On the Role of Stream Reasoning in Run-time Monitoring and Analysis in Autonomic Systems

Rustem Dautov[1], Mike Stannett[2] and Iraklis Paraskakis[1]

[1] South-East European Research Centre (SEERC),
City College – International Faculty of the University of Sheffield,
24 Proxenou Koromila Street, 54622 Thessaloniki, Greece
{rdautov, iparaskakis}@seerc.org

[2] Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield S1 4DP, United Kingdom
m.stannett@dcs.shef.ac.uk

Abstract. Both academia and industry have been putting a lot of effort into investigating various solutions to the problem of autonomic management of complex distributed computer systems. Existing approaches, based on the principles of autonomic computing, typically rely on monitoring a managed system, analysing the monitored values, and executing corresponding adaptation actions. By addressing existing limitations, we present a novel research area – stream reasoning – in the context of run-time monitoring and analysis in autonomic systems. This research area aims at enhancing existing stream processing techniques by adding dynamic reasoning support, thereby fitting scenarios when timely, yet precise analysis of dynamically flowing data is needed. In this paper, by discussing benefits and shortcomings, we speculate on the potential role of the approach in autonomic computer systems.

1 Introduction

The increasing complexity, cost and heterogeneity of distributed computer systems have motivated researchers to focus on the development of self-management mechanisms following the principles of autonomic computing. The goal of autonomic computing is to create computer systems capable of managing themselves to a far greater extent than they do today [1]. The fundamental requirements for a self-managing system to be effective are self- and context-awareness: in order to adapt to changes in its operating environment, a computer system should first observe these changes and then correctly interpret them – that is, to support self-management, it should perform both self-monitoring and contextual analysis.

Even though the interpretation of monitored values is the key to prompt and efficient adaptations, the analysis support in existing approaches aiming at implementing self-managing systems, is still hindered with certain limitations [2] – analysis rules are hard-wired into the code, analysis of the monitored values is delayed, it is not well

automated, etc. To address (at least partially) these constraints, in this paper we describe the potential of *stream reasoning* – a novel research area at the intersection of the Semantic Web and stream processing – to support run-time monitoring and analysis. We present a non-exhaustive list of benefits of utilising stream reasoning techniques when implementing monitoring and analysis mechanisms within autonomic systems, including, e.g., timeliness, intelligence, separation of concerns, and higher automation. The ideas to be presented have potential shortcomings as well, which are also discussed in this paper. However, positive aspects of this approach make us believe that they are worth further investigation.

The rest of the paper is organised as follows. Section 2 gives a short introduction to the motivation behind the presented work – the need for enhanced run-time monitoring and analysis in distributed computer systems. It lists some challenges associated with monitoring such complex systems, and outlines some basic principles an effective analysis mechanism should follow. Section 3 addresses these challenges and principles by introducing the notion of data streams. It also briefly explains existing approaches in this field, such as data stream management systems and Complex Event Processing. Section 4 presents the concept of stream reasoning as a Semantic Web-enabled extension to stream processing – it lists potential benefits stream reasoning can bring in the context of run-time monitoring and analysis. Section 5 discusses some potential shortcomings associated with the presented ideas, and Section 6 concludes the paper.

2 Background and Motivation

2.1 Need for Monitoring

Due to the growing importance of distributed systems such as service-based applications, clouds and grids in recent years, the scientific community focused on the theoretic foundations and research on how to make such complex systems adaptive and sustainable, often referring to the original self-* principles of *autonomic computing* [3, 4] – a concept that brings together many fields of computing with the purpose of creating computer systems that self-manage. IBM was the first to introduce this term in 2001 [5] to refer to systems which are able to adapt themselves to changes happening in the surrounding environment, and suggested a reference model for creating closed adaptation loops known as *MAPE-K* (Monitor, Analyse, Plan, and Execute based on Knowledge). IBM compared complex computer systems to the human body, and suggested that such systems should also demonstrate certain autonomic properties, that is, should be able to independently take care of regular maintenance and optimization tasks, thus reducing the workload on system administrators [6]. These challenges of creating closed control loops had been already addressed in other research areas, including both Agent Theory [6, 7] and Control Theory [6], which inspired IBM in their vision of autonomic computer systems.

