire a response immediately. CEP systems are lightweight components. Several notable CEP systems have appeared recently in both industry and academia. Esper, Drools, Siddhi, ruleCore and Cayuga are some of the notable CEP systems that have appeared recently. A comparison of the features and scalability of Epser, Siddhi, ruleCore and Cayuga which has been performed by Dayarathna and Srinath (2018) identified Siddhi as the best CEP system, especially in terms of scalability. The comparison does not include, unfortunately, the widely adopted Drools system. This was one of the reasons that we opted to test and compare Drools, as explained in Section 4.

Context-Aware Event Processing

Context has become a significant abstraction for modelling the EP (Etzion et al. 2011). Event context and the context awareness are two very important factors in IoT applications. Three key uses of the context in EP applications can be summarized as temporal context, spatial/segment context, and state-oriented context (Dayarathna and Srinath, 2018). In temporal context, the stream is divided into windows and the operations are defined in terms of the processing done on the events stored in the window. Spatial/segment-oriented context allows for assigning related events to different context partitions. The EP agent can be active in some contexts and inactive in others, which is called state-oriented context. Akbar et al. (2015) described a context-aware method to extract and analyse high-level knowledge from data streams. The proposed approach has been implemented on µCEP, which is a lightweight CEP that runs on embedded devices. Similarly, Wang and Cao (2012) described a high-performance context-aware CEP architecture and method for the IoT. They modelled the context as fuzzy ontology that supports linguistic variables and uncertainty in event queries.

3. Situation Metamodel & Situation Detection Approach

3.1 Situation Metamodel

In the PrEstoCloud project, we follow an event-based approach for situation modelling and detection. We consider sensor data or event encompassing raw (or minimally processed) data retrieved from both physical sensors and 'virtual' sensors observing systems, services and applications such as network traffic. These data are used to form context – the environment in which the system operates (D3.5 "Mobile Context Analyser" reports on how we handle it in the PrEstoCloud project), and situations, which are considered as an abstraction of the events occurring in the real world.

We define a situation as an external semantic interpretation of events. Interpretation means that situations assign meanings to events; external means that the interpretation is performed from the perspective of applications, rather than from events; semantic means that the interpretation assigns meaning on events based on structures and relationships within the same type of events and between different types of events (Ye et al. 2012). The latter part is achieved by the combination with the output of the Mobile Context Analyzer, MCA (D3.5) to be reported in the deliverable D5.6 "Resources Adaptation & Data-intensive Application Recommenders". A situation can uncover meaningful correlations between events, labelling them with a descriptive name. The descriptive name can be called a descriptive definition of a situation, which is about how a human defines a state of affairs in reality.

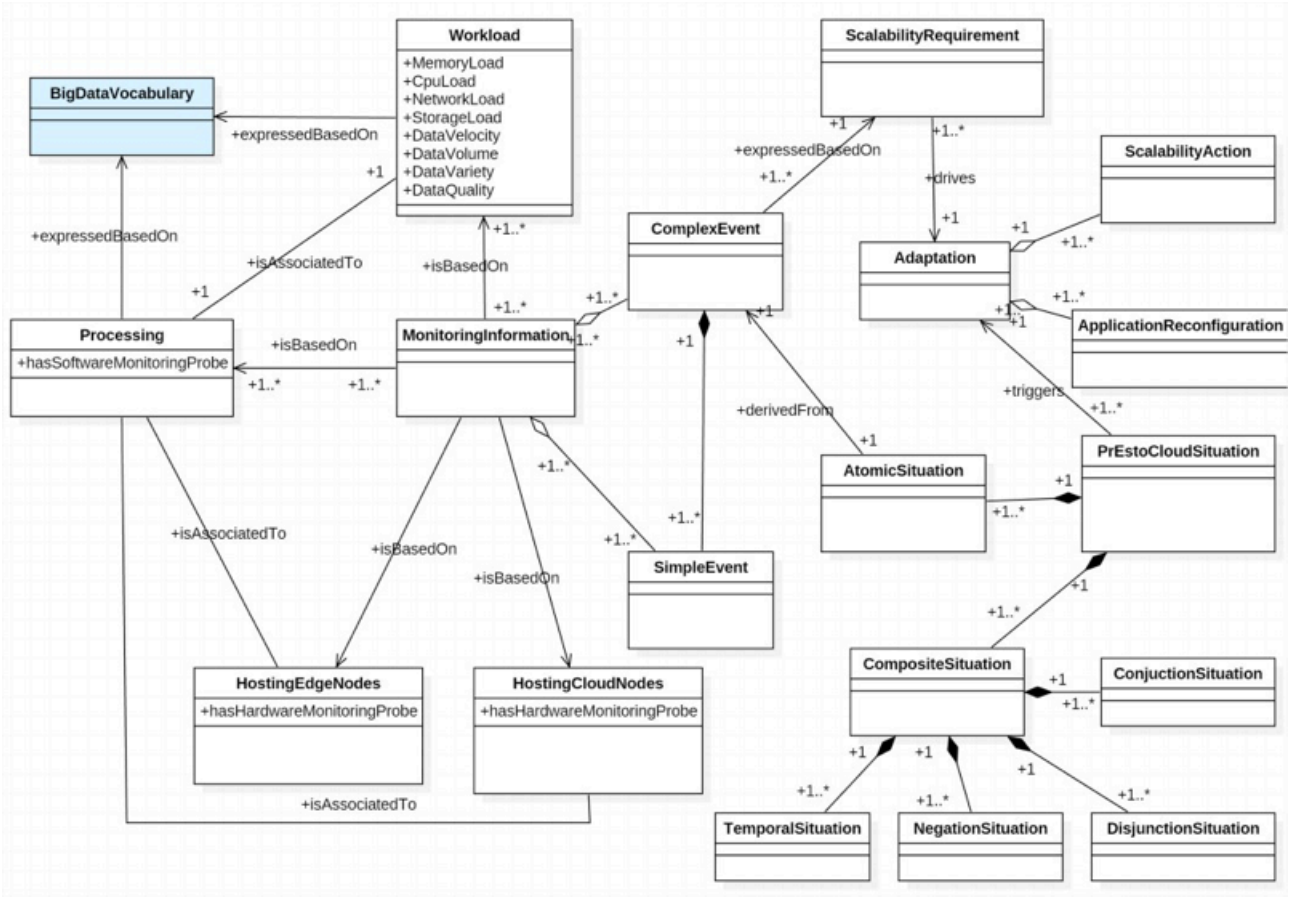

Figure 2. PrEstoCloud Situation Metamodel

In Figure 2, we describe the PrEstoCloud Situation Metamodel that captures the concepts and artefacts based on which the SDM will be able to detect meaningful PrEstoCloud situations. Such situations may reveal impending failures or even opportunities for increasing the performance of the deployed Big Data intensive applications over multi-cloud and edge resources. Specifically, these detected situations will be used by the Resources Adaptation Recommender along with the derived context and any workload predictions available, coming from other PrEstoCloud Meta-Management Layer components in order to recommend reconfigurations concerning the application fragments used or adaptations on the employed

cloud and edge resources. Although, metamodeling refers to the analysis and development of a number of rules and constraints, applicable for modelling a predefined class of problems, here we use the term Metamodel to describe the frame of concepts and their associations that should be followed for implementing SDM's core capabilities.

According to the PrEstoCloud approach the *PrEstoCloudSituation* comprise *AtomicSituation* and *CompositeSituation* (Figure 2). An *AtomicSituation* represents any basic situation whose value is directly derived from the value of a *ComplexEvent.* A *ComplexEvent* is composed of *SimpleEvents* (e.g. raw incoming events) and expresses a ScalabilityRequirement (e.g. if RAM >80% and CPU > 60% for at least 5 minutes…) that should drive the Adaptation of the big data intensive application according to a ScalabilityAction (e.g. … then scale horizontally).

The *CompositeSituation* represents complicated situations pertained to the logical composition and temporal composition of *AtomicSituations*. The logical composition over other situations refer to the *ConjunctionSituation* (i.e. combining two or more AtomicSituations using the logical AND operator), *DisjunctionSituation* (i.e. combining two or more AtomicSituations using the logical OR operator), and *NegationSituation* (i.e. combining two or more AtomicSituations using the logical NOT operator); the temporal composition can be implemented using the *TemporalSituation* that describes certain time-related dependencies or sequence associations between two or more AtomicSituations. A situation may occur before, or after another situation, or interleave with another situation.

A *CompositeSituatio*n can be decomposed into a set of smaller situations, which is a typical composition relation between situations. For example, a 'Cold VM migrating' situation is composed of a 'Relocating configuration and storage files' situation, a 'Moving VM to new host' situation and a 'Powering off VM' situation. According to our metamodel aggregating *SimpleEvents* and *ComplexEvents we acquire the related MonitoringInformation* which is necessary for checking the health status and QoS of both the deployed big data intensive application and the underlying multi-cloud and edge resources. Thus, all the *MonitoringInformation* is based on the *Processing, HostingEdgeNodes, HostingCloudNodes and* current *Workload* detected through the appropriate software, hardware and workload related monitoring probes, respectively. Both *Processing* and *Workload* are expressed based on the *BigDataVocabulary* in order to abstractly map types of big-data streams to big data processing services types revealing their importance for the detected *PrEstoCloudSituations. T*he *BigDataVocabulary* was discussed in the deliverable D2.5 and refers to an external class that includes all the concepts and properties to be used for describing Big Data characteristics that should be considered for making Big Data application placement decisions.

3.2 Situation Detection Approach

In industry, cloud platforms that support automatic or semi-automatic adaptation use event driven rules in order to decide the time of adaptation. Amazon AWS, for example, provides auto-scaling services (Amazon 2018 that trigger adaptation actions based on user-configurable rules that are evaluated in real-time using internal or external monitoring infrastructure. Kubernetes (2018) provides auto-scaling capabilities based on internal or external metrics. In Google Cloud (2018), users can specify a target CPU utilisation for a group of (service) instances, the platform will try to maintain it by scaling it up or down. OpenStack (2018) also supports auto-scaling policies by deploying the Heat service. Autoscaling in OpenstackHeat is triggered by Alarms produced by the telemetry service (Ceilometer). In research approaches situation detection has been performed with several advanced techniques as described in Chapter 2 of this document.

Since, the PrEstoCloud environment combines multi-cloud and edge resources, we need a mechanism to detect situations from heterogeneous devices and services with very different capabilities in terms of computational resources and provide the ability to control and customize the execution environment. For example edge devices may have very low computational resources or a very restricted (due to security reasons) environment for custom applications. Very often those devices have low network bandwidth, unpredictable disconnections from the network and data transmission spikes that are caused by external events (such as social events, weather conditions or other). In this environment we need infrastructure and mechanisms for data-driven event detection. Therefore, we opted for an approach that relies on complex event processing technologies, which are capable of processing in real-time a large number of events

generated by a variety of distributed cloud and edge computing resources as well as other data generating sensors. A complex event is an event derived from a group of events using either aggregation of derivation functions. Information enclosed in a set of related events can be represented (i.e., summarized) through such a complex event.

Arguably, situation detection in a big-data generating and distributed environment such as PrEstoCloud needs to take care of network bandwidth consumption. Similarly, to commercial systems, it is important to support parts of the situation detection at the edge. For example Cisco routers with Cisco IOS® XE (2018) are able to run KVM virtual machines or LXC containers. In many use cases, it would be desirable to deploy situation detection services near the edge or at the extreme edge. In this way we could lower resource consumption in the cloud, limit the required bandwidth or process events from edge devices with lower latency and lower rates of event loss (due to network outages at the extreme edge). So, it is crucial for the situation detection mechanism to have low computation resource consumption (memory and CPU) and ability to efficiently distribute and process events in multiple stages.

The approach that we propose for realising situation detection in PrEstoCloud has the following characteristics:

- A homogenous solution for data intensive and data-driven situation detection at the edge or near the edge and in the cloud.

- Components (containers) that can be deployed with existing cloud orchestration technologies (such as Kubernetes (https://kubernetes.io/), Rancher (https://rancher.com/), Ansible (https://www.ansible.com/) or the technologies that PrEstoCloud develops.

- A distributed hierarchical approach for event-driven and rule-based situation detection with complex event patterns.

4. Architecture and Implementation

This section presents the conceptual architecture of SDM as well as two concrete instantiations of the conceptual architecture. We also present a walkthrough of how to deploy SDM.

4.1 Situation Detection Mechanism Architecture

Figure 3 depicts the conceptual architecture of SDM. The architecture includes the PrEstoCloud distributed broker engine (D3.1), which serves as the event messaging and routing backbone in PrEstoCloud. Metrics from the cloud infrastructure (physical and virtual machines, containers, applications, services, etc.) and edge devices (mobile phones, drones, IoT devices) are published as events to the Broker in specific topics. One or more SDM service instances subscribe to the desired topics and receive streams of events that contain up-to-date information about the current state of those entities (e.g. used RAM, CPU consumption, disk I/O, requests per second, etc.).

