The Tiny Instrument Element

Francesco Lelli, Cesare Pautasso

Faculty of Informatics
University of Lugano
via Buffi 13
6900 Lugano, Switzerland
firstname.lastname@lu.unisi.ch

Abstract. In the past few years, the idea of extending the Grid to cover also the remote access, control, management of instrument devices has been explored in a few initiatives. Existing tools lack in generality and require advanced specialized computer science knowledge, thus making them difficult to be broadly adopted in the scientific community. In this paper we present a new open source initiative that is designed to overcome these problems. The Tiny Instrument Element project defines a high level architecture for plugging instruments into the Grid and provides the corresponding skeleton implementation. This lightweight approach, as opposed to existing middleware-based solutions, reduces the effort required to Gridify existing instruments. The paper evaluates the proposed abstraction with a case study from a pervasive computing scenario.

1 Introduction

The term Grid refers to a set of technologies for sharing and accessing storage space and computational power. Additionally, the desire to access, control, and acquire data from pervasive, widely-networked and distributed instruments reflects the need to include such scientific equipment as sensors and probes directly into the Grid. In previous works ([1], [2]) we defined the term Instrument Element (IE) as a set of services that provide the remote control and monitoring of physical instruments. In Grid terminology the words "instrument", "sensor", "actuator", and "device" are synonyms used to identify any piece of equipment that needs to be initialized, configured, calibrated, operated (with commands such as start, stop, standby, resume, reset), and monitored. Unlike the classical computing infrastructure composed of the Computing Element (CE) and the Storage Element (SE), the IE must be accessed using interactive computational job execution and usually requires a tightly coupled interaction with the users. In the past few years, the concept of Instrument Element and its definition has been adopted by few international cooperations (GridCC [3], RINGrid [4], DORII [5]). Not only complex instruments such as High Energy Physics experiments require a direct access to computational infrastructure. In this paper we consider additional use case scenarios where instruments:

– are large in number.
– are widely distributed.
– have a highly dynamic behavior: for instance they often go on and off or can appear and disappear in a working net of sensors or probes.
– operate in embedded systems with low resources: for example FPGA based instrumentation.

Some example application domains that would require such widely sparse instrumentation are: (i) power grids, (ii) territory monitoring: to prevent geo-hazardous situations and detect forest fires, (iii) sea monitoring: for tsunami surveillance for example, (iv) distributed laboratories, (v) transportation remote control and monitoring.

Whereas in these applications a single device may not produce an amount of data comparable with the one produced by high-energy physics experiments, including such large collection of devices in the Grid can be very useful to run complex data processing and leverage the available distributed storage facilities. This way, the data produced by the sensor equipment may be directly and continuosly stored in a distributed storage system, making it possible to submit analysis jobs on the Grid as new data arrives.

In order to address the requirements of these usage scenarios, a new version of the Instrument Element is needed, since the complexity of the current middleware makes it very expensive and difficult to adopt by third party scientific institutions, end-users or programmers that have expressed an interested to use it for these applications. In this paper we present the Tiny Instrument Element project [6], which aims a developing a light-weight implementation of similar concepts. However, it uses a template-based approach based on widely adopted technology as opposed to a plug-in based middleware. Moreover, the project iself is developed according to the Web 2.0 open colleboration paradigm, in order to provide transparent access to the development and spur the growth of a large user and developer community. The Tiny Instrument Element Project [6] has started as a simplification of the code for the Run Control of the Compact Muon Solenoid (CMS) experiment at CERN [7] and now the first stable release is available for download. The rest of this paper is structured as follows: Section 2 presents a classification of related work and discusses its main limitations. Section 3 outlines the principles guiding the design of the Tiny Instrument Element. Section 4 introduces the architecture of the Tiny Instrument Element from a technical point of view. The case study presented in Section 5 is used to validate the present implementation. Section 6 concludes the paper with a discussion of the project roadmap.