Inspired by IBM's Autonomic Computing manifesto and referring to the original self-* characteristics of autonomic systems, early theoretical works served as a groundwork for more and more prototypical implementations of self-managing mech-

anisms in various computer systems [3]. Existing self-adaptation mechanisms typically follow the agent-driven approach of sensing and acting upon sensed values by (fully or partially) implementing control feedback loops, such as the already mentioned MAPE-K model or the CADA model (Collect, Analyse, Detect, Act) [8]. The common component in most of the solutions implementing autonomic computing principles is a monitoring mechanism which is the "trigger" either for further run-time adaptations, or else for post-mortem data analysis. In a broad sense, monitoring may be defined as a process of collecting and reporting relevant information about the execution and evolution of a computer system [9].

Depending on a particular purpose of a designed service-based system, monitoring activities can be used [3]:

- to perform run-time analysis of system correctness, i.e., checking the system execution and its parameters against certain specifications that distinguish "correct" and "failing" situations;
- to perform the diagnosis and recovery of any identified faults;
- to instantiate and guide the optimisation activities regarding resource allocation (e.g., in a cloud computing scenario);
- to support dynamic adaptation of the application to the relevant change in its environment;
- to support the long-term adaptation (i.e., evolution) of a computer system, etc.

All these monitoring processes target the collection of data for a specific artefact, referred to as the *monitored subject* [9]. Depending on the context, the range of monitored subjects can be rather broad, including applications, operating systems, networks, Web servers, etc.

Two types of monitoring are usually identified in the literature [6]:

- *Passive* monitoring, also known as *non-intrusive*, assumes that no changes are made to the managed element. This kind of monitoring is generally targeted at the context of the managed element.
- *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).

Usually, monitoring is performed under one or both of the following main modes for collecting information [9]:

- *Polling mode*: querying or directly interacting with the monitored subject within regular time intervals.
- *Push mode*: propagating events or other data from the information sources of the monitored subject to the monitoring mechanism.

Even though monitoring activities also include such techniques as post-mortem log analysis, data mining, online and offline testing, etc. [10], throughout the rest of the paper we will only refer to the "classical" notion of run-time monitoring.

2.2 Need for Analysis

Monitoring on its own is not enough to enable adaptations to be carried out. For example, suppose we are monitoring response times from a Web service, and the observed value is 5 seconds. What does this value tell us? How can we know if it is a slow response or not? What is really needed is the analysis support, so that comparing observed values against expected ones, we can decide whether the current state of the system is faulty or potentially leads to a failure. The analysis component's main responsibility is to assess the current situation and detect failures or suboptimal behaviours of the managed system. In other words, when coupled with an analysis component, monitoring will be important in supporting problem determination and recovery when a fault is found or suspected [11]. Some of the common drawbacks existing analysis approaches tend to suffer from, are [2, 10]:

- *Rigidness*: analysis rules are "hard-coded", which hinders possible modifications.
- *Low intelligence*: analysis mechanisms are not sophisticated and intelligent enough to produce a precise diagnosis when it comes to complex, non-trivial scenarios.
- *Delayed execution*: analysis processes take place after monitored values have been observed, which decreases the actuality of the results.
- *Low level of automation*: the analysis is (fully or partially) performed by human operators, which again results in low actuality of the results and human errors.

With the development of highly distributed, complex and dynamic systems, such as service-based application systems, clouds, grids, etc. where large volumes of data are continuously generated and consumed, timely problem determination becomes an even more challenging task. In this respect, from an information management point of view, we can distil the following challenges of existing computer systems [12], which have to be taken into consideration when implementing monitoring and analysis mechanisms:

- *Dynamism*: various sources of information are constantly generating data (which is then processed, stored, deleted, etc.) at an unpredictable rate. Moreover, various system components are always evolving, so that new sources of information are coming up, while 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. The first means that it may originate from databases, file systems, running applications, or external Web services. The latter refers to the fact that all these "logical" sources may be deployed in separate virtual machines, servers and even distinct datacentres.
- *Large volumes*: the amount of raw data being generated (for example, within a social network) is huge. Even if we neglect those information flows that are not directly relevant to the context of self-management (i.e., the so-called "noise"), the amount of data remaining is still considerable.
- *Heterogeneity*: originating from various distributed sources such as applications, databases, user requests, external services, etc., the information is a priori hetero-

geneous. 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 completely separate applications from different domains 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.

