Distributed Systems
Introduction to Apache Avro

Tom De Schepper - University of Antwerp
(tom.deschepper@uantwerpen.be - M.G.215)

2016 – 2017

1 Introduction

This lab session serves as an introduction to Apache Avro[1]. Avro is a remote procedure call (RPC) and data serialization framework for language-neutral inter-process communication. It serves a similar purpose to middleware technologies such as CORBA and Java RMI. However, it does not suffer the drawbacks and design faults of these older technologies. Moreover, Avro does not require static code generation, based on an Interface Definition Language (IDL) schema, but rather employs dynamic schemas written in JavaScript Object Notation (JSON)[2]. Avro supports a wide range of programming languages, such as Java, Scala, C#, C, C++, Python and Ruby. In this lab, we will use it in conjunction with Java.

2 Required libraries

We will be using the latest stable release of Apache Avro, version 1.7.7. For Java, it comes as a pre-compiled JAR file that needs to be included in the classpath of your project. It additionally relies on the Jackson JSON library[3], and the SLF4J logging framework. Concretely, the following JAR files are needed:

- avro-1.7.7.jar:
  `http://apache.cu.be/avro/avro-1.7.7/java/avro-1.7.7.jar`
- avro-tools-1.7.7.jar:
  `http://apache.cu.be/avro/avro-1.7.7/java/avro-tools-1.7.7.jar`
- avro-ipc-1.7.7.jar:
  `http://apache.belnet.be/avro/avro-1.7.7/java/avro-ipc-1.7.7.jar`
- jackson-core-asl-1.9.13.jar:
  `http://repo1.maven.org/maven2/org/codehaus/jackson/jackson-core-asl/1.9.13/jackson-core-asl-1.9.13.jar`
- jackson-mapper-asl-1.9.13.jar:
  `http://repo1.maven.org/maven2/org/codehaus/jackson/jackson-mapper-asl/1.9.13/jackson-mapper-asl-1.9.13.jar`

[1] `http://avro.apache.org/`
[2] `http://www.json.org/`
[3] `http://jackson.codehaus.org/`

- slf4j-api-1.7.7.jar:
  `http://www.slf4j.org/dist/slf4j-1.7.7.tar.gz`
- slf4j-simple-1.7.7.jar:
  `http://www.slf4j.org/dist/slf4j-1.7.7.tar.gz`

The avro-tools library is used for code generation, the other jar files need to be included in your project's classpath (see below).

3 A simple Hello World example

This section will run the reader through the creation of a simple Avro-based client-server application. The server exposes a single procedure `sayHello`, which takes as argument the client's name and returns a textual greeting.

3.1 Including libraries in Eclipse projects

Start by creating a new Eclipse Java project, titled AvroHelloWorld. Next, we need to import the required libraries into the project classpath, so we can use them in our code. First, create a new directory, called "lib" inside your newly created project folder structure and copy avro-1.7.7.jar, avro-tools-1.7.7.jar, avro-ipc-1.7.7.jar, jackson-core-asl-1.9.13.jar, jackson-mapper-asl-1.9.13.jar, slf4j-api-1.7.7.jar and slf4j-simple-1.7.7.jar into it. You may need to press F5 in Eclipse to refresh the project structure and show the libraries. Next, select "Project → Properties", click "Java Build Path" in the left hand menu of the Properties window, and go to the "Libraries" tab. Subsequently, press the "Add JARs..." button and select all the libraries except for the avro-tools-1.7.7.jar. The Properties window should now look similar to what is shown in Figure 1. Finish by clicking the "Ok" button.

