GLIMPSE: A Generic and Flexible Monitoring Infrastructure*

Antonia Bertolino, Antonello Calabrò, Francesca Lonetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo".
CNR, Pisa
{firstname.secondname}@isti.cnr.it

Antonino Sabetta†
SAP Research
Sophia Antipolis
antonino.sabetta@sap.com

ABSTRACT

To respond to the growing needs of evolution and adaptation coming from the modern open connected world, applications must continuously monitor their own execution and the surrounding context. The events to be observed, belonging to guaranteed functional and non-functional properties, can themselves vary in scope and along time. Therefore the monitor must be easily configurable and able to serve differing event consumers. To address these requirements, we developed the GLIMPSE monitoring infrastructure conceived having flexibility and generality as main concerns. The paper introduces the architecture of GLIMPSE and shows how it can support runtime performance analysis through a simple example.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2 [**Software Engineering**]: General; D.2.8 [**Software Engineering**]: Metrics; D.2.5 [**Testing and Debugging**]: Monitors

General Terms

Measurement, Languages, Performance, Management

Keywords

Monitoring, Model-driven engineering, Complex-event processing, CONNECT

1. INTRODUCTION

Modern software applications are more and more conceived as dynamically adaptable and evolvable sets of components that must be able to modify their behavior at run time to tackle the continuous changes in the unpredictable open-world settings [8]. Changes can be due for example to the adoption of highly dynamic business integration infrastructures in the web service domain or to inherent mobility of ubiquitous systems.

Coping with context changes requires that software be able to self-organize its structure and self-adapt its behavior to provide the desired quality of service. Assumptions taken by developers at design-time, in fact, may become invalid after the system is deployed and running and not be able to fully take into account the changing external world interacting with the system. A runtime evaluation process is hence needed to timely detect unpredictable changes that affect the software system behavior.

This on-line analysis can involve non functional aspects that are to be taken into account for addressing performance measurement and run-time performance management. In particular, runtime analysis is the basic step for collecting the execution-time values of specified parameters in order to fill program executions models or perform on-line performance prediction.

The runtime assessment process, as opposed to activities that are carried out in the development/coding phase, can be identified as a *monitoring* activity. The monitoring process involves several different activities carried out during the execution of a system, dealing with data collection, interpretation and presentation of information concerning the "observable object". Each of these activities is the topic of a heterogeneous literature and addresses specific problems and techniques providing different solutions (see a brief overview in Section 5).

The most commonly used approach for observing the behavior of distributed systems is *event-based monitoring*. In a large system a lot of events can occur, which need to be filtered and combined to detect unexpected behaviors and to estimate performance or dependability measures of the system. In this paper we present an event-based monitoring infrastructure, called GLIMPSE[1], that implements the key components of a generic, flexible and robust architecture for composite event detection by means of publish/subscribe messaging pattern. Current event-based monitoring

*This work is partially supported by the EU-funded CONNECT project (FP7–231167)

†Antonino Sabetta's contribution to the research presented in this paper was given while he was a Researcher at Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo".

EWDC '11, May 11-12, 2011, Pisa, Italy

[1]Generic fLexIble Monitoring based on a Publish-Subscribe infrastructurE. A prototype is available at http://labsewiki.isti.cnr.it/labse/tools/glimpse/public/main

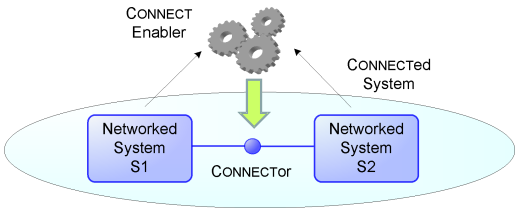

Figure 1: Overview of the Connect architecture

systems mainly focus on simple and composite event specification languages, but provide a limited flexibility in dealing with the large and dynamic context of monitor applications and the existing technologies. Our contribution is a highly flexible architecture decoupling high-level events specification from their underlying observation and analysis.