2 Related Work

The goal is to produce a middleware that can act as glue between devices allowing their abstraction for making them accessible on the Grid. If we try to classify these efforts we can distinguish two levels of abstraction:

– **Low Level**: these provide general interfaces and mechanisms therefore they are very flexible but incomplete. A significant effort by developers is required in order to build an artifact design for a particular kind of instrument.
– **High Level**: these give management interfaces more specific to the domain of instrument control. However, they are less applicable to a large variety of instruments and they may require complex configuration. Still, less development effort is required in order to support a specific instrument.

In the first category (low level) we find the following: A WS-* based standard, WS-Notification [8], describes asynchronous publish/subscribe notification protocols that can be used for listening to remote service data element updates representing the state of a Grid-enabled instrument. The WSRF framework such as OGSA [9], [10], Apache-WSRF [11] and WSRF.NET [12] implement this standard. The Java Management Extensions (JMX) [13] technology is an open system for management and monitoring. Via its Instrumentation, Agent, and Distributed Services layers, this standard can be used for implementing management tools, and providing monitoring solutions. Jini [14] provides mechanisms to enable adding, removing, and locating devices and services on the network. The Java Message Service (JMS) define a common set of publish/subscribe APIs [15] that allow different peers of a distributed system to communicate using messages or data streams.

In category "high level" we find CIMA [16] and the Instrument Element (IE) [1], [2]. CIMA proposes a common instrument middleware based on Web Services using SOAP over HTTP as a communication layer and specific WSDL interfaces. The first reliable implementation of the IE has been provided by the GridCC project [3] and then few additional implementations have followed [17], [5]. Built on high level middleware that can fit in all possible use case, the integration of IE require not trivial efforts because each specific use case is not perfectly covered by the middleware itself.

In this paper we present another implementation of the Instrument Element concept. In the design of this version we tried to take the benefits of both high and low abstraction levels in order to create a transparent, open source project, independent from any particular initiative. As we will describe in the next section, instead of building yet another middleware framework, we propose to use a semi-finite artifact (i.e., a skeleton software) that can be extended, tailored, and customized in order meet the requirements of a specific use case. As we will show in our case study, this approach grants more flexibility and reusability than existing high level solutions. Also, it does not suffer from the generality of low level solutions, as it is designed around the instrument abstraction.

3 Design Principles

In this section we present how to integrate pervasive devices in the classical grid computing infrastructure. As outlined in the related work section many solutions have been proposed at a high and low level of abstraction, which require specialized expertise. In proposing this solution to a new community we can encounter a natural resistance from the people that have to learn how to use it. Moreover, high level solutions usually require the development of a plug-in and a deep knowledge of a complex and specialized middleware. Therefore even if some of the proposed solutions appear to be of general applicability they are hard to apply in practice due to the amount of time that has to be spent in learning how to use and extend them. Low level solutions instead require non-trivial computer programming expertise, because they are not designed targeting the Gridification of scientific instruments. Therefore solutions built starting from this abstraction level may be quite advanced but hard to reuse: customization to similar or other instruments may be performed only by experts.

To introduce our model for instruments Gridification, we take inspiration from modern Web development practices. For example, many Web 2.0 services, such as blogs, wikis, and social networking sites, target a variety of user categories:

– **End User**: definitely not a computer expert user, it has no understanding about the technology that he is using but he is able to use the functionality of a tool. For example, a blog writer may post his ideas on Web pages; a scientist may retrieve data from a pre-configured instrument.
– **Advanced User**: with some basic computer science knowledge, she is capable of following the instructions for the installation of the tool and for performing some simple customization. Advanced blog writers can create and customize the layout and appearance of their blog. Advanced scientists can setup and calibrate their instrument as they share it on the Grid.
– **API Developer**: thanks to their programming skills, these developers know how to develop applications using the API offered by the tool. Developer may write programs to retrieve and aggregate posts using the API of their favourite blogs. Scientist developers may extend the Tiny Instrument Element to support new kinds of instruments, as discussed in the case study.
– **Tool Programmer**: the builder of the tool itself, he has the ultimate knowledge on how to use, customize, and extend it for any application. Once the user community starts to grow, programmers should provide support to the users of the other categories and use their feedback to improve the tool.

