MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DISTRIBUTED SYSTEMS

Assignment 1

Request-Reply Communication Paradigm

A1.1: The Basics

Ioan Salomie
Marcel Antal
Teodor Petrican

Tudor Cioara
Claudia Daniela Pop

Ionut Anghel
Dorin Moldovan
Ciprian Stan

2017

Contents

1. Introduction

The continuous technological breakthroughs of the recent decades in computer hardware, software and networking led to the growth of distributed applications over the Internet. Nowadays, most of the applications communicate over the Internet in order to access remote resources. The Internet has become a mesh of interactive applications and services over the physical stack of interconnected resources.

There are **two main actors** on the Internet, those that make a **request** and those that respond to the **request**. These actors can be easily mapped to **the two-tier client-server software architecture**: Web Clients (or simply **clients**) make a request, and Web Servers (or simply **servers**) that process the request and send a response to the client.

There are several communication paradigms and classifications, but one of the basic methods used by computers to communicate to each other is the **request-response** (or request-reply) mechanism, a message exchange pattern especially common in **client-server** architectures. Basically, the client sends a request message to the server that receives the request message, processes it and returns a response message. It is a **synchronous** communication because it keeps the connection open until the **response** is **delivered** or the **timeout period expires**.

Because the client and the server applications communicate over the Internet, they need a **common format for their messages** (or a **protocol**) that is **mutually accepted and known** by the parts involved in the communication process. Furthermore, several clients can communicate with the same server, thus they all need to know the messaging format. Last but not least, this messaging format must be general enough to cover most use cases for complex clients and simple enough to be used by clients that make only simple requests.

2. Problem Requirements

Suppose we are requested to create a distributed application with the following requirements:

- A central database is located on a server.
- The database stores a table with students.
- The teachers (Remote Clients) must access the database in order to:
    - add student information
    - retrieve students by their IDs
- The information retrieved form the central database is displayed for the users of the client application on a user-friendly GUI.

3. Problem Analysis

The problem can be decomposed into the following subsystems:

- Communication protocol
- The server application:
  - Database
  - Data access layer over the database
  - Communication layer over the network
- The client application:
  - Communication layer over the network
  - The user interface

4. Designing a Solution

We need to create a distributed application over the network. We choose a **client-server** software architecture.

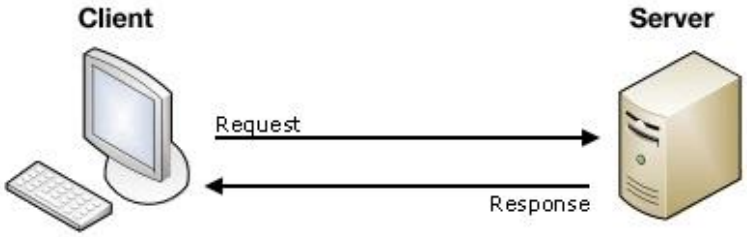

Figure 1. Client-Server software architecture

For the transport layer of the application we use **sockets** [1] that assure a two-way communication between the client and the server, thus allowing us to implement a synchronous request-reply communication method.

4.1. Defining the Message Structure

Because we use sockets for communication, the messages are sent as a stream of bytes. We intend sending strings as a stream of bytes through the sockets. Thus, all our messages will be encoded as strings.

We analyze the operations that we need to perform:

- OP1: Add a new student:
  - Parameters: Student.lastname, Student.firstname, Student.mail
  - Return: success followed by the new student's id or fail
- OP2: Return a student by its ID
  - Parameters: Student.ID
  - Return: Student

Each operation sends several pieces of information to the server, as strings. We choose to concatenate these strings and separate the information from the message by the following tokens: _ and #.

For executing each operation, two steps are involved: the **client request** and the **server response**:

4.2. Client Request

We determine that the client needs to send to the server the following information:

- OP1: send data in order to store an entity with all its fields in the database (e.g. student)
- OP2: request a resource from the server by specifying a resource identifier

In order to be able to perform these operations, the message that is passed from the client to the server needs to contain the following information:

- What kind of operation we perform (send data to the server –POST method ; request resource from the server –GET method)
- Which is the entity upon which the operation will be executed (the URI or URL, in our case the DAO class that handles students)
- Encode the data that needs to be passed to the server – method parameters already described for OP1 and OP2. These parameters will be converted to strings and concatenated with # used as separator.

