Addressing Self-Management in Cloud Platforms: A Semantic Sensor Web Approach

Rustem Dautov
South-East European Research Centre
International Faculty, The University of Sheffield
24 Proxenou Koromila Street
Thessaloniki, 54622, Greece
rdautov@seerc.org

Dimitrios Kourtesis
South-East European Research Centre
International Faculty, The University of Sheffield
24 Proxenou Koromila street
Thessaloniki, 54622, Greece
dkourtesis@seerc.org

Iraklis Paraskakis
South-East European Research Centre
International Faculty, The University of Sheffield
24 Proxenou Koromila Street
Thessaloniki, 54622, Greece
iparaskakis@seerc.org

Mike Stannett
Department of Computer Science
The University of Sheffeld
Regent Court, 211 Portobello Street
Sheffeld S1 4DP, United Kingdom
m.stannett@dcs.shef.ac.uk

ABSTRACT
As computing systems evolve and mature, they are also expected to grow in size and complexity. With the continuing paradigm shift towards cloud computing, these systems have already reached the stage where the human effort required to maintain them at an operational level is unsupportable. Therefore, the development of appropriate mechanisms for run-time monitoring and adaptation is essential to prevent cloud platforms from quickly dissolving into a non-reliable environment. In this paper we present our approach to enable cloud application platforms with self-managing capabilities. The approach is based on a novel view of cloud platforms as networks of distributed data sources - sensors. Accordingly, we propose utilising techniques from the Sensor Web research community to address the challenge of monitoring and analysing continuously flowing data within cloud platforms in a timely manner.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: *Reliability, availability, and serviceability* D.4.8 [**Operating Systems**]: Performance – Monitors, Operational analysis

Keywords
Cloud application platform; PaaS; Autonomic computing; MAPE-K; Self-management; Sensor Web; Semantic Sensor Web.

1. INTRODUCTION
With the paradigm shift towards cloud computing, the complexity of next generation service-based computing systems is soon expected to outgrow our capacity to manage them in a manual manner [5]. A similar problem was faced in the 1920s with telephony [14], when increased telephone usage led to the introduction of the automatic branch exchanges which eventually substituted human operators. Autonomic computing aims at an analogous goal today, seeking to improve complex computing systems by decreasing human intervention to a minimum. It is a concept that brings together many fields of IT with the purpose of creating computing systems that are capable of self-management – a feature that is central to the concept of cloud computing. In particular, the combination of two research areas, cloud computing and autonomic computing, has been attracting more and more attention over the past few years. As a recognition of the importance of research in this direction, the first Autonomic and Cloud Computing Conference[1], organised by ACM[2], will take place this year (2013).

To date, attempts to enable clouds with autonomic behaviour have focussed on the Infrastructure-as-a-Service (IaaS) level of cloud computing. Both academia and industry have been investigating and trying to develop efficient mechanisms of adapting to varying volumes and types of user requests by allocating the incoming workload across computational instances (i.e., load balancing), or by reserving and releasing computational resources upon demand (i.e., elasticity) [1, 19]. Both load balancing and elasticity are essential characteristics of cloud computing according to the National Institute of Standards and Technology [18].

However, more sophisticated adaptation scenarios, such as modifying the actual structure and/or behaviour of a deployed application at run-time, are much more difficult to automate, and at the moment are beyond the capabilities of common cloud platforms. As an example, consider a situation when hundreds of applications deployed on a cloud platform are using the platform's built-in notification service (e.g., for e-mail notifications). At some point this service crashes, affecting the quality of service of all the dependent applications. A possible solution in such circumstances would be to switch to an external notification service, automatically and transparently to the users. Unfortunately, at the moment there seem to be no self-management mechanisms of such a kind at the Platform-as-a-Service (PaaS) level. Even though much effort has been put into

HotTopiCS'13, April 20–21, 2013, Prague, Czech Republic.

[1] http://www.autonomic-conference.org/

[2] http://www.acm.org/

the development of self-management mechanisms at the IaaS level, self-adaptation capabilities of services at the PaaS level are as yet immature and not well theorised.

To achieve self-management at the PaaS level we are developing a self-adaptation framework, based on our novel concept of viewing cloud platforms as *sensor networks*. A sensor network is a computer accessible network of spatially distributed devices using sensors to monitor conditions at different locations, such as temperature, sound, pressure, etc. [4]. Accordingly, the presence of multiple distributed sources of information on a cloud platform, which have to be monitored to support self-management, allows us to draw a parallel between cloud platforms and sensor networks.

This similarity gives us confidence that we can re-use the positive experience of the Sensor Web research community in the context of dynamic monitoring and analysis of continuously flowing streams of data. To be more precise, we propose to employ techniques from the Semantic Sensor Web (SSW) [22] – a research field at the intersection of the Sensor Web and the Semantic Web. The SSW technologies, namely RDF data streams and SPARQL query engines, allow us to address the challenge of situation assessment and detection of critical conditions by processing multiple heterogeneous data streams. Moreover, by reasoning over OWL ontologies and SWRL rules we can benefit from existing reasoning machinery for diagnosis and identification of adaptation strategies. In our work we are following IBM's MAPE-K reference model for creating adaptation loops. As a first step towards the validation of this vision, in this paper we present a simple use case study which demonstrates the viability of our proposed approach.

