Controlling and Monitoring Devices with REST

Francesco Lelli and Cesare Pautasso

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

Abstract. In this paper we apply the REST principles to the problem of defining an extensible and lightweight interface for controlling and monitoring the operations of instruments and devices shared on the Grid. We integrated a REST Service in the Tiny Instrument Element (IE) that has been used for an empirical evaluation of the approach demonstrating that this implementation can coexist with a Web Service back-end and be used in parallel where is needed. Finally we present a preliminary performance comparison with the WS-* compliant implementation.
Keywords: Instrument Element, REST, Control and Monitor

1 Introduction

The remote control and monitoring of devices and experiments require many interactions between the instruments and the computational Grid. Scientific equipment must be accessed by running jobs that need to interact with the instrument while performing some computation. These jobs are also interactive, as the users need to be able to use them to monitor and steer the instrument operations. In addition, in most demanding use cases, such as instruments for high energy physics experiments, achieving the required performance and quality of service guarantees represents an important challenge [1].

In the past few years modern web based companies offer multiple ways for accessing the services that they provide. The most popular approaches are based on the Web Service technology stack or on Representational State Transfer (REST) [2]. From a technical point of view both the approaches have strengths and weaknesses and the adoption of a particular solution is really use case dependent [3].

The REST architectural style has been introduced to give a systematic explanation to the scalability of the HTTP protocol and the rapid growth of the World Wide Web. Its design principles have been recently adopted to guide the design of the next generation of Web services called RESTful Web services [4]. The benefits of REST lie in the simplicity of the technology stack required to build a Web services and in the recognition that most Web services are indeed stateful entities.

In this paper we investigate how to apply the REST design principles to give a lightweight solution to the problem of monitoring and controlling scientific instruments. We build a set of APIs for controlling and monitoring devices

consisting of an oculate selection of the resource URIs and a precise description of their representation, as it is exchanged in the request and response of each method applied to a URI published by the device. We integrated these APIs in the Tiny Instrument Element (IE) [5], [6] that has been used for an empirical evaluation of the approach. The Original Tiny IE Web Service interface expose methods that are similar to the ones developed in international cooperations like GridCC [7], RINGrid [8] and DORII [9]. Since that no standards have been produced by the RISGE [10] Research Group yet we decided to demonstrate the feasibility of a REST design by implementing all the methods that are exposed by the original WS-* compliant interface used to monitor and control instruments. Our tests and measurements indicate better performance than the classical Web Service implementation. Finally it is worth noting that a WS-* and a REST based implementation can coexist and be used in parallel where is needed.

The rest of this paper is structured as follows: Section 2 presents a selection of works that are relevant for the purpose of this paper while Section 3 introduce our proposed REST APIs. Finally in Section 4 we presents some performance comparison between Web Service (WS) and REST and in Section 5 we summarize our experience and we draw our conclusions.

2 Related Work and Background

Traditional efforts in providing remote access to equipment propose a common instrument middleware based on Web Services using SOAP over HTTP as a communication layer and specific WSDL interfaces [11], [1]. However modern software products propose different approaches for consuming stateful resources.

In [12] Foster et al. present and compare the four most common techniques for defining interactions among Web Service to support stateful resources (REST, WS-RF [13], WS-Transfer and "non standard"). REST is emerging as a lightweight technique for the remote consumption of services in modern e-business applications. Therefore we decided to investigate if this approach may be adopted in the context of accessing remote instruments. So far not much effort has been spent in this task however the following contributions are relevant for this objective.

In [4] techniques on how to build RESTful Web Services are underlined and a detailed comparison between REST and WS-* can be found in [3]. REST has also started to make inroads in different application domains. In [14], for example, a set of REST APIs for accessing mobile phones information such as photos, tags and manipulating device properties is presented.

