Unit 7. Remote Procedure Calls using Web Services

Outcomes Summary. You will learn

- how remote procedure calls compare with REST principles and the distributed objects model as a basis for developing a DS
- about the important Web Services standards SOAP and WSDL; their purpose, design, operation and good practices
- to use the Axis tools via Eclipse to develop Java clients and servers for Web Services based on SOAP and WSDL

Further Reading: CDK 2005 Ch.19.

Context. We have studied how to build systems using REST principles and we established that they have indisputable benefits but force developers to use designs that are quite far from the distributed objects model of computation. REST has been gaining in popularity and industry acceptance only fairly recently, partially due to some level of disappointment with the other, slightly older and more wide-spread forms of Web services (WS). (We studied REST first because it can be implemented using a much simpler middleware than the more established RCP WS.) Even these "older" WS standards are quite young and have been going through rapid development and only recently have started to mature and stabilise.

The idea of a remote procedure call (RPC) predates the object-oriented paradigm and people usually find it to be the most straightforward way of making remote computers communicate. Therefore, the first generation of WS implement RPC. We shall see that RPC, like REST, is quite far from OO principles but in a very different way. One of the recent trends in WS is to combine the two ideas (ie RPC and REST) to achieve a model that is almost as powerful as the distributed objects model of Java RMI. We will address this trend in unit 8.

7.1 Definition and comparison to previous models

A remote procedure call consists of the following steps:

- distributed node A locates a "port" or "remote procedure" on distributed node B;
- node A sends a call to this port together with appropriate parameters and waits for a response;
- node B receives the parameters, does some work and sends a response to node A.

This model closely mirrors a normal procedure call as presented by any imperative programming language, eg C, Ada. It is also analogous to calling a *static* method in Java.

The differences between a local procedure call and an RPC are analogous to the differences between a local method invocation and an RMI: ie parameters must be either serialisable or valid remote references. Nevertheless, RPC standards do not usually specify any format for remote references — RPC is often restricted to serialised parameters. In the context of WS, the parameters of remote procedures are usually declared to be either of standard primitive types (such as integer or string) or they are specified by an XML schema. In either case, their serialisation is straightforward.

7.1.1 Conceptual comparison

Figure 7.1 illustrates the conceptual view of a typical WS RPC system focusing on how it differs from a similar system designed using REST or RMI. In RPC WS each server node usually exposes in its remote interface one "object" (representing the whole node) and a number of methods for this object. This is in contrast with REST where each node exposes a large number of resources all of which are accessed using the same methods. In RMI, a node allows other nodes to access some of its objects in a controlled way as specified in the remote interfaces for these exposed objects.

7.1.2 Overview of WS RPC standards

Keeping focus on the openness of the resulting systems, in RPC WS the remote procedures available on a server have to be fully described without any assumptions on programming language or libraries used to implement the server and client nodes. The only acceptable assumption is that the communication will take place over the Internet, most likely using HTTP.

It is a surprisingly complex task to fully describe all operations on a server, their parameters and the way in which the operations are specified and parameters are represented within RPC messages. Fortunately, some aspects of RPC descriptions can be and should be the same or very similar for all services. Such aspects have been described as open standards and can be used off-the-shelf when specifying a new service. For example, XML should be used to represent parameter *values* of WS remote procedures and XML schemas should be used to describe the parameter *types* inside service specifications.

SOAP is a standard protocol for wrapping the XML data related to WS RPC calls in so-called envelopes. (Also, the SOAP envelope itself uses an XML format.) The purpose of this wrapping is to allow extra data such as digital signatures (think of a wax seal), transport error descriptions (think of postman's notes), binary attachments, origin and destination addresses to be added to each RPC message in a standard way. Another purpose of SOAP envelopes is to allow transport using other protocols than HTTP, although this is rarely used.