By putting all this information in a String, we obtain a message of the following form:

Method_URL_messageBody

e.g. *OP1: POST_student_1#0George#Popescu#mail@mail.com*
OP2: GET_student_1

NOTES:

- Even if both operations can be performed using the same message structure, for the GET operation we will adopt another encoding, because this operation has another semantic meaning. When requesting a resource from the server, the client knows that resource location (in this case the student located in the DB at a given id), thus we will encode this information in the URL:
  - *OP2: GET_student?id=1_*
- For a GET method the body of the message send will be empty. That is because the URL (the name of the entity) together with the integer (the database identifier of the entity) forms the resource identifier. i.e. it contains all the information needed by the server to identify the resource (student) and return it to the client.
- For a POST method, the URL will specify the entity that will be sent to the server. The actual data will be encoded in the body of the message. These encoding will contain all the fields of the entity in the order that they appear in the class.

4.3. Server Response

We determine that the server needs to answer to the client request by specifying the following:

- OP1: return a code that represents the status of the operation. (e.g 200 – the operation was successful).
- OP2: return the resource requested by the client as a string encoded with the same rule as the request. In case of an error, return the code corresponding to the encountered error (e.g. 404 if the resource was not found).

Based on this information we determine the following response message structure:

StatusCode_messageBody

The message_body contains the returned values as strings separated by #.

e.g. *OP1: 200_*

OP2: 200_1#George#20#Cluj#Romania

We define the following status codes for our operations:

- 200 – the operation was successful
- 400 – bad request
- 404 – if the resource was not found
- 405 – operation not allowed

5. Implementing the Solution

The solution is implemented in 3 different modules:

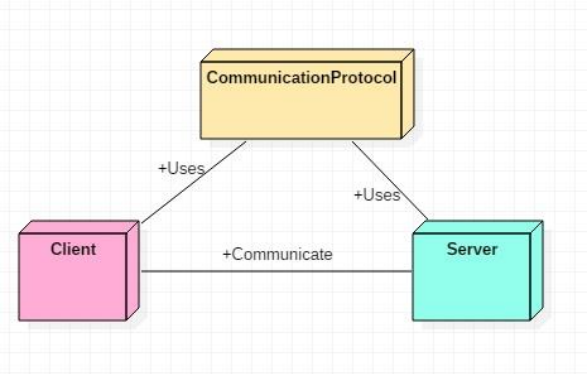

Figure 2. Conceptual architecture

Each module has the following architecture:

- **The Client Application**

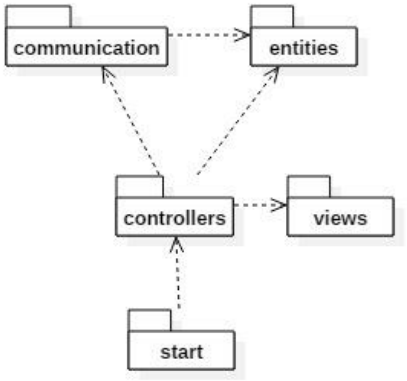

Figure 3. Client Package Diagram

- Communication - package that contains the classes responsible for the communication
- Controllers - package that contains the controller classes
- Entities - package that contains the entity classes
- Start - package that contains the class that starts the application
- Views - package that contains the UI class

- **The Server Application**

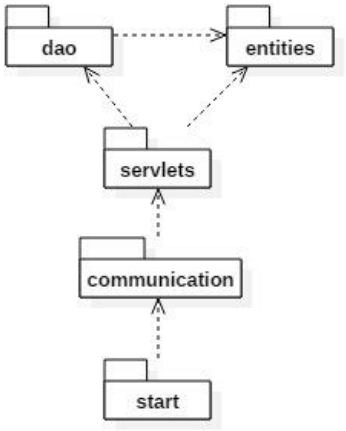

Figure 4. Client Package Diagram

- Communication - package that contains the classes responsible to the communication
- DAO - package that contains the classes responsible to the database access
- Entities - package that contains the entity classes
- Servlets - package that contains the classes that extend the *AbstractServlet*
- Start - package that contains the class that starts the application

- **The Communication Protocol** – library that contains the protocol definition

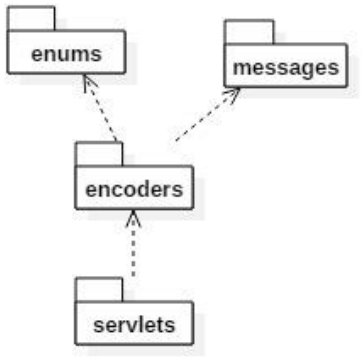

Figure 5. Communication Protocol Package Diagram

- Encoders - package that contains the classes responsible with the serialization and deserialization of objects
- Enums – package that contains enumerations
- Messages – package that contains the request and response message classes
- Servlets - package that contains an abstract class with the definition of the servlet

In the next sections we will present the functionality of the client and server applications by means of sequence diagrams.