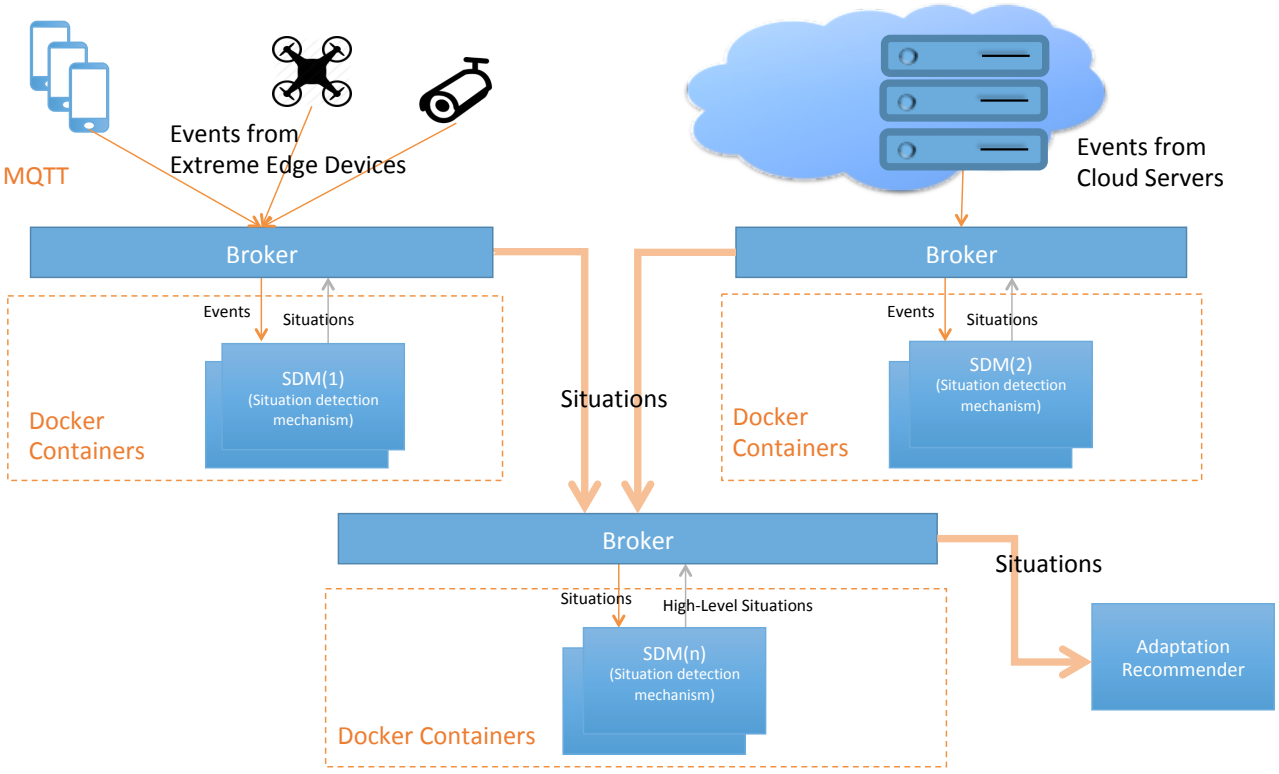

Figure 3 - PrEstoCloud Situation Detection Architecture

The SDM instances process these events based on the supplied CEP rules which are defined in order to detect interesting situations. Several SDM instances can be used in parallel or in series in order to process the incoming event streams. High level situations can be detected by processing low-level situations from many SDM instances. The final consumer of the situations that are detected by the SDM is be the PrEstoCloud Adaptation Recommender, which will be delivered in the upcoming deliverable D5.5.

4.2 Implementation

The Situation Detection Mechanism has been developed in Java. It can be deployed as a set of Docker containers. The Java source code and the Docker configuration files are uploaded on gitlab:

https://gitlab.com/prestocloud-project/situation-detection-mechanism.git .

SDM receives events from the Communication and Message Broker, which is implemented by a RabbitMQ container. RabbitMQ, which has been used to implement the PrEstoCloud Communication Broker (D3.1), supports different transport and messaging protocols such as the different versions of AMPQ (0-9-1, 0-8-1, 1.0), MQTT, STOMP, JSON-RPC over HTTP and Web-STOMP. A Logstash Docker container can be used for the preprocessing of the incoming events. Logstash can subscribe to RabbitMQ topics, process incoming events, and publish the events back to RabbitMQ (in different topics). In our use cases some very common tasks assigned to Logstash included the transformation of the event payload (from CSV to JSON, or from an initial JSON format to a JSON format with additional fields) and the mapping of different event types to new topics based on which were extracted from the event payload. Logstash can be used also as an input interface that can support additional messaging protocols like UDP, Graphite, UDP, XMPP or Beats with different input plugins (https://www.elastic.co/guide/en/logstash/current/input-plugins.html).

After pre-processing, the input events are published through RabbitMQ to a Docker container that embeds and runs a CEP engine. We demonstrate that the implementation of the proposed approach can be agnostic to the CEP engine used. This allows the PrEstoCloud adopter to use the engine of his or her choice based on the characteristics of the monitored big data-intensive application and her expertise with event processing software. There are many CEP libraries that can be used at this stage with SDM such as Drools, WSO2 Siddhi and Esper. The basic functional requirements from the CEP Docker container include the a) ability to read all the necessary configuration (input parameters and topics, output parameters and topics, rules) from files or environment variables, b) the ability to consume events from RabbitMQ in JSON format, c) the ability to produce new events that denote the detection of a situation, d) the ability to publish events to RabbitMQ in JSON format , d) the ability to read, process, and produce different JSON event formats dynamically by changing only the rule file (without the need for example to write and compile new code in Java or any other programming language in order to create new event models).

It may be very critical for the selected CEP library to present the lowest possible computing resource consumption, latency or footprint as well as the maximum throughput. This depends on the environment that a CEP container is deployed (cloud, edge, extreme edge), the type of situations that it will be instructed to detect, and the expected input workload (in terms of events per second). Important factors for the selection of a CEP library are also the expressivity of the supported rule description language and the complexity from the user's point of view for the implementation of the required CEP patterns.

In this first version of SDM, we have selected and compared two different CEP libraries, the WSO2 Siddhi 4 and the Drools CEP engine. Both are distributed with open source licenses. Drools is a widely used rule engine and CEP library, which PrEstoCloud partners have successfully used in previous projects and have gained substantial experience. Siddhhi on the other hand is newer and is reported to offer state of the art performance when compared to other libraries including Esper (Dayrathna and Perera, 2018).

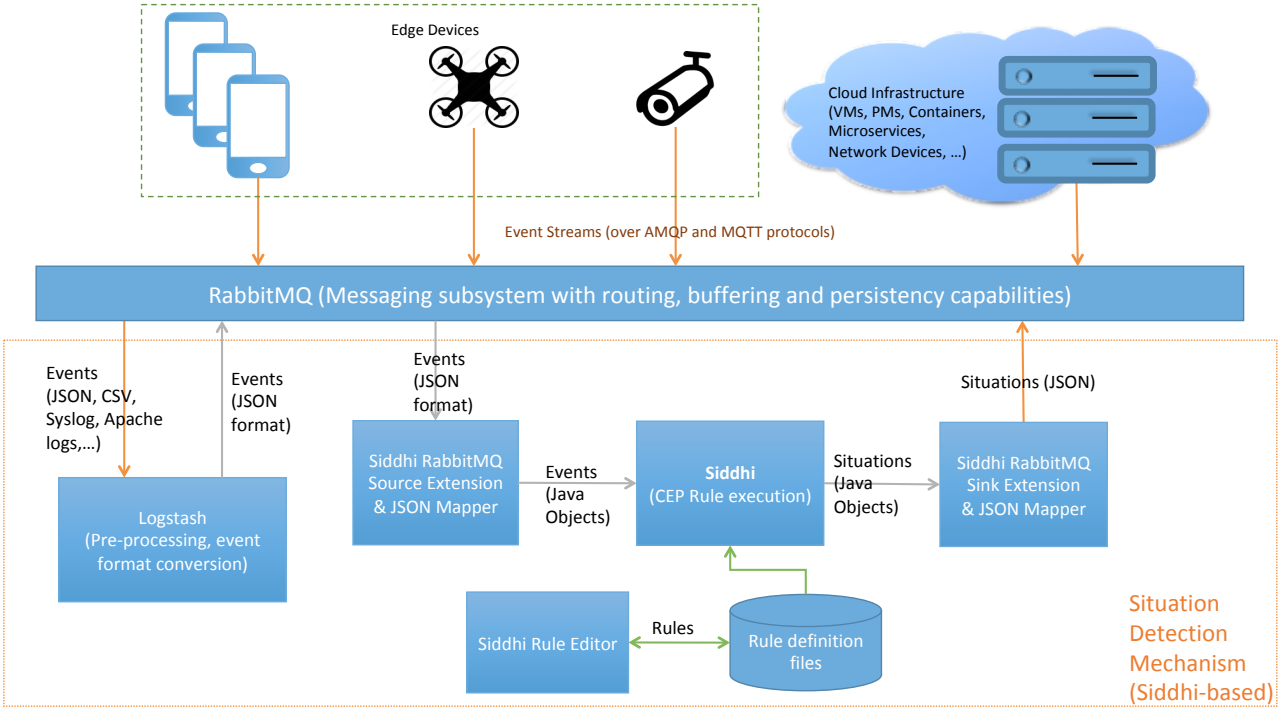

Figure 4. Situation Detection Mechanism (Siddhi-based)

Siddhi CEP patterns are implemented in the SQL-like rule language SiddhiQL. It supports out-of-the box extensions that can interface directly with RabbitMQ (WSO2 2018) and map JSON events to event streams (WSO2 2018b). In the following figure (Figure 4) we can see a grounded architecture diagram of SDM that uses Siddhi, Logstash and RabbitMQ.

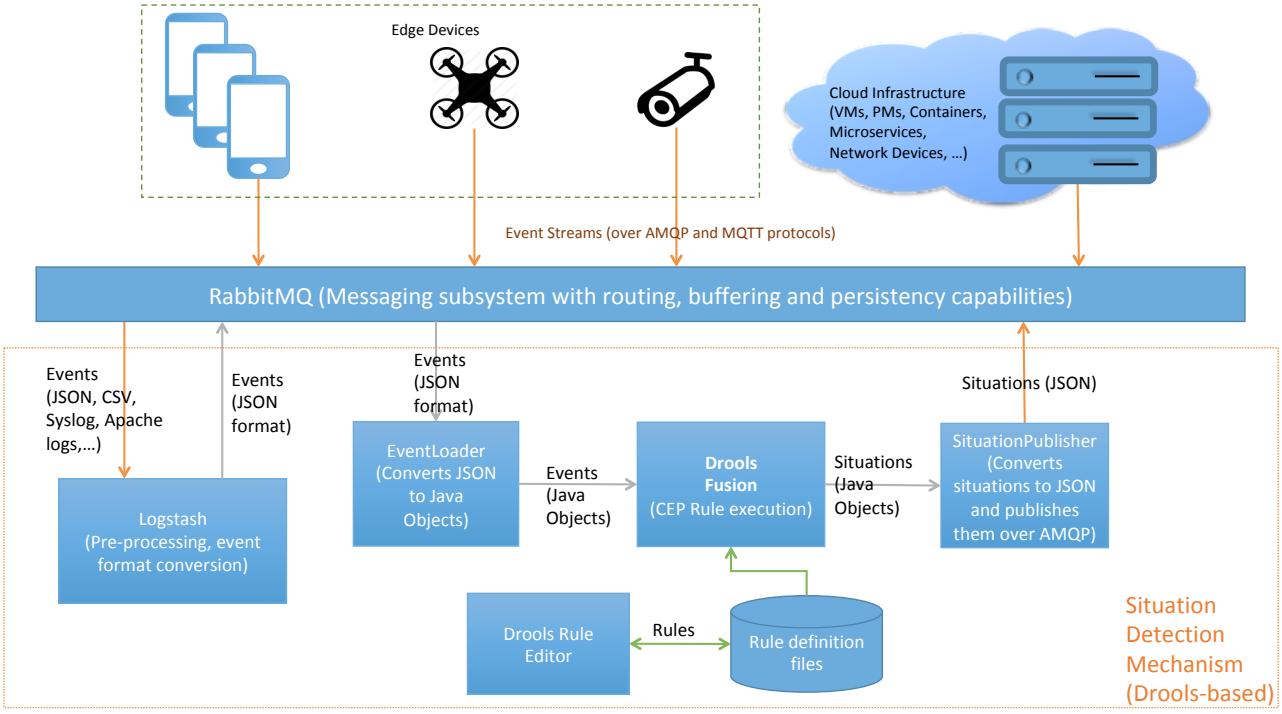

Figure 5 - Situation Detection Mechanism (Drools-based)

Figure 5 depicts an implementation of SDM with Drools as the CEP container. Drools was initially developed as a rule engine. In Drools, rules and facts constitute a knowledge base. Rules are present in the production memory and the facts are kept in a database called working memory, which maintains current system knowledge. There is an Inference Engine based on Charles Forgy's Rete Algorithm, which efficiently matches the facts from working memory to conditions of the rules in the production memory. Knowledge based systems require a conflict resolution is required when there are multiple rules on the agenda. As firing a rule may have side effects on working memory, the rule engine needs to know in what order the rules should fire (for instance, firing 'ruleA' may cause 'ruleB' to be removed from the agenda). The default conflict resolution strategies employed by Drools are: Salience and LIFO (last in, first out), (JBoss 2013). Later, new functionalities were added allowing Drools to perform complex event processing. The CEP rules in Drools CEP are Event Condition Action (ECA)-style rules.

In the case of the Drools based implementation of SDM, we have implemented in Java two additional classes. The EventLoader class subscribes to RabbitMQ topics and converts input JSON events to Java Objects. The SituationPublisher converts output situation events from Java to JSON and publishes them to the configured RabbitMQ topic. In both these implementations, we have used the RabbitMQ Java Client Library (https://www.rabbitmq.com/java-client.html) and the google-gson JSON serialization/deserialization library (https://github.com/google/gson). The EventLoader class converts JSON events to instances of the JsonObject class. The rule language of Drools CEP allow to declaratively define new event types and access JsonObject fields , as shown in the following example :

```
declare JsonObject
  @role ( event )
end

declare CPUEvent
  @role( event )
  @timestamp( edate )
  edate: Date @key
  cpu: Double
end

rule "parseCPUEvent"
when
   $event : JsonObject() from entry-point INPUT
then
   SimpleDateFormat sdf = new SimpleDateFormat("yy.MM.dd'_'HH.mm.ss");
   Date edate = sdf.parse( $event.get("ts").getAsString() );
   Integer cpu = $event.get("cpu_load_pct").getAsDouble();
   CPUEvent $c = new CPUEvent(edate, cpu);
   insert($c);
end
```

In this way we are able to read and process any JSON-formatted event with Drools CEP without the need to write and compile new code in Java for each use case. Drools CEP on startup reads the rule file which can contain expressions in a scripting language that is able to call Java Object methods (as we have shown above), parses it and converts it in runtime in Java Objects.