We have established that standards such as XML and SOAP help to make service specifications shorter. Even more importantly, the standards make it easier for the programmers as most programming environments would have libraries that support these standards well. To go one step further, it is often the case that the programmer

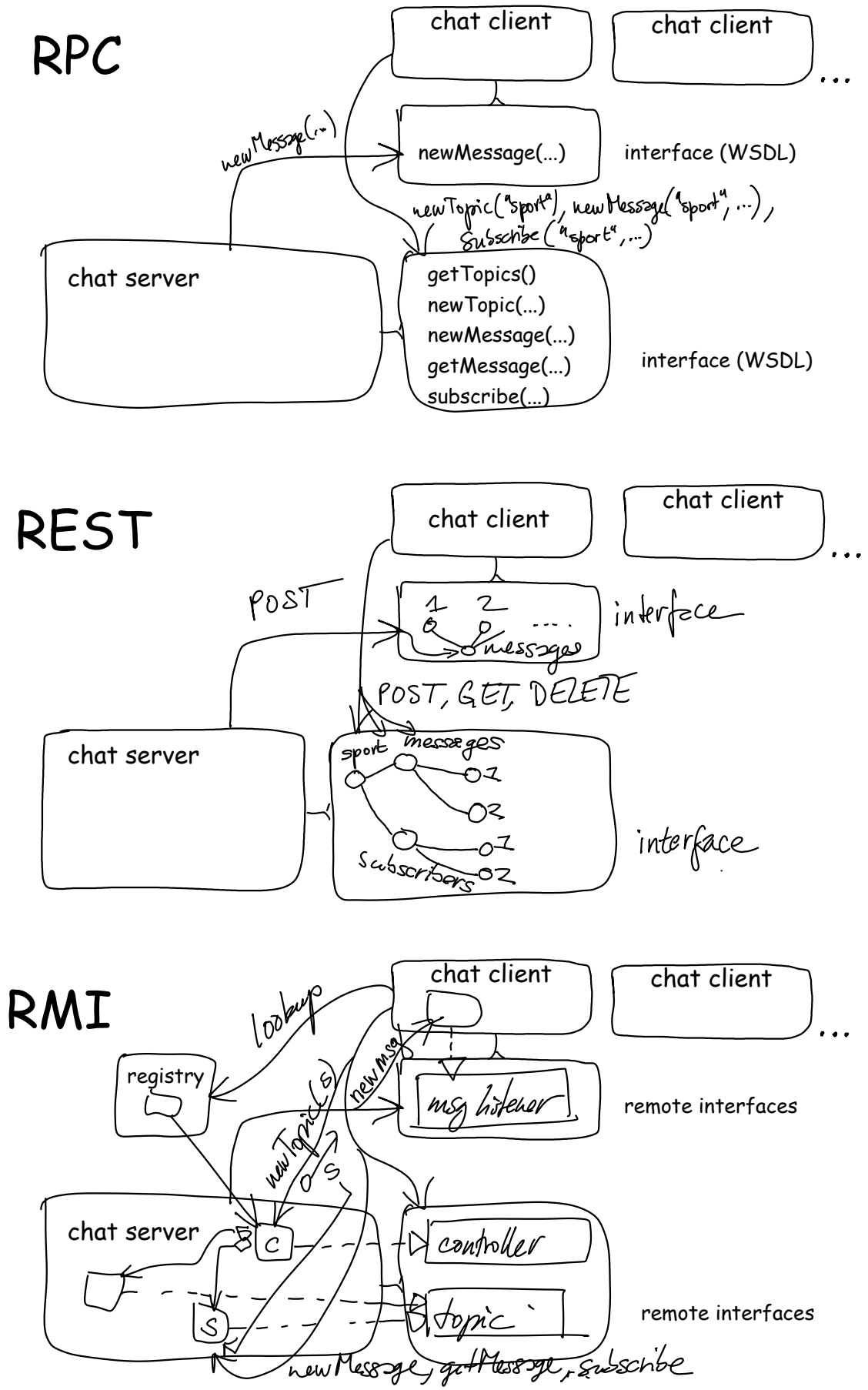

Figure 7.1: Comparing RPC with REST and RMI using the chatroom example.