In this paper we present a definition of a RESTful service for accessing, controlling, and monitoring remote instruments. Our proposed REST APIs (described in Section 3) maintains all the functionality that was previously exposed via a Web Service interface thus showing that REST can be a good example of an alternative technology platform for instrument management. The rest of this section continues with a description of the Resource Oriented Instrument Model

that we are considering (Section 2.1) and with a brief background description about REST where we outline its most relevant features (Section 2.2).

2.1 Resource Oriented Instrument Model

Considering the heterogeneous nature of instruments, one current shortcoming is that the applications that use them must have a complete operational model of the instruments and sensors they work with.

We can consider a generic model for a device consisting of a collection of parameters, attributes and a control model, plus an optional description language [1]:

- **Parameters**: are variables on which the instrument configuration depends, like range or precision values of a measure;
- **Attributes**: refer to the properties of the actual object that the instrument is measuring, such as the values that are being measured;
- **Finite State Machine**: this defines a control model, which represents the list of commands that the instrument can support. Commands are usually related using a Finite State Machine but in principle different formalisms (such as Petri Nets, Rule-based systems, etc.) may be adopted.
- **XML-based description language** that provide information about the semantic of the particular instrument such as SensorML [15] or OWL [16] etc.

The main difference between parameters and attributes concerns the access patterns that should be supported to read their data values. While parameters are typically read by polling, attributes should additionally support also an event-based or stream-based publish/subscribe approach. Therefore, both push and pull access patterns must be supported for some kinds of attributes.

This model is used for the representation of generic instruments. Our goal is also to provide support for more complex systems, where devices are logically (or physically) grouped into hierarchies for aggregating data and/ or distributing commands in more convenient ways. Therefore a way to retrieve the topology of the devices must be provided as well.

The code developed for controlling and monitoring devices is usually difficult to develop and expensive to maintain especially when the underlying instrument hardware is changed and/or improved. A primary design goal of this model is to externalize the instrument description so that applications can build an operational model on the fly. This approach makes it possible to preserve investments in codes as instrument hardware evolves and to allow the same code to be used with several similar types of instruments. Different representations of this model can be provided in order to let user decide the most convenient way for accessing and controlling the physical instrument.

2.2 REST Background

In the following we give a quick overview over the design principles and constraints of the REST architectural style. For more information we refer the interested reader to [2,4,17].

A RESTful service exposes its state, data, and functionality through the *resource* abstraction. Each resource is identified by a Uniform Resource Identifier (URI [18]). Its state can have one or more representations and can be handled with a limited and predefined set methods. A resource can negotiate with clients which representation format should be used to exchange data and also can inform its clients about the supported methods. The set of methods that can be used to manipulate and interact with a resource are the following ones.

- **GET**: retrieve the current state of a resource.
- **PUT**: update the state of a resource[1].
- **POST**: create a resource within the current one.
- **DELETE**: delete a resource.
- **OPTIONS**: reflect upon which methods are allowed on a given resource.

These methods are commonly found in the HTTP protocol and carry additional semantics which helps to support stateless interactions (where each request is self-contained) and idempotency (where requests can be repeated without side-effects). The only method which is unsafe, i.e., cannot be retried without side-effects, is **POST**.

Whereas — according to the *uniform interface principle* — the set of methods provided by a resource if fixed to the previous ones, REST does not constrain the set of resources that are published by a service. Thus, the expressiveness of the interface lies in the selection of a URI naming scheme and in the definition of the resource representations that are exchanged for each request method, as opposed to the freedom of defining a specific set of methods for each service interface, like in traditional Web services.

3 REST APIs for Remote Controlling and Monitoring Instruments

In this section we apply the REST design guidelines to the definition of an API to control and monitor instruments. We first define the set of resource URIs (Section 3.1) and specify which of the GET, POST, PUT, and DELETE methods can be applied. Then in Section 3.2 we define the resource representation formats used to exchange data with an instrument.

This REST API has been directly implemented using the HTTP protocol, which also provides security, access control, accounting, and exception signaling where needed.

[1] If a resource does not exists, create it.