With CEP engines we can detect situations with complex temporal event patterns. For example we can detect if after CPU utilization was detected above 90%, it was detected again in a period of ten to twenty seconds after the first event.

```
rule "cpu > 90.0 again after 10 to 20 sec"
when
   $e2 : CPUEvent( cpu > 90.0 )
   $e1 : CPUEvent( cpu > 90.0 , this != $e2, this after[ 10s, 20s ] $e2 )
```

```
then
   log("TEMPORAL1: cpu>90.0 twice after 10s-20s triggered by : " + $e1 + " & " +
$e2) ;
end
```

Or we can detect that in a time period that spans from 10 seconds before to ten seconds after the CPU utilization was above 80% an event that denotes that used memory was also above 80%.

```
rule "memory and cpu events coincide"
when
   $e2 : CPUEvent( cpu > 80.0 )
   $e1 : MemoryEvent( used > 80.0 , this coincides [ 10s, 10s ] $e2 )
then
   log("TEMPORAL2: cpu>80.0 and mem>80.0 coincide: " + $e1 + " & " + $e2) ;
end
```

4.3 Execution Walkthrough

In this section, we describe how SDM can be used in order to detect situations from one or more services. For the purposes of this walkthrough, we assume that services run a Netdata metric collection agent (http://my-netdata.io/). Netdata is an agent that can be installed on systems that run different flavours of Linux or MacOS. It can collect over 5000 different metrics from the operating system and from a plethora of applications (databases, application servers, web servers, etc.). Netdata can be configured to export the collected metrics to a backend (https://github.com/firehol/netdata/wiki/netdata-backends). If we configure the Graphite type backend we can send the metrics to a Logstash service over the Graphite protocol (Figure 6).

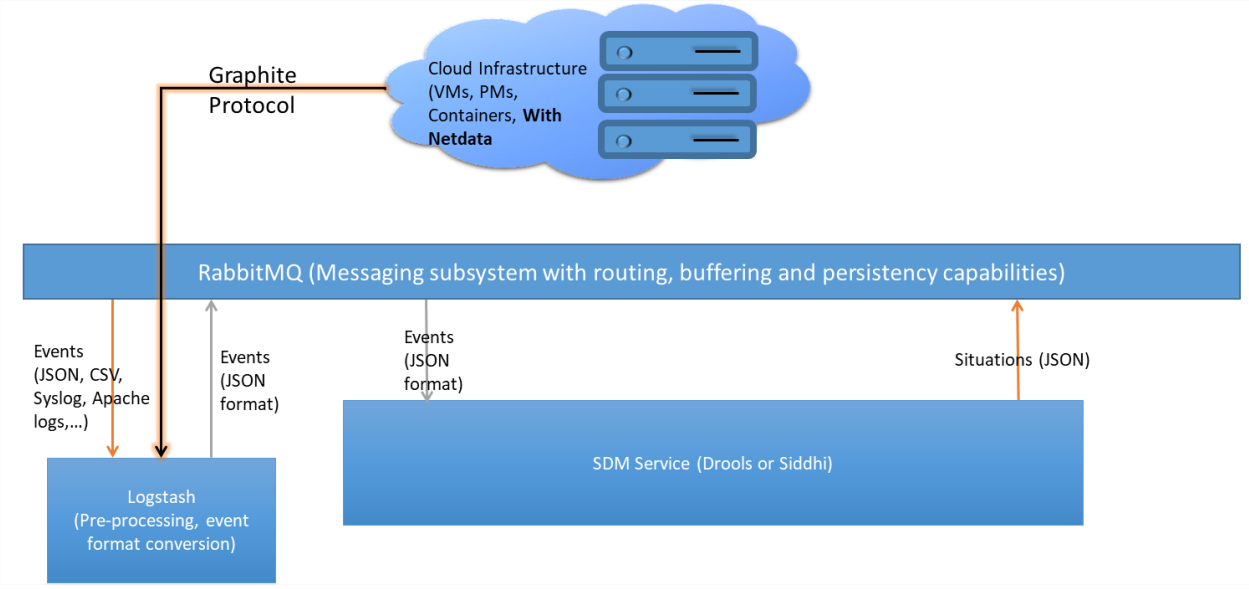

Figure 6 – Situation Detection Mechanism processing messages from Netdata

The graphite protocol encodes metrics in a plain text format which contains on metric per line along with the value of the metric and a timestamp separated by spaces, as depicted bellow:

```
netdata.mysrv1.system.cpu.irq  1.7481766  1525950748
```

```
netdata.mysrv1.system.cpu.user  30.8244400  1525950748
netdata.mysrv1.system.cpu.system  5.7389183  1525950748
netdata.mysrv1.system.cpu.nice  0.1666761  1525950748
netdata.mysrv1.system.cpu.iowait  14.7725093  1525950748
netdata.mysrv1.system.cpu.idle  46.7492800  1525950748
netdata.mysrv1.system.cpu.guest_nice  0.0000000  1525950751
netdata.mysrv1.system.cpu.guest  0.0000000  1525950751
```

Logstash with the appropriate pipeline configuration (example in Appendix 8.2) can convert the metrics from Netdata to JSON and publish them to RabbitMQ topics that are defined dynamically by the Netdata metric name. For example Logstash will publish the metric with name "netdata.mysrv1.system.cpu.user" to a RabbitMQ topic with the same name and so on. An SDM instance running Siddhi or Drools can be configured to subscribe to these topics (in Appendix **8.3** we show a Siddhi rule file that subscribes to two RabbitMQ topics).

In the remaining part of this section we present the steps needed for starting and running SDM. Docker-compose configures and runs one RabbitMQ service, one Logstash service, one Siddhi CEP engine and multiple instances of Linux containers that run Netdata. We are using the (container) scaling capabilities of Docker-Compose to increase or decrease the Docker containers that run Netdata. The Docker-compose file is included in Appendix 8.1. An example of the Netdata backend configuration section of its configuration file (netdata.conf) is included in Appendix 8.4. It shows how to configure a graphite backend with metric filtering.

Step 1: Prepare the docker-compose images

With the following commands we can build and prepare all the Docker containers.

```
$ docker-compose build

Successfully built 663056462f28
Successfully tagged sdmubidemo_netdata:latest


$ touch sdmsrv2/resources.netdata/RULES.siddhi
```

Step 2: Start rabbitmq service

First, we start the rabbitmq service. All other services depend on this.

```
$ docker-compose up -d rabbitmq
Creating network "sdmubidemo_esnet" with the default driver
Creating sdmubidemo_rabbitmq_1 ... done
```

If we check the logs we can confirm that rabbitmq was started successfully when we see the following lines:

```
$docker-compose logs -f rabbitmq
rabbitmq_1  | 2018-05-18 20:39:42.293 [info] <0.5.0> Server startup complete; 3 plugins
started.
rabbitmq_1  |  * rabbitmq_management
rabbitmq_1  |  * rabbitmq_management_agent
rabbitmq_1  |  * rabbitmq_web_dispatch
```

Step 3: Start the logstash service

Logstash subdcribes to RabbitMQ topics when it starts. With the following command we can start the logstash service after we check that rabbitmq has started.

```
$ docker-compose up -d logstash
sdmubidemo_rabbitmq_1 is up-to-date
Creating sdmubidemo_logstash_1 ... done
```

If we check the logs we can confirm that logstash was started successfully when we see the following lines .

```
$docker-compose logs -f logstash

logstash_1  | [2018-05-18T20:42:14,183][INFO ][logstash.runner          ] Starting
Logstash {"logstash.version"=>"6.2.0"}
logstash_1  | [2018-05-18T20:42:14,706][INFO ][logstash.agent           ] Successfully
started Logstash API endpoint {:port=>9600}
logstash_1  | [2018-05-18T20:42:18,799][INFO ][logstash.pipeline        ] Starting
pipeline {:pipeline_id=>"main", "pipeline.workers"=>2, "pipeline.batch.size"=>125,
"pipeline.batch.delay"=>50}
logstash_1  | [2018-05-18T20:42:19,317][INFO ][logstash.outputs.rabbitmq] Connected to
RabbitMQ at
logstash_1  | [2018-05-18T20:42:19,798][INFO ][logstash.inputs.graphite ] Automatically
switching from plain to line codec {:plugin=>"graphite"}
logstash_1  | [2018-05-18T20:42:19,854][INFO ][logstash.inputs.graphite ] Starting tcp
input listener {:address=>"0.0.0.0:5000", :ssl_enable=>"false"}
logstash_1  | [2018-05-18T20:42:20,084][INFO ][logstash.pipeline        ] Pipeline
started succesfully {:pipeline_id=>"main", :thread=>"#<Thread:0x793fbce6 run>"}
logstash_1  | [2018-05-18T20:42:20,220][INFO ][logstash.agent           ] Pipelines
running {:count=>1, :pipelines=>["main"]}
```

Step 4: Start the sdmsrv2 service (Siddhi CEP engine)

The sdmsrv2 service executes a Siddhi instance. When we see the message "CEP STARTED NETDATA INPUT" the service has started successfully.

```
$ docker-compose up sdmsrv2
sdmubidemo_rabbitmq_1 is up-to-date
sdmubidemo_logstash_1 is up-to-date
Creating sdmubidemo_sdmsrv2_1 ... done
Attaching to sdmubidemo_sdmsrv2_1
sdmsrv2_1   | [INFO] Scanning for projects...
sdmsrv2_1   | [INFO]
sdmsrv2_1   | [INFO] ------------------------< gr.iccs.presto:sdmsrv2 >------------------
-----
sdmsrv2_1   | [INFO] Building sdmsrv2 1.0-SNAPSHOT
sdmsrv2_1   | [INFO] --------------------------------[ jar ]---------------------------
-----


sdmsrv2_1   | Created inputstream from : RULES.siddhi
sdmsrv2_1   | copy ok
sdmsrv2_1   | LOADED RULES.siddhi .
sdmsrv2_1   | 23:44:54 : CEP STARTED NETDATA INPUT
```

Step 5a: Start 2 Netdata containers (event producers)

The service netdata simulates a scalable micro-service which sends metrics about its state to SDM with a Netdata agent. With docker-compose, we can scale this service up and down by increasing or decreasing the number of containers that execute it. The supplied SDM rules, in Siddhi rule format, count every 10 seconds the number of distinct netdata instances by processing the events that they publish to RabbitMQ through Logstash. As shown in the following example if we start 2 netdata service instances (with the docker-compose option –scale) we must see after some seconds the message "HOSTCNT : DISTINCT HOSTS LAST 10s : 2".

```
$ docker-compose up -d --scale netdata=2 netdata
sdmubidemo_rabbitmq_1 is up-to-date
```

```
The "netdata" service specifies a port on the host. If multiple containers for this
service are created on a single host, the port will clash.
Creating sdmubidemo_netdata_1 ... done
Creating sdmubidemo_netdata_2 ... done


sdmsrv2_1    | -------------------------------------------------------
sdmsrv2_1    |  T E S T S
sdmsrv2_1    | -------------------------------------------------------
sdmsrv2_1    | Running gr.iccs.presto.AppTest
sdmsrv2_1    | Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.014 sec
sdmsrv2_1    |
sdmsrv2_1    | Results :
sdmsrv2_1    |
sdmsrv2_1    | Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
sdmsrv2_1    |
sdmsrv2_1    | [INFO]
sdmsrv2_1    | [INFO] >>> exec-maven-plugin:1.1.1:java (default) > validate @ sdmsrv2 >>>
sdmsrv2_1    | [INFO]
sdmsrv2_1    | [INFO] <<< exec-maven-plugin:1.1.1:java (default) < validate @ sdmsrv2 <<<
sdmsrv2_1    | [INFO]
sdmsrv2_1    | [INFO]
sdmsrv2_1    | [INFO] --- exec-maven-plugin:1.1.1:java (default) @ sdmsrv2 ---
sdmsrv2_1    | Created inputstream from : RULES.siddhi
sdmsrv2_1    | copy ok
sdmsrv2_1    | LOADED RULES.siddhi .
sdmsrv2_1    | 00:14:30 : CEP STARTED NETDATA INPUT
sdmsrv2_1    | 00:14:40 : HOSTCNT : DISTINCT HOSTS LAST 10s : 2
sdmsrv2_1    | 00:14:40 : ALLCPUAVG : Average CPU of all hosts LAST 10s :
42.48388320207596
sdmsrv2_1    | 00:14:46 : HOSTCPUAVG : sdmubidemo_netdata_1.sdmubidemo_esnet: 15 sec avg
cpu.idle=65.87990093231201
sdmsrv2_1    | 00:14:46 : HOSTCPUMAX : sdmubidemo_netdata_1.sdmubidemo_esnet: 15 sec max
cpu.user=79.72789
sdmsrv2_1    | 00:14:46 : HOSTCPUAVG : sdmubidemo_netdata_2.sdmubidemo_esnet: 15 sec avg
cpu.idle=65.881844997406
sdmsrv2_1    | 00:14:46 : HOSTCPUMAX : sdmubidemo_netdata_2.sdmubidemo_esnet: 15 sec max
cpu.user=79.720535
sdmsrv2_1    | 00:14:49 : ALLCPUAVG : Average CPU of all hosts LAST 10s :
3.3208606243133545
sdmsrv2_1    | 00:14:49 : HOSTCNT : DISTINCT HOSTS LAST 10s : 2
sdmsrv2_1    | 00:15:01 : HOSTCPUMAX : sdmubidemo_netdata_2.sdmubidemo_esnet: 15 sec max
cpu.user=4.4910984
sdmsrv2_1    | 00:15:01 : HOSTCNT : DISTINCT HOSTS LAST 10s : 2
sdmsrv2_1    | 00:15:01 : HOSTCPUAVG : sdmubidemo_netdata_2.sdmubidemo_esnet: 15 sec avg
cpu.idle=95.87651672363282
sdmsrv2_1    | 00:15:01 : HOSTCPUAVG : sdmubidemo_netdata_1.sdmubidemo_esnet: 15 sec avg
cpu.idle=95.87651672363282
sdmsrv2_1    | 00:15:01 : HOSTCPUMAX : sdmubidemo_netdata_1.sdmubidemo_esnet: 15 sec max
cpu.user=4.4910984
sdmsrv2_1    | 00:15:01 : ALLCPUAVG : Average CPU of all hosts LAST 10s :
3.161834269762039
```

