Situation Detection on the Edge

Nikos Papageorgiou[1], Dimitris Apostolou[1, 2], Yiannis Verginadis[1], Andreas Tsagkaropoulos[1], Gregoris Mentzas[1]
[1]National Technical University of Athens, 9 Iroon Polytechniou str., 157 80 Zografou, Athens, Greece
{npapag, jverg, atsagkaropoulos, gmentzas}@mail.ntua.gr
[2]University of Piraeus, 80 Karaoli & Dimitriou str., 185 34, Piraeus, Greece
dapost@unipi.gr

Abstract. Situation Awareness in edge computing devices is necessary for detecting issues that may hinder their computation capacity and reliability. The Situation Detection Mechanism presented in this paper uses Complex Event Processing in order to detect situations where the edge infrastructure requires an adaptation. We designed the 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 it to be independent of Complex Event Processing libraries and we have shown that it can operate with both the Siddhi and Drools libraries. We evaluated our work with a real-world scenario indicative of the usage of our component, and its capabilities.

Keywords: situation awareness, complex event processing, edge computing

1 Introduction

Mobile Edge Computing (MEC) enables a computing and storage infrastructure provisioned closely to the end-users at the edge of a cellular network. Combining MEC in multi-cloud infrastructures can help to combat latency challenges imposed by cloud-centric architectures. However, edge devices are highly dynamic in nature: they are not as reliable as server and cloud computing resources; they computing capacity is limited and varies greatly depending on their workload; their operating environment (temperature, humidity, etc.) may impact their performance; they often include sensors which send data at a very high rate which can sometimes swamp the available network bandwidth. Perception of these elements in the environment of edge devices within a volume of time and space and the comprehension of their meaning is typically referred to as 'situation' [6], [7]. Situation detection can enhance the capacity to manage edge resources effectively as part of a computing environment.

Situations in edge computing infrastructures are highly related to the current status and context of edge devices and behavior of deployed applications. Situations are in a rich structural and temporal relationship, and they are dynamic by definition, continuously evolving and adapting. To cope with the dynamicity of situations, one needs to sense and process data in large volumes, in different modalities [18]. To realize systems for Situation Awareness (SA) "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" [13]. The power of using 'situations' lies in their ability to provide a simple, human-understandable representation of, for instance, sensor data [13]. In the context of dynamic computing systems, a situation is defined as an event occurrence that might require a reaction [1]. In 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 [17]. SA is one of the most fundamental features to support dynamic adaptation of entities in pervasive computing environments.

The research objective of our work has been to design and develop a Situation Detection Mechanism (SDM) capable of processing in real-time events generated by the edge infrastructure and detecting situations where the edge infrastructure requires an adaptation. Moreover, the SDM should be modular and easily deployable on heterogeneous edge infrastructures. The deployment flexibility of SDM is quite important since it allows using it on as many as possible different types of edge devices. SDM is part of the PrEstoCloud dynamic and distributed software architecture that manages cloud and fog resources proactively, while reaching the extreme edge of the network for an efficient real-time BigData processing (http://prestocloud-project.eu/).

The rest of the paper is organized as follows. Section 2 discusses enabling technologies and works related to situation detection and awareness for enterprise decision making with an emphasis on supporting proactivity. Section 3 outlines the proposed situation model and described the approach followed to realise situation detection. Section 4 presents a scenario in which SDM has been evaluated. Section 5 discusses the main findings of our work and our future plans.

2 Related Work

Situation detection is mainly accomplished using two approaches: specification- and learning-based ones. 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 [18]. Existing approaches range from earlier attempts in first-order logic [16] to complex event processing and more advanced logic models that aims to support efficient reasoning while keeping expressive power, see, e.g., [14]. With their powerful representation and reasoning capabilities, ontologies have been widely applied, see [3]. As more and more sensors are deployed in real-world environments for a long-term experiment, the uncertainty of sensor data starts poses the need for probabilistic techniques [5] capable of coping with incompleteness, accuracy, timeliness, and reliability of sensor data [9];[12];[4].

Learning-based techniques have been widely applied to learning complex associations between situations and sensor data [18]. Typically, learning-based techniques use supervised learning methods. Supervised learning methods train models using available labeled data. Manual labelling of training data can be cumbersome and laborious in cases of many situations and corresponding data to be used for training [10]. Unsupervised learning methods can help ameliorate this challenge but there are limited works reported in the literature. Learning-based methods can cope well with uncertainty when trained with noisy real-world data [18].

3 Situation Model & Situation Detection Approach

3.1 Situation Model

We follow an event-based approach for situation modeling 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. 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 [18]. A situation can uncover meaningful correlations between events, labeling 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.