Our work on monitoring originated in the context of the European Project CONNECT[2], addressing the challenges set out in the "ICT forever yours" initiative[3]. The CONNECT world (see Figure 1) envisions dynamic environments populated by technological islands which are referred to as the Networked Systems (NSs), and by the components of the CONNECT architecture, called the CONNECT Enablers. The ambitious goal of the project is to have eternally functioning systems within a dynamically evolving context. This is achieved by synthesizing *on-the-fly* the CONNECTors through which NSs communicate. The resulting CONNECTors then compose and further adapt the interaction protocols run by the CONNECTed System.

The very vision of CONNECT, i.e., achieving automated and eternal interoperability, puts on-line approaches, and therefore monitoring, in a central position. Monitoring supports the construction of feedback loops whereby approaches to dependability analysis, CONNECTor synthesis, behavior learning can be applied to an on-line setting and can be enhanced to cope with change and dynamism. An evaluation of the requirements imposed on the monitoring subsystem by the other elements of CONNECT has led us to identify several key features that a generic, flexible monitoring infrastructure should possess, among which:

Distribution, dinamicity, heterogeneity: the monitoring system must be able to observe systems that are inherently distributed, highly dynamic and composed of heterogeneous entities that were not necessarily conceived for interoperation. Monitoring should address distribution by adopting an architecture that is itself distributed.

Modularity and Flexibility: a key goal of our work is to realize a monitoring infrastructure that can be employed for different purposes (including monitoring of functional and non-functional properties) and in different settings (even beyond its application within CONNECT). This is achieved by structuring our infrastructure around a modular architecture whose coupling with the observed system is minimal.

Efficiency: the performance penalty incurred because of monitoring should be minimized (and possibly predictable), while achieving the intended observation goals. Approaches for minimizing the impact of monitoring involve the mechanisms of monitoring (e.g., sampling, self-tuning [10]) as well as its architecture (e.g., by improving scalability through a hierarchical organization [18]).

An important issue addressed in our proposal is decoupling high-level specification of properties and events of interest to be monitored, from the underlying observation architecture, thus yielding the greatest generality and facility of use. For this purpose, the quantitative functional and non-functional properties to be monitored by GLIMPSE can be formally specified as instances of an eCore[4] meta-model. The main advantage of expressing properties as models which conform to a meta-model is that of providing a machine-processable language. From such models designers can thus use automated procedures (in form of ModelToModel or ModelToText transformations) to instrument the GLIMPSE architecture to monitor properties of interest. An overview of a specific Property Meta-Model (CPMM) we devised to express properties relevant for the CONNECT project, is presented in [12][5]. Such meta-model allows defining elements and types to specify prescriptive (required) and descriptive (owned) quantitative and qualitative properties that CONNECT actors may expose or must provide. The models conforming to such meta-model can be used to drive the instrumentation of the monitoring Enabler that generates suitable probes to monitor useful properties on the CONNECTors.

The remainder of the paper is structured as follows. Section 2 outlines the key concepts and elements of our monitoring framework. These concepts are mapped onto the components of a generic architecture. A reference implementation of such architecture is presented in Section 3. An application to a concrete scenario is presented in Section 4. Section 5 gives a brief overview of related work. Finally, Section 6 draws conclusions and identifies possible future developments.

2. THE MONITORING ARCHITECTURE

Monitoring has been defined as *the process of dynamic collection, interpretation, and presentation of information concerning objects or software processes under scrutiny* [15].

Elaborating on [16], five core functions can be identified for a generic monitoring system:

1. Data collection: this function concentrates on the collection of raw data from the execution of the observed components. This can be done by either instrumenting the subject component (when this is possible) or by intercepting interactions among components through a proxy-based probe. A special case is represented by the built-in logging facilities that many systems provide natively.