3.1 Resource URI Design

In defining the URI for addressing instrument resources we follow this structure:

/Context/<id>/Instrument/<id>/<instrument-resource>

where */Context/<id>* represents a configuration of the instrument itself, while */Instrument/<id>* represents a unique identifier of the instrument within the instrument element. A different naming convention for representation of the same concept may be adopted without changing the final result, however this approach to the design of "nice" URIs is one of the most used [4]. Note that this URI structure maintains the same structure and granularity of the addressing information of the original Web Service interface. Therefore this interface does not change the number of requests needed by the clients to perform the same operations.

Finally *<instrument-resource>* represents a resource published by the instrument such as a Parameter, an Attribute or the State machine. As presented in Section 2.1, an instrument must also allow introspection. To do so, for each instrument we define the following URIs that clients can use to discover more information about the instrument capabilities:

/Context/<id>/Instrument/<id>/Description
/Context/<id>/Instrument/<id>/Status
/Context/<id>/Instrument/<id>/FSM
/Context/<id>/Instrument/<id>/Parameters
/Context/<id>/Instrument/<id>/Attributes

where:

- **Description**: represents the description of the instrument in a given Language (SensorML [15] or OWL [16] , plain text, etc)
- **Status**: get the current status of the instrument.
- **Finite State Machine (FSM)**: inspect the finite state machine representation that is mapped as a set of transition plus an initial state
- **Parameters**: retrieve the list of parameters exported by the instrument
- **Attributes**: retrieve the list of attributes exported by the instrument

Few of these resources (like the FSM) may have a non trivial representation, which will be defined in Section 3.2.

Tables 1 and **??** summarize the resources used for representing an IE and give a detailed specification of which methods are allowed for each resource.

More in detail, the URI */Context/<id>/Instrument/< id >* has the following semantics when used in conjunction with each method:

- GET */Context/<id>/Instrument/< id >* : Retrieve the list of instruments controlled supervised by the given instrument-id
- PUT */Context/<id>/Instrument/< id >* : If it is not already present, it creates an instance of the given instrument-id by instantiating the proxy for the real instrument. Otherwise it simply configures an existing proxy.

Table 1. REST Model for Controlling and Monitoring Instruments

Method	**URI**
Description	
GET	/Context
return the list of possible instrument configurations or instruments topologies	
GET	*/Context/<id>/*
return the list of intruments that are accessible in a given configuration	
GET	*/Context/<id>/Instrument/<id>/Description*
Read the description of the device(s)	
GET	*/Context/<id>/Instrument/<id>/Status*
Read the current instrument status	
GET	*/Context/<id>/Instrument/<id>/Parameters*
Retrieve a list of parameters of the given instrument	
GET/PUT/POST/DELETE	*/Context/<id>/Instrument/<id>/Parameter/<id>*
Access the description of individual parameters	
GET	*/Context/<id>/Instrument/<id>/Attributes*
Retrieve a list of attributes of the given instrument	
GET/PUT/POST/DELETE	*/Context/id/Instrument/<id>/Attributes/<id>*
Access the description of individual attributes	
GET	*/Context/<id>/Instrument/<id>/FSM*
Read the finite state machine description of the instrument	
GET	*/Context/<id>/Instrument/<id>/FSM/Transition/<id>*
Read the description of a transition	
GET	*/Context/<id>/Instrument/<id>/Commands*
Read the description of a command	
POST	*/Context/<id>/Instrument/<id>/Command/<id>*
Execute a command	
PUT	*/Context/<id>/Instrument/<id>/Transition/<id>*
Execute a Transition	

– DELETE /Context/<id>/Instrument/< *id* > : De-instantiate the proxy.