The PrEstoCloud Situation Metamodel (Figure 1) captures the concepts based on which the SDM will be able to detect situations, which may reveal impending failures or even opportunities for increasing the performance of the deployed applications over multi-cloud and edge resources. The PrEstoCloudSituation comprises AtomicSituations and CompositeSituations. 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.

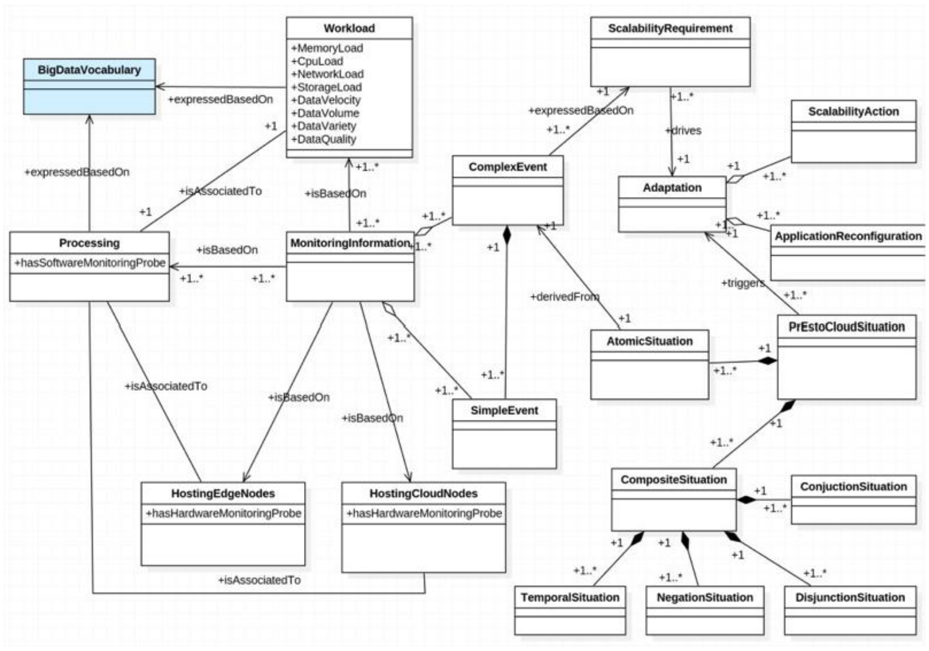

Fig. 1. PrEstoCloud Situation Metamodel

A CompositeSituation 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 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.

3.2 Situation Awareness 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 [2] that trigger adaptation actions based on user-configurable rules that are evaluated in real-time using internal or external monitoring infrastructure. Kubernetes [11] provides auto-scaling capabilities based on internal or external metrics. In Google Cloud [8], 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 [15] also supports auto-scaling policies by deploying the Heat service. Autoscaling in OpenstackHeat is triggered by Alarms produced by the telemetry service (Ceilometer).

Since a MEC 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.

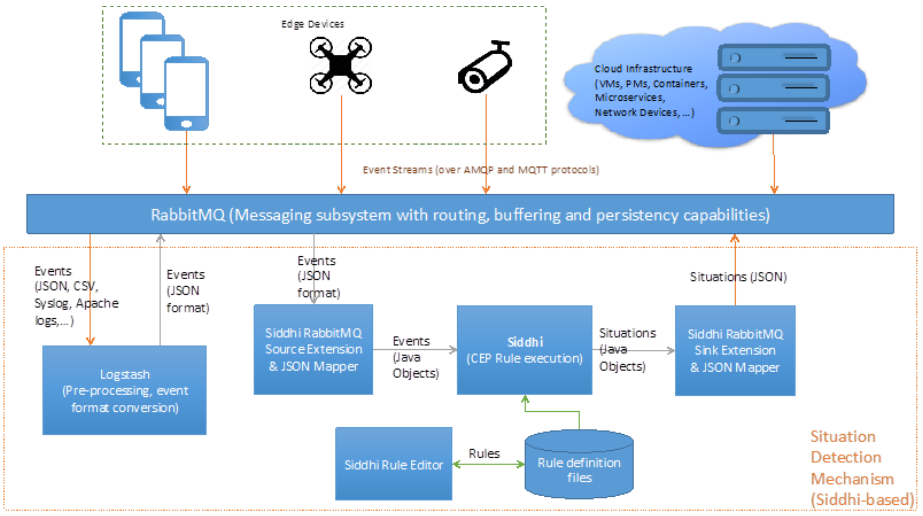
Edge Devices
Cloud Infrastructure (VMs, PMs, Containers, Microservices, Network Devices, ...)
Event Streams (over AMQP and MQTT protocols)
RabbitMQ (Messaging subsystem with routing, buffering and persistency capabilities)
Events (JSON, CSV, Syslog, Apache logs,...)
Events (JSON format)
Events (JSON format)
Situations (JSON)
Logstash (Pre-processing, event format conversion)
Siddhi RabbitMQ Source Extension & JSON Mapper
Events (Java Objects)
Siddhi (CEP Rule execution)
Situations (Java Objects)
Siddhi RabbitMQ Sink Extension & JSON Mapper
Siddhi Rule Editor
Rules
Rule definition files
Situation Detection Mechanism (Siddhi-based)