The rest of the paper is organised as follows. Section 2 gives a short introduction to the state of the art in existing self-adaptation mechanisms in clouds and presents our motivation for this research in more detail. Section 3 introduces the principles of autonomic computing and explains the MAPE-K model. Section 4 is dedicated to related technologies from the areas of Semantic Web standards, the Sensor Web and the field of data streams. In section 5 we present and justify our vision of cloud platforms as sensor networks, as well as sketch out the architecture of the envisaged self-adaptation framework. A simple use case study, which has already been carried out, is also described in section 5.

2. MOTIVATION

2.1 Existing Self-Adaptation Mechanisms at the PaaS level

To support elasticity and load-balancing, two important features for cloud computing services, and ensure that appropriate service levels are maintained, cloud platforms continuously monitor the usage of deployed applications and available resources. In response to reaching a critical level of CPU/memory utilisation, additional computational instances can be launched and incoming user requests can be spread across instances evenly. In some cases, the process of adding/removing computational instances is managed by the cloud platform in a completely automated way (e.g., in OpenShift[3], Google AppEngine[4], or AWS Elastic Beanstalk[5]). In other cases, application developers are expected to build such capability by themselves or integrate it into their applications' source code using appropriate cloud platform APIs (e.g., in CloudFoundry[6] or Heroku[7]). External solutions may be utilised to automate the task of scaling applications up/down (e.g., the commercial offering HireFire[8] for Heroku). Some platforms, including OpenShift and Heroku, are also able to idle inactive applications based on users' recent activity to save resources and users' money.

A basic technique employed by several existing self-management mechanisms is "heart beat monitoring": detecting if an application has crashed and restarting it. Self-management behaviour of this kind is rather simple, and does not involve any sophisticated analysis of what the underlying problem in an application might be. Applications are treated as "black boxes" and usually, if they still do not operate after several attempts to restart, they are fully stopped and further action needs to be taken by users.

To support more in-depth monitoring of deployed applications and provide customers with visibility into resource utilisation and operation performance, some cloud platform operators either offer built-in monitoring tools (e.g., Amazon CloudWatch[9] or Google AppEngine Dashboard) or employ external monitoring frameworks (e.g., New Relic[10]). Common monitored metrics are CPU and memory utilisation, disk reads and writes, network traffic, etc. However, the monitoring frameworks only deal with problem detection and do not provide any means of automatically fixing a problem once it appears – this task is left to the administrators.

2.2 Problem Statement

In the era of the Internet of Services, when applications are increasingly dependent on third-party services, a failure of an external component at one point may lead to malfunctioning of a whole cloud-based application, without the hosting platform noticing it. Presently, cloud platforms are incapable of detecting and reacting to such situations [8]: they cannot understand whether a response from a service is correct or not, cannot substitute a malfunctioning or underperforming service with another, etc. The existing limitations make it necessary for platform administrators and application developers to be involved in the lifecycle of an application after it has been deployed to a cloud environment. That is, adaptations need to be performed manually by rewriting the application/platform code, recompiling, redeploying the application or restarting the platform.

As cloud platforms become more and more complex, the number of applications hosted on a platform grows, and changes to these applications become more frequent, we are running into the risk of a cloud platform becoming unmanageable, because the effort required to manage it has outgrown the human resources available for the task. The development of appropriate mechanisms for run-time monitoring and adaptation are therefore essential to prevent

[3] http://www.openshift.com

[4] https://appengine.google.com/

[5] http://aws.amazon.com/elasticbeanstalk/

[6] http://www.cloudfoundry.com

[7] http://www.heroku.com

[8] http://hirefireapp.com/

[9] http://aws.amazon.com/cloudwatch/

[10] http://newrelic.com/

cloud platforms from quickly dissolving into non-reliable environments.

3. BACKGROUND THEORY

3.1 Autonomic Computing

Inspired by the biological concept of autonomic systems, IBM introduced the term *autonomic computing* [13] in 2001 to refer to systems which are able to self-manage. IBM compared complex computing systems to the human body, and suggested that such systems should also demonstrate certain autonomic properties, that is, should be able independently to take care of regular maintenance and optimisation tasks, thus reducing the workload on system administrators. Though autonomic computing is the most commonly used and established term, throughout the rest of this paper we will be using the terms *autonomic computing*, *self-management*, and *self-adaptation* interchangeably.

According to Paul Horn [13] from IBM, who first suggested the systematic scientific approach of autonomic computing, the four fundamental properties of self-managing (i.e., autonomic) systems are: *self-configuration*, *self-optimisation*, *self-healing*, and s*elf-protection*. One of the possible ways of achieving these 4 characteristics is through *self-reflection*. A self-reflective system refers to the use of a causally connected self-representation to support the inspection and adaptation of that system [3]. It means that such a system is context-aware and self-aware of its internal structure and able to perform run-time adaptations, so that applied adaptations dynamically reflect on the state of the system (thus, possibly, triggering another adaptation cycle) [8]. The motivation behind self-reflection stems from the necessity to have systems which, when deployed in hostile and/or dynamically changing settings, are capable of reacting to various changes in the environment, based on the knowledge they already possess. In such scenarios, the capability of a remote system to perform automatic adaptations at run-time within a specific time frame is often of a great importance.