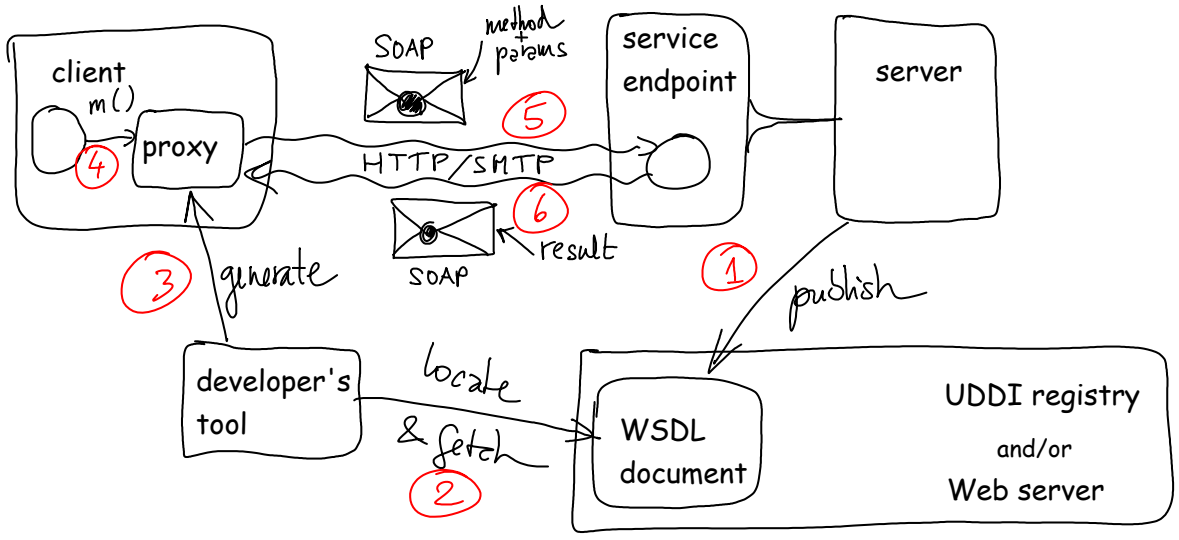

Figure 7.2: Collaboration of participants in a typical WS RPC exchange.

does not even have to read the RPC service specification at all! The specification is understood by a computer program that translates it to a much simpler specification in their preferred programming language. For example, we shall soon see how one can take a description of a WS in the *Web Service Description Language* (WSDL) and have it automatically converted to a Java class whose instance acts as proxies for the service (see Figure 7.2). Thus the remote service becomes a local object to the client programmer in a similar fashion as it did after a `Naming.lookup` call in Java RMI. The key to making this possible is that not only most of the WS RPC *interaction* follows open standards (namely SOAP, XML, HTTP), also the RPC service *description* follows an open standard (namely WSDL).

For a RPC WS developer WSDL is at the heart of their system design. We will explore WSDL in some detail now and also in the following units.

7.2 Introduction to WSDL

Upon first inspection, a typical WSDL seems to be a very verbose document. For a simple service, it seems to require an inappropriate amount of information and appears to contain duplication. The purpose of many of its components becomes clearer only when one surveys a large number of services of different kinds. We will consider only one or two examples here and thus some aspects may not be entirely obvious. You can explore further WSDL documents on your own to get a thorough understanding of the language.

7.2.1 Structure overview

Let us consider a WSDL for the chatroom service. WSDL is based on XML and each WSDL is rooted in a `definitions` element. The attributes of the root element are references to various namespaces. One of these namespaces is special — it is called

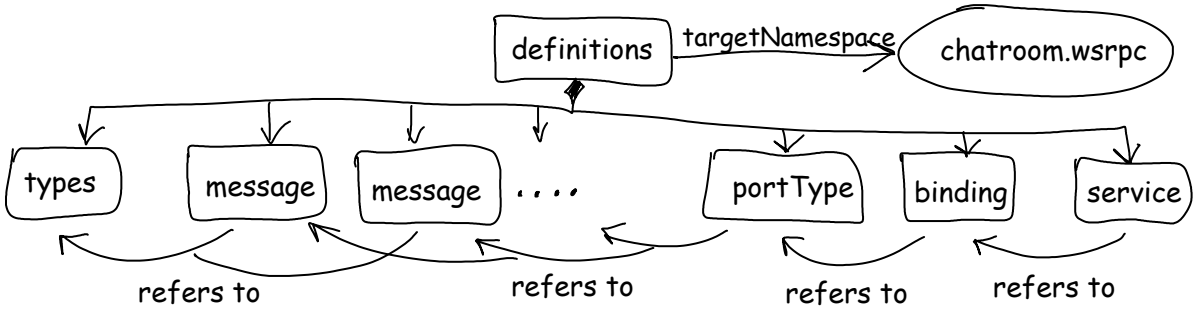

Figure 7.3: The top-level structure of `ChatRoom.wsdl`.