From this classification we notice how many different groups of people can contribute to the success of a tool. We can also notice that non-specialized know-how is enough for performing simple adjustments. Therefore users may become familiar with the technology incrementally. This will foster the establishment of a community that will support the tool, contribute to its development, testing, extension, and application at the best of their knowledge. Interested users (and scientists in our case) will progress from simple user to more advanced ones as their familiarity of the tool increases, thus becoming able to apply a tool to more advanced and specialized use cases.

The Tiny Instrument Element project is centered around the previously described “gentle learning curve” principle. Additionally, the following guidelines are at the foundation of its design.

- **Skeleton Architecture**: the Tiny Instrument Element is a semifinal artifact that – used as a skeleton – can simplify and homogenize the construction of the final solution.
- **Technology Reuse**: we prefer to reuse existing and adopted technology as opposed to develop new middleware frameworks. This way, potential new developers may quickly contribute to the project by leveraging existing knowledge and skills.
- **Standard Packaging**: the project packaging follows a standard structure (e.g., the one of Maven) in order to be easy to understand by its users.
- **Template Customization**: several examples are given for common use cases to guide developers as they customize and extend the skeleton architecture to their needs.
- **Transparent Development**: the Tiny Instrument Element is an open source project driven by its user and developer community. Recent studies have shown that the adoption of these methodologies improves the quality of the software and reduces its development time [18].

Instead of presenting yet another complex middleware framework and showing how to develop a new plug-in, we are focusing our attention on how the customization of existing code can be integrated in the given target application. In [1] we showed with empirical evidence that different devices have different needs even if they share similar functionalities. Therefore the development of a single middleware for all the instruments results in a complex solution, which is difficult to maintain and customize. This also hinders the creation of a large user community.

In order to overcome such limitations we provide a well-defined modular architecture and we reduce the dependencies with external libraries. The goal is to have new programmers become familiar with the project in a short time. In addition, the adoption of Web based software deployment simplifies the installation procedure of the tool making it easy and convenient for scientists to access its functionality through a Web-based interface.

4 Architecture and API Design

A Web service interface (WS) acts as front-end between external components and the tiny Instrument Element (IE) itself. Using the Proxy/Wrapper pattern, inside the IE one or more Control Manager (or Instrument Manager) map the exposed WS to the actual devices. In the simplest scenario only one instrument is controlled and the control manager acts just as a proxy for the information contained in the device. Depending on the controlled equipment the control manager could also perform fault tolerant and or autonomic control functionality. A detailed technical documentation for the API can be found in the project Web site [19]. Figure 1 summarizes the overall architecture of the system.

We describe the object-oriented design of the Tiny Instrument Architecture following two common use cases. The first (detailed in Section 4.1) is about tailoring the framework to new kinds of instruments. The integration of an instrument

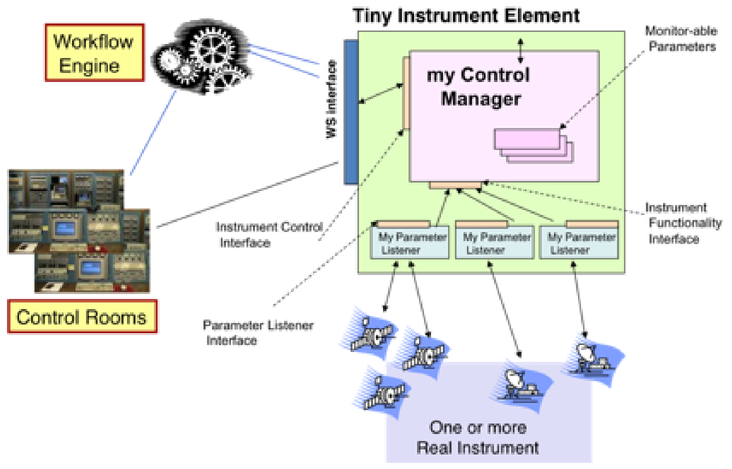

Fig. 1. Architecture of the Tiny Instrument Element

consists in the implementation of one or more interfaces that take care of the communication with the device. One or more instruments can be controlled using an object that implements the Parameter Listener interface. These objects present new events to the Control Manager using the Instrument Functionality interface. The proposed interfaces can support both stateless and stateful instruments, which can communicate both in a synchronous or asynchronous way.