Depending on the extent to which modern computing systems support self-management, we can distinguish 5 levels of autonomicity [10], starting from the *basic* level, through *managed*, *predictive* and *adaptive* levels, and finally to the fully *autonomic* level, which is the ultimate goal of our work. At this level, system operation is governed by business policies and objectives, established by administrators at design-time. Users only interact with the system to monitor the business processes or alter the objectives.

3.2 MAPE-K Reference Model

One of the fundamental components of IBM's vision of autonomic computing is a reference model for autonomic control loops [14], which is usually referred to as the *MAPE-K* (Monitor, Analyse, Plan, Execute, Knowledge) adaptation loop, and depicted in Figure 1.

IBM's vision of autonomic computing was influenced by agent theory, and the MAPE-K model is similar to and was probably inspired by the generic model for intelligent agents proposed by Russell and Norvig [21]. According to this model, agents are equipped with sensors to perceive their environment. Then, based on the sensed values, they execute some actions on the environment. This process of sensing and acting upon sensed values clearly corresponds to the closed adaptation loop of the MAPE-K model. Applying the model to the domain of adaptations at the PaaS level, we now consider each of its elements in more detail.

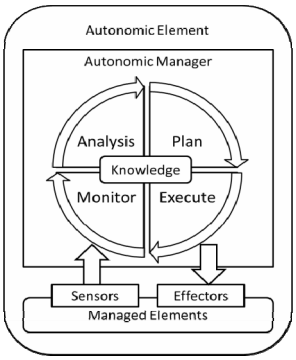

Figure 1. IBM's MAPE-K reference model for autonomic control loops (modified from [14]).

The *managed elements* represent any software and hardware resources which are enhanced with autonomic behaviour by coupling with an autonomic manager. In our case, the managed element may be the cloud platform as a whole, a web server, an operating system, an application, etc. Managed elements are equipped with *sensors* – they are software or hardware components responsible for collecting information about the managed elements. Managed elements are also equipped with *effectors* – components responsible for carrying out changes to the managed elements. Changes may be coarse-grained (e.g., substituting a Web service) or fine-grained (e.g., changing configuration of a Web service).

The *autonomic manager* is the core element of the model – it is a software component which implements the whole MAPE-K functionality. Usually it is configured by human administrators using high-level goals and uses the monitored data from sensors and internal (i.e. self-reflective) *knowledge* of the system to plan and execute low-level actions that are necessary to achieve these goals. The internal knowledge of the system, shared between the four MAPE components, includes an architectural model of the managed element (i.e., its components, connections between them, adaptation points, etc.), topological information, system logs, adaptation policies, etc. Autonomic computing policies can be expressed using some form of *event-condition-action* (ECA) rules, *goal* policies or *utility functions* [16]. Often the knowledge base is not static (i.e., populated once by humans at design-time), but rather evolves dynamically by accumulating new information at run-time (e.g., keeping track of detected failures and applied solutions).

The data collected by sensors allows the autonomic manager to *monitor* the execution of the managed element. For instance, we may be interested in monitoring such properties of deployed applications as CPU and memory utilisation, response times to user requests, I/O operations frequency, up time, etc. Two types of monitoring are usually identified in the literature [14]:

- *Passive* monitoring, also known as *non-intrusive*, assumes that no changes are made to the managed element. This kind of monitoring is targeted at the context of the managed element, rather than the element itself.
- *Active* monitoring, also known as *intrusive*, entails designing and implementing software in such a way that it provides some entry-points for capturing required properties (e.g., APIs).

The *analysis* component's main responsibility is to assess the current situation and detect failures or sub-optimal behaviour of the managed element. In its simplest form, the analysis engine, based on ECA rules, detects when a single monitored value is exceeding its threshold (e.g., CPU utilisation reaches 100%), and sends this diagnosis to the *planning* component, which is responsible for generating an appropriate adaptation plan (i.e., an action or a set of actions to be executed). Similarly, in its simplest form, it just follows the "action" part of an ECA rule. To make planning more effective, the autonomic manager uses an architectural model of the entire managed system, which reflects the managed system's behaviour, requirements, available resources, goals, etc. The model is updated through sensor data and used to reason about the managed system to plan adaptations. An advantage of the architectural model-based approach when planning is that, under the assumption that the model correctly mirrors the managed system, the architectural model can be used to verify that system integrity is preserved when applying an adaptation; that is, we can guarantee that the system will continue to operate correctly after the planned adaptation has been executed [20]. The *execution* stage is the final step in the MAPE-K adaptation cycle. This component is responsible for carrying out the adaptation plan generated at the previous stage to the managed element by means of effectors. Once changes have been applied to the system, a new adaptation cycle is triggered – newly generated values are monitored and analysed, an appropriate adaptation plan is generated and executed, and so on.

