Implementation of a distributed auction application using REST Web Services

Luis Fernando Nicolás Alonso, Álvaro Fernández Fernández

ct1x01

Abstract

This report discussed about the Lab Assignment 2 that consists on developing a distributed application that manages an auction system employing a Java framework for developing REST Web Services: Restlet. The report is mainly focused on the problems found and tries to establish a comparison with Object-Oriented middleware. After the development process, the report shows how REST Web services allows to create a loosely-coupled distributed system using URIs for identifying resources and HTTP methods as the uniform interface.

1. Introduction

Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web [1]. The term was defined by Roy Fielding in his PhD Thesis [7]. The REST architectural style describes several constraints applied to the architecture, such as client-server, stateless communication, cacheable and uniform interface.

The client-server constraint means that clients and servers are connected by a uniform interface. This client-server separation implies that the portability of the client code is improved, and also servers are simpler and more scalable. This is because of two issues, on the one hand, clients are not aware of data storage on servers, and on the other hand servers are not concerned with the user interface or the user state.

The stateless communication improves reliability and scalability due to the server has not to maintain state information.

The cacheable constraint allows clients to cache responses. Thanks to a well-managed caching, some client-server interactions could be eliminated, improving also scalability and performance.

The uniform interface simplifies and decouples the architecture, and enables servers and clients to be developed independently. The uniform interface is considered fundamental to the design of any REST service.

Nowadays, REST can be found in many places on the Web, such us the blogosphere (which is mostly REST based) and Amazon.com.

A REST web service is a simple web service implemented using HTTP and the principles of REST. REST web services offer a collection of resources identified by URIs (Uniform Resource Identifier) and support HTTP methods as the uniform interface. These methods are GET, PUT, POST and DELETE [1].

In this report, it is discussed the development of a very simple distributed application that supports the management of an on-line auction company [2]. The aim is to have a

first approach with the development of REST Web Services using Restlet. In addition, the report is focused on the advantages and drawbacks of REST Web Services and a comparison with other middleware technologies, such as Java RMI.

Section 2 discusses the development process of a REST Web Service using Restlet. Section 3 compares that development process whit the centralized case and with the Java RMI case. Then, section 4 describes the distribution of the services in different machines, as well as the tests executed with the clients/servers of other groups.

2. Development process

Thanks to Restlet, the distributed application implementation is not much more difficult than the centralized case. On the server side, a new component is created. Then, a HTTP server connector is added to the component. Finally, the application is attached to the default host (in this case, there is only one host) and the component is started. This is done with the following code:

```
Component component = new Component();

// Add a new HTTP server listening on port 8009
component.getServers().add(Protocol.HTTP, 8009);
// Attach the application.
component.getDefaultHost().attach("",
          new AuctionApp());

component.start();
```

The code above is right only if the server is not going to be client of other service. In terms of this assignment, the code above is part of the *AuctionManager* code. When developing the *ItemManager*, it is very important to be aware that it will be server and client at the same time. So, the code is very similar, but a HTTP client connector must be added to the component so that the *ItemManager* can communicate with the *AuctionManager*. Then, the code will be like this:

```
// Create a new Component.
Component component = new Component();

// Add a new HTTP server listening on port 8008.
component.getServers().add(Protocol.HTTP, 8008);
//Add a new HTTP client listening
component.getClients().add(Protocol.HTTP);
// Attach the sample application.
component.getDefaultHost().attach("",
          new ItemApp());

// Start the component.
component.start();
```

When the component is started (this will start the server and the default host), a router restlet is created, whose role is to route each call to a new instance of a resource. That is, it establishes a relation between an URI and a resource. So, the router helps the developer to manage the calls: the router will find the target resource which is in the framework (an instance of the Resource class). Basically, with the attach() method, the router creates a Finder (a subclass of Restlet), which takes a Resource class reference and will automatically instantiate it when a request comes in. Then it will dynamically dispatch the call to the newly created instance, to one of its methods (GET, POST…)

depending on the request method [6]. All of this is achieved with the following code (only is shown the code for the *ItemManager*):

```
// Create a router Restlet that routes each call to a
// new instance.
Router router = new Router(getContext());

// Defines several routes
router.attach("/items", ItemListResource.class);
router.attach("/items/{itemId}", ItemResource.class);

return router;
```

Now, in order to finish the servers, the last thing to do is to implement the GET, POST… methods of a resource in order to offer the services required in the Auction application. As explained before, the router will dispatch incoming calls to the corresponding method of the resource instance, thanks to the URI. Then, the method will perform the necessary actions to offer the service, will inform about the status of the operations (modifying the Status) and will return a representation with the information required in order to inform the user. This returned representation can change depending on the functionality accessed. If the functionality is going to be accessed with a browser, HTML representations should be served. But, if the functionality is going to be accessed with a dedicated client, the representations used for communicating the server and the client are web forms.