5.1. The Client Application

The client makes a request by pressing a button on the GUI, the *CatalogView* class. When pressing the button, the action listener from the controller class, *CatalogController*, is called. There are two buttons on the UI, each of them corresponding to one operation specified and has the corresponding listeners (OP1- *PostActionListener* and OP2- *GetActionListener*).

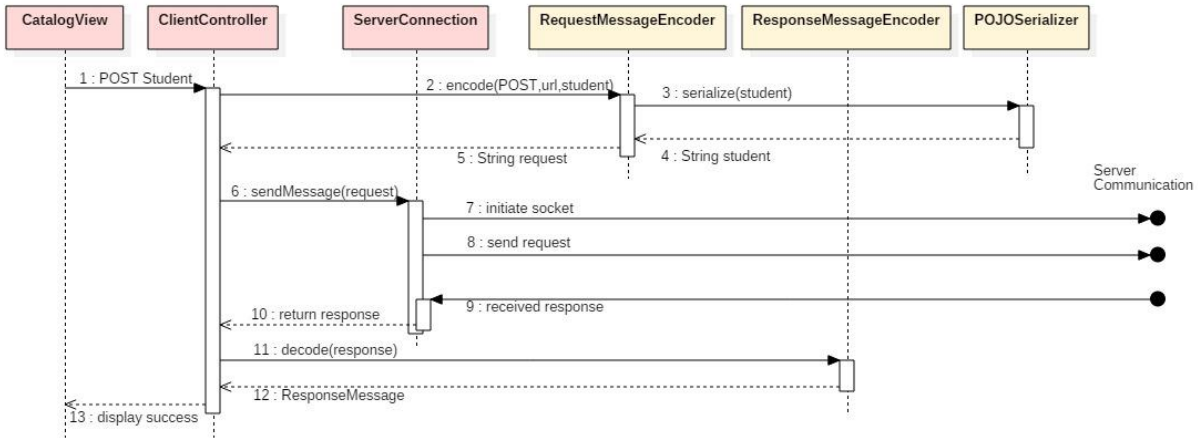

Figure 6. Sequence Diagram- client performing the POST operation

We present the steps involved in performing the OP1 operation (POST a student). The action is similar for the OP2 operation (GET operation). In order to make a request and display the response, the client application performs the following steps:

1. When a button is pressed on the *ClientView*, it triggers the *actionPerfomed* method from the corresponding action listener located in the *ClientController* class. It takes the information from the *ClientView* and creates a *Student* Object.
2. The student is encoded with the convention described in section 4.2 by calling the *encode* method from the class *RequestMessageEncode*

```
String messageEncoded = RequestMessageEncoder.encode(ProtocolMethod.POST, "student", student);
```

The *RequestMessageEncoder* creates a message with the format described in section 4.2. by concatenating the method, the URL and the serialization of the object sent as parameter.

```
String messageString = method + "_" + url + "_";

    if (o != null) {
       messageString+= POJOSerializer.serialize(o);
    }
```

3. In order to serialize an object, we adopt a method that concatenates its fields and adds a special separator character (# in our case) between them. This is the *serialize* method from the *POJOSerializer* class. It is important to notice that this method reads all the fields of an object and its corresponding values using **reflection techniques**.
4. The serialized student is returned and added to the message.
5. Finally, the message is encoded and saved in the *messageEncoded* string in the *ClientController*.
6. Having the message encoded as a stream of bytes, the client application will send it to the server. Because it is a synchronous communication, the client waits until the server sends back the response.

```
String response = serverConnection.sendRequest(messageEncoded);
```

7. A socket connection to the server is opened (method *sendRequest* from *ServerConnection* class)
8. The message is sent through the socket(method *sendRequest* from *ServerConnection* class)
9. The client waits until it receives the message(method *sendRequest* from *ServerConnection* class)
10. The response is returned to the client application and stored in the *response* String.
11. The response message is de-serialized by the *ResponseMessageEncoder* class and a *ResponseMessage* object is created. This class also uses the encoding conventions presented in section 4.2.

```
ResponseMessage decodedResponse = ResponseMessageEncoder.decode(response);
```

12. Because it is a POST action, it will return only the status code of the operation. However, we also return the ID of the student because it is auto-generated and would not be known otherwise.
13. The information about the POST operation is displayed on the UI.