[2]http://connect-forever.eu
[3]http://cordis.europa.eu/fp7/ict/fet-proactive/ictfy_en.html
[4]The eCore metamodel is an instrument for designing Model-Driven Architecture (MDA).
[5]A first release is available at http://labsewiki.isti.cnr.it/labse/tools/cpmm/public/main

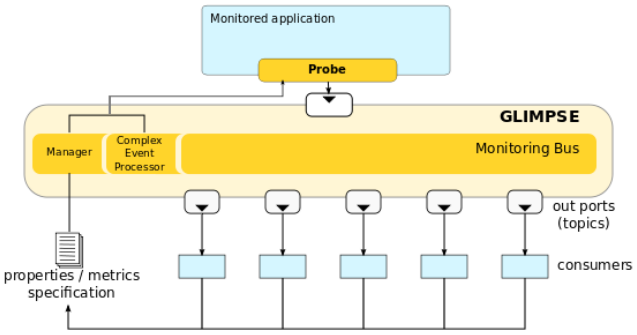

Figure 2: The architecture of GLIMPSE

2. Local interpretation: this function refers to the filtering that raw data go through before being interpreted at an aggregated level, to remove redundant or irrelevant information.

3. Data transmission: in distributed systems, this function takes care of gathering information from different originating nodes to a central (possibly not unique) node. Data transmission might exploit smart optimization algorithms (e.g., to delay data transmission or to give higher priority to certain information, when the network is subject to congestion).

4. Global interpretation: (also known as "correlation" or "aggregation"), this function makes sense of pieces of information that come from several nodes and puts them together in order to identify interesting conditions/events that can be observed only at an aggregated level. This function can be realized by means of a complex-event processing engine.

5. Reporting: this function deals with presenting the results of monitoring in a format that is meaningful to the "consumer" of the monitoring system. The consumer can be a human (e.g., a system administrator) or a program (e.g., a software component that implements the feed-back loop in a self-controlled software system).

GLIMPSE, the monitoring infrastructure presented in this paper, was designed in order to cover these five functions in a modular, flexible way and it is based on messages exchange. It can be improved and enriched, even at runtime, with components that respect a minimal set of requisites.

GLIMPSE was initially proposed in the context of the CONNECT project, where it is used to support behavioural learning, performance and reliability assessment, security, and trust management. However, the infrastructure is totally generic and can be easily applied to different contexts.

The architecture of GLIMPSE, shown in Figure 2, is composed of five main components:

Probes (Collector/Data Suppliers). Probes intercept primitive events when they occur into the software and send them to the GLIMPSE Monitoring Bus. Probes are usually realized by injecting code into an existing software or by using proxies. In addition, they may be configured to use a primitive event filter in order to reduce the amount of generated raw data.

Monitoring Bus. The Monitoring Bus is the communication backbone where all information (events, questions, answers) is sent by: Probes, Consumers, Complex Event Processor and by all the services querying information to GLIMPSE. In this prototype, we adopt a publish-subscribe paradigm devoting the communication handling to the Manager component; this communication pattern allows more consumers to fetch the same CEP evaluation results and offers data dissemination at low computational cost.

Complex Event Processor. The Complex Event Processor (CEP) is the rule engine which analyzes the primitive events, generated from the probes, to infer complex events matching the consumer requests. There are several rule engines that can be used for this task (like Drools Fusion [2], RuleML [6]). In the current implementation, we adopt the Drools Fusion rule language [2] that is open source and can be fully embedded in the realized Java architecture. Note that, the proposed flexible and modular architecture allows for easily replacing this specific rule language with another one.

Consumer. It may be a learning engine, a dependability analyzer or a simple customer that requests some information to be monitored. It sends a request to the Manager using the Monitoring Bus and waits for the evaluation results on a dedicated answer channel provided by the Manager.