`targetNamespace`. Its value for our example is "chatroom.wsrpc". A WSDL document defines some concepts that can be imported and reused in other documents. The target namespace is the namespace into which these new concepts are added. For example, whenever WSDL defines a named XML schema datatype or a named remote interface, their name will be inserted into the target namespace and can be referred to from other documents using this namespace. (Recall that namespaces are like Java packages — they help to hierarchically organise concepts that are represented as XML elements and prevent/resolve naming conflicts among them.) When using a WSDL document to generate Java code, the target namespace is used to derive the package name for the Java code, in our example the package name will be `wsrpc.chatroom`. (The order is swapped because the namespace name is viewed as a shortened URI containing only a virtual DNS hostname and thus is read from right to left.)

Apart from the namespaces, the root element contains a number of other elements. The complete content of the root element is illustrated by an object diagram in Figure 7.3. The diagram also shows how the elements are subordinate to one another in terms of their interpretation. While data types make complete sense without any reference to the other elements, definitions of messages (more accurately: message types) have to refer to data types in order to describe their components. Similarly, a port type is defined in terms of message types, a binding applies to a port type and a service is a binding attached to a URL.

How these relationships are represented is shown in Figure 7.4. A port type's operation `newTopic` refers to a message type `newTopicRequest` and this message type refers to a schema element type `newTopic`, which contains a sequence of parameters for the operation `newTopic` (in this case only one parameter called `topicName`). Similarly, the return values, if any, are grouped into one schema element type `newTopicResponse`. This makes it possible to define operations that return more than one value.

In general, a WSDL document can specify any number of port types, each having any number of operations.

The example in Figure 7.4 shows a particular style of encoding RPC parameters — all of the parameters are encoded by a single element defined by the embedded XML schema. This style is called "document" style because all parameters are passed as one document from the point of view of the SOAP envelope. At present, this style is the preferred one. An alternative is the so-called "rpc" style in which the parameter

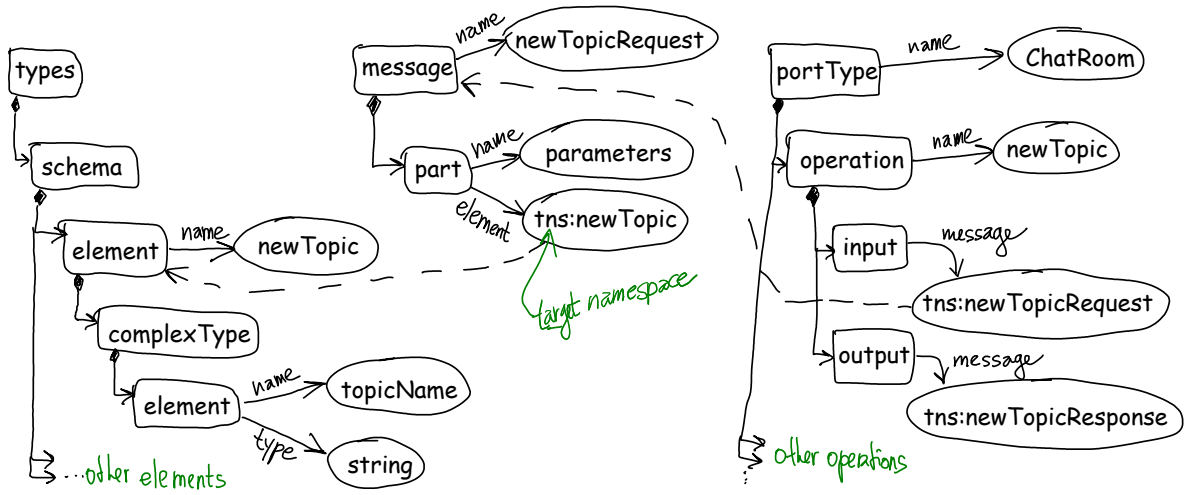

Figure 7.4: Example internal references inside `ChatRoom.wsdl`.