This approach to the modeling of instruments with resources has the following implications:

– Clients can browse the possible set of configurations and the list of instruments using the */Context* and *Context/<id>* .
– Clients can get information and about the instruments using /Description, /Status, /Parameters, /Attributes, /Commands and /FiniteStateMachine.
– Clients can use the URI */Context/<id>/Instrument/< id >* for introspection and for instantiating the instrument proxy control structure.
– Clients can execute commands and trigger FSM transitions using the the */Command/<id>* and */Transition/<id>* URIs.
– We decided to allow a POST and DELETE commands on Parameters and attributes. However few instruments may not allow such operations. In this case the error 405 (Method Not Allowed) can be used.
– Parameters, Attributes, Commands and FiniteStateMachine may return empty values because not all instruments may implement all these functionalities.
– The URI structure map the model presented in Section 2.1 trying to minimize the number of service requests needed in order to retrieve a conceptual information.
– Few complex structures have been used for representing information related to the instrument.

3.2 Resource Representation Format

Concerning the data, the API supports the exchange of data for different applications using formats such as ATOM, JSON, or binary JSON. ATOM is a popular XML format that is commonly adopted in Web data feeds while JSON is, compared to XML, a lightweight format for exchanging structured data [19].

In this paper we concentrate our attention in the XML representation of the information but similar considerations could be repeated for different serialization formats.

What follows is an XML representation of a Parameter:

```
<Parameter>
.       <Name>
.       <Value>
.       <Unit>
.       <Min>
.       <Max>
.       <Description>
.       <Parameter>
.       ....
.       </Parameter>
.       <Parameter>
.       ....
```

. </Parameter>
</Parameter>

We choose a complex XML representation of the information because parameters have the possibility of containing other parameters. Therefore a representation */Parameter/<id>/<Subparameter/<id>* may cause a eccesive fragmentation of the interface thus increasing the number of requests needed in order to retrieve the complete parameter value.

Additional resources are represented as follows.

- **Attributes**: They are represented in a structure that is quite similar to the one used by parameters
- **Finite State Machine**: It is represented as a set Transitions plus an initial state. (<initialState> <Transition> <Transiton> ...)
- **Transitions**: are represented as a command, an initial state and an arriving state. (<fromState> <toState> <Command>)
- **Commands**: are represented as a Name plus a set of Parameters. (<commandName> <Parameter>)

3.3 Usage Scenarios

In this section we present few usage scenarios of the proposed REST APIs. Service requests related to instrument introspection may require 1 or 2 calls while the majority of the requests for controlling and monitoring the devices can be handled in a single call. We can also note that in terms of the supported use cases there are not many difference between the REST APIs and the original WS-* APIs [1].

Controlling the List of available Instruments
An Operator can control the list of available Instruments sending a GET request to the IE (URI: /Context) in order to retrieve the list of possible configurations and use this information in order to access each context (URI: */Context/<id>*).

Accessing to the current setting of an instrument
The Parameters of an Instrument represents the current setting of an instrument. The list can be retrieved using a GET request to */Context/<id>/Instrument/<id>/Parameters* where */Instrument/<id>* identify a device with configuration */Context/<id>/*.

Retrieve the list of Instruments controlled by a given instrument
Assuming that the URI */Context/<id>/Instrument/<id>* represents the instrument supervisor, the list of controlled instrument can be retrieved using a GET request to */Context/<id>/Instrument/<id>/* that return the list of URIs identifying the instruments controlled by the supervisor.

Retrieve the list of available commands of a given instrument
The URI */Context/<id>/Instrument/<id>/* addresses the instrument that we

want to use. We can retrieve the list of available commands sending a GET request at the URI */Context/<id>/Instrument/<id>/Command/*.

Execute a Measure
We can execute a measurement command sending a POST request at the URI */Context/<id>/Instrument/<id>/Command/measure* the results of the measure can be fetched sending a GET request to the URI */Context/<id>/Instrument/<id>/Attributes/<id>*.