Manager. The Manager component is the orchestrator of all the GLIMPSE architecture. It manages all the communications among the GLIMPSE components. Specifically, the Manager fetches requests coming from Consumers, analyzes them and instructs the Probes. Then, it instructs the CEP Evaluator, creates and notifies to the Consumer a dedicated channel on which it will provide results produced by the CEP Evaluator.

The usage of a messaging system in Enterprise Service Bus (ESB) allows GLIMPSE to use a variety of protocols such as HTTP/SOAP and REST. Moreover, with the usage of JBI [4] components, GLIMPSE may interact even with legacy systems, binary transports, document-oriented transports, and Remote Procedure Call systems. Adopting a messaging system reduces execution bottlenecks that might occur using Remote Procedure Call [5] or Database-centric architecture.

3. IMPLEMENTATION

A prototype of the GLIMPSE architecture, was developed and is being used in the CONNECT project. According with the five core functions identified in Section 2 we implemented the components of the GLIMPSE architecture as follows:

1. Data collection is entrusted to the Probes, that collect primitive data and send them to the Monitoring Bus.

2. Local interpretation is even realized by Probes; the pa-

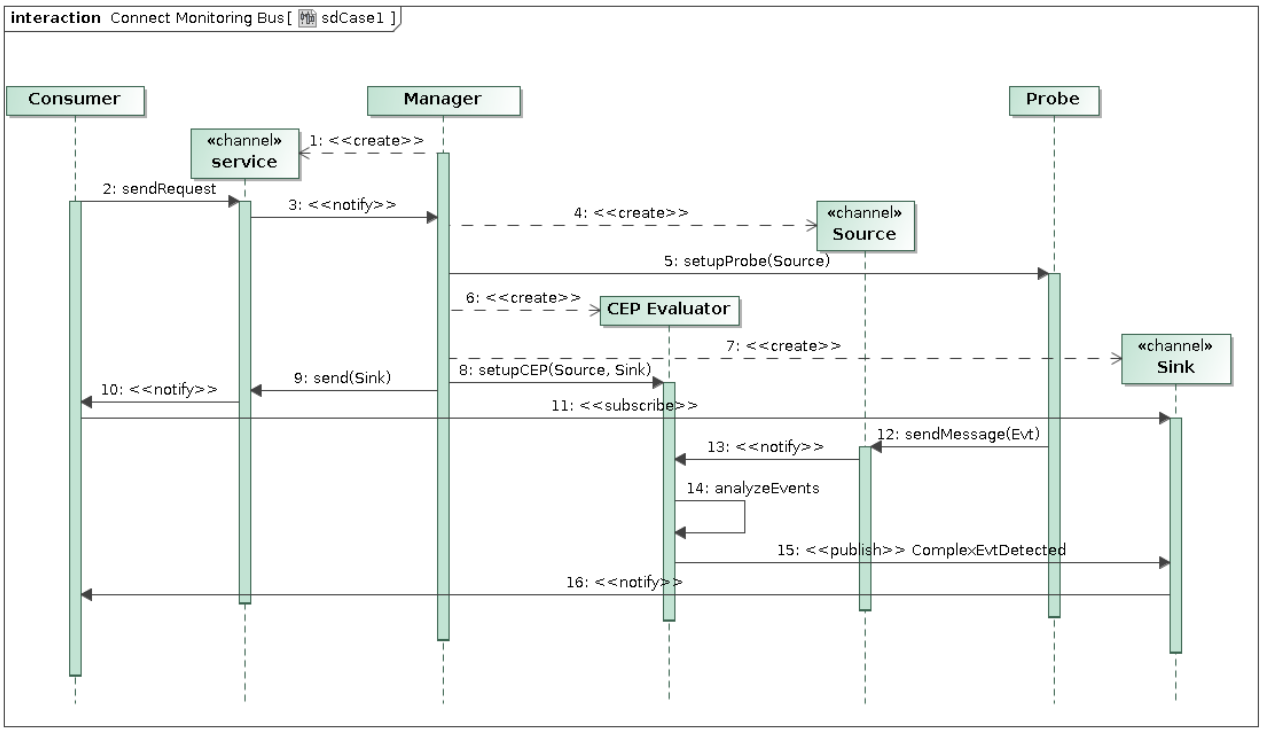

Figure 3: Sequence diagram of a typical interaction among GLIMPSE components

rameters for Local Interpretation are provided by the Manager component after processing a specific request from a Consumer.