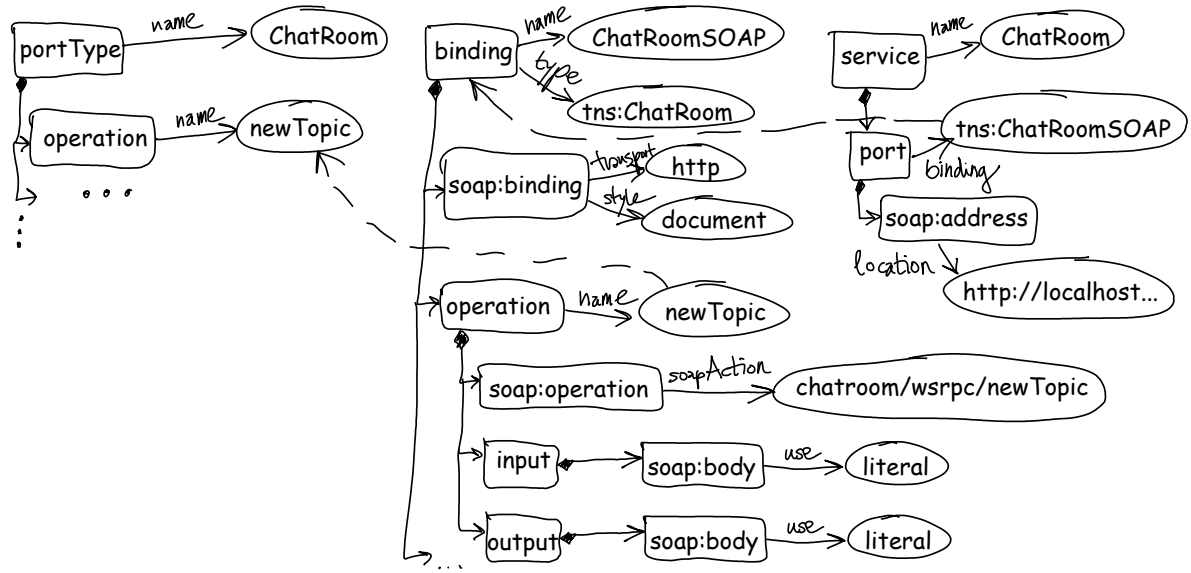

Figure 7.5: Linking of service with binding and binding with port type.

types are specified by the schema one-by-one and visible at the SOAP level. With the rpc style, the parameters should be specially encoded.

Figure 7.5 shows in further detail the contents of the other top-level elements in the WSDL document: a *binding* that associates a port type with a way to wrap its operations in SOAP envelopes; a *service* consisting of a *port*, which is a concrete URL through which this service is to be accessed. This WSDL defines only one binding of one port type and one service with only one port. The language supports multiple port types, bindings and services in one document. Also, a service is allowed to have multiple ports.

7.2.2 WSDL document in multiple files

A WSDL structure such as the one we have just descibed is usually contained in one file. WSDL has a mechanism to refer to other files in case it is beneficial to have the same structure defined in multiple files.

The simplest and most common case is when the types schema includes another XML schema using the standard XSD include element, eg:

```
<wsdl:types>
  <xsd:schema
   elementFormDefault="qualified"
   targetNamespace="http://s3.amazonaws.com/doc/2006-03-01/"
  >
    <xsd:include schemaLocation="AmazonS3.xsd"/>
  </xsd:schema>
</wsdl:types>
```

The point of storing a schema separately from the service specification is to be able to reuse the schema in other specifications. The `schemaLocation` attribute can be any URL. Thus it is easy to publish and share an XML schema across nodes in a DS.

Notice that the same schema can be included into schemas with different target namespaces. The elements of the included schema are added to the target namespace of the schema where it was included into. Thus the same elements may end up in different namespace when included multiple times. If this is not desirable, one should use an *import* statement rather than an *include* statement. An import statement preserves the namespace of the imported schema — all its elements will be available but only with the correct namespace, ie the target namespace specified inside the imported schema.