4 Performance Results

In this Section we present some preliminary benchmark tests. The goal is to understand the scalability, the overhead and the flexibility our proposed REST APIs compared with the existing Web Service (WS) implementation. We run two different tests:

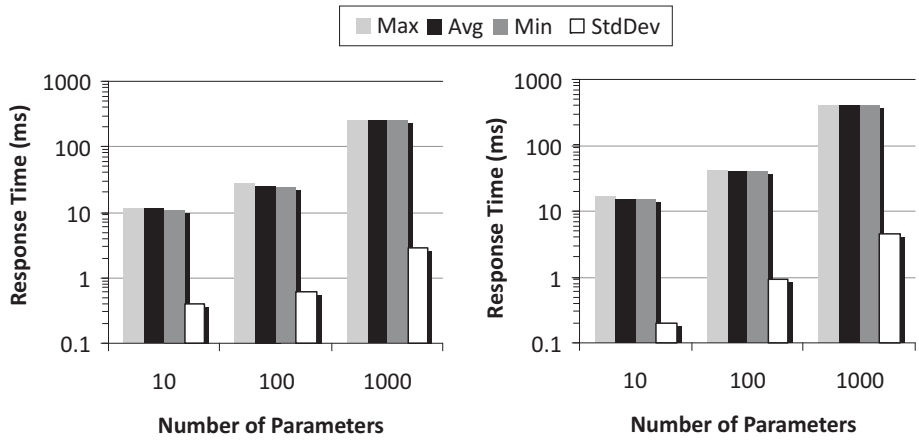
- **Throughput**: we measure the maximum number of requests per second that can be handled by the server as it is saturated with requests from clients.
- **Service Request Time**: we measure the time needed by a single client in order to perform a service invocation.

The experimental setup consists of a Tiny Instrument Element (IE) [5] containing only one Instrument Manager controlling a dummy instruments which can be configured using a variable number of parameters (10, 100, 1000). The values of these parameters were automatically generated and fixed. One or more clients were retrieving the parameters from the instrument using the Web Service and the REST APIs.

The service was running on an Intel dual core 2.1 Ghz with 2 GB of RAM with java 1.5. Apache Axis 1.4 [20] was used as WS provider while REST APIs were implemented using the RESTlet framework [21]. Both alternatives used an XML encoding of the parameters.

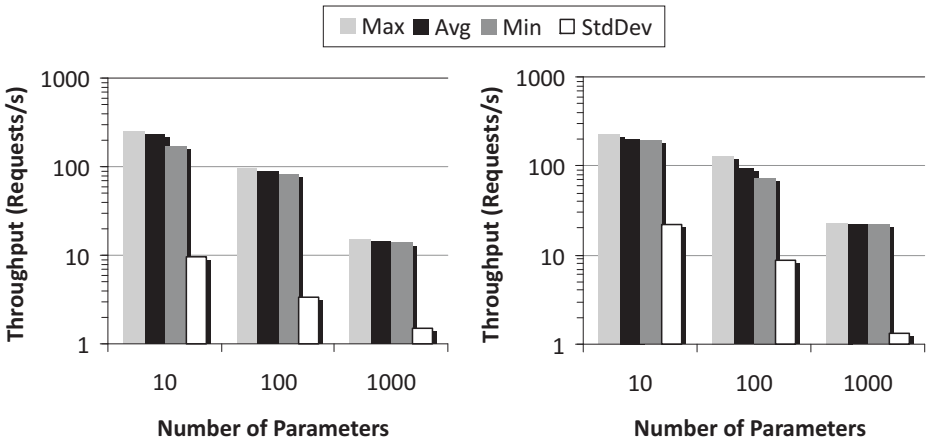
In Figure 1 the service response time for both WS and REST is presented, while in Figure 2 the maximum number of request per second is shown. Each measure has been taken 1000 times and in the plots Average, Minimum, Maximum and Standard Deviations are reported. We note that in each experienced case (10, 100, 1000 Parameters) an IE based on REST offers a shorter response time. In terms of the throughput, the WS performed better for small messages (10 Parameters) while for big messages (1000 Parameters) REST achieved a higher throughput. This is a counter intuitive behavior as the overhead of SOAP should be higher fractionally for small messages. Moreover the experienced standard deviation of REST is larger than WS-*. This may be due to the RESTlet framework [21] which begin to be stable but has not yet reached its maturity.