Accordingly, in order to address these challenges, the analysis mechanism should follow the following principles:

- *Automation*: obviously, with the ever-expanding complexity of software systems, the process of analysing monitored data should be automated. The automated analysis may be complemented by human supervision, but the main process should be performed in an autonomic manner.
- *Run-time execution*: the analysis should be performed dynamically – that is, as soon as new data arrives it has to be evaluated. As opposed to static approaches, where monitored values are first stored (e.g., in a database or log files) for later analysis, dynamic analysis assumes we will be processing streams of incoming data "on the fly", without storing it on a hard drive.
- *Flexibility:* the analysis mechanism should be flexible enough so as to accommodate the evolving and expanding nature of existing software systems. Consider a scenario where the analysis logic is "hardcoded" with numerous "if-then" and "switch-case" operators – as opposed to declarative approaches to defining rules, introduction of a new rule to the analysis mechanism in this circumstances will require rewriting the code, recompiling, and redeploying/restarting the whole system [13].

3 Data Streams

The challenges associated with run-time monitoring and analysis of contemporary complex distributed software systems, listed in the previous section, are already attracting researchers' attention. In the era of Big Data, we create 2.5 quintillion bytes of data every day – so much that 90% of the data in the world today has been created in the last two years alone [14]. 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 [15]. A key research area addressing these issues is Information Flow Processing (IFP); in contrast to the traditional static processing of data, IFP focuses on *flow processing* and *timeliness* [16]. 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. In this context, a *data stream* comprises an unbounded sequence of continuously appended values, each of which carries a timestamp that typically indicates when it has been produced [12, 15]. 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* [15] 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 [12, 17]. Windows can be specified both in terms of *tuples*, where a window comprises a given number of most recent elements regardless of arrival time; and *time*, where a window comprises all elements which have arrived during some specified time frame.

The concepts of unbounded data streams and windows are visualised in Figure 1. 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 [12]. As time passes and new values are appended to the data stream, old values are pushed out of the specified window, i.e. they become irrelevant and may be discarded (unless there is a need for storing historical data for later analysis).

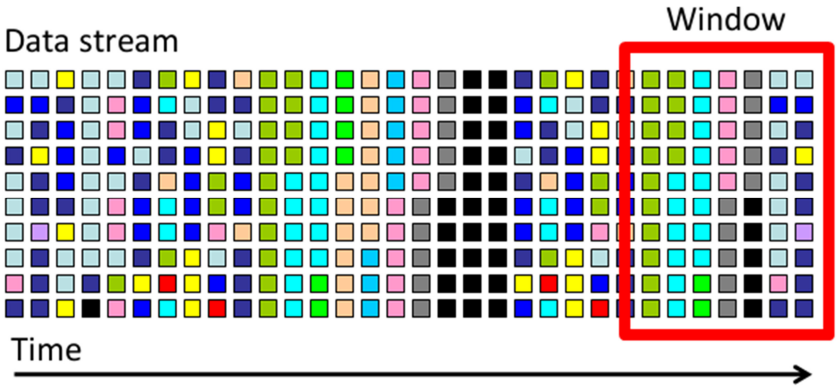

Fig. 1. A data stream and a window (modified from [17]).

Data stream management systems (DSMSs) – an evolution from traditional static database management systems – have been developed to process streams of data coming from different sources to produce new data streams as output [16]. They target transient, continuously updated data and run standing (i.e., continuous) queries, which fetch updated results as new data arrives. Complex Event Processing (CEP) goes one step further and tries to detect complex event patterns, themselves consisting of sim-

pler atomic events, within a data stream [16]. Accordingly, from this point of view, flowing information items can be seen as notifications of events happening in the external world. An event may be anything that changes the current state of affairs, e.g. a fire alarm, social status update, change of traffic light, etc. Accordingly, the focus of this perspective is on detecting occurrences of particular patterns of (low-level) events that represent higher-level events. A notification of a complex event to the interested parties is triggered if and only if a corresponding pattern of lower-level events is detected.