4. RELATED TECHNOLOGIES

4.1 Semantic Web

The Semantic Web is the extension of the World Wide Web that enables people to share content beyond the boundaries of applications and websites [12]. It is a mesh of information linked up in such a way as to be easily readable by machines, on a global scale. It can be understood as an efficient way of representing data on the World Wide Web, or as a globally linked database. As shown in Figure 2, the Semantic Web is realised through the combination of certain key technologies [12]. The technologies from the bottom of the stack up to the level of OWL have already been standardised by the W3C and are widely applied in the development of Semantic Web applications. The technologies are:

Universal Resource Identifiers (URIs) provide means for uniquely identifying Semantic Web resources. The Semantic Web should be able to represent text documents in different human languages, and *Unicode* serves this purpose. *XML* provides an elemental syntax for content structure within documents. It is not really a necessary component of the Semantic Web technologies in most cases, as alternative syntaxes exist, such as Turtle[11] or N3[12] for RDF. *Resource Description Framework* (RDF) is a framework for creating statements in the form of triples: subject – predicate – object. It enables the representation of information about resources in the form of graph – that is why the Semantic Web is sometimes called a Giant Global Graph. As noted above, an RDF-based model can be represented in a variety of syntaxes (e.g., RDF/XML, N3, and Turtle). *RDF Schema* (RDFS) provides a basic schema language for RDF. For example, using RDFS it is possible to create hierarchies of classes and properties.

Web Ontology Language (OWL) is used to formally define an ontology – "a formal, explicit specification of a shared conceptualisation" [23]. OWL extends RDFS by adding more advanced constructs to describe resources on the Semantic Web. By means of OWL and other ontology specification languages it is possible to explicitly define knowledge (i.e. concepts, relations, properties, instances, etc.) and basic rules in order to reason about this knowledge. OWL allows stating additional constraints, such as cardinality, restrictions of values, or characteristics of properties such as transitivity. It is based on Description Logics and thus brings reasoning power to the Semantic Web.

Semantic Web Rule Language (SWRL) extends OWL with even more expressivity, as it allows defining rules so that whenever the conditions specified in the body of a rule hold, then the conditions specified in the head must also hold. *SPARQL* is an RDF query language - it can be used to query any RDF-based data, including statements involving RDFS and OWL.

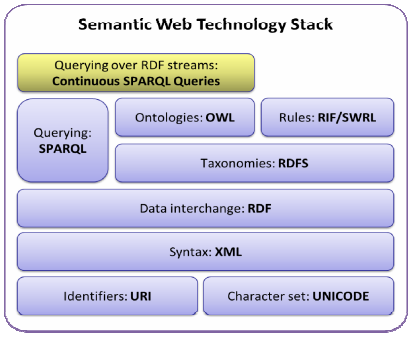

Figure 2: The Semantic Web technology stack.

4.2 Semantic Sensor Web

The rapid development and deployment of sensor technology involves many different types of sensors, not necessarily limited to physical devices. Anything that can calculate or estimate a data value can be perceived as a sensor [22] – an application component, an SQL query, a Web service, etc. The Open Geospatial Consortium[13] has recently launched the Sensor Web Enablement (SWE) project [4] – an initiative aiming at developing a suite of specifications related to sensors, sensor data models, and sensor Web services that will be accessible and controllable via the Web.

To address the requirements of SWE, the Semantic Sensor Web community has combined two research areas, the Semantic Web and the Sensor Web, so as to enable situation awareness by providing enhanced meaning for sensor observations [22]. It does this by adding semantic annotations to existing standard sensor languages of the SWE project. These annotations provide more meaningful descriptions and enhanced access to sensor data than SWE alone, and act as a linking mechanism to bridge the gap between the primarily syntactic XML-based metadata standards of the SWE and the RDF/OWL-based metadata standards of the Semantic Web. In association with semantic annotations, ontologies and rules play an important role in SSW for interoperability, analysis, and reasoning over heterogeneous sensor data [22]. For example, the Semantic Sensor Network (SSN) ontology [7] provides a common vocabulary to support modelling sensor networks of any complexity.

[11] http://www.w3.org/TeamSubmission/turtle/

[12] http://www.w3.org/TeamSubmission/n3/

[13] http://www.opengeospatial.org/

4.3 Data Streams

According to IBM, the world creates 2.5 quintillion bytes of data every day [15]. This data comes from everywhere: sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals, to name a few. Even though existing technologies seem to succeed in storing these overwhelming amounts of data, on-the-fly processing of newly generated data is a challenging task. An increasing number of distributed applications are required to process continuously streamed data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries [6]. A key research area addressing the issues involved in processing streamed data is Information Flow Processing (IFP); in contrast to the traditional static processing of data, IFP focuses on *flow processing* and *timeliness* [11]. The former means that data is not stored, but rather continuously flowing and being processed, and the latter refers to the fact that time constraints are crucial for IFP systems. These two main requirements have led to the emergence of a number of systems specifically designed to process incoming information streams according to a set of pre-deployed processing rules. A *data stream* consists of an unbounded sequence of values continuously appended, each of which carries a timestamp that typically indicates when it has been produced [6]. Usually (but not necessarily) recent values are more relevant and useful, because most applications are interested in processing current observations to achieve near real-time operation. Examples of data streams include sensor values, stock market tickers, social status updates, heartbeat rates, etc.