Step 5b: Start 8 additional Netdata containers (scale up to 10 Netdata containers)

If we scale up netdata to 10 instances (with the docker-compose option –scale), we see after some seconds the message "HOSTCNT : DISTINCT HOSTS LAST 10s : 10".

```
$ docker-compose up -d --scale netdata=10 netdata
sdmubidemo_rabbitmq_1 is up-to-date
The "netdata" service specifies a port on the host. If multiple containers for this
service are created on a single host, the port will clash.
Starting sdmubidemo_netdata_1 ... done
Starting sdmubidemo_netdata_2 ... done
Starting sdmubidemo_netdata_3 ... done
Starting sdmubidemo_netdata_4 ... done
Starting sdmubidemo_netdata_5 ... done
Creating sdmubidemo_netdata_6  ... done
Creating sdmubidemo_netdata_7  ... done
Creating sdmubidemo_netdata_8  ... done
Creating sdmubidemo_netdata_9  ... done
Creating sdmubidemo_netdata_10 ... done
```

If we check the log we can see that SDM detects after some seconds that 10 distinct hosts are sending metrics.

```
sdmsrv2_1    | 00:21:01 : HOSTCNT : DISTINCT HOSTS LAST 10s : 10
sdmsrv2_1    | 00:21:01 : ALLCPUAVG : Average CPU of all hosts LAST 10s :
3.8740233212709425
```

Step 5c: Stop 8 Netdata containers (scale down to 2)

If we scale down netdata to 2 instances (with the docker-compose option –scale), we see after some seconds the message "HOSTCNT : DISTINCT HOSTS LAST 10s : 2".

```
$ docker-compose up -d --scale netdata=2 netdata
sdmubidemo_rabbitmq_1 is up-to-date
The "netdata" service specifies a port on the host. If multiple containers for this
service are created on a single host, the port will clash.
Stopping and removing sdmubidemo_netdata_3  ... done
Stopping and removing sdmubidemo_netdata_4  ... done
Stopping and removing sdmubidemo_netdata_5  ... done
Stopping and removing sdmubidemo_netdata_6  ... done
Stopping and removing sdmubidemo_netdata_7  ... done
Stopping and removing sdmubidemo_netdata_8  ... done
Stopping and removing sdmubidemo_netdata_9  ... done
Stopping and removing sdmubidemo_netdata_10 ... done
Starting sdmubidemo_netdata_1               ... done
Starting sdmubidemo_netdata_2               ... done
```

If we check the log we can see that SDM detects after some seconds that now only 2 distinct hosts are sending metrics.

```
sdmsrv2_1    | 00:22:31 : ALLCPUAVG : Average CPU of all hosts LAST 10s :
0.7142319902777672
sdmsrv2_1    | 00:22:31 : HOSTCNT : DISTINCT HOSTS LAST 10s : 2
```

5. Evaluation

In Section 4, we described the implementation of the SDM services. We have used two different CEP engines, Drools and Siddhi, along with our custom components. In this section we will evaluate and compare these implementations of SDM. We will present two experiments. In both experiments the software and hardware configuration is the same:

- Hardware
  - A KVM Virtual Machine with 4 cores and 8GB RAM running on a server with Intel Xeon E7 @ 2.4 Ghz CPU
- Software
  - SDM services run under Ubuntu 17.10 with the following software packages installed:
    - Docker version 17.12.0-ce, build c97c6d6
    - Docker-compose version 1.19.0, build 9e633ef
    - OpenJDK Runtime Environment (build 1.8.0_171-8u171-b11-0ubuntu0.17.10.1-b11)
  - Container-based libraries:
    - Siddhi version v4.0.0 with RabbitMQ extension v1.0.14
    - Drools version 6.5.0.Final
    - RabbitMQ 3.7.5 (Docker image rabbitmq3.7.5-management)

5.1 Experiment 1 (Load-test with PerfTest)

In this experiment the RabbitMQ load-testing tool is used to generate and publish events (https://github.com/rabbitmq/rabbitmq-perf-test) to SDM services (through a RabbitMQ instance) (Figure 7). In this experiment, we use only RabbitMQ and two SDM instances, one implemented with the Drools CEP library and one implemented with the Siddhi library. With PerfTest we can select the number of event producers, the length of the period that we want to send events, the frequency with which the event producers should generate events and the payload of the events (from a list of files). The AMQP exchange name and the topic are also configurable.

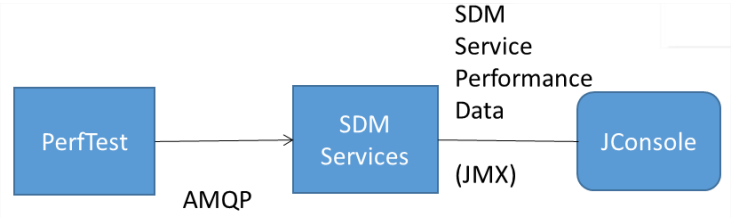

Figure 7 - Experiment 1 (Load-testing SDM with PerfTest)

With a Java Management Extensions (JMX) tool such as JConsole or VisualVM (https://visualvm.github.io/) we can monitor many metrics of Java applications. We use JMX to monitor the Drools and Siddhi version of SDM. We run (with docker-compose) one Siddhi CEP engine and one Drools CEP engine in parallel and configure them to subscribe to the same AMQP exchange and topic. In this way both CEP engines receive the same events from PerfTest.

The payload of the events is a JSON file that contains different values of two attributes named "memory" and "cpu", (without any timestamp for simplification reasons) like the following:

```
{
        "event": {
                "cpu": 45.0,
                "memory": 37.0
        }
}
```

Both Drools and Siddhi were configured to produce every 10 sec two events containing:

- The average CPU and MEMORY (during the last 10s)
- The number of MEMORY and CPU events that it received (during the last 10s)

In the following series of diagrams we can see some representative results. All diagrams have been generated using a python Jupyter notebook with the pandas 0.23 (https://pandas.pydata.org) and matplotlib (https://matplotlib.org) libraries.

First we run PerfTest for 60 seconds with increasing number of event producers that send one event per second.

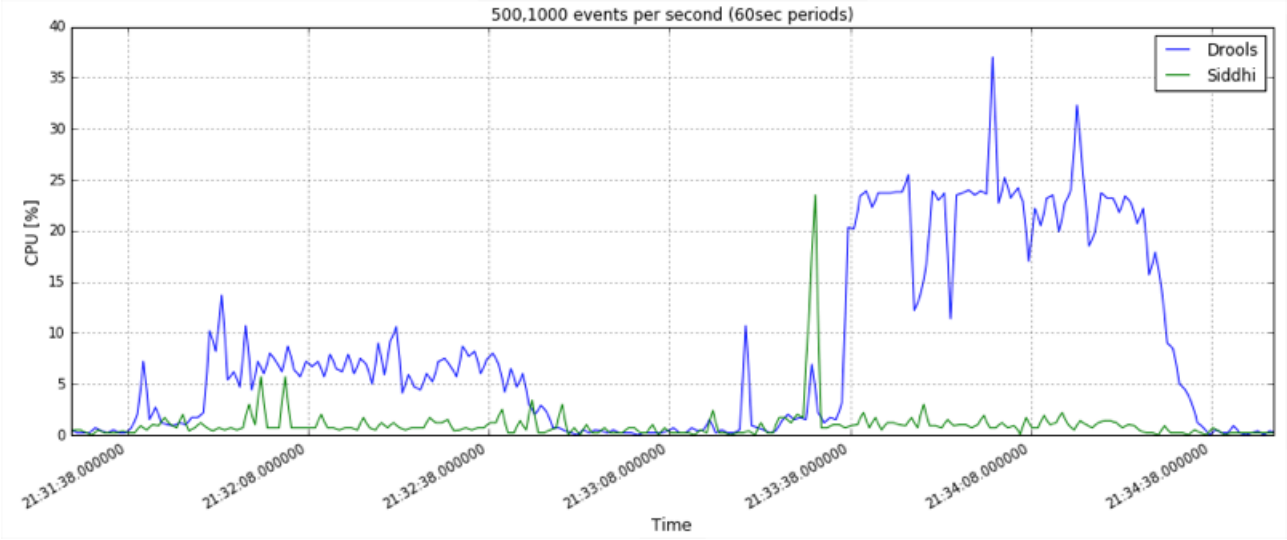

Figure 8. SDM load testing with PerfTest. CPU utilization of Drools-based implementation vs Siddhi-based implementation (500,500,1000 events/sec)

In Figure 8, we compare the CPU consumption of Drools and Siddhi while sending 500 events per second (twice) and 1000 events per second, for two consecutive periods of 60 seconds. We can clearly see that the Siddhi-based implementation of SDM has much lower total CPU utilization than the Drools-based implementation which increases in a bigger proportion as the rate of incoming events increases.

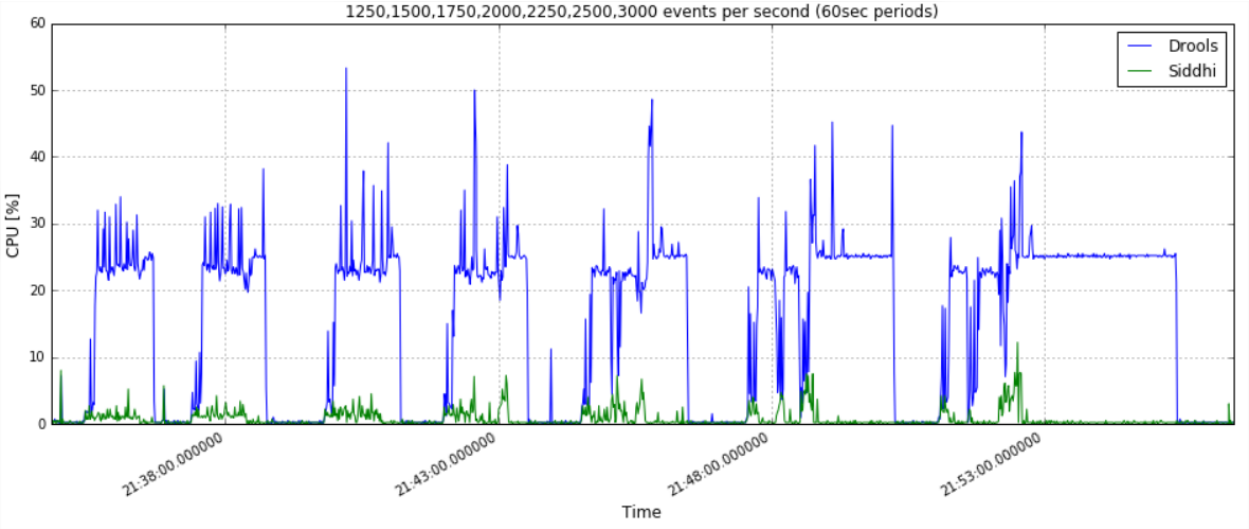

Figure 9. SDM load testing with PerfTest. CPU utilization of Drools-based implementation vs Siddhi-based implementation (1250 to 3000 events/sec)

In Figure 9, we continue the same experiment with increasing number of events per second (generated by PerfTest) : 1250, 1500, 1750, 2000, 2250, 3000. It is again clear that Siddhi has much lower CPU utilisation. It is also notable that after 1500 events per second the Drools-based implementation of SDM queues the incoming messages and continues processing an increasing number of seconds after PerfTest has finished sending events. Siddhi processes all the events in almost real-time in the above tests.

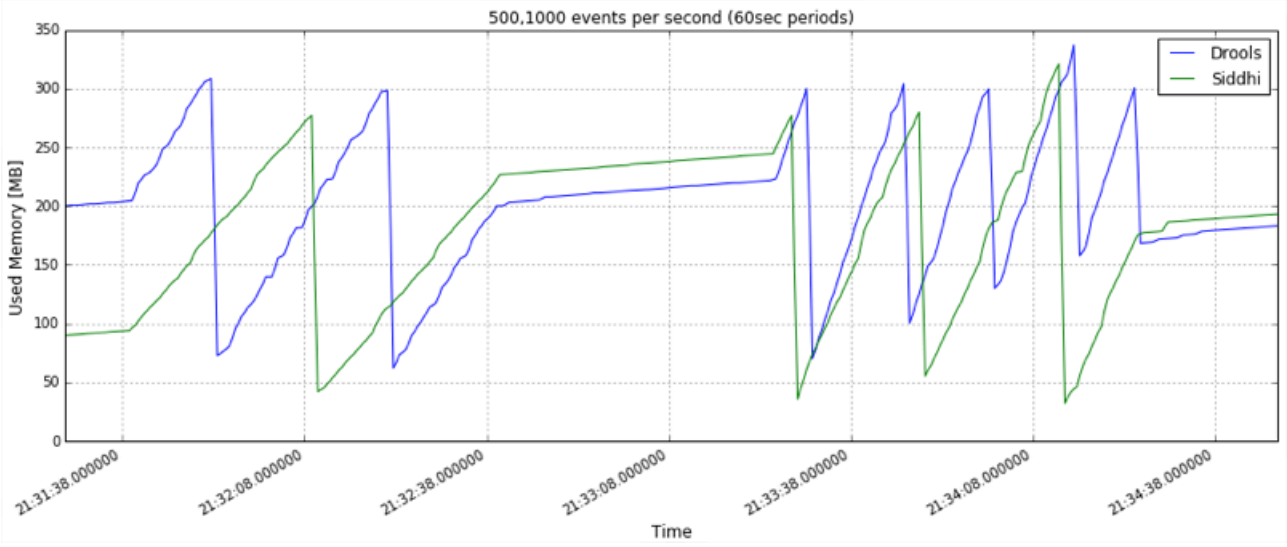

Figure 10. SDM load testing with PerfTest. Used memory of Drools-based implementation vs Siddhi-based implementation (500,500,1000 events/sec)

In Figure 10, we can see the memory consumption of Drools and Siddhi when sending 500 events per second (twice) and 1000 events per second. In these event rates, both CEP engines have similar memory consumption.