4 Stream Reasoning

Stream Reasoning research [18] goes even further by enhancing the continuous querying with the run-time reasoning support – that is, with capabilities to infer additional, implicit knowledge based on already given, explicit facts. Barbieri et al. [21], who were the first to use the term 'stream reasoning' for the new concept, defined it as "*reasoning in real time on huge and possibly noisy data streams, to support a large number of concurrent decision processes*". As data streams have become more and more common on the Web (e.g. stock exchange movements, weather information, sensor readings, social networking activity notifications, etc.), the combination of data stream processing techniques with data streams distributed across the Web came as a natural fit, and this in turn required new ways of coping with the typical openness and heterogeneity of the Web environment – in this context, the role of Semantic Web technologies is to facilitate data integration in open environments, and thus help to overcome these problems by using uniform machine-readable descriptions to resolve heterogeneities across multiple data streams [19]. For example, the Semantic Sensor Web [20] represents an attempt to enable more expressive representation, advanced access, and formal analysis of avalanches of sensor values in such domains as traffic surveillance, environmental monitoring, house automation and tracking systems, by encoding sensor descriptions and sensor observation data with Semantic Web languages.

As Semantic Web technologies are mainly based on Description Logics, their application to data stream processing also offers new opportunities to perform reasoning tasks over continuously and rapidly changing flows of information. In particular, stream reasoning utilises and benefits from the following Semantic Web technologies:

- *Resource Description Framework*[1] (*RDF*) as a uniform format for representing streamed heterogeneous data as a collection of (*subject*, *predicate*, *object*) triples using a vocabulary defined in an OWL ontology;
- *OWL* ontologies[2] and *SWRL* rules[3] as a source of static background knowledge, which may also act as a vocabulary of terms for defining RDF triples;

[1] http://www.w3.org/RDF/
[2] http://www.w3.org/TR/owl-features/
[3] http://www.w3.org/Submission/SWRL/

- *SPARQL*-based[4] continuous query languages as a way of querying RDF streams and performing reasoning tasks by combining them with the static background knowledge.

As a result, several prominent stream reasoning approaches have emerged: C-SPARQL [22], CQELS [23], ETALIS [24], and $SPARQL_{stream}$ [15] to name a few. These stream reasoning systems aim at preserving the core value of data stream processing, i.e. processing streamed data in a timely fashion, while providing a number of additional features [19]:

- *Support for advanced reasoning*: depending on the extent to which stream reasoning systems support reasoning, it is possible not only to detect patterns of events (as CEP already does), but in addition to perform more sophisticated and intelligent detection of failures by inferring implicit knowledge based on pre-defined facts and rules (i.e., static background knowledge).
- *Integration of static background knowledge with streamed data*: it is possible to match data stream values against a static background knowledge base (usually represented as an ontology), containing various facts and rules. This separation of concerns allows for seamless and transparent modification of analysis rules constituting the static knowledge base.
- *Support for expressive queries and complex schemas*: ontologies, acting as static background knowledge, also serve as a common vocabulary for defining complex queries. This means that classes and properties constituting an ontology provide "building blocks" and may be used for defining queries of required expressivity.
- *Support for logical, data and temporal operators*: to cope with the unlimited nature of data streams, stream reasoning systems extend conventional SQL-based logical and data operators with temporal operators. This allows us to limit an unbounded stream to a specific window, and also to detect events following one after another chronologically.
- *Support for time and tuple windows*: as described earlier, windows may be specified either by time-frame, or else by the number of entries to be retained, regardless of arrival time.

Taking into consideration these features of stream reasoning systems, we believe that they provide a promising foundation for implementing monitoring mechanisms with built-in analysis support. Our initial experiments implementing monitoring and analysis mechanisms within a cloud application platform using C-SPARQL, an OWL ontology, and SWRL rules, are reported in [12], and suggest that this approach is viable and has the potential to provide the required level of analysis support.

[4] http://www.w3.org/TR/rdf-sparql-query/

5 Demonstrating Benefits of Stream Reasoning

To demonstrate the benefits of Stream Reasoning techniques in the context of run-time monitoring and analysis of data streams, we now present a simple scenario and describe how it can be potentially implemented using: (i) traditional stream processing, (ii) Complex Event Processing, and, finally, (iii) Stream Reasoning.