3. Data transmission is realized by the system backbone (Monitoring Bus), implemented by means of ServiceMix4 [7], an open source Enterprise Service Bus, used to combine advantages of event-driven architecture and service-oriented architecture functionality. We chose ServiceMix4 because it offers a Message Oriented Bus and is able to run an open source message broker like ActiveMQ [1].
4. Global interpretation is provided by the CEP Evaluator component. As said above, this has been implemented using Drools Fusion, chosen for the useful integration with ServiceMix4, and the flexible and powerful rule language. It fetches messages generated from Probes on the Monitoring Bus, analyzes them, and answers on the dedicated channel provided by the Manager.
5. The Reporting function has been implemented in the Manager component that creates a dedicated channel on which it will provide monitoring results to the Consumer.

Figure 3 represents a sequence diagram of a typical interaction among the GLIMPSE components. On startup, the system initializes all the connections through ServiceMix4, activates the ActiveMQ broker and starts the Manager.

The Manager is listening for incoming requests on the *service* channel (see message 1 in Figure 3) from the Consumer.

The consumer sends a XML-like request on *service* channel (see message 2 in Figure 3). This request, including parameters like window-time to evaluate, pattern to match, events to match, is notified to the Manager (see message 3 in Figure 3).

After receiving the request, the Manager starts the monitoring environment setup creating the channel *source* (see message 4 in Figure 3). Then, it instructs the Probes about the channel identifier to use and the filter to apply for reducing the amount of raw data to send on the Monitoring Bus (see message 5 in Figure 3).

The Manager starts the Complex Event Processor (see message 6 in Figure 3) and creates the *sink* channel where to write the monitoring results (see message 7 in Figure 3).

The Manager provides to the Complex Event Processor the evaluation request received from the Consumer and the channel where the monitoring results will be notified (message 8 in Figure 3). After that, the Manager notifies to the Consumer the channel name where it will find the results concerning the submitted request (see message 9 and 10 in Figure 3).

The Consumer subscribes on the *sink* channel (see message 11 in Figure 3).

The Probes send primitive events on the *source* channel (see message 12 in Figure 3), these messages are captured by the CEP Evaluator (see message 13 in Figure 3), that analyzes

them (message 14 in Figure 3) and publishes the evaluation results on the *sink* channel where the Consumer is listening for (messages 15 and 16 in Figure 3).

The events taken into account in the GLIMPSE infrastructure are an abstraction of transitions between states of an LTS machine. Specifically, we identify the following event fields:

- a Timestamp;
- a Data field representing the event payload;
- a SourceID, identifying the probe sending the event;
- a SequenceID, used by the CEP to infer the sequence of events sent from the same source;
- a SourceState, indicating the state from where the transition started.

4. APPLICATION EXAMPLE

As an example to show the application of GLIMPSE, in this paper we refer to the Photo-Sharing scenario, which is one of the demonstration examples chosen in CONNECT.

It depicts a service provided in a public infrastructure (e.g, a stadium) for the dependable and secure exchange of pictures among the heterogeneous handheld devices of spectators. In particular, we consider a constraint that defines the acceptable response-time for operations used by a Photo-Sharing client. Hence, we want to measure the latency between submitting a query and obtaining a list of matching photos.

Other properties can be checked using the same framework, as long as they are expressed in an input language (the eCore meta-model mentioned earlier) for which a mapping exists onto the complex event specification language used in GLIMPSE. The notation we use in the example below is expressed in the specification language used by Drools Fusion [2]. We plan to release automated transformers from our eCore meta-model to other possible specification languages.