To cope with the unbounded nature of streams and temporal constraints, so-called *continuous query languages* [6] have been developed to extend the conventional SQL semantics with the notion of *windows*. A window transforms unbounded sequences of values into bounded ones, allowing the traditional relational operators to be applied. This approach restricts querying to a specific window of concern which consists of a subset of statements recently observed on the stream, while older information is (usually) ignored [2]. In our research we are mainly interested in SPARQL-based continuous query languages (see Figure 2), which are not yet part of the standardised Semantic Web stack, but becoming more and more used to query over RDF data streams. Up to date, there are several languages, mainly developed by the SSW community – CQELS [17], C-SPARQL [2], and $\text{SPARQL}_{\text{stream}}$ [6], to name a few.

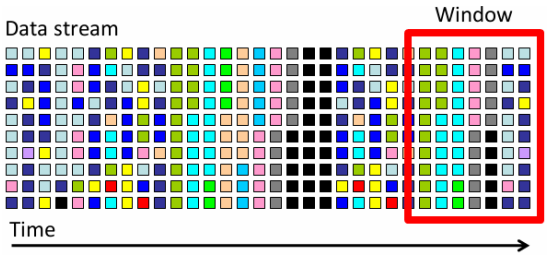

Figure 3: A data stream and a window [9].

The concepts of unbounded data streams and windows are visualised in Figure 3. The small squares represent tuples continuously arriving over time and constituting a data stream, whereas the thick rectangular frame illustrates the window operator applied to this unbounded sequence of tuples. As time passes and new values are appended to the data stream, old values are pushed out of the specified window, i.e. become no more relevant and may be discarded (unless there is a need for storing historical data for later analysis).

5. OUR APPROACH

5.1 Our Vision

Being highly complex and dynamic, cloud platforms are also characterised with extremely large volumes of information continuously generated and consumed both within and outside the platforms. In this context, looking at the cloud platform from an information management point of view, we can distil the following characteristics:

- *Dynamism*: in such dynamic systems as cloud platforms various sources of information are constantly generating data (which is then processed, stored, deleted, etc.) at an unpredictable rate. Moreover, various platform components are always evolving, so that new sources of information are coming up, while the old ones are disappearing, thus making the whole system even more dynamic.
- *Distributed nature*: the information may come from various logically and physically distributed sources (i.e., sensors). The first means that it may originate from databases, file systems, running applications, external Web services. The latter refers to the fact that all these "logical" sources may be deployed in separate virtual machines, servers and even datacentres.
- *Volume*: the amount of raw data being generated by (hundreds of) deployed applications, components of the platform, users, external services, etc. is huge. Even if we neglect the information flows that are not directly relevant to the context of self-management (i.e., the so called "noise"), the amount of information remaining is still considerable.
- *Heterogeneity*: originating from various distributed sources such as applications, databases, user requests, external services, etc., the information is *a priori* heterogeneous. Apart from the heterogeneity in the data representation, such as differences in data formats/encodings, there is also heterogeneity in the *semantics* of the data. For example, two separate applications with different business logic may store logging data in XML. In this case, the data is homogeneous in its format and, potentially, structure, but completely heterogeneous at the semantic level.

These characteristics are not unique to the problem domain of cloud platform monitoring. They are shared by many other problem domains where solutions based on Sensor Web technologies have been very successful, such as, for instance, environmental monitoring and traffic surveillance. The similarities between the problem domains give us confidence that we can apply a similar solution to the domain of cloud platform monitoring. Accordingly, we can think of a particular data source in a cloud platform as a *sensor* and the whole platform as a *network* of such sensors, similar to a distributed network of weather or city traffic sensing devices. Treating a cloud platform as a sensor network allows us to re-use existing solutions, developed and validated by the Sensor Web community, in the context of monitoring and analysis of streaming heterogeneous sensor data. A particularly promising direction to pursue is applying techniques from the Semantic Sensor Web area to implement the self-adaptation framework and enable self-management at the PaaS level. The next subsection presents a conceptual architecture of the proposed future framework and explains the role and benefits of SSW technologies.

5.2 Framework Architecture

Based on our Sensor Web vision of cloud platforms and employing the MAPE-K as an underlying model for developing our future self-adaptation framework, we describe our SSW-driven approach by sketching out a high-level architecture of the framework (see Figure 4).

In order to support both self-awareness and context-awareness of the managed elements, we need to employ some kind of architectural model describing the adaptation-relevant aspects of the cloud environment (e.g., platform components, available resources, connections between them, etc.) and the managed elements (e.g., entry-points for monitoring and execution). For these purposes we propose using OWL ontologies to represent the self-reflective knowledge of the system. Such an architectural model, represented with OWL, will also serve as a common vocabulary of terms, shared across the whole managed system. At this stage of our research we distinguish 3 main elements of the framework: triplification engine, continuous SPARQL query engine and OWL/SWRL reasoning engine. Accordingly, our ontological classes and properties will serve as building blocks for creating RDF streams, SPAQRL queries and SWRL rules.