Fig. 2. Situation Detection Mechanism Architecture

Arguably, situation detection in a MEC environment 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. This way we can 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 has the following characteristics: (i) A homogenous solution for data intensive and data-driven situation detection at the edge or near the edge and in the cloud. (ii) 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/). (iii) A distributed hierarchical approach for event-driven and rule-based situation detection with complex event patterns.

The primary input for Situation Detection Mechanism (Figure 2) is considered any health status event or application performance-related event transmitted to the Communication and Message Broker (RabbitMQ). 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. Events from the cloud infrastructure (physical and virtual machines, containers, applications, services, etc.) and edge devices (mobile phones, routers, 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.). 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.

4 Evaluation

We have evaluated SDM using two different CEP engines, Drools and Siddhi. The software and hardware configuration is as follows: Hardware (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, Docker version 17.12.0-ce, build c97c6d6, Docker-compose version 1.19.0, build 9e633ef, OpenJDK Runtime Environment, 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). We used the RabbitMQ load-testing tool to generate and publish events (https://github.com/rabbitmq/rabbitmq-perf-test). We used 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. With a Java Management Extensions (JMX) tool such as JConsole or VisualVM (https://visualvm.github.io/), we monitored the Drools and Siddhi version of SDM.

We ran (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 received the same events from PerfTest. First we run PerfTest for 60 seconds with increasing number of event producers that send one event per second. 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). Both Drools and Siddhi were configured to produce every 10 sec two events containing: (i) the average CPU and MEMORY (during the last 10s); (ii) the number of MEMORY and CPU events that it received (during the last 10s).

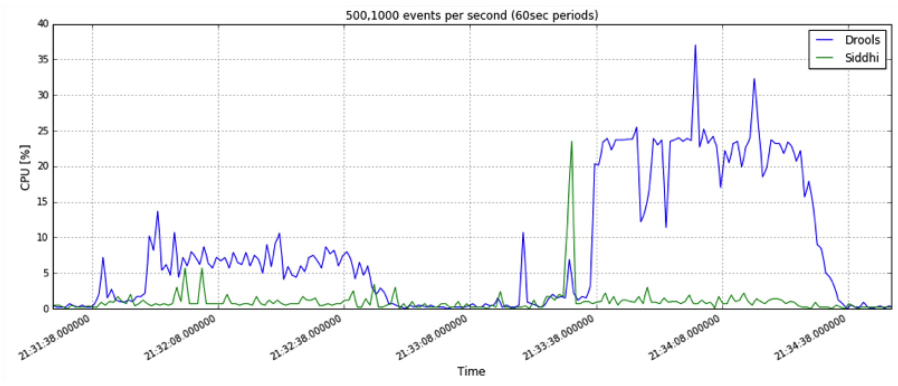

Fig. 3. SDM load testing with PerfTest. CPU utilization of Drools-based implementation vs Siddhi-based implementation (500,500,1000 events/sec)

We compared 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 see clearly in figure 3 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.

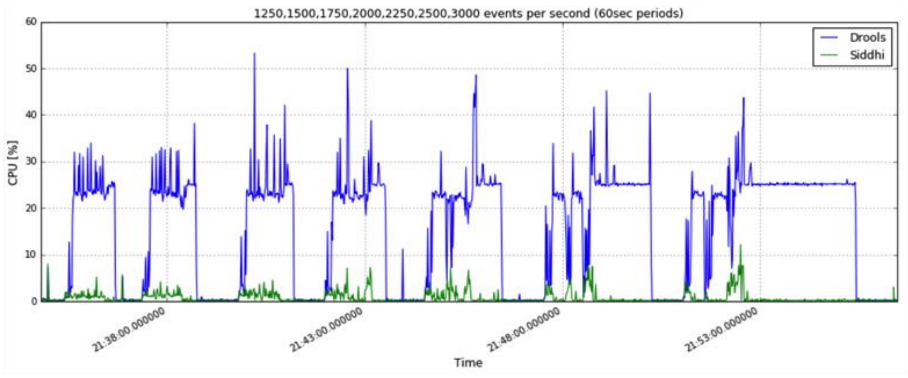

Fig. 4. SDM load testing with PerfTest. CPU utilization of Drools-based implementation vs Siddhi-based implementation (1250 to 3000 events/sec)

In Figure 4, 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.