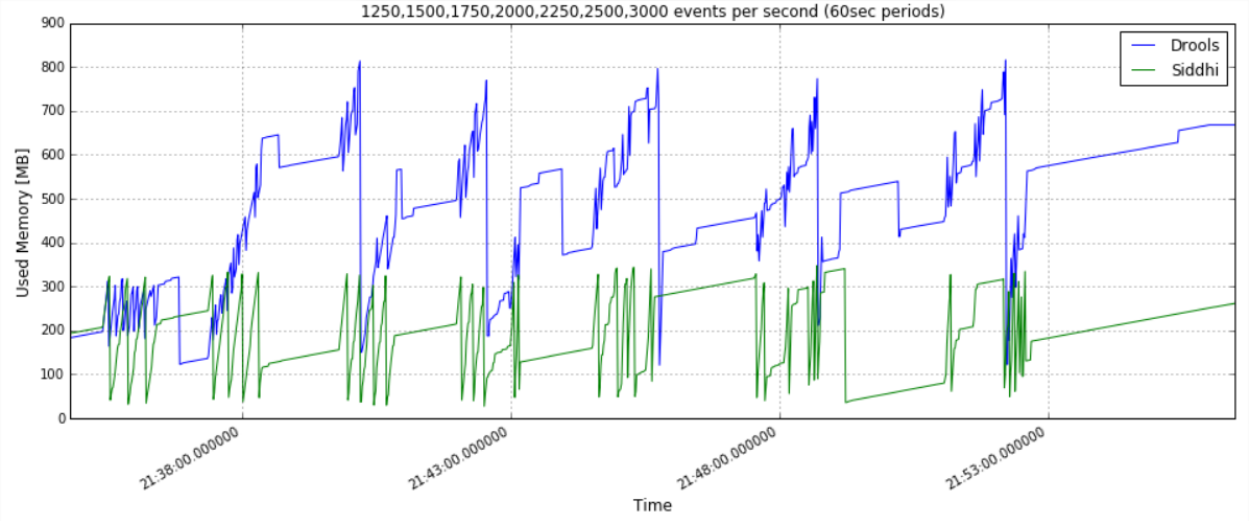

Figure 11. SDM load testing with PerfTest. Used memory of Drools-based implementation vs Siddhi-based implementation (1250 to 3000 events/sec)

In Figure 11, we can see the memory consumption of Drools and Siddhi when sending in range from 1250 to 3000 events per second, in consecutive 60 second periods. After 1500 events per second Drools needs more memory than Siddhi (the peak of difference is about 500MB).

In the following two diagramms we present in the left axis the CPU consumption and in the right axis the memory consumption of Drools during the tests that we described before.

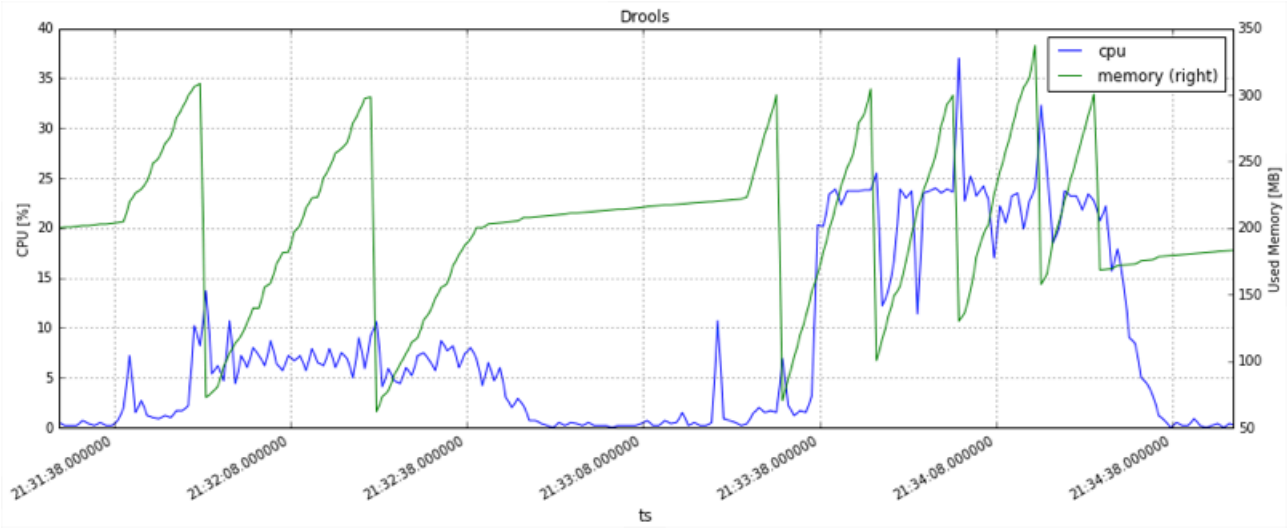

Figure 12. SDM load testing with PerfTest. Drools-based implementation CPU utilization (blue) and used memory (green) (500, 1000 events/sec)

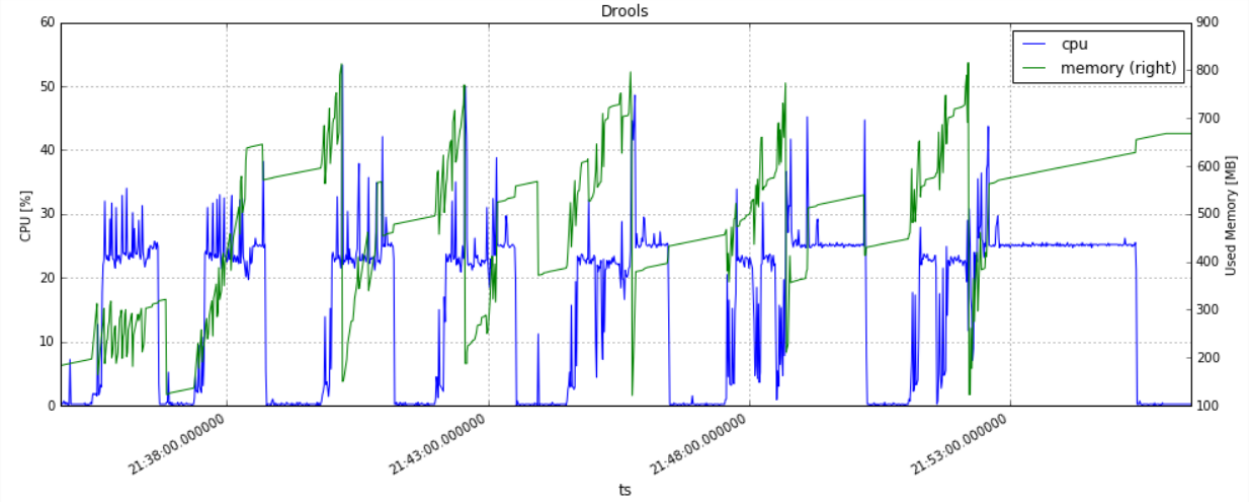

Figure 13. SDM load testing with PerfTest. Drools-based implementation CPU utilization (blue) and used memory (green) (1250 to 3000 events/sec)

In the following two diagramms we present in the left axis the CPU consumption and in the right axis the memory consumption of Siddhi during the tests that we described before.

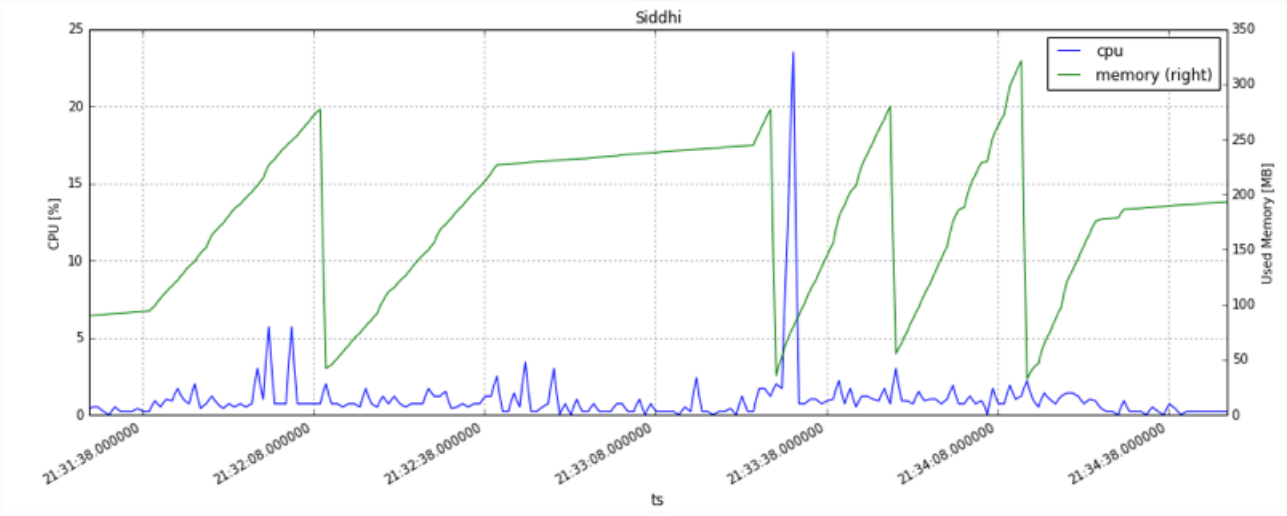

Figure 14. SDM load testing with PerfTest. Siddhi-based implementation CPU utilization (blue) and used memory (green) (500, 1000 events/sec)

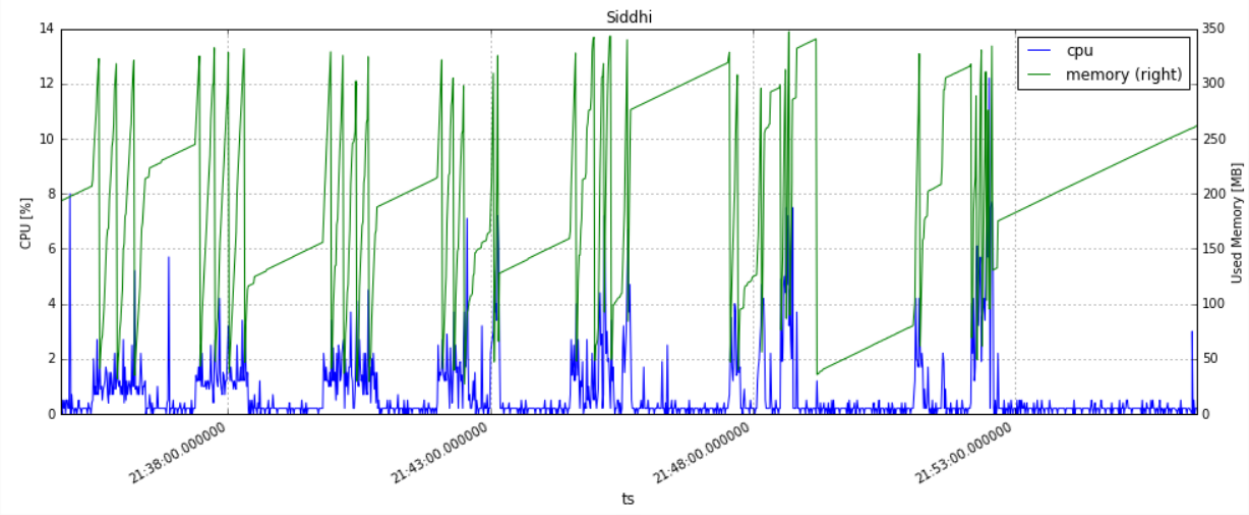

Figure 15. SDM load testing with PerfTest. Siddhi-based implementation CPU utilization (blue) and used memory (green) (1250 to 3000 events/sec)

Figure 16 depicts the queued messages in RabbitMQ (red colour) and the message rates (in yellow the publish rate and in green the deliver rate). We can observe that RabbitMQ was queuing many messages above 3000 evens per second rate.

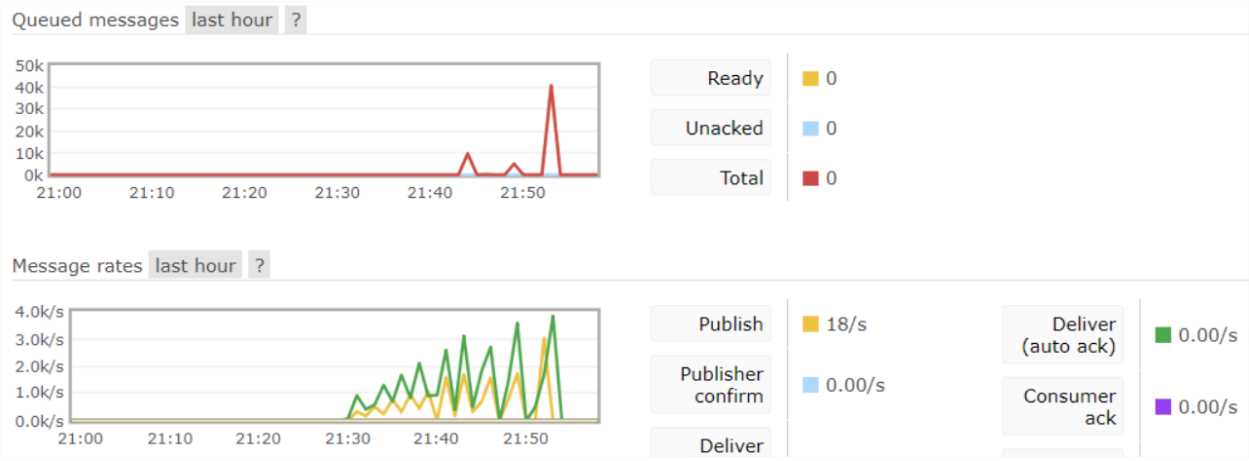

Figure 16. RabbitMQ management console metrics during SDM load testing with PerfTest (consecutive 60s period tests with increasing rates from 500 to 3000 events per second)

If we test Drools and Siddhi for bigger time periods, over 1500 events per second we can see clearly in the following diagrams that Drools takes much more time to process the incoming events. These diagrams have been produced by sending 1500 events per second for 5 minutes (300 sec) to Siddhi and Drools with PerfTest.