The second use case concerns the remote access to the real instruments using the aforementioned Web service APIs (Section 4.2). External components like a control room or a workflow engine can access the controlled instrument(s) through the Web Service that retrieves the requested information via the instrument control interface.

4.1 How to plug a new instrument

From a conceptual point of view an Instrument Manager (IM) (i.e., an implementation of your control manager) is completely described by its parameters, attributes, commands, and a finite state machine:

- **Parameters** hold configuration information of the instrument.
- **Attributes** hold instrument variables (inputs and outputs).
- **Commands** hold actions that the device should perform.
- **Finite State Machine** specifies a state transition automata, used to constrain in which states can commands be executed.

This model is general enough to be applicable to different classes of instruments, since some of the elements are optional. Therefore we can have devices that, for instance, do not use the Finite State Machine because they only support one command, or that do not have input attributes.

As an example, consider an instrument manager for a simple Voltmeter (i.e., an instrument for measuring the voltage between two points in an electric circuit). Parameter are: Maximum Voltage, Minimum voltage. These characterize the instrument and do not change unless the given voltmeter provides the possibility to tune its measurement scale. Attributes: measured Voltage or set of measures. Commands: Perform a measure or Perform a set of measures. Finite State Machine: IM-Linked (the IM is connected the instrument), IM-Unlinked, Error.

From an practical point of view, developing a controller for a new kind of instrument involves implementing from 1 to 3 interfaces, depending on the control features supported by the device. In the simplest case, the instrument can be controlled by implementing the `InstrumentControl` interface, shown in Figure 2. This interface represents the abstraction of a generic instrument, and includes methods used for its remote control (such as `create()`, `destroy()`, `get/setParameters()`, `get/setAttributes()`, `executeCommand()`, `getStateMachine()`) and should be implemented by a controller that acts as a protocol adapter between the APIs and the actual instrument. In the UML diagram of Figure 2 the class `Command` represents a command while the class `Parameter` holds information about both attributes and parameters. Note also that commands can contain parameters and that parameters may contain arbitrarly typed objects. This design enables the instrument manager to execute complex command scripts to manage the instruments.

In more complex scenarios, additional interfaces `InstrumentFunctionality` and `ParameterListener` come into play, when the following assumptions do not hold:

- There is one controller for each instrument
- The instrument does not send asynchronous messages
- The instrument can process commands in a short amount of time (less than one second)

Like the majority of control systems, the design supports instrument aggregation and grouping, as an Instrument Manager (IM) can bee also seen as an instrument. This is a very important feature for controlling a collection of instruments of similar kind through the same control instance.

More complex instruments require handling asynchronous inputs coming from other devices or subcomponents. The UML diagram in Figure 3 shows the additional interface that has to be implemented by a controller. Generic input like State changes or Errors from the equipment may be presented to the IM implementing the `InstrumentFunctionality` interface and the `ParameterListener` interface.

These two objects collaborate following the Observer pattern and add a method `onMessage()` to the `InstrumentControl` interface to support asynchronous communication. Both the method `init()` and `destroy()` come from the interface intrumentControl itself. Therefore only the method `onMessage()`