Instead of importing a schema, it is possible to import the whole another WSDL. This is desirable when several services share not only data types but also message definitions, port types or other components. With the import feature it makes sense to have incomplete WSDL documents that cannot be used on their own but only imported to other WSDL documents to make them complete.

7.3 Developing WS clients and servers

The quickest way to interact with an existing Web service is to get hold of its WSDL specification and use a generic human-interface client. Such clients let people manually select one of the available procedures, construct procedure parameters, make the RPC and then inspect the response. It is useful for testing one's own services as well as for studying the details of other people's services. WSDL is usually not sufficient for humans to fully understand a service. While it has all the technical details to allow correct RPC, it does not even attempt to explain the *effect* of the procedures nor the full meaning of their parameters. By invoking the service a few times we can often gain much better understanding of the service than by reading the WSDL.

The Eclipse Web Tools Platform (WTP) framework includes such a generic WS client. It can be started from the main menu by Run > Launch Web Services Explorer. There are also free web-based generic clients, eg `http://www.soapclient.com/soaptest.html`.

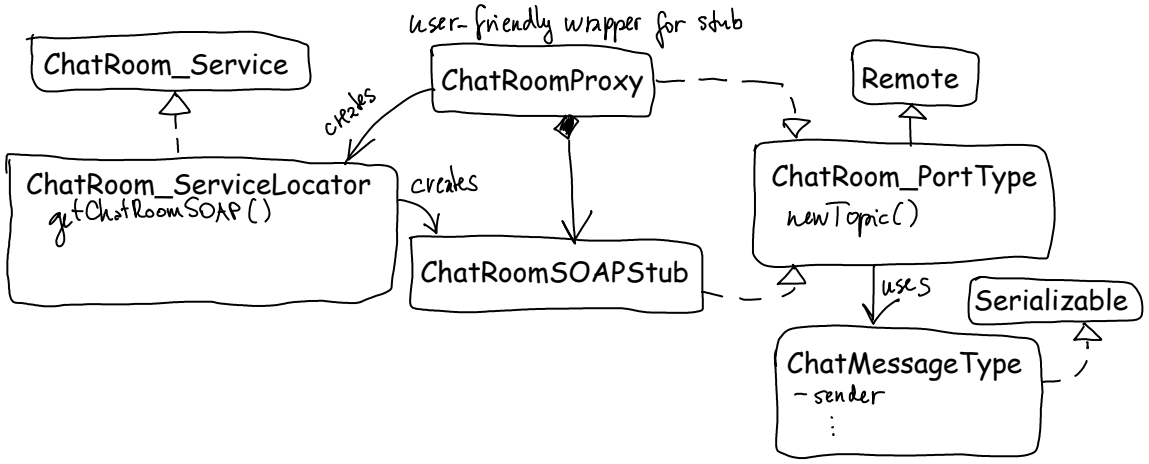

Figure 7.6: Files generated by Axis to provide a proxy for a Web service.