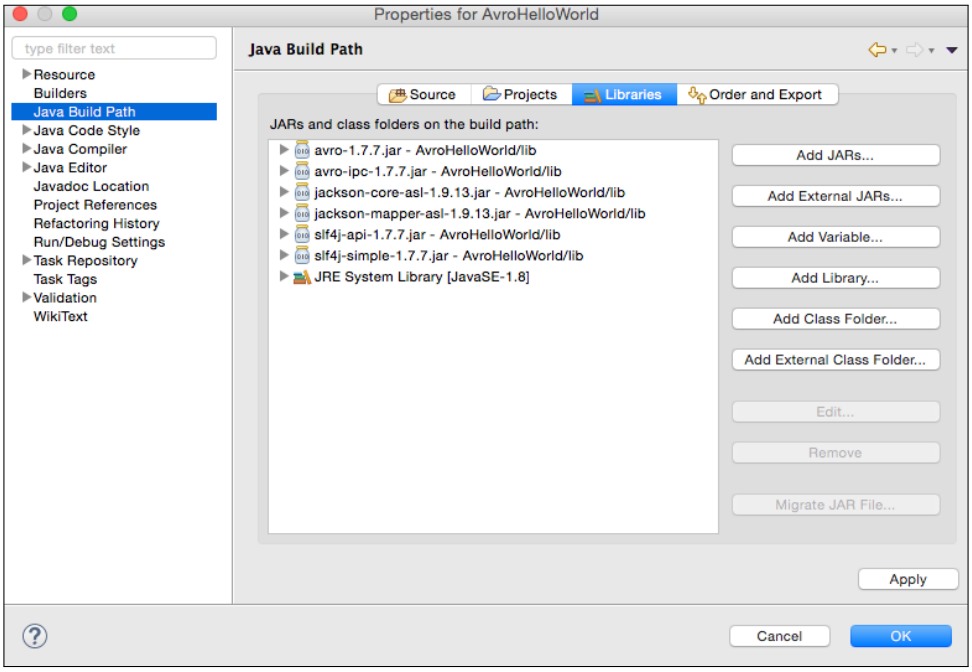

Figure 1: The Project Properties window after adding the Avro, Jackson, and SLF4J libraries to the classpath

3.2 Schema definition

The schema defines the contract between communicating processes. Avro schema files can define data types as well as protocols. A protocol consists of one or more messages, which are the equivalent of methods in Java. Avro supports schema definitions in JSON as well as its own proprietary IDL format. As the Avro IDL format is still in an experimental phase, we will use JSON.

Our Hello World application consists of a single protocol, which should define the equivalent of the following Java method:

```
public String sayHello(String name)
```

Using JSON syntax, this is translated into the following protocol schema:

```
{ "namespace": "avro.hello.proto",
  "protocol": "Hello",

  "messages": {
    "sayHello": {
      "request": [{ "name": "username", "type": "string" }],
      "response": "string"
    }
  }
}
```

A protocol declaration has several mandatory and optional attributes. The attribute "namespace" is optional and qualifies the protocol name, similar to a package declaration of a Java class. The "protocol" attribute is required and defines the protocol's name. The "messages" attribute is a map whose keys are message names and whose values are message objects. Each message object defines at least a "request" (an list of named, typed parameter schemas) and "response" (a primitive/complex type name or `null`) attribute. For a complete list of possible attributes, see the Avro specification[4].

Create a new empty file in the src folder of your Eclipse project (File → New → File), name it "hello.avpr" and copy the above protocol schema into it. Note that Eclipse is sometimes unable to open '.avpr' files with the default editor. You can use the text editor instead (Right click on the file → Open With → Text Editor).

As stated above, this schema can now be executed and interpreted dynamically. However, Avro also supports the traditional method of statically compiling the schema. As the latter approach results in cleaner application code (at the expense of less runtime flexibility) we will use that for now. To compile the schema into Java code, open a terminal window in the "lib" directory and execute the following command:

```
java -jar avro-tools-1.7.7.jar compile protocol ../src/hello.avpr ../src/
```

As a result, the "src" directory of your project should now contain the file avro/hello/proto/Hello.java.

3.3 Server implementation

The server implementation is straightforward. It implements the `Hello` interface we generated above and provides an implementation of the `sayHello` method. Our implementation will greet the user and tell it how many users have called the method before. Moreover, we provide a main method that starts the server so it is ready to receive remote procedure calls. The server is implemented as follows:

[4] `http://avro.apache.org/docs/1.7.7/spec.html#Protocol+Declaration`

```
package avro.hello.server;

import java.io.IOException;
import java.net.InetSocketAddress;
import org.apache.avro.AvroRemoteException;
import org.apache.avro.ipc.SaslSocketServer;
import org.apache.avro.ipc.Server;
import org.apache.avro.ipc.specific.SpecificResponder;
import avro.hello.proto.Hello;

public class HelloServer implements Hello {
  private int id = 0;

  @Override
  public CharSequence sayHello(CharSequence username) throws AvroRemoteException
      {
    System.out.println("Client connected: " + username + " (number: " + id +
        ")");
    return "Hello " + username + ", you are user number " + id++;
  }

  public static void main(String[] args) {
    Server server = null;
    try {
      server = new SaslSocketServer(new SpecificResponder(Hello.class,
          new HelloServer()), new InetSocketAddress(6789));
    } catch (IOException e) {
      System.err.println("[error] Failed to start server");
      e.printStackTrace(System.err);
      System.exit(1);
    }
    server.start();
    try {
      server.join();
    } catch (InterruptedException e) { }
  }
}
```

The `HelloServer` class implements the Avro `Hello` remote interface and its method `sayHello()`. The method simply returns a greeting to the user, that includes its username and an ID variable (which is incremented by 1 every time the method is called). The main method creates a new Avro `Server` object. The `Server` is responsible for providing clients access to the remote objects and offering RPC functionality. Avro provides different server implementations, of which we used the `SaslSocketServer`. It takes two arguments, a `Responder` and an address to which to bind. For generated interfaces, the `SpecificResponder` has to be used, which takes as arguments the generated interface's `Class` object and an instantiation of the class that implements the interface. The responder binds the interface name to its implementation, allowing clients to retrieve a reference to the remote object and call its methods. Subsequently, the server should start listening to incoming client requests, by calling the `start()` method on the server object. Finally, the `SaslSocketServer` implementation will stop if the thread in which it was started finishes execution. To prevent this, let the current thread wait as long as the server is running, by calling `server.join()`.