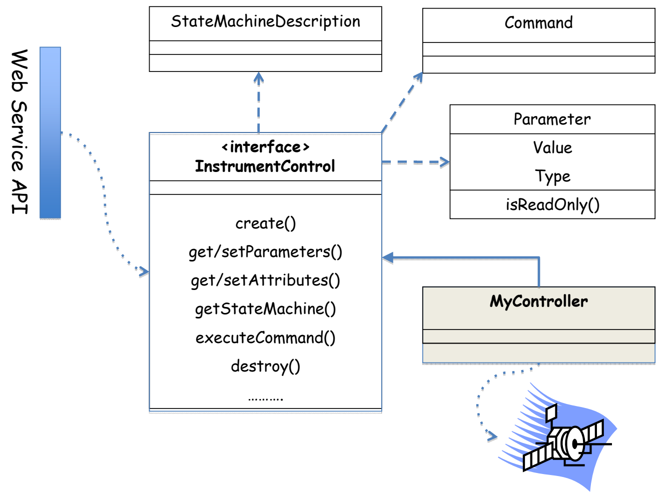

Fig. 2. How to Plug a New Instrument: Basic Functionality

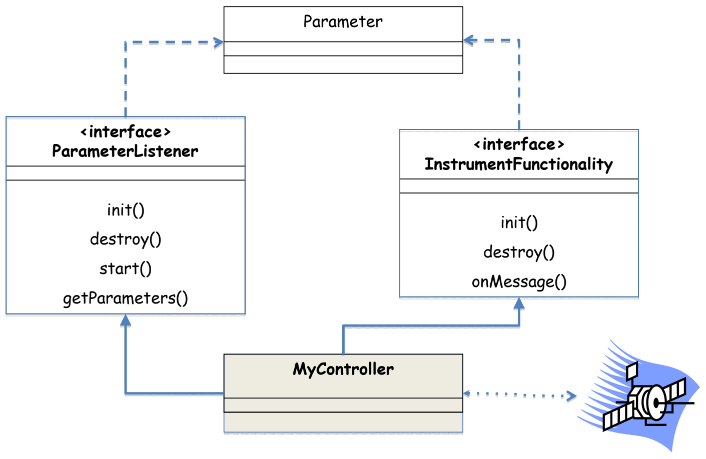

Fig. 3. How to Plug a New Instrument: Advanced Functionality

has to be implemented in order to update the logic of the controller when an asynchronous message coming from the equipment has to be handled. On the other side of the observer pattern a `ParameterListener` interface can receive in synchronous or asynchronous way messages coming from the controlled devices and can call back the `onMessage()` method of the `instrumentFunctionality` interface.

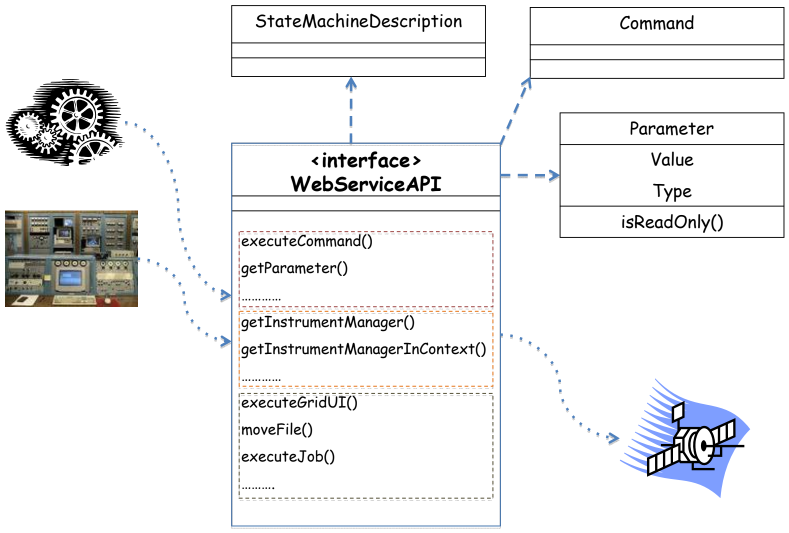

Fig. 4. Grid API design of an instrument element

4.2 How to share instruments on the Grid

Instruments are included in the Grid through a service-oriented API, providing the following operations:

– Monitor and control of one or more instruments
– Access and retrieve an instrument configuration parameters and topology.
– Collect the current measurements performed by the instrument
– Provide a set of APIs for the integration between instrument and Computational Infrastructure.