On the client side, the code is very simple. First of all, a form is created in order to store the information that has to be passed to the server. Then, a new instance of ClientResource is created, so that the client can connect to the server connector which is listening on the server side (at the specified port and host). When the connection is established, and the ServerResource class is instantiated (what is done thanks to the router), the ClientResource will call the corresponding methods passing the information required by means of a web form. Finally, the returned representation and the status are checked in order to inform the user about the success or the failure of the operation. For example, this part of the *PostItem* client code follows the previous explanation:

```
Form form = new Form();
form.add("name",args[0]);
form.add("seller",args[1]);

ClientResource cr = new
      ClientResource("http://localhost:8008/items");

Representation r = cr.post(form.getWebRepresentation());
System.out.println(r.getText());
Reference uri = r.getLocationRef();
System.out.println("The uri for the posted Item is: " +
      uri.toString());
```

If the functionality offered by the servers is accessed with a web browser, the operations on the server side are the same, and HTML representations are served.

3. Comparison with centralized case and assignment A1

When comparing with the centralized case, it is important to notice that when developing a centralized application, clients and servers will be in the same machine. So, the interaction between them is handled directly by means of method invocations. The only thing that has to be done is to import the corresponding packets (if the

different classes are in different packets, what is very common). As a result, the implementation will be very simple, as the developer doesn´t need to worry about communication issues. In that case, the errors are handled by means of exceptions.

On the other hand, the development process using Java RMI [3] [5] is quite different. Of course, in that case client and server could run in different machines, but there are several differences when implementing a REST Web Service with Restlet. Basically, in Java RMI, the server has to create the remote objects that will provide the different services, and export them to obtain the stub that has to be passed to the clients. Then, that stub is register within the Java RMI Registry, so that the client could get a reference (a stub) to the remote object. When the stub is registered, the client obtains the stub for the remote object from the registry (it is important to be aware that the client had to know the interface, in order to instance the stub) and then, the client is ready to invoke the methods on the remote object´s stub. From the client´s point of view, invoking remote methods is like calling ordinary Java methods, but the stub will communicate transparently with the server (doing the marshalling and unmarshalling processes) in order to return the corresponding information.

In the Restlet case, the first difference noticed is that there is no registry. But the client can invoke methods on the server side. The idea is that every resource in the server is identified by a URI. So, using HTTP methods, the client (which is a component) can access resources on the server side and obtain its representation by using connectors and the URI of the proper resource. And then, client and server can interchange information of resources through representations. In conclusion, the role of the registry is played by the URI.

Furthermore, HTTP methods are used as the uniform interface. By doing this, the architecture is decoupled, and clients and servers can evolve independently. In addition, the interactions are simpler, as the distributed system is not using a specific interface, as it is done in RMI for example. This allows using clients from other developers, improving interoperability.

As a result, any off-the-shelf application that uses that uniform interface (that is, HTTP methods) can be used as the client. That´s why it is possible to use an off-the-shelf web browser as the client, as it is using the HTTP method GET to access the functionalities of the servers.

Another difference between Java RMI and Restlet is the way to handle errors. When using Java RMI, errors are handled by means of exceptions. If there is a failure during the process, the server will throw an exception, and it will be caught by the client. Furthermore, the client will treat that remote exception as it were a “normal” exception. In the Restlet case, to inform about errors (and also about the success of the operation and other information) HTTP codes are used. These codes represent a previously defined status, which will inform the client about the type of error (client error, server error…) and also contains a description of the error in order to obtain more information. When the status is changed to a status associated to an error code, the Restlet will also catch an exception (a ResourceException), and then the status and the description of what had happened can be obtained.

4. Distribution

The ItemManager and the AuctionManager were running in two different machines in order to test the distributed application in a more realistic context. The ItemManager

server was running in a machine with IP 192.168.1.35 and the AuctionManager was located in another machine with IP 192.168.1.34 (Figure 1). There was no difference from the case using an only machine because the same interface is used for the communication, that is, the same protocol is used (HTTP) as well as the same web forms. Notice that to make IP and port configurations easier, a new class called Configuracion has been created.