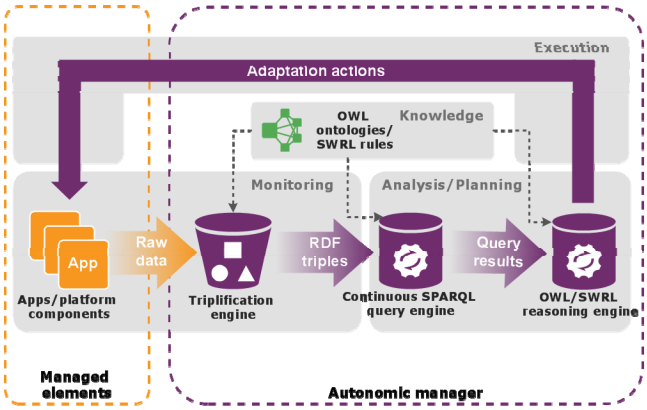

Figure 4: Framework architecture

The triplification engine is responsible for consuming and "homogenising" the data generated by deployed applications, platform components, external services, etc. The engine takes as input streams of raw data, and generates streams of RDF triples. There are already existing tools for converting data stored in relational databases into RDF, using special mapping languages (e.g., R2RML[14]), and analogous tools can be envisaged for RDF stream generation. Using RDF as a common format for representing streaming data, and OWL concepts as subjects, predicates and objects of the RDF triples, allows us to benefit from human-readability and extensive support of query languages. The triplification step may be omitted if the monitored data is already represented in RDF format using ontological classes and properties.

The next step of the information flow within our framework is the continuous SPARQL query engine. This component takes as input the flowing RDF data streams generated by the triplification engine and evaluates pre-registered continuous SPARQL queries against them, to support situation assessment. In the first instance, situation assessment includes distinguishing between usual operational behaviour and critical situations by matching them against critical condition patterns. The process of encoding patterns is iterative, that is, new critical conditions are constantly added to ensure the pattern base remains up-to-date. Machine learning techniques may be employed in this context to assist with maintaining the list of critical condition patterns. We propose using one of the existing continuous SPARQL languages to encode critical condition patterns. By registering an appropriate SPARQL query against a data stream, we will be able to detect a critical situation with a minimum delay - the continuous SPARQL engine will trigger as soon as RDF triples in the stream match the WHERE clause of the query. With SPARQL as a query language it is also possible to benefit from inference capabilities – that is, apart from just querying data, we may also be able to perform some reasoning over RDF triples. The reasoning capabilities depend on the entailment regime of the continuous SPARQL language to be selected, and respective tool support. Supporting "on-the-fly" processing of constantly flowing data, generated by hundreds of sources, as well as employing one of the existing RDF streaming engines, help us in achieving near real-time behaviour of the adaptation framework.

Once a critical condition has been detected, a corresponding adaptation plan has to be generated. This step requires not just associating the "event" and "condition" parts of an ECA rule with its "action" part, but rather more complex reasoning over the possible reasons for a problem, and identification of potential adaptation strategies. We envisage addressing this challenge (at least partially) with OWL ontologies and SWRL rules, which provide sufficient expressivity to define adaptation policies, and will exempt us from the effort-intensive and potentially error-prone task of implementing our own analysis engines from scratch. Rather, we will rely on the built-in reasoning capabilities of OWL ontologies and SWRL rules, so that the routine of reasoning over a set of situations and adaptation alternatives is done by an existing, tested, and highly optimised mechanism.

When defining adaptation policies with OWL and SWRL, we, as application developers and platform administrators, may benefit from the following [8]:

- Separation of concerns: with ontologies and rules separated from the platform/application programming code, it is easier to make changes to adaptation policies on the fly (i.e. without recompiling, redeploying and restarting the platform/application) and to maintain the adaptation framework.
- Flexible adaptation at any adaptation scope: from the lowest level of programming classes and variables to the upper-most level of the whole platform. The ontology-based approach is generic and can be potentially applied to adaptations at the IaaS level, and any other distributed system as well.
- Increase in reuse, automation and reliability: once implemented and published on the Web, ontologies are ready to be re-used by third parties, thus saving ontology engineers from "reinventing the wheel". Since the reasoning process is automated and performed by a reasoning engine, it is not prone to "human factors" and is more reliable.

5.3 Use Case

Let us consider a scenario where a number of applications are deployed on a cloud platform and rely on the platform's built-in notification service. In this case, the managed elements are the applications, and the response time of the notification service is

[14] http://www.w3.org/TR/r2rml/

the parameter being monitored. The autonomic manager possesses knowledge about the applications and their dependency on the notification service, as well as the list of third-party services that can act as substitutes. The knowledge base, among other things, also includes a policy stating that if the internal notification service is slow to respond, then it needs to be replaced with an external one. At some point in time the notification service gets overloaded and cannot process all incoming requests. In order to satisfy Service Level Agreements we need to detect such situations and switch over all the dependent applications to an external substitute. In this scenario the main focus has been on the monitoring and analysis steps of the MAPE-K model, whereas the planning and execution steps have been left aside (this example is intended only to demonstrate the viability of our approach, and is correspondingly simplified).

```
@prefix : <http://www.seerc.org/ontology.owl#> .
<http://www.seerc.org/ontology.owl> rdf:type owl:Ontology .
  :Service rdf:type owl:Class .
  :Time    rdf:type owl:Class .
  :hasResponseTime rdf:type owl:ObjectProperty ,
                   rdfs:domain :Time .
  :isEquivalent    rdf:type owl:ObjectProperty ,
                            owl:SymmetricProperty ;
                   rdfs:range :Service ;
                   rdfs:domain :Service .
  :hasHighResponseTime rdf:type owl:DatatypeProperty ,
                       rdfs:range xsd:Boolean .
  :hasValue            rdf:type owl:DatatypeProperty ,
                       rdfs:range xsd:int .
  :needsSubstitution   rdf:type owl:DatatypeProperty ,
                       rdfs:range xsd:Boolean .
```

Listing 1. Simplified OWL ontology.