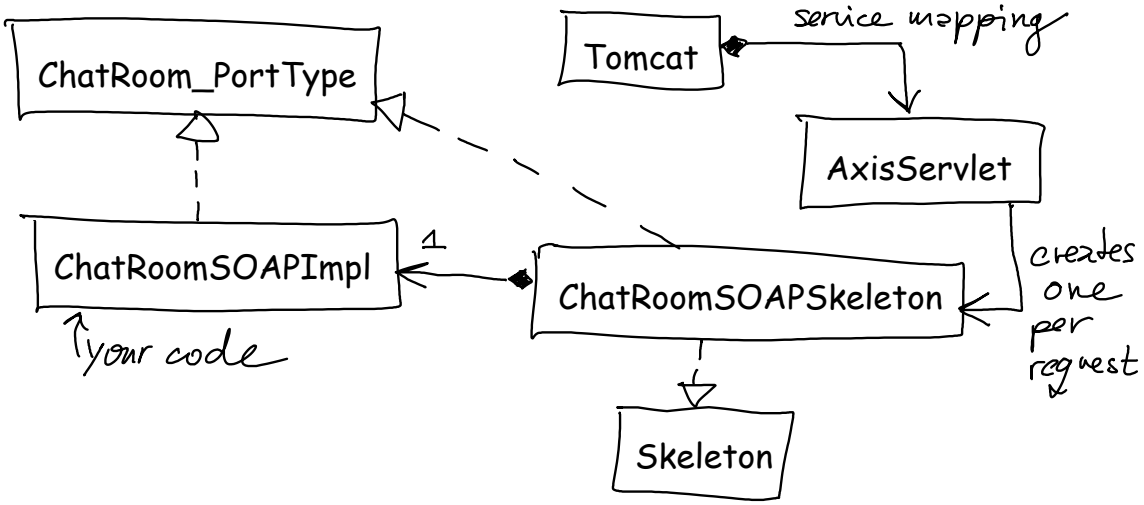

Figure 7.7: Files generated by Axis to provide a skeleton implementation of a Web service.

Using Apache Axis. To develop a DS node that interacts with a RPC WS, we usually generate a proxy as shown in Figure 7.2. For example, the Apache Axis framework includes a compiler `WSDL2Java` that takes a WSDL specification and produces a number of Java files that include a proxy class for the service (see Figure 7.6 for an overview of what is generated). Eclipse makes use of the Axis libraries to generate these files.

One normally creates the proxy object using the default constructor of the generated `...Proxy` class. Since WSDL contains the actual URL where the service's port is located, there is enough information in the generated classes to establish a connection. Nevertheless, quite often there are multiple services with equal or compatible interfaces. In such cases one can use the same proxy class to create proxies for any of these (almost) equal services using a constructor that takes the service's port URL as a parameter.

Notification listeners. One of the most common scenarios where there are multiple services with the same port type and binding is when these WS services are listening to notifications from the same source WS client. The server that notifies the listeners only needs to know their URLs and which port type and binding they all use. If using

Axis, the notifier creates a proxy object for each subscriber, passing the subscriber's URL to the proxy constructor. Notice that with notification the logical roles of client and server are reversed — a listener is a client of a notification service but technically a listener is a WS server waiting for notification "requests".

7.3.1 Developing new services with Apache Axis

In general, there are two ways to make a Web Service:

1. **code-first** or **bottom-up**: Expose an existing program unit containing some operations (methods, procedures, functions or whatever the PL supports) as a Web service.
2. **WSDL-first** or **top-down**: Agree a WSDL specification according to the requirements of potential clients and generate a dummy *skeleton* server in your PL. Then complete the logic of the server into the skeleton or, better, connect the skeleton to independently implemented logic.

Both methods are usually well supported by developer tools and largely automated. The second option is actually quite similar to the way one would develop a WS client.

There are arguments for and against both approaches:

- Code-first approach with automated generation often results in a very untidy WSDL specification. Nevertheless, it is the fastest way to get a Web service out there, especially when converting legacy software to a Web service.
- With the WSDL-first approach, one has full control over the specification and gives nicer WSDL, revealing less about the implementation. This way the whole service is likely to be better designed because WSDL allows one to focus on the client-server interaction protocol, which is the most important aspect of a Web service, without being distracted by aspects of the implementation.

  With the WSDL-first approach, it is harder to evolve the interface of an existing service. One has to often re-generate the skeleton from scratch with every change (having saved the old implementation portion of the skeleton) and then refactor the old implementation back into the new skeleton. The refactoring is usually not very complicated, eg adding a new parameter, changing parameter type etc but still it can be a nuisance.

  The fact that it is harder to change the interface with WSDL-first approach is a disadvantage in the early stages of development when the service is not yet public. Nevertheless, once the service is public, this property is an advantage — it helps the service maintainers to keep the interface stable even as the implementation is being improved.

Figure 7.7 shows an overview of the classes generated by Axis from a given WSDL to serve as a basis of an implementation. The only class that should be further edited in order to put some logic into the service is the class whose name ends with `Impl`. Nevertheless, unlike Jersey `@Singleton` resource classes, there are no persistent instances of this class — it is instantiated with each request. Persistent data, such as the indices of chatroom topics and chat messages, has to be stored in a **`static`** field of this class.