The assumption we make is that the quantitative properties to be monitored (latency in this example) are formally defined using our meta-model based notation. Such definition must be formulated in terms of the operations belonging to the interface of the monitored entity. In the example of the photo-sharing client, the monitoring could be used to detect whenever the time it takes to get a list of photos that match a query is above a certain threshold.

It must be noted that in general, observing the latency on the client side or on the server side can yield very different results. Besides, the deployment of probes usually is subject to constraints, and this affects the observability of certain properties significantly. This is a well-know problem, that is outside the scope of this paper (for a discussion of observability of non-functional properties in distributed systems, see, e.g., [20]).

What matters for the purposes of this work, as hinted above, is that the property is eventually translated into a language that is readily understood by the event-correlation engine. The example in Listing 1 shows the latency constraint.

```
1  rule "PhotoSearchLatency"
2  when
3    // this is the complementary of Call
4    $call: ReceiveCall(
5      operation == "SearchPhotos",
6      $session : session_id),
7      from entry-point "PhotoSharingMediator";
8    $reply: Reply(
9      session_id == $session,
10     operation == "SearchPhotos",
11     this after[1200ms] $call )
12     from entry-point "PhotoSharingMediator";
13 then
14   // inject alarm on the Monitoring Bus
15 end
```

Listing 1: Sample rule for checking a latency policy

The rule *PhotoSearchLatency* matches a *ReceiveCall* event followed by a *Reply event* (lines 4 and 8 respectively), ensuring that both refer to the same session (lines 6 and 9) and that the latter happens not earlier than 1200 ms (which is assumed to be the latency threshold) after the former (line 11). If all these conditions are verified, i.e., the response is received after the latency threshold, an alarm is injected into the Monitoring Bus (line 14) so that the subscribers for that kind of complex event can be notified.

Possible consumers of this type of notifications are SLA checkers, loggers (of aggregated events), parameterization/-tuning of runtime analysis models. In CONNECT, this notification is used by the dependability analysis Enablers.

5. RELATED WORK

Monitoring is recognized as a key functionality in many different types of systems, spanning web services applications, grid environments [22], networks management mechanisms. A first attempt to overview the most important problems and issues about monitoring in distributed systems is [16]. Due to different aspects and activities of monitoring, research on it appears fragmented. The many works on monitoring are hard to compare as most of them focus only on some of the aspects of monitoring.

The works more related to our proposal are those approaching the definition of a monitoring architecture [19, 14]. In particular, [19] presents an extended event-based middleware with complex event processing capabilities on distributed systems. Similar to GLIMPSE this work adopts a publish/-subscribe infrastructure but it is mainly focused on the definition of a complex-event specification language. The aim of GLIMPSE is to give a more general and flexible monitoring infrastructure for achieving a better interpretability with many heterogeneous systems.

Another monitoring architecture for distributed systems management is presented in [14]. Differently from GLIMPSE, this architecture employs a hierarchical and layered event filtering approach. The goal of the authors is to improve monitoring scalability and performance for large-scale distributed systems, minimizing the monitoring intrusiveness.

A very attractive research field is the event-based modeling

and computing. Many works focus on the definition of expressive complex event specification languages [17, 11, 13]. Among them, GEM [17] is a generalized and interpreted event-based monitoring language. It is rule-based (similar to other event-condition-action approaches) and also provides a tree-bases detection algorithm taking into account communication delay. Also the Snoop language [11] follows an event-condition-action approach supporting temporal and composite events specification but it is especially developed for active databases. A more recent formally defined specification language is TESLA [13]. It has a simple syntax and a semantics based on a first order temporal logic. The authors of [13] also provide an efficient event detection algorithm by translating TESLA rules into automata. Some existing open-source event processing engines are Drools Fusion [2] and Esper [3]. They can be fully embedded in existing Java architectures and provide efficient rule processing mechanisms.