Following our approach, the framework utilises RDF data streams to support on-the-fly querying and achieve minimal reaction delays. The monitored values are represented in RDF format using resources (i.e., subjects and objects) and properties (i.e., predicates) defined in an OWL ontology. Its simplified version in the Turtle notation is presented in Listing 1. Among other things, it includes the classes *Service* and *Time*, and the properties *hasResponseTime*, *hasHighResponseTime*, *isEquivalentTo* and *needsSubstitution*. Listing 2 illustrates the concept of RDF data streams. Each line is an RDF triple (in the N3 notation), representing an event (i.e. the fact that a service's response time has changed), and is annotated with a timestamp.

```
@prefix ex:<http://www.seerc.org/ontology/>

ex:#Service1 ex:hasResponseTime; 890.  [2012-09-18 13:24:54]
ex:#Service1 ex:hasResponseTime; 1110. [2012-09-18 13:24:56]
ex:#Service1 ex:hasResponseTime; 1300. [2012-09-18 13:24:58]
ex:#Service1 ex:hasResponseTime; 5450. [2012-09-18 13:25:13]
ex:#Service1 ex:hasResponseTime; 6000. [2012-09-18 13:25:20]
ex:#Service1 ex:hasResponseTime; 6700. [2012-09-18 13:26:15]
```

Listing 2. RDF stream.

Listing 2 shows a sudden increase in the response time of *Service1*. In order to detect such a critical situation, we need to have registered a continuous SPARQL query, returning relevant results whenever some monitored value has been continuously exceeding 5000 milliseconds for more than 60 seconds. Listing 3 contains a simple C-SPARQL query and demonstrates the usage of continuous query languages.

```
PREFIX ex:<http://www.seerc.org/ontology/>
SELECT DISTINCT ?service
FROM STREAM http://www.seerc.org/stream
    [RANGE 60s STEP 1s]
WHERE { ?service ex:hasResponseTime ?time .
    FILTER (?time > 5000) }
```

Listing 3. C-SPARQL query.

Once we have detected a critical condition we need to diagnose the problem and reason about possible adaptation actions. Listing 4 demonstrates the usage of SWRL in the context of diagnosing a problem and inferring a possible adaptation strategy.

Rule 1: Has high response time

```
    Service(?s1) ^ Time(?t)
  ^ hasResponseTime(?s1, ?t)
  ^ greaterThan(?t, 5000)
->  hasHighResponseTime(?s1, true)
```

Rule 2: Needs substitution

```
    hasHighResponseTime(?s1, true)
  ^ Service(?s2) ^ isEquivalentTo(?s1, ?s2)
->  needsSubstitution(?s1, ?s2)
```

Listing 4. SWRL rules.

The first rule states that if a response time from a service is greater than 5 seconds, that service can be considered to have a high response time. The second rule says that if a service has a high response time and there is an equivalent service, then the slow service should be substituted by an alternative one. Please note that the RDF stream, the C-SPARQL query and the SWRL rules are defined using concepts from one and the same ontology, stored at http://www.seerc.org/ontology. Having such a common shared vocabulary increases the reliability, consistency and reusability of our approach.

To implement this use case scenario, we have extended the SSN ontology with the concepts presented above, using the ontology editor Protege[15]. We also developed two applications using the Java Spring framework - an autonomic manager and a client which continuously calls a Web service. Both were deployed to CloudFoundry. The choice of CloudFoundry as a sandbox for our use case was because it offers: (i) an easy-to-use Eclipse plug-in for developing, testing and deploying Spring applications, (ii) a portable version of the CloudFoundry cloud platform, which can run on a single laptop for testing purposes (the so-called MicroCloudFoundry), (iii) extensive support by the developers' community.

To support communication between its components, this platform uses the RabbitMQ[16] messaging service – a convenient environment for implementing RDF data streams, where each RDF triple is sent as a separate message. Accordingly, every time the client calls the Web service, it sends an RDF triple containing information about the response time to a RabbitMQ queue, to which the autonomic manager is already subscribed. We use the C-SPARQL engine library to enable the autonomic manager with capabilities to process RDF streams. Accordingly, the autonomic manager, by registering an appropriate C-SPARQL query against the queue, is notified as soon as the RDF triples in the stream satisfy the WHERE clause of the query.