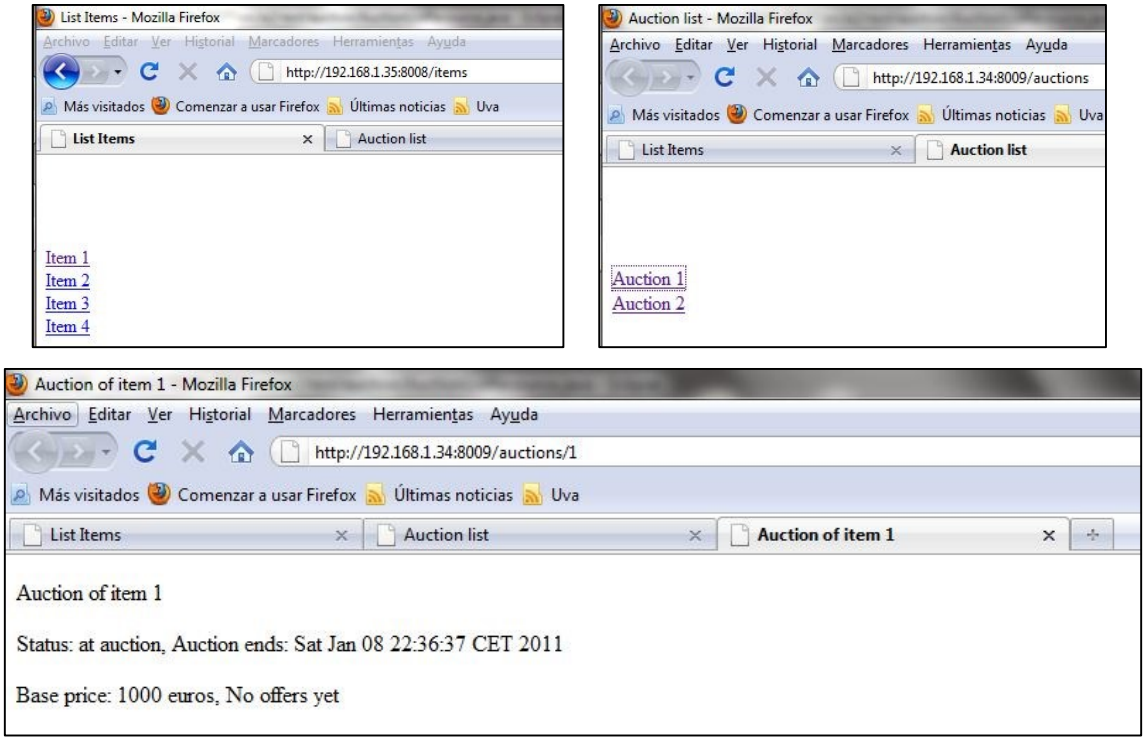

Figure 1. Screenshots of the distributed application running.

Also our clients were tested with other servers developed by other group (ct1x03) and vice versa. No problems were found because the interface is the same in both implementations. This is one of the advantages of Restlet.

5. Conclusion

This report tries to focus on identifying important aspects about the development of a REST Web Service using Restlet. It also tries to compare the Service-Oriented approach with Object-Oriented middleware.

REST allows developers to build loosely-coupled distributed systems, with a well-defined client-server separation. The uniform interface between clients and servers provides simplicity and independent evolvability. So, clients and servers are not concerned with the issues of the other part. The uniform interface provides a generality of access, and in the REST Web Services case, this allows users to access the services offered by the servers with any client that uses HTTP methods as the uniform interface, improving software reuse. But, in contrast, when the applications that have to be developed are very specific, the uniform interface could be less efficient.

A REST Web Service is a service in the Web following the REST constraints and using Web technologies. As a result, the information is split into resources, which are identified by URIs. In addition, those resources are manipulated through representations using HTTP methods as the uniform interface, and are accessed by means of URIs.

When using Object-Oriented Middleware, the goal is to achieve transparency from developer´s point of view and establish the principles of Object-Oriented paradigm. But this is only at "Object level". On the other hand, Service-Oriented middleware goes beyond the concept of object, and can wrap more complex concepts as component. So, Service-Oriented Computing can provide "self-describing, platform-agnostic computational elements" [1].

6. References

[1] G. *Vega, "Introduction to Distributed Systems and Middleware"*. Department of Signal Theory, Communications and Telematic Engineering. ETSIT, UVA. October 2010.

[2] G. Vega, *"Lab Assignment 2: CT1-RESTAuction REST Web Services"*. Department of Signal Theory, Communications and Telematic Engineering. ETSIT, UVA. November 2010.

[3] G. *Vega, "Object-Oriented Middleware Java RMI & Java RMI-IIOP"*. Department of Signal Theory, Communications and Telematic Engineering. ETSIT, UVA. November 2010.

[4] G. Vega, *"REST Web Services Introduction"*. Department of Signal Theory, Communications and Telematic Engineering. ETSIT, UVA. November 2010.

[5] L. F. Nicolás, A. Fernandez, *"Implementation of a distributed auction application using Java RMI and Java RMI-IIOP"*, Complementos de Telemática I, Lab Assignment 1. November 2010.

[6] *"Restlet 1.0–Tutorial"*. Noelios Technologies. 2005-2010. Available at http://www.restlet.org/documentation/1.0/tutorial, last visited: 5 January 2011.

[7] R.T. Fielding, *"Architectural Styles and the Design of Network-Based Software Architectures"*, PhD Thesis, 2000.