Monitoring can be used to observe functional and non functional properties. An approach to perform automated and distributed monitoring for guaranteeing Service Level Agreements (SLA) in the web services scenario is presented in [21]. It gives a flexible but precise formalization of SLAs and provides an engine to automate the monitoring of these SLAs. Other monitoring frameworks exist that address the monitoring of performance, in the context of system management. Among them, there are Nagios [9] and Ganglia [18]: the first offers a monitoring infrastructure to support the management of IT systems spanning network, OS, applications; the latter is especially dedicated for high-performance computing and is used in large clusters, focusing on scalability through a layered architecture.

6. CONCLUSIONS

We have presented the inspiring principles, the architecture, an implementation and a simple example of usage of GLIMPSE, a generic and flexible monitoring infrastructure. It has been conceived to address the requirements for adaptation and context-awareness of modern distributed systems operating in an open changing world.

GLIMPSE adopts a model-driven approach, in that the simple or complex events to be monitored conform to a specified GLIMPSE meta-model. The properties are defined by the consumers in their own language and are then translated into our complex event language using model-driven standard transformations. GLIMPSE is currently being integrated in the CONNECT architecture, aiming at ensuring seamless eternal connection among heterogenous networked systems. We focused here on the usage of GLIMPSE for performance on-line analysis, but other usages are being considered in CONNECT, such as reliability assessment or trust evaluation. The idea is to use one comprehensive and configurable monitoring infrastructure for different purposes and different property specifications.

7. REFERENCES

[1] ActiveMQ: A complete message broker. http://activemq.apache.org.

[2] Drools Fusion: Complex Event Processor. http://www.jboss.org/drools/drools-fusion.html.

[3] Esper: Event Stream and Complex Event Processing for Java. http://www.espertech.com/products/esper.php.

[4] JBI: Java Business Integration. http://jcp.org/aboutJava/communityprocess-/final/jsr208/index.html.

[5] RPC: Model for programming in a distributed computing environment. http://msdn.microsoft.com/en-us/library/ms691207(VS.85).aspx.

[6] RuleML: The Rule Markup Initiative. http://ruleml.org.

[7] ServiceMix: an open source ESB. http://servicemix.apache.org/home.html.

[8] Luciano Baresi, Carlo Ghezzi, and Elisabetta Di Nitto. Toward open-world software: Issues and challenges. *Computer*, 39(10), 2006.

[9] Wolfgang Barth. *Nagios. System and Network Monitoring*. No Starch Press, U.S. Ed edition, 2006.

[10] Antonia Bertolino, Guglielmo De Angelis, Antonino Sabetta, and Sebastian G. Elbaum. Scaling up SLA monitoring in pervasive environments. In *Proc. of ESSPE*, pages 65–68, 2007.

[11] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[12] CONNECT Consortium. Deliverable 5.2 – Design of approaches for dependability and initial prototypes, 2011.

[13] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proc. of DEBS*, pages 50–61, 2010.

[14] Ehab Al-Shaer Hussein, Hussein Abdel-wahab, and Kurt Maly. HiFi: A New Monitoring Architecture for Distributed Systems Management. In *Proc. of ICDCS*, pages 171–178, 1999.

[15] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987.

[16] Masoud Mansouri-Samani and Morris Sloman. Monitoring distributed systems. *Network and distributed systems management*, pages 303–347, 1994.

[17] Masoud Mansouri-Samani and Morris Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[18] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[19] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44 – 55, January 2004.

[20] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service SLAs. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 170–180, 2008.

[21] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad van Moorsel, and Fabio Casati. Automated SLA Monitoring for Web Services. In *Proc. of DSOM*,

pages 28–41. 2002.

[22] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Gener. Comput. Syst.*, 21:163–188, January 2005.