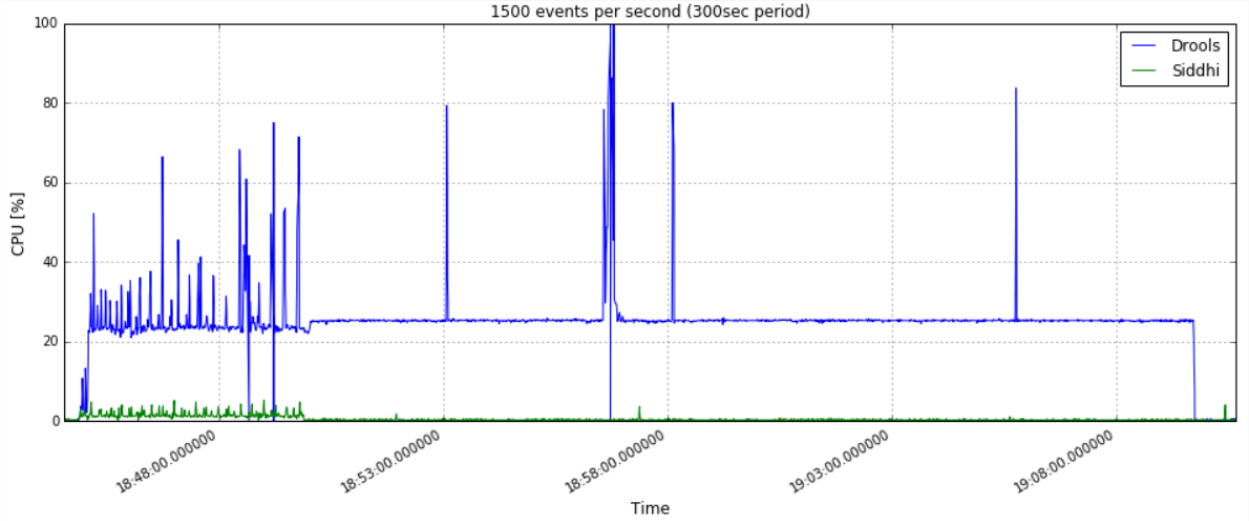

Figure 17. SDM load testing with PerfTest. CPU utilization of Drools-based implementation vs Siddhi-based implementation (sending 1500 events/sec for 5 minutes)

Figure 17 depicts the CPU consumption of the two implementations of SDM while load-testing the with 1500 events/sec for of a period of 300 seconds. From the CPU consumption diagrams we can confirm that the Drools-based implementation of SDM still processes the data 20 minutes after PerfTest has completed sending events.

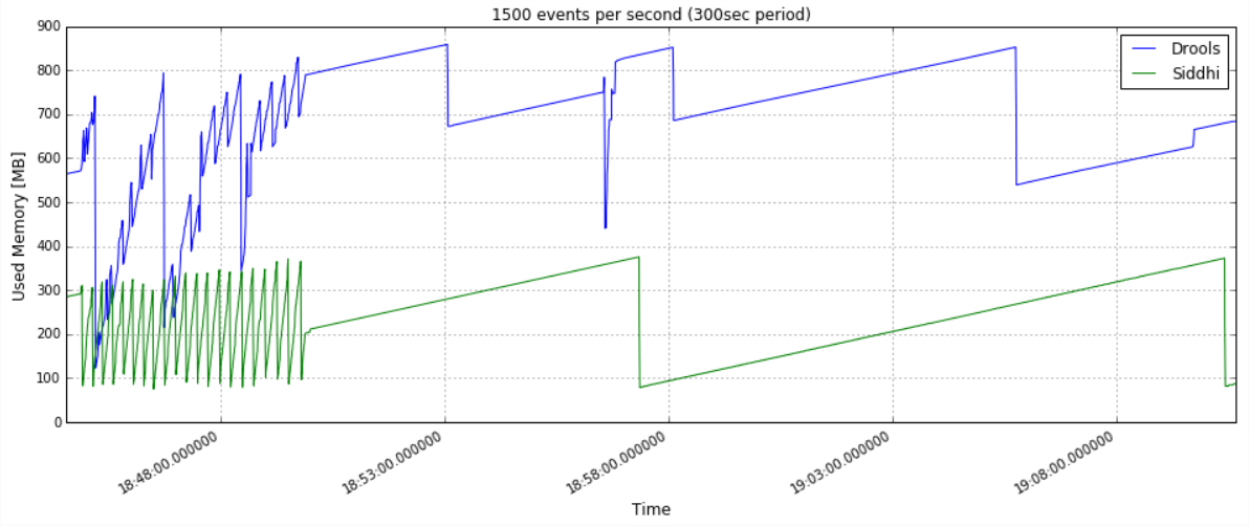

Figure 18. SDM load testing with PerfTest. Used memory of Drools-based implementation vs Siddhi-based implementation (sending 1500 events/sec for 5 minutes)

Figure 18 depicts the memory consumption of the two implementations of SDM while load-testing the with 1500 events/sec for of a period of 300 seconds.

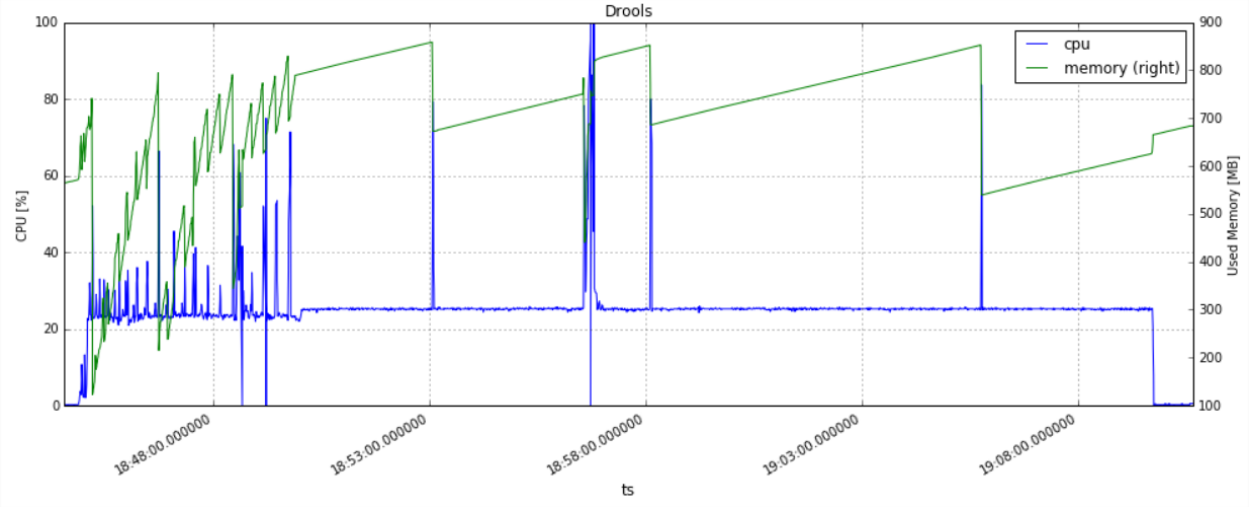

Figure 19. SDM load testing with PerfTest. CPU utilization vs used memory of Drools-based implementation (sending 1500 events/sec for 5 minutes)

Figure 19 depicts in the same diagram the CPU utilization and the memory consumption of the Drools-based implementation of SDM while load-testing it with 1500 events/sec for of a period of 300 seconds.

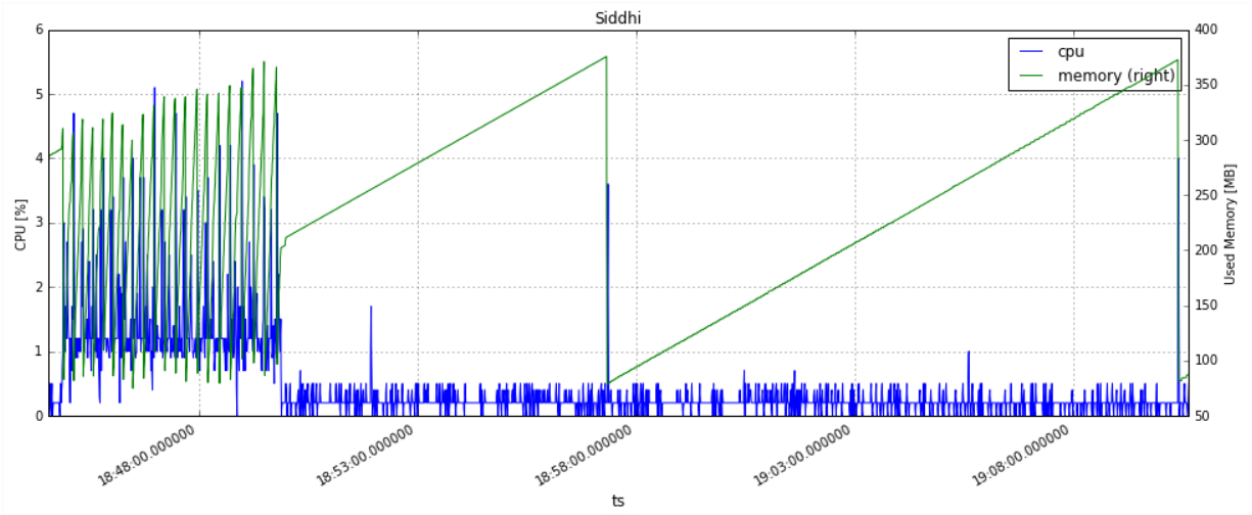

Figure 20. SDM load testing with PerfTest. CPU utilization vs used memory of Siddhi-based implementation (sending 1500 events/sec for 5 minutes)

Figure 20 depicts in the same diagram the CPU utilization and the memory consumption of the Siddhi-based implementation of SDM while load-testing it with 1500 events/sec for of a period of 300 seconds. In contrast with the Drools-based implementation Figure 19 CPU utilization falls to much lower than 1% after PerfTest has finished sending events.

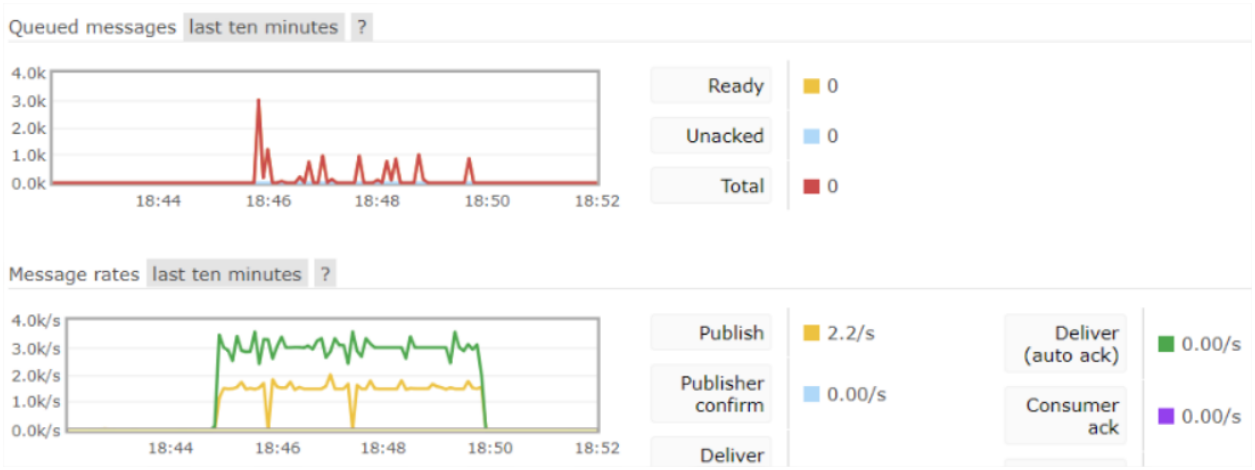

Figure 21. RabbitMQ management console metrics during SDM load testing with PerfTest (5min period)

The diagrams in Figure 21 are produced by the RabbitMQ management console. The diagram named "Queued messages" depict the amount of queued messages in RabbitMQ during the load-testing of the system with PerfTest for 5 minutes (or 300 seconds). In the diagram named "Message rates" the green line represents the deliver message rate while the yellow line represents the publish message rates. As we can see the publish message rate is very close to 1500 events/sec (as we instructed PerfTest to do). The deliver message rate is twice because we have two subscribers (the Drools-based and the Siddhi-based instances of SDM).

5.2 Experiment 2 (Proxy Server)

In the second experiment we use SDM services to monitor a proxy server that receives over 150 requests per second. The network traffic in Squid is from a real environment.

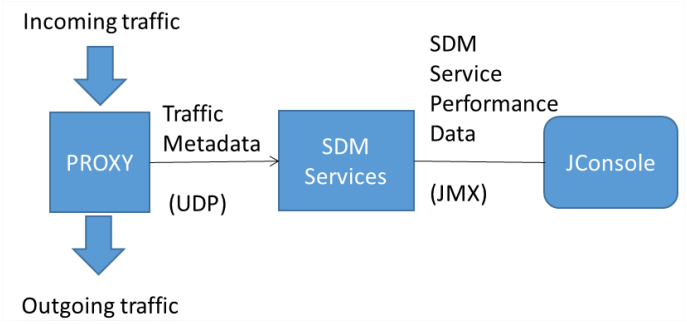

Figure 22. Experiment 2 (Load-testing SDM when monitoring network traffic in a production environment)

Squid runs in a separate machine than SDM services, which include Logstash, RabbitMQ and two instances of SDM, on Drools-based and one Siddhi-based. It exports metadata about the requests (source IP, destination IP, request_url, reponse_time) that it serves in real time to SDM over the UDP protocol. A Logstash instance listens on the corresponding UDP port, converts the Squid format to JSON and publishes each event back to RabbitMQ. Two instances of SDM, one running Drools and one running Siddhi subscribe to this topic, receive the events and detect the following set of complex event patterns:

1) Detect more than 100 requests with the same src_ip every 10 sec
2) Detect more than 100 requests with the same dst_ip every 10 sec
3) Detect more than 20 requests with the same request_url every 10 sec
4) Count the number of requests every 10 sec
5) Calculate the average response_time (of all requests) during the last 10 sec

In the second experiment we detect more and more complex event patterns then the previous one.