Let us assume that we are interested in monitoring response times from various Web services constituting a composite service-based application system, so as to detect, for example, if a web server hosting a group of services has crashed or a particular service is getting overloaded. The examples presented below are intended for illustrative purposes, and are correspondingly simplified.

5.1 Monitoring and Analysis with Traditional Stream Processing

Using traditional stream processing techniques, we are able to register corresponding continuous queries, which would trigger every time a new response time value appears on the data stream. Then these monitored values should be fed to an analysis engine – typically, a hard-coded software component, which implements the analysis routine. For example, to detect if response time from a particular service has been gradually increasing during a given time interval – that is, to predict if it will soon become overloaded, the analysis engine should evaluate input values against a predefined set of hard-coded rules (e.g., numerous "if-then" and "switch-case" operators). Obviously, this approach suffers from complexity, low performance, low flexibility and potential human error.

5.2 Monitoring and Analysis with Complex Event Processing

CEP can help improve the situation by partially shifting the analysis routine to the monitoring component. For example, a single change in response time from a Web service can be seen as an atomic event, whereas a sustained pattern of increasing response times from a given Web service represents a complex event. By employing CEP techniques, we are exempted from the routine of defining temporal relations between monitored events – instead, we use a temporal query operator, and in the WHERE clause of the corresponding continuous query we specify the order of the events to be detected. It is worth noting, however, that defining CEP queries is not something straightforward, and, depending on the complexity of the monitored events, may result in clumsy, multi-level and nested constructions.

5.3 Monitoring and Analysis with Stream Reasoning

As we have seen, stream reasoning allows us to benefit from integrating static background knowledge with data streams, and performing reasoning tasks over values observed on those streams based on this knowledge. Let us assume that we have a background ontology, which, among other things, includes knowledge about which

Web services are hosted on which Web servers. Unlike CEP, which only allowed us to detect an increase in response time from a particular Web service, adding in the background knowledge allows us to detect cases where *several* services, all hosted on the same server, get overloaded, indicating a situation where the *Web server itself* is becoming overloaded.

A capability to resolve subclass relationships at run-time is another benefit of stream reasoning in the context of run-time monitoring and analysis. Instead of registering (hundreds of) separate queries for every single instance of the Web services being monitored (as we would do in CEP), we can register a single query for the superclass `Service`, and then define a simple background taxonomy specifying its subclasses. This taxonomy helps the stream reasoner understand whether an element observed on the stream is a subclass of `Service`, and should therefore be evaluated against the defined query.

6 Potential Shortcomings

Despite the benefits we have described of using stream reasoning in the context of run-time monitoring and analysis, this approach is not a "silver bullet" and has potential shortcomings. We will try to summarise them in this section:

- *Need for unified data representation*: in order to apply reasoning to a data stream, its heterogeneous values must first be represented in a common format – RDF. Only after monitored data is represented in the appropriate RDF can it be queried and reasoned about.
- *No standards yet:* while other technologies constituting the Semantic Web technology stack have been already standardised [25], research effort in the stream reasoning area is still disjoint and scattered. Standardisation by the World Wide Web Consortium[5] (W3C) is currently under way, but it may take another several years until agreed standards are widely adopted.
- *Immature reasoning support:* as opposed to the reasoning capabilities of SPARQL – a standardised static SQL-like query language for RDF data, widely used within the Semantic Web community – querying over data streams with reasoning support has not as yet been fully implemented. None of the existing continuous query languages for RDF streams fully supports the SPARQL 2.0 specification.
- *Low performance*: stream reasoning research is still in its infancy, and the performance of dynamic reasoning over a data stream is an issue. Expressivity of a query language is known to be inversely related to its performance [19] – the more expressive queries are, the longer it takes to execute them. This affects both the scalability of stream reasoning systems, and the actuality and accuracy of the results obtained. At the moment, much research effort is concentrated on advancing the reasoning support, while performance issues, such as query optimisation, caching, and parallel querying are yet to be explored.