The API (shown in Figure 4) exposes a subset of the design elements introduced in the previous section. Following the principe presented in section 3, most of the Objects presented in section 4.1 have been reused in the Web Service APIs In in order to minimize the knowledge that potential new programmers should have in order to master the overall software.

The APIs take into account that more that an Instrument Element may contain more than one Instrument Manager and that IMs may be organized in certain topologies. Therefore graph navigation APIs have been provided like `getInstrumentManager()` and `getInstrumentManagerInContext()`. The Access to the computational infrastructure is provided by a wrapper around the Grid User Interface. Therefore the produced data can then be moved to a storage element using familiar commands sent to the instrument service via the `executeGridUI()` method or simplified methods like `moveFile()` or `submitjob()`. In the current stable release of Tiny Instrument Element, the Grid interface is only implemented using SOAP calls. We are currently extending it also to support a RESTful design to make it easier to invoke it from JavaScript applications.

5 Case Study

Instead of using a quantitative approach for the validation of the project, we present a case study that helps to evaluate the efforts needed for applying the tiny Instrument Element in a pervasive computing scenario. Our aim is to understand the reusability of the software where classical Grid middleware can encounter difficulties.

The case study is tracking a large collection of instruments, which are widely distributed in the environment. These instruments perform various kinds of measurements and data aquisition (GPS location, CPU usage, available memory, QoS properties, availability status, data traffic) and can be deployed by scientists at their location with minimal effort. The collected information can be then conveyed to dedicated instruments that act as information providers enabling the display, the geo-tagging and consulting of the aggregated data coming from different remote sources. An important requirement of the case study is that the instrument control software should run on Linux/Windows operating systems, but also in FPGA based embedded systems. The distribution of the software is handled in 3 different ways:

- Web Start Application (click a web link an the instrument element is installed to monitor your local machine).
- WAR Based Deployment (copy a file in the webapps folder of a Web application server and the instrument element service is ready to be invoked).
- Cross compilation and deployment script (run the instrument element on low resource systems such as FPGA).

Following the classification given in section 3 we can classify the people involved in the development as follow:

- **End User**: any user that want to perform the demo that is availlable at the web site.
- **Advanced User**: a small number of system administrators that deployed the software one expert that knows how to deploy application on FPGAs.
- **API Developers**: two expert programmers that have no previous knowledge of the given technology

– **Tool Programmer**: One of the people directly involved in the tiny IE project that was giving mail assistance in case of problems.

The result of this case study helps to support the claim that the Tiny Instrument Element can be extended in order to cover this pervasive computing application scenario with minimal effort. Also, it is important to point out that the case study demo [20] was built by two members of the project user community and not by its original authors. Using the Tiny Instrument Element as a starting point, the two programmers were able to apply it to 6 different kinds of instruments [21], extend it with a distributed index for data aggregation and complete it with a graphical user interface in less than two months. The feedback that was provided by these users was very positive, thus showing the potential of the Tiny Instrument Element to cater for the needs of its user community. More in detail, the majority of the time was spent implementing a controller compatible with the `InstrumentControl` interface.

6 Conclusion

In this paper we present the Tiny Instrument Element project showing our proposed novel approach to the integration between instruments and the Grid. Instead of building a new middleware we propose to use a semifinite artifact (i.e., a skeleton software) that can be tailored to meet the requirements of a specific instrument characteristics. This approach not only provides an uniform access to the Gridified instruments but also leaves the flexibility to customize and tune the Tiny Instrument Element for optimal monitoring and control of the instruments. From the case study we have seen that none of the code included in the Tiny Instrument Element release was redundant and that the time required in order to gain a good understanding of the API and the corresponding skeleton was quite small. This supports the idea of template-based software development.

The Project has started full open source activities in September 2008. If we exclude our personal activity and the one performed by the case study participants, until the end of 2008, the project website attracted 304 unique visitors (7 Returning many times). The source code was downloaded 40 times and the authors were contacted with positive feedback by 2 users of the community. Whereas the project has been running for a relatively short time, these numbers are promising and show the benefit of a transparent development process to achieve wider dissemination of our research ideas.