The following diagrams (Figure 23, Figure 24) depict the number of requests per second that the Squid proxy processes during a period of 30 minutes.

Figure 23. Squid Workload (requests/sec) – 30 min period

Figure 24. Squid Workload (requests/sec) – last 15 minute of 30 min period

The following diagrams depict the resource consumption of Drools-based and Siddhi-based SDM services during the same time period.

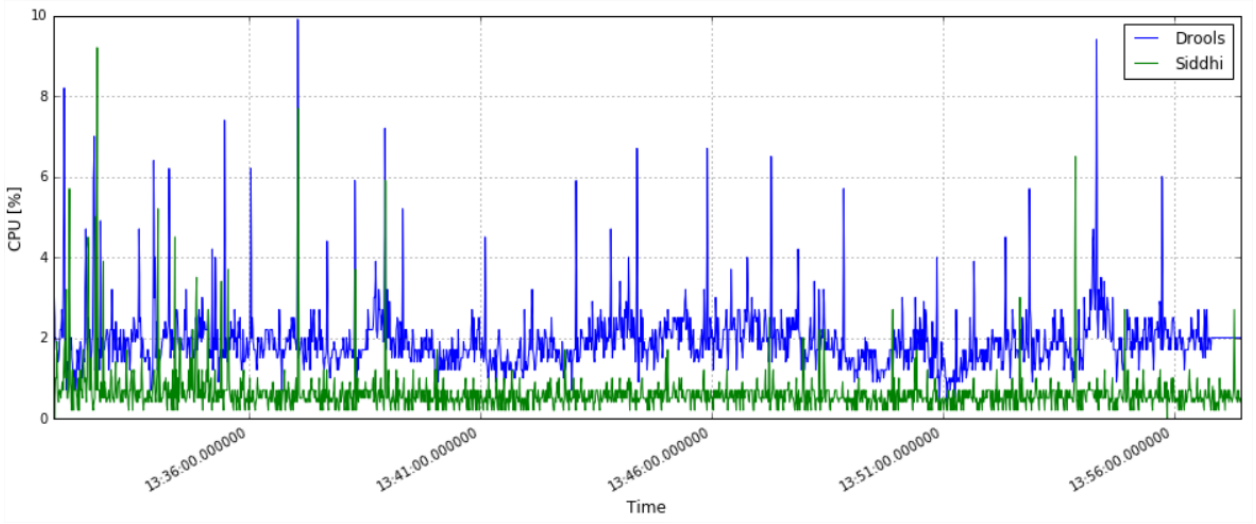

Figure 25. SDM evaluation with Squid proxy. CPU utilization comparison (Drools vs Siddhi)

Figure 25 depicts a comparison of the CPU utilization of Drools and Siddhi-based SDM while processing request metadata from a Squid proxy that receives up to 200 events/sec. The CPU utilization on both SDM instances (Drools and Siddhi-based) is lower than 4%. Siddhi clearly has lower CPU utilization but the difference with Drools is under 3%.

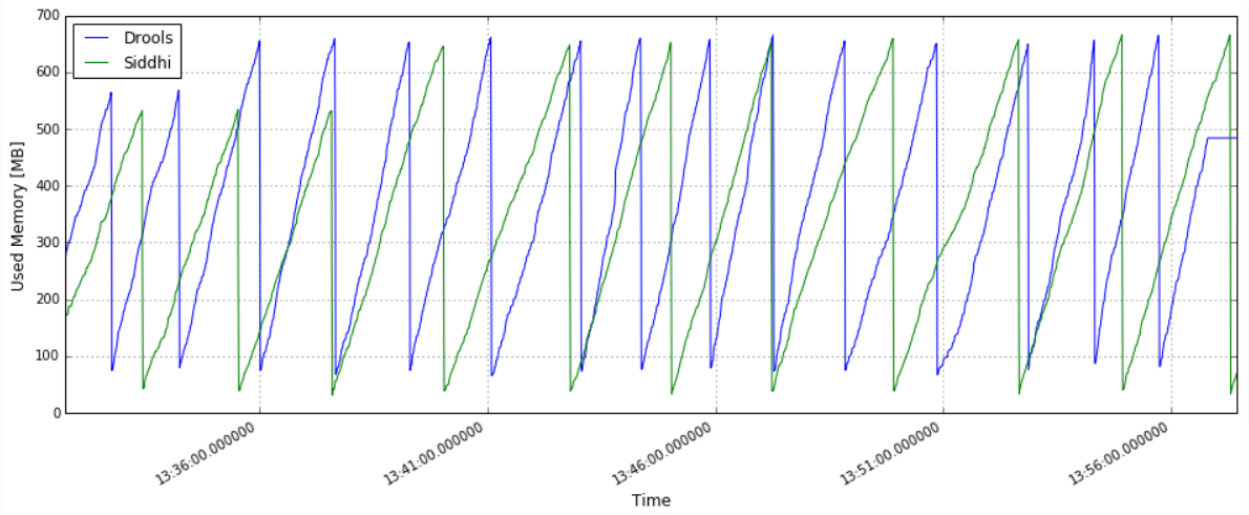

Figure 26. SDM evaluation with Squid proxy. Memory usage comparison (Drools vs Siddhi)

Figure 26 depicts a comparison of the memory usage of Drools and Siddhi-based SDM while processing request metadata from a Squid proxy that receives up to 200 events/sec. The memory usage patterns are similar.

In the following two diagrams (Figure 27, Figure 28) we show for each SDM implementation using the blue line the CPU utilization pattern and the green line the memory usage pattern during the same experiment with the Squid proxy.

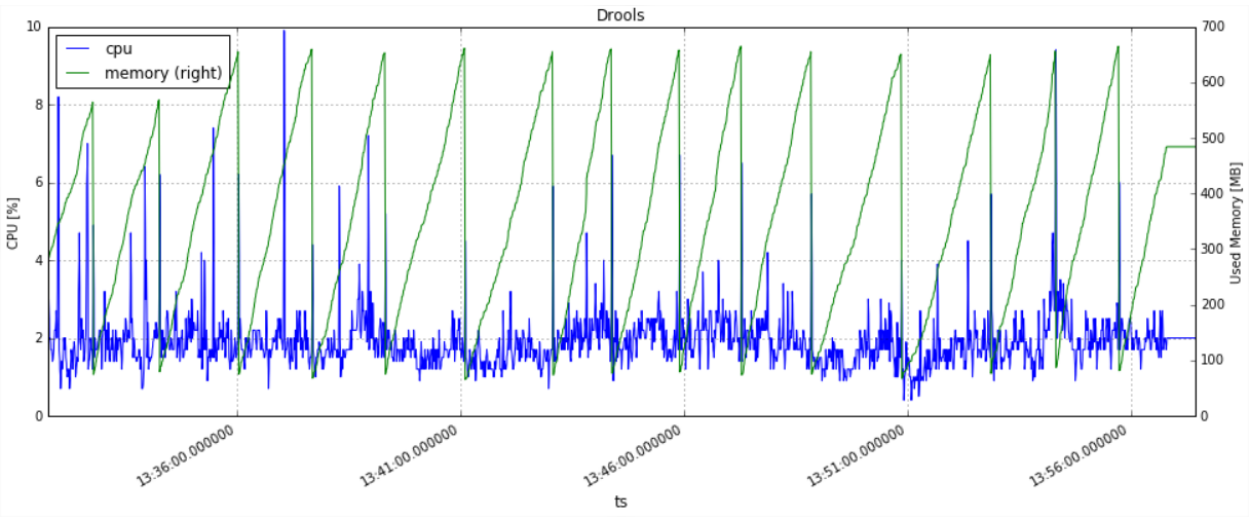

Figure 27. SDM evaluation with Squid proxy. CPU utilization vs used memory (Drools-based implementation)

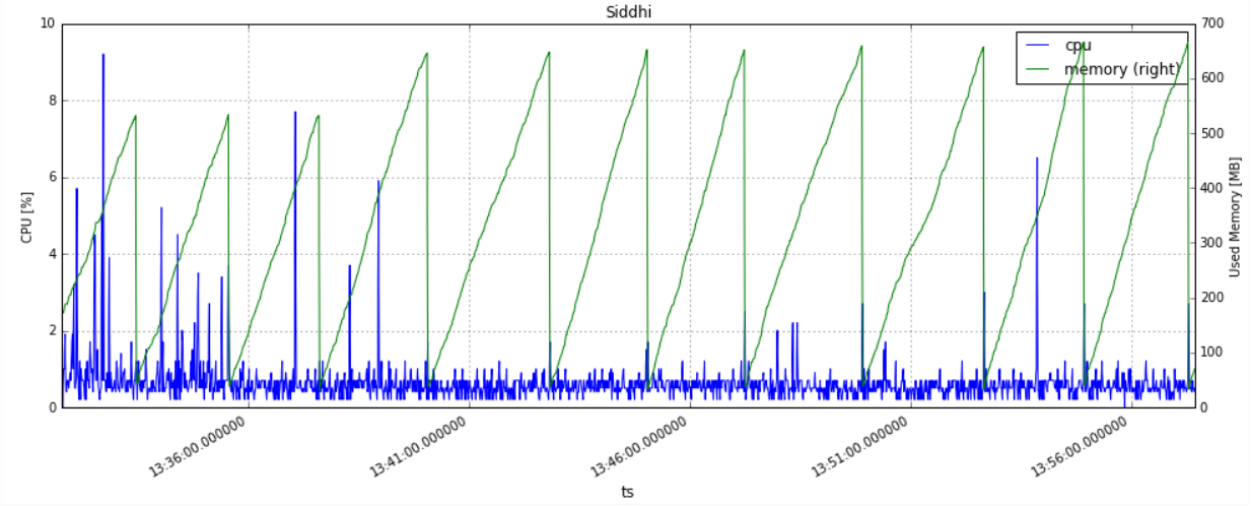

Figure 28. SDM evaluation with Squid proxy. CPU utilization vs used memory (Siddhi-based implementation)

5.3 Differences between Siddhi and Drools CEP languages

In this paragraph, we discuss some differences in the expressivity of Drools and Siddhi rule languages. Siddhi rule language is SQL-like while Drools language follows the paradigm of Event-Condition-Action (ECA) rules.

As we can see in the following examples, regarding the calculation of simple functions like the average, the minimum or the maximum of a metric over a period of time both rule languages have different but straightforward expressions.

```
rule "Average response time"
   timer ( int: 10s 10s )
when
   $rt: Number() from accumulate( $event: SquidEvent( rt : response_time ) over
window:time(10s),  average( rt ) )
then
   log("[RESPTM] " + $rt + " average response time");
end
```

Drools rule to calculate average response time

```
from squidStream#window.timeBatch(10 sec)
select str:concat("[RESPTM] " , convert(avg(rtime),'string'), " average response
time" ) as msg
insert into msgStream;
```

Siddhi rule to calculate average response time

When we want to aggregate over different objects and group the results over a group key, according to our knowledge, Drools rule language is not as expressive as Siddhi rule language. In the following examples we show how we can detect a situation where more than 20 requests for the same URL are detected in a period of time of 10 seconds. In Drools we have to produce an event (RequestAlert) which expires in 10 seconds and insert it in the fact base in order to prevent Drools to continuously produce the situation after the first time it was detected.

```
declare RequestAlert
  @role( event )
  @timestamp( ts )
  @expires ( 10s )
```

```
  ts: Date @key
  url: String @key
end

rule "request_url alert"
   timer ( int: 10s 10s )
when
   $e1 : SquidEvent( $url : request_url )
   not RequestAlert( url == $url )
   $c : Number(intValue > 20 ) from accumulate (
       $e2 : SquidEvent( this != $e1, request_url == $url ) over
window:time(10s) , count( $e2 ) )
then
   insert ( new RequestAlert( new Date(),  $url ) );
   log( "[REQURL] " + $c + " requests to the same url in 10s" );
end
```

Drools rule to alert when more than N=20 requests for the same url are detected in a period of 10 seconds.

With Siddhi we can group by the attribute "rurl" (that corresponds to the request URL) and check every 10 seconds (with #window.timeBatch) for each one if the total is over 20 (having cnt>20).

```
from squidStream#window.timeBatch(10 sec)
select  count(rurl) as cnt , rurl
group by rurl
having cnt > 20
insert into rurlStream;

from rurlStream
select str:concat("[REQURL] " , cnt, "  requests to the same url in 10s" ) as
msg
insert into msgStream;
```

Siddhi rule to alert when more than N=20 requests for the same url are detected in a period of 10 seconds.

6. Conclusions

This deliverable presented the first version of the Situation Detection Mechanism. SDM allows the detection of situations that require some kind of infrastructure or application adaptation. SDM does so by processing and analysing data streams generated by data-intensive applications and services deployed on cloud resources or at computing resources at the extreme edge of the network, which are monitored by PrEstoCloud.

The fist version of SDM focused on providing detection capabilities for situations that are few and can be modelled manually. Hence, we followed a specification-based approach. During the course of the project, we will evaluate the specification-based approach and, if needed, we will augment it with learning-based methods and techniques to cope with more and more complex situations, which can be manually specified as well as with imperfect sensors.

We designed the SDM component so as it is modular and can be easily deployed as a Docker container or a set of Docker containers. Moreover, we designed SDM to be independent of CEP libraries and we have shown that it can operate with both the Siddhi and Drools CEP libraries. We also demonstrated a real-world scenario indicative of the usage of our component, and its capabilities.