[5] http://www.w3.org/

7 Conclusion

We have presented run-time monitoring as a foundation for implementing autonomic computer systems. However, in order for the monitored values to be of use in the further adaptation processes, there should be some form of evaluation over these values, so that potential failures of the monitored system can be detected and/or predicted. By listing challenges associated with run-time monitoring of complex distributed software systems, we have distilled principles to be followed when implementing a suitable analysis mechanism. In this context, we introduced stream reasoning – an extension of data stream processing, which allows for run-time reasoning over data streams – as a way of enhancing monitoring mechanisms with analysis capabilities. Stream reasoning techniques facilitate the evaluation of monitored data streams in a timely manner by supporting static background knowledge, expressive continuous query languages and temporary query operators. We have also described some of the potential shortcomings associated with stream reasoning in run-time monitoring and analysis solutions, arising from the relative immaturity of the underlying research areas. However, stream reasoning has attracted increasing attention in recent years, and we can reasonably expect these limitations to be addressed in the near future.

References

1. Tesauro, G., Chess, D.M., Walsh, W.E., Das, R., Segal, A., Whalley, I., Kephart, J.O., White, S.R.: A multi-agent systems approach to autonomic computing. Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1. pp. 464–471 (2004)
2. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. Ieee Trans. Softw. Eng. 30, 859–872 (2004)
3. Breskovic, I., Haas, C., Caton, S., Brandic, I.: Towards Self-Awareness in Cloud Markets: A Monitoring Methodology. Ninth IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC) (2011)
4. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer. 36, 41– 50 (2003)
5. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. Comput. Syst. 15, 1–40 (2001)
6. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing—degrees, models, and applications. Acm Comput Surv. 40, 1–28 (2008).
7. Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. Knowl. Eng. Rev. 10, 115–152 (1995)
8. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. Acm Trans. Auton. Adapt. Syst. Taas. 1, 223–259 (2006).
9. Bratanis, K.: Towards engineering multi-layer monitoring and adaptation of service-based applications. SEERC Technical Reports. Thessaloniki, Greece: South-East European Research Centre (SEERC) (2012)
10. Kazhamiakin, R., Benbernou, S., Baresi, L., Plebani, P., Uhlig, M., Barais, O.: Adaptation of Service-Based Systems. In: Papazoglou, M.P., Pohl, K., Parkin,

M., and Metzger, A. (eds.) Service Research Challenges and Solutions for the Future Internet. pp. 117–156. Springer Berlin Heidelberg (2010)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer. 36, 41 – 50 (2003)
12. Dautov, R., Paraskakis, I., Kourtesis, D., Stannett, M.: Addressing Self-Management in Cloud Platforms: a Semantic Sensor Web Approach. Proceedings of the International Workshop on Hot Topics in Cloud Services (HotTopiCS 2013). , Prague, Czech Republic (2013)
13. Dautov, R., Paraskakis, I., Kourtesis, D.: 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)
14. IBM: Bringing Big Data to the Enterprise, http://www-01.ibm.com/software/data/bigdata/
15. Calbimonte, J.-P., Jeung, H., Corcho, O., Aberer, K.: Enabling Query Technologies for the Semantic Sensor Web. Int. J. Semantic Web Inf. Syst. (2012)
16. Gucola, G., Margara, A.: Processing Flows of Information: From Data Stream to Complex Event Processing. Acm Comput. Surv. (2011)
17. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: 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)
18. Della Valle, E., Ceri, S., Barbieri, D., Braga, D., Campi, A.: A First Step Towards Stream Reasoning. In: Domingue, J., Fensel, D., and Traverso, P. (eds.) Future Internet. pp. 72–81. Springer Berlin / Heidelberg (2009)
19. Lanzanasto, N., Komazec, S., Toma, I.: Reasoning over real time data streams. http://www.envision-project.eu/wp-content/uploads/2012/11/D4.8-1.0.pdf
20. Sheth, A., Henson, C., Sahoo, S.S.: Semantic sensor web. Internet Comput. Ieee. 12, 78–83 (2008)
21. Barbieri, D., Braga, D., Ceri, S., Valle, E.D., Huang, Y., Tresp, V., Rettinger, A., Wermser, H.: Deductive and Inductive Stream Reasoning for Semantic Social Media Analytics. Intell. Syst. Ieee. 25, 32 –41 (2010)
22. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. Proceedings of the 18th international conference on World wide web. pp. 1061–1062. ACM, New York, NY, USA (2009)
23. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: 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. pp. 370–388. Springer-Verlag, Berlin, Heidelberg (2011)
24. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in ETALIS. Semantic Web. (2010)
25. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. CRC Press (2009)