In the future road map of the project, we plan to begin investigating a REST API for the remote access to the instruments, and the ability to publish instruments on different Grid/Cloud Middleware frameworks such as ARC [22] and Amazon EC2 [23] in order to prove the reusability of our proposed solution across different middleware. We also plan to embed the instrument control API into a scientific workflow system [24,25]

Acknowledgements

We would like to thank Pietro Molini, Eric Frizziero and Silvano Squizzato for their participation in the Case Study.

References

1. Lelli, F., Frizziero, E., Gulmini, M., Maron, G., Orlando, S., Petrucci, A., Squizzato, S.: The many faces of the integration of instruments and the grid. Int. J. Web Grid Serv. **3**(3) (2007) 239–266
2. Frizziero, E., Gulmini, M., Lelli, F., Maron, G., Oh, A., Orlando, S., Petrucci, A., Squizzato, S., Traldi, S.: Instrument Element: A New Grid component that Enables the Control of Remote Instrumentation. In: CCGRID'06. (2006)
3. GridCC Project Web Site: `http://www.gridcc.org/`
4. RINGrid Project web site: `http://www.ringrid.eu/`
5. DORII Project web site: `http://www.dorii.eu/`
6. Tiny Instrument Element Project: `http://instrumentelem.sourceforge.net/`
7. The CMS Collaboration: The CMS experiment at the CERN LHC. Int. Journal of Instrumentation **3**(3) (2008)
8. WS Notification specification: `http://www-128.ibm.com/developerworks/webservices/library/specification/ws-notification/`
9. Globus Open Grid Services Architecture: `http://www.globus.org/ogsa/`
10. Foster, I., Kesselman, C.: The Globus Toolkit. In: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers (1999) 259–278
11. Apache WSRF Project: `http://ws.apache.org/wsrf/`
12. WSRF.NET Project: `http://www.cs.virginia.edu/~gsw2c/wsrf.net.html`
13. McManus, E.: JSR 160: JavaTM Management Extensions (JMX) Remote API 1.0 (October 2003)
14. Jini project: `http://www.jini.org/`
15. JMS standard API: `http://java.sun.com/products/jms/`
16. McMullen, D., Devadithya, T., Chiu, K.: Integrating Instruments and Sensors into the Grid with CIMA Web Services. Proc. of the 3rd APAC Conference on Advanced Computing, Grid Applications and e-Research (APAC05) (Sept 2005)
17. Vuerli, C., Taffoni, G., Coretti, I., Pasian, F., Santin, P., Pucillov, M.: Instruments in grid: The new instrument element. In: Grid Enabled Remote Instrumentation. Signals and Communication Technology. Springer US (October 2008)
18. Jantunen, S., Smolander, K., Malinen, S., Virtanen, T., Kujala, S.: Utilizing Firm-Hosted Online Communities: Research Challenges and Needs. In: Proc. of 1st Int'l Workshop on Social Software Engineering and Applications. (Sept. 2008)
19. InstrumentElement User Guide: `http://instrumentelem.sourceforge.net/wiki/index.php/User_Guide`
20. Case Study Demo: `http://sadgw.lnl.infn.it:2002/MapsMonitor/`
21. Released Instruments: `http://sadgw.lnl.infn.it:2002/MapsMonitor/marker_guide.htm`
22. Advanced Resource Connector ARC: `http://www.nordugrid.org/middleware/`
23. Amazon Elastic Compute Cloud EC2: `http://aws.amazon.com/ec2/`
24. Pautasso, C., Bausch, W., Alonso, G.: Autonomic computing for virtual laboratories. In: Dependable Systems: Software, Computing, Networks. Number 4028 in LNCS. Springer (2006)
25. Stirling, D., Welch, I., Komisarczuk, P.: Designing workflows for grid enabled internet instruments. In: CCGRID '08. (May 2008) 218–225