Testing and evaluation of SDM revealed that it is capable to detect situations defined as complex event patterns. Specifically, we tested SDM in conjunction with both Drools and Siddhi in two scenarios: first, we stress-tested it using the PerfTest load-testing tool of RabbitMQ and, second, to detect situations in computer network traffic in a real production computing environment. Tests indicated that SDM can be used to detect situations expressed as complex event patterns. Moreover, out tests have shown that Siddhi can scale better than Drools.

Our next objective will be the development of the PrEstoCloud Adaptation Recommender (D5.5), which will consume the output of SDM, i.e., the situations that are detected by SDM are require adaptation. The Adaptation Recommender will be fed also with the output of other WP5 components, namely, the Mobile Context Analyser and Workload Predictor and will generate as output specific recommendations for adaptations in the PrEstoCloud infrastructure resources. This will enable us afterwards to evaluate the complete capabilities of WP5 components, i.e. its capabilities to adapt and reconfigure the processing topology so that its performance requirements can be satisfied.

7. References

Adi, A. and O. Etzion, "Amit - the situation manager", The VLDB Journal, vol. 13, no. 2, pp. 177–203, May 2004.

Aggarwal, C. (2013). A Survey of Stream Clustering Algorithms, In" Data Clustering: Algorithms and Applications", ed. C. Aggarwal and C. Reddy.

Akbar, A., F. Carrez, K. Moessner, J. Sancho, and J. Rico. 2015. Context-aware stream processing for distributed IoT applications. In Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT'15). 663–668.

Allen, J. F. "Maintaining knowledge about temporal intervals," Communications of the ACM, vol. 26, Nov. 1983, pp. 832–843.

Amazon (2018), https://aws.amazon.com/autoscaling.

Amini, A., Wah, T. Y., & Saboohi, H. (2014). On density-based data streams clustering algorithms: A survey. Journal of Computer Science and Technology, 29(1), 116-141.

Anind K. Dey, "Providing Architectural Support for Building Context-Aware Applications", PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

Barwise, J., The Situation In Logic, CSLI Lecture Notes 17, 1989.

Chen, H., T. Finin, A. Joshi, An ontology for context-aware pervasive computing environments, Knowledge Engineering Review 18 (3) (2004) 197–207. Special Issue on Ontologies for Distributed Systems.

Cisco (2018), https://www.cisco.com/c/dam/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/q-and-a-c67-737653.pdf.

Cohen, N.H., H.Lei, P.CastroII, J.S.D,A.Purakayastha. Composing pervasive data using iQL, in: WMCSA'02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications, 2002, pp. 94–104.

Costa, et al. "A model-driven approach to situations: Situation modeling and rule-based situation detection." Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International. IEEE, 2012.

Costa, P. D., G. Guizzardi, J.P.A. Almeida, L. Ferreira Pires, M. van Sinderen, "Situations in Conceptual Modeling of Context". Workshop on Vocabularies, Ontologies, and Rules for the Enterprise (VORTE 2006) at IEEE EDOC 2006, IEEE Computer Society Press, 2006.

Dawar, S., van der Meer, S., Fallon, E., Keeney, J., & Bennett, T. (2013). Building a scalable event processing system with messaging and policies–test and evaluation of RabbitMQ and drools expert. Architectural Research Quarterly, 17, f1-f2.

Dayarathna, M., & Perera, S. (2018). Recent Advancements in Event Processing. ACM Computing Surveys (CSUR), 51(2), 33.

Delir, P., Haghighi, S. Krishnaswamy, A. Zaslavsky, M.M. Gaber, Reasoning about context in uncertain pervasive computing environments, in: EuroSSC'08: Proceedings of the 3rd European Conference on Smart Sensing and Context, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 112–125.

Dey, A. K., & Abowd, G. D. (2000, June). The context toolkit: Aiding the development of context-aware applications. In Workshop on Software Engineering for wearable and pervasive computing (pp. 431-441).

Endsley, M. Designing for Situation Awareness: An Approach to User-Centered Design, Second Edition. CRC Press, 2016.

Etzion, O., Yonit Magid, Ella Rabinovich, Inna Skarbovsky, and Nir Zolotorevsky. 2011. Context-Based Event Processing Systems. Springer, Berlin, Germany, 257–278.

Flouris, I., Giatrakos, N., Garofalakis, M., & Deligiannakis, A. (2015, August). Issues in complex event processing systems. In Trustcom/BigDataSE/ISPA, 2015 IEEE (Vol. 2, pp. 241-246). IEEE.

Franke, U. and J. Brynielsson, "Cyber situational awareness - A systematic review of the literature", Computers & Security, vol. 46, pp. 18–31, 2014.

Gaber, M. M., Gama, J., Krishnaswamy, S., Gomes, J. B., & Stahl, F. (2014). Data stream mining in ubiquitous environments: state-of-the-art and current directions. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 4(2), 116-138.

Google Cloud (2018), https://cloud.google.com/compute/docs/autoscaler/.

Gray, P.D., D. Salber, Modelling and using sensed context information in the design of interactive applications, in: EHCI'01: Proceedings of the 8th IFIP International Conference on Engineering for Human–Computer Interaction, Springer-Verlag, London, UK, 2001, pp. 317–336.

Gu, T., S. Chen, X. Tao, J. Lu. A non supervised approach to activity recognition and segmentation based on object-use fingerprints, Data and Knowledge Engineering 69 (6) (2010) 533–544.

Guizzardi, G. "Ontological foundations for structural conceptual models," CTIT, Centre for Telematics and Information Technology, Enschede, 2005.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B., & Grimm, R. (2014). A catalog of stream processing optimizations. ACM Computing Surveys (CSUR), 46(4), 46.

Hoehndorf, R. "Situoid theory, An ontological approach to situation theory", M.Sc. Thesis, University of Leipzig 2005.

Kalyan, A., Gopalan, S., & Sridhar, V. (2005, March). Hybrid context model based on multilevel situation theory and ontology for contact centers. In Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on (pp. 3-7). IEEE.

Kokar, M. M., C. J. Matheus and K. Baclawski, "Ontology-based situation awareness," Information Fusion, vol. 10, Jan, 2009, pp. 83- 98.

Kubernetes (2018), https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/.

Lei, H., D.M. Sow, S. John, I. Davis, G. Banavar, M.R. Ebling, The design and applications of a context service, SIGMOBILE Mobile Computing and Communications Review 6 (4) (2002) 45–55.

Loia, V., G. D'Aniello, A. Gaeta, and F. Orciuoli, "Enforcing situation awareness with granular computing: A systematic overview and new perspectives", Granular Computing, vol. 1, no. 2, pp. 127–143, 2016.

Loke, S.W. Incremental awareness and compositionality: a design philosophy for context-aware pervasive systems, Pervasive and Mobile Computing 6 (2) (2010) 239–253.

Matheus, C. J., M. M. Kokar, and K. Baclawski, "A Core Ontology for Situation Awareness", Proc. 6th Int'l Conf. on Information Fusion, 2003, pp. 545 –552.

McCarthy, J. "Situation Calculus with Concurrent Events and Narrative", http://wwwformal.stanford.edu/jmc/narrative/ narrative.html, 2000.

OpenStack (2018), https://docs.openstack.org/senlin/latest/scenarios/autoscaling_heat.html.

Ranganathan, A., J. Al-Muhtadi, R.H. Campbell, Reasoning about uncertain contexts in pervasive computing environments, IEEE Pervasive Computing 03 (2) (2004) 62–70.

Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., De Carvalho, A. C., & Gama, J. (2013). Data stream clustering: A survey. ACM Computing Surveys (CSUR), 46(1), 13.

Sobral, V. M., Almeida, J. P. A., & Costa, P. D. (2015, March). Assessing situation models with a lightweight formal method. In Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2015 IEEE International Inter-Disciplinary Conference on (pp. 42-48). IEEE.

Wang, Y. and K. Cao. 2012. Context-aware complex event processing for event cloud in Internet of Things. In Proceedings of the 2012 International Conference on Wireless Communications Signal Processing (WCSP'12). 1–6.

Yau et al. 2006 proposed an OWL-based situation ontology to model situation hierarchically to facilitate sharing and reusing of situation knowledge and logic inferences is presented. The same work converts the OWL situation ontology specification to a First-Order Logic (FOL) representation.

Yau, S. S., & Liu, J. (2006, April). Hierarchical situation modeling and reasoning for pervasive computing. In Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on (pp. 6-pp). IEEE.

Yau, S. S., F. Karim, Y. Wang, B. Wang, and S.Gupta, "Reconfigurable Context-Sensitive Middleware for Pervasive Computing," IEEE Pervasive Computing, 1(3), July- September 2002, pp.33-40.

Yau, S. S., Huang, D., Gong, H., & Yao, Y. (2006). Support for situation awareness in trustworthy ubiquitous computing application software. Software: Practice and Experience, 36(9), 893-921.

Yau, S. S., Y. Wang, and F. Karim, "Development of Situation-Aware Application Software for Ubiquitous Computing Environments", Proc. 26th Ann. Int'l Computer Software and Applications Conf., 2002, pp. 233-238.

Ye, J., Dobson, S., & McKeever, S. (2012). Situation identification techniques in pervasive computing: A review. Pervasive and mobile computing, 8(1), 36-66.

Ye, J., L. Coyle, S. Dobson, P. Nixon, Using situation lattices to model and reason about context, in: MRC 2007: Proceedings of the Workshop on modeling and reasoning on context (coexist with CONTEXT'07), Roskilde, Denmark, August 2007, pp. 1–12.

WSO2 (2018), https://wso2-extensions.github.io/siddhi-io-rabbitmq/.

WSO2 (2018b), https://wso2-extensions.github.io/siddhi-map-json/.

8. Appendix I – Example configuration files

8.1 Docker-compose

```
version: '2.4'

services:

  rabbitmq:
    build: ./rabbitmq
    hostname: "rabbitmq-broker"
    environment:
      RABBITMQ_ERLANG_COOKIE: "SWQOKODSQALRPCLNMEQG"
      RABBITMQ_DEFAULT_USER: "rabbitmq"
      RABBITMQ_DEFAULT_PASS: "rabbitmq"
      RABBITMQ_DEFAULT_VHOST: "/"
    ports:
      - "15672:15672"
      - "5672:5672"
    labels:
      NAME: "rabbitmq1"
    networks:
      - esnet
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:15672"]
      interval: 30s
      timeout: 5s
      retries: 5

  logstash:
    image: docker.elastic.co/logstash/logstash-oss:6.2.0
    volumes:
      - ./logstash/netdata.pipeline:/usr/share/logstash/pipeline:ro
      - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro
    ports:
      - 5002:5000
    networks:
      - esnet
    depends_on:
      rabbitmq:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9600"]
      interval: 30s
      timeout: 5s
      retries: 5

  sdmsrv2:
    build: ./sdmsrv2
    image: sdmsrv2:1.0
    environment:
      - TZ=Europe/Athens
      - MAVEN_OPTS=-Xmx1024m -Xms1024m -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.port=1898 -Dcom.sun.management.jmxremote.rmi.port=1898 -
Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.local.only=false -Djava.rmi.server.hostname=0.0.0.0 -
Dkie.mbeans=enabled
    volumes:
      - ./sdmsrv2/resources.netdata:/sdm/src/main/resources:ro #netdata example
      #- /docker/maven/.m2:/root/.m2:rw #mount maven .m2 repository on a local volume
    command: mvn install #exec:java
    ports:
      - 1898:1898 # HOST port : CONTAINER port
    networks:
      - esnet
    depends_on:
      logstash:
        condition: service_healthy

  netdata:  #services running netdata that push events to logstash->rabbitmq->sdmsrv2(siddhi)
    build: netdatagen/.
    ports:
      #- "19999:19999"
```

```
      - "19999-20100:19999" #needed for multiple instances with scale=n
    labels:
      NAME: "netdata"
    networks:
      - esnet
    volumes:
      - ./netdatagen/netdata.conf:/etc/netdata/netdata.conf:ro
    depends_on:
      - rabbitmq

networks:
  esnet:
```

8.2 Logstash pipeline for Netdata

```
input {
  graphite {
    port => 5000
  }
}

filter {
  grok {
    match => [ "message", "%{DATA:mname} %{NUMBER:mvalue:float} %{POSINT:timestamp}" ]
  }
}

output {
  stdout {
    codec => rubydebug
  }
  rabbitmq {
    host => "rabbitmq"
    user => "rabbitmq"
    password => "rabbitmq"
    exchange => "LOGSTASH2_TO_SDM"
    exchange_type => "topic"
    durable => "false"
    key => "%{mname}"
  }
}
```

8.3 Siddhi rule file

```
@App:name('BenchmarkExecutionPlan')

define stream outputStream (memory double, cpu double);
define trigger TenSecTriggerStream at every 10 sec;
define trigger StartTriggerStream at 'start';
define stream msgStream (msg string);

@source(
        type ='rabbitmq',
        uri = 'amqp://rabbitmq:rabbitmq@rabbitmq:5672',
        exchange.name = 'SDM_INPUT',
        routing.key= 'input.data',
        @map(
                type='json'
        )
)
define stream serverStream (memory double, cpu double);

from StartTriggerStream select 'CEP STARTED' as msg insert into msgStream;

from serverStream#window.timeBatch(10 sec)
select str:concat("CPU AVG:",convert(avg(cpu),'string'))  as msg
insert into msgStream;

from serverStream#window.timeBatch(10 sec)
select str:concat("CPU COUNT:",convert(count(cpu),'string'))  as msg
insert into msgStream;

from serverStream#window.timeBatch(10 sec)
```

```
select str:concat("MEMORY AVG:",convert(avg(memory),'string'))  as msg
insert into msgStream;

from serverStream#window.timeBatch(10 sec)
select str:concat("MEMORY COUNT:",convert(count(memory),'string'))  as msg
insert into msgStream;
```

8.4 Netdata backend configuration (netdata.conf)

```
[backend]
        enabled = yes
        data source = average
        type = graphite
        destination = logstash:5000
        prefix = netdata
        hostname = mypc
        update every = 3
        buffer on failures = 10
        timeout ms = 20000
        send names instead of ids = yes
        send charts matching = system.cpu*
```