In Figure 5, 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. After 1500 events per second Drools needs more memory than Siddhi (the peak of difference is about 500MB). If we test Drools and Siddhi for bigger time periods, over 1500 events per second we can see that Drools takes much more time to process the incoming events (figures not included for brevity).

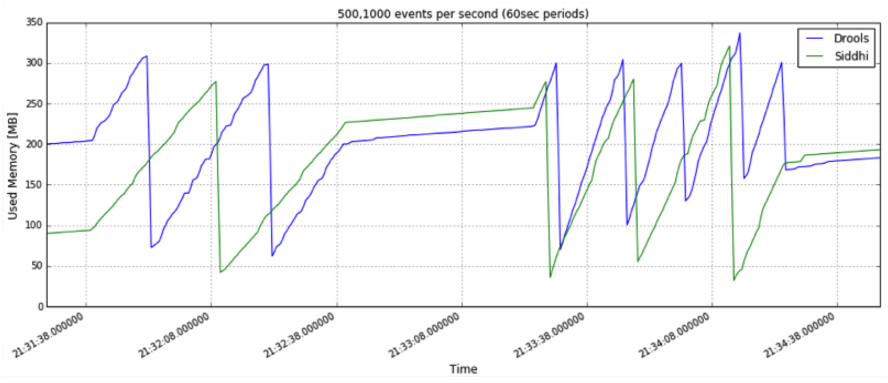

Fig. 5. SDM load testing with PerfTest. Used memory of Drools-based implementation vs Siddhi-based implementation (500,500,1000 events/sec)

Further, we compared the expressivity of the Drools and Siddhi rule languages in expressing situations. Siddhi rule language is SQL-like while Drools language follows the paradigm of Event-Condition-Action (ECA) rules. Both rule languages have different but straightforward expressions for dealing with simple functions like the average, the minimum or the maximum of a metric over a period of 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.

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

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).

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

5 Conclusions

SDM has been designed and developed focusing on providing detection capabilities for situations pertaining to edge resources computing capabilities. In implementing SDM, we followed a specification-based approach. 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. The deployment flexibility of SDM is quite important since it allows using the Complex Event Processing engine of choice based on the processing capabilities required in each case and the prior expertise regarding a certain Complex Event Processing engine.

We evaluated SDM with 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.

In the future, we will further test SDM with more complex patterns and scenarions. Moreover, we will augment the specification-based approach we followed for SDM with learning-based methods and techniques to cope with more and more complex situations, which cannot be manually specified as well as with imperfect sensors. Moreover, our future work will focus on enhancing SDM with capabilities to recommend adaptations to the edge processing topology in order to optimize their usage so that its performance requirements can be satisfied.

Acknowledgments. This work is partly funded by the H2020 PrestoCloud project– Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing (732339).

References

1. Adi, A. and O. Etzion, "Amit - the situation manager", The VLDB Journal, vol. 13, no. 2, pp. 177-203, May 2004.
2. Amazon (2018), https://aws.amazon.com/autoscaling.
3. 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.
4. 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.
5. 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.
6. Endsley, M. Designing for Situation Awareness: An Approach to User-Centered Design, Second Edition. CRC Press, 2016.
7. Franke, U. and J. Brynielsson, "Cyber situational awareness - A systematic review of the literature", Computers & Security, vol. 46, pp. 18-31, 2014.
8. Google Cloud (2018), https://cloud.google.com/compute/docs/autoscaler/.
9. 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.
10. 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.
11. Kubernetes (2018), https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/.
12. 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.
13. 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.
14. Loke, S.W. Incremental awareness and compositionality: a design philosophy for context-aware pervasive systems, Pervasive and Mobile Computing 6 (2) (2010) 239-253.
15. OpenStack (2018), https://docs.openstack.org/senlin/latest/scenarios/autoscaling_heat.html.
16. Ranganathan, A., J. Al-Muhtadi, R.H. Campbell, Reasoning about uncertain contexts in pervasive computing environments, IEEE Pervasive Computing 03 (2) (2004) 62-70.
17. 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. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on (pp. 6-pp).
18. Ye, J., Dobson, S., & McKeever, S. (2012). Situation identification techniques in pervasive computing: A review. Pervasive and mobile computing, 8(1), 36-66.