[15] http://protege.stanford.edu/

[16] http://www.rabbitmq.com/

To support manipulating and reasoning over the OWL ontology and SWRL rules we have used the OWL API[17] Java library, which implements several reasoners, and is easy to use. As a result, having detected a critical situation, the autonomic manager is able to analyse the problem and infer one or more possible adaptation strategies (the actual generation of plans for executing adaptation strategies is outside the scope of this work). By means of reasoning over the SWRL rules, it is able to deduce that the service has high response time and needs to be substituted.

6. CONCLUSION

In this paper we have presented a framework for introducing self-management in Platform-as-a-Service environments. Devising this framework, we drew parallels between the problem domain of self-adaptation in cloud platforms, and other problem domains where successful solutions have come from applying the Sensor Web technology, such as traffic surveillance and environmental monitoring. Accordingly, our approach is based on a novel view of cloud platforms as networks of distributed data sources - sensors. Based on this vision and applying principles of the Semantic Sensor Web, we described our approach and outlined a conceptual architecture for our framework. As a proof of concept, we are planning to further develop the framework by introducing more sophisticated and complex adaptation policies, increasing the number of monitored parameters, and trying other existing continuous SPARQL languages. Future work also includes investigations of the scalability of the framework and portability across different cloud platforms.

7. REFERENCES

[1] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. and Zaharia, M. 2009. Above the Clouds: A Berkeley View of Cloud Computing.

[2] Barbieri, D., Braga, D., Ceri, S., Della Valle, E. and Grossniklaus, M. 2010. Stream Reasoning: Where We Got So Far. *Proceedings of the 4th International Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic (NeFoRS)* (2010).

[3] Blair, G.S., Coulson, G. and Grace, P. 2004. Research directions in reflective middleware: the Lancaster experience. *Proceedings of the 3rd workshop on Adaptive and reflective middleware* (New York, NY, USA, 2004), 262–267.

[4] Botts, M., Percivall, G., Reed, C. and Davidson, J. 2008. OGC® sensor web enablement: Overview and high level architecture. *GeoSensor networks*. (2008), 175–190.

[5] Brazier, F.M.T., Kephart, J.O., Van Dyke Parunak, H. and Huhns, M.N. 2009. Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda. *IEEE Internet Computing*. 13, 3 (Jun. 2009), 82 –87.

[6] Calbimonte, J.-P., Jeung, H., Corcho, O. and Aberer, K. 2012. Enabling Query Technologies for the Semantic Sensor Web. *International Journal On Semantic Web and Information Systems*. (to appear. 2012).

[7] Compton, M. et al. 2012. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*. 17, 0 (Dec. 2012), 25–32.

[8] Dautov, R., Paraskakis, I. and Kourtesis, D. 2012. An ontology-driven approach to self-management in cloud application platforms. *Proceedings of the 7th South East European Doctoral Student Conference (DSC 2012)* (Thessaloniki, Greece, 2012), 539–550.

[9] Della Valle, E. 2012. Challenges, Approaches, and Solutions in Stream Reasoning.

[10] Ganek, A.G. and Corbi, T.A. 2003. The dawning of the autonomic computing era. *IBM Systems Journal*. 42, 1 (2003), 5–18.

[11] Gucola, G. and Margara, A. 2011. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys*. (2011).

[12] Hitzler, P., Krötzsch, M. and Rudolph, S. 2009. *Foundations of Semantic Web Technologies*. CRC Press.

[13] Horn, P. 2001. Autonomic Computing: IBM's Perspective on the State of Information Technology. *Computing Systems*. 15, Jan (2001), 1–40.

[14] Huebscher, M.C. and McCann, J.A. 2008. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv*. 40, 3 (2008), 1–28.

[15] IBM 2012. Bringing Big Data to the Enterprise. http://www-01.ibm.com/software/data/bigdata.

[16] Kephart, J.O. and Walsh, W.E. 2004. An artificial intelligence perspective on autonomic computing policies. *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings* (Jun. 2004), 3 – 12.

[17] Le-Phuoc, D., Dao-Tran, M., Parreira, J.X. and Hauswirth, M. 2011. A native and adaptive approach for unified processing of linked streams and linked data. *Proceedings of the 10th international conference on The semantic web - Volume Part I* (Berlin, Heidelberg, 2011), 370–388.

[18] Mell, P. and Grance, T. 2009. The NIST definition of cloud computing. *National Institute of Standards and Technology*. 53, 6 (2009), 50.

[19] Natis, Y.V., Knipp, E., Valdes, R., Cearley, D.W. and Sholler, D. 2009. *Who's Who in Application Platforms for Cloud Computing: The Cloud Specialists*. Gartner Research.

[20] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S. and Wolf, A.L. 1999. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and their Applications*. 14, 3 (Jun. 1999), 54 –62.

[21] Russell, S.J., Norvig, P., Canny, J.F., Malik, J.M. and Edwards, D.D. 1995. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, NJ.

[22] Sheth, A., Henson, C. and Sahoo, S.S. 2008. Semantic sensor web. *Internet Computing, IEEE*. 12, 4 (2008), 78–83.

[23] Studer, R., Benjamins, V.R. and Fensel, D. 1998. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*. 25, 1–2 (Mar. 1998), 161–197.

[17] http://owlapi.sourceforge.net/