Create a new class file named `avro.hello.server.HelloServer` in your Eclipse project and copy the above code into it.

3.4 Client implementation

The client implementation is very straightforward and contains only a main method:

```
package avro.hello.client;

import java.io.IOException;
import java.net.InetSocketAddress;
import org.apache.avro.ipc.SaslSocketTransceiver;
import org.apache.avro.ipc.Transceiver;
import org.apache.avro.ipc.specific.SpecificRequestor;
import avro.hello.proto.Hello;

public class HelloClient {
  public static void main(String[] args) {
    try {
      Transceiver client = new SaslSocketTransceiver(new
          InetSocketAddress(6789));
      Hello proxy = (Hello) SpecificRequestor.getClient(Hello.class, client);
      CharSequence response = proxy.sayHello("Bob");
      System.out.println(response);
      client.close();
    } catch (IOException e) {
      System.err.println("Error connecting to server...");
      e.printStackTrace(System.err);
      System.exit(1);
    }
  }
}
```

The main method performs four tasks. First, it creates a new `Transceiver` object, which is used to connect to the server. This transceiver object needs to be of the same type as the server, in our case `SaslSocketTransceiver`. As argument, it takes an address to connect to. Second, the client retrieves a proxy object, to communicate to the remote implementation of the `Hello` interface. Third, it calls the `sayHello()` method of the proxy object, and prints out the server's response. Finally, it closes the connection to the server, by calling the `close()` method on the transceiver and shuts down. Note that communication with the server may fail, throwing an `IOException`. The client catches this exception and notifies the user in case it cannot connect to the server.

Create a new class file named `avro.hello.client.HelloClient` in your Eclipse project and copy the above code into it.

3.5 Running your application

To run the application, first open `HelloServer` and press the "Run" button in the toolbar. Subsequently, open `HelloClient` and run it as well. Try running several instances of the client and see what happens.

3.6 Types of communication

The Avro RPC protocol is essentially client-driven (i.e., pull-based communication), the client invokes a method on the server, which executes it and sends back the resulting return value. Avro supports two modes of communication:

- **Synchronous**: The client thread invoking the method will block until the server sends back the result. This is the mode employed in the hello world example described above.
- **Asynchronous**: The client thread will continue execution while the server executes the method. To use this mode, the client should request a proxy of the `Callback` subclass

(e.g., `Hello.Callback` in the example above). The client implementation of the hello world example would look as follows:

```
Hello.Callback proxy = SpecificRequestor.getClient(Hello.Callback.class,
    client);
CallFuture<CharSequence> future = new CallFuture<CharSequence>();
proxy.sayHello("Bob", future);
System.out.println(future.get());
```

The `CallFuture` object provided to the `sayHello()` method, will contain the method call's return value as soon as it is sent by the server. The `sayHello()` call returns immediately and the client can subsequently perform other tasks. Only when the `get()` method of the `CallFuture` object is called, the client thread will block until the result arrives. Note that the server should **not** implement the `Callback` version of the interface.

The synchronous and asynchronous modes of communication are discussed in more detail in the theoretical lectures of this course.

Neither of these modes allow the server to invoke methods on the client (i.e., push-based communication). In many applications, the server will however need to send clients unsolicited information (e.g., messages sent by other clients). A simple mechanism to emulate this behaviour is client-based polling. Instead of the server directly initiating communication, the client asks the server if new information is available at specific intervals. For the application to be responsive, this operation needs to be performed frequently, generally many times per second. This approach clearly introduces a lot of extra network traffic.

As an alternative, clients can start their own server, making available an Avro protocol object that can be invoked by the server. Although this approach consumes some more computing resources on the client side, it is more scalable and efficient in terms of network resources and improves responsiveness of the application. As such, this will be the approach that should be used in the project. Note that if you want to start multiple clients (on the same machine), their `SaslSocketServer` should bind to a different port.

4 Further reading

Below is a list of additional documentation to familiarize yourself with Avro:

- Avro 1.7.7 Documentation: `http://avro.apache.org/docs/1.7.7/index.html`
- Avro 1.7.7 Getting Started (Java): `http://avro.apache.org/docs/1.7.7/gettingstartedjava.html`
- Avro 1.7.7 Schema Specification: `http://avro.apache.org/docs/1.7.7/spec.html`
- Avro RPC Quick Start: `https://github.com/phunt/avro-rpc-quickstart`
- Avro 1.7.7 Java API: `http://avro.apache.org/docs/current/api/java/index.html`