In order to provide a fair comparison, both WS and REST implementations were configured to exchange XML data. However with REST it would have been possible to use different resource representations, and – for example – choose a more lightweight data format such as JSON [19], for which we expect an additional reduction of the overhead.



(a) REST (b) Web Service

Fig. 1. Response Time Comparison



(a) Web Service (b) REST

Fig. 2. Throughput Comparison

5 Conclusion

In this paper we investigate the possibility of applying the REST design principles to the problem of monitoring and controlling scientific instruments in order to provide a lightweight solution. We present a set of APIs for controlling and monitoring devices that follows the REST design principles. Our prototype demonstrates the feasibility of implementing all the methods that are exposed by the original WS-* compliant interface used to monitor and control instruments. Both implementations can coexist and be used in parallel. Tests and measurements of our prototype using XML payloads indicate better performance of the RESTlet container as opposed to the Axis-based Web service implementation. We plan to further leverage the flexibility of REST by introducing more lightweight data formats with the potential to further reduce the overhead of the instrument manager interface. Also, the possibility of directly using HTTP streams to perform measurement data streaming looks promising.

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. Fielding, R., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology **2**(2) (2002) 115–150
3. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In: 17th International World Wide Web Conference (WWW2008), Beijing, China (April 2008) 805–814
4. Richardson, L., Ruby, S.: RESTful Web Services: Web Service for Real World. O'Reilly (May 2007)
5. Tiny Instrument Element Project: `http://instrumentelem.sourceforge.net/`
6. Lelli, F., Pautasso, C.: The Tiny Instrument Element Project. in proc of 4th International Conference on Grid and Pervasive Computing (GPC 2009) (May 2009)
7. GridCC Project Web Site: `http://www.gridcc.org/`
8. RINGrid Project web site: `http://www.ringrid.eu/`
9. DORII Project web site: `http://www.dorii.eu/`
10. Remote Instrumentation Services In Grid Environment: `http://forge.gridforum.org/sf/projects/risge-rg`
11. McMullen, D., Devadithya, T., Chiu, K.: Integrating Instruments and Sensors into the Grid with CIMA Web Services. Proceedings of the Third APAC Conference on Advanced Computing, Grid Applications and e-Research (APAC05) (September 2005)
12. Foster, I.T., Parastatidis, S., Watson, P., Mckeown, M.: How do I model state?: Let me count the ways. Commun. ACM **51**(9) (2008) 34–41
13. OASIS: Web Services Resources Framework (WSRF 1.2). (April 2006) `http://www.oasis-open.org/committees/wsrf/`.
14. Riva, C., Laitkorpi, M.: Designing Web-Based Mobile Services with REST. In: Proc. of Standard Performance Evaluation Corporation (SPEC) Benchmark Workshop. (January 2009)

15. OGC Sensor Web Enablement:: www.opengeospatial.org/functional/page=swe
16. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide,W3C. also available at http://www.w3.org/TR/owl-guide/
17. Fielding, R.: A little REST and Relaxation. The International Conference on Java Technology (JAZOON07), Zurich, Switzerland. (June 2007) http://www.parleys.com/display/PARLEYS/A%20little%20REST%20and%20Relaxation.
18. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): generic syntax. IETF RFC 3986. (January 2005)
19. Crockford, D.: JSON: The fat-free alternative to XML. In: Proc. of XML 2006, Boston, USA (December 2006) http://www.json.org/fatfree.html.
20. Graham, S., Simeonov, S., Boubez, T., Daniels, G., Davis, D., Nakamura, Y.: Building, Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI. Sams (December 2001)
21. RESTlet framework: http://www.restlet.org/