5.2. The Server Application

The server responds to each client request. It has a thread that runs and listens for incoming connections from clients, in the *Server* class. Each time a new client sends a request, it establishes a connection to the client, creates a separate thread for that client (Session class), receives the message, processes it and returns a reply to the client.

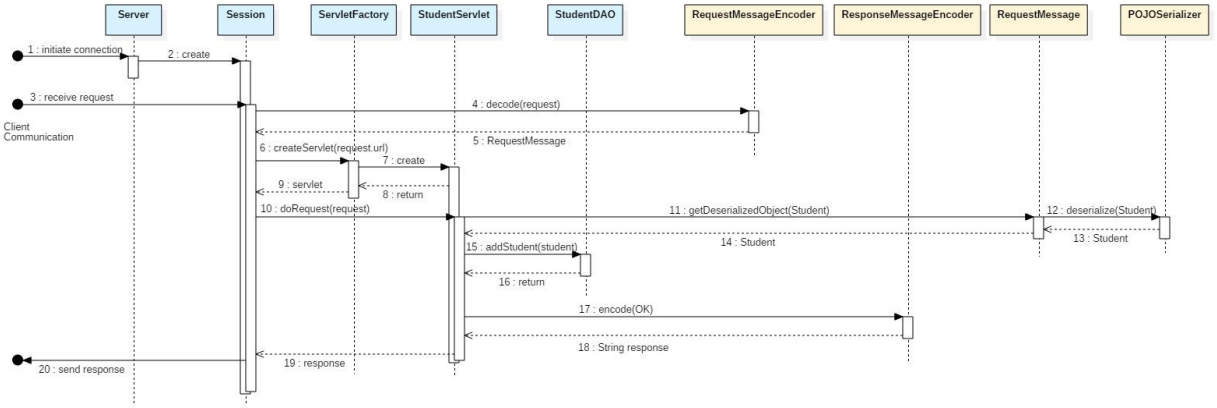

Figure 7. Sequence Diagram- server performing the POST operation

We present the processing on the server side when a POST message is received from a client:

1. The client initiates a connection. The thread from the Server class listens and accepts the connection.

```
Socket clientSocket;
clientSocket = serverSocket.accept();
```

2. Then, a Session is created. This will receive the actual message, process the message and return the reply.

```
Session cThread = new Session(clientSocket, this);
cThread.start();
```

3. In the Session class, the message is received from the client. It is important to notice that this class is also a thread, because the server can handle multiple client connections simultaneously. After the response is sent back to the clients, this thread ends its execution.

```
messageReceived = (String) inFromClient.readObject();
```

4. The message must be decoded. This is performed by the RequestMessageEncoder class.

```
RequestMessage decodedRequest = RequestMessageEncoder.decode(messageReceived);
```

5. The *RequestMessage* is returned, containing the method (POST or GET), the URL of the resource, the URL parameters sent in a <key,value> map, and the body in string format.
6. Based on the request URL the Session calls the *ServletFactory* to create the Servlet asked by the client.

```
AbstractServlet abstractServlet = ServletFactory.createServlet(decodedRequest.getUrl());
```

In our implementation we consider that each URL is mapped to a resource named Servlet that has implemented methods for each request type (GET or POST). If there is no Servlet for an URL or the method is not implemented, then an exception is thrown.

7. The *ServletFactory* creates the Servlet mapped to the URL or throws an exception if no servlet is found matching the given URL. It uses reflection to create an instance of an object given its class name. The naming convention assumes that the servlets are located in the package *servlets* and each servlet has the name in the format *UrlServlet* (e.g. for the url = "Student" the class name will be *servlets.StudentServlet).* The object is instantiated using reflection techniques.

```
public static AbstractServlet createServlet(String url) throws ClassNotFoundException{
    String className = "servlets.";
    className += url.replace(url.charAt(0),Character.toUpperCase(url.charAt(0)));
    className += "Servlet";

    AbstractServlet abstractServlet = null;

    Class<?> clazz;
    try {
       clazz = Class.forName(className);

       if (clazz == null) {
          return null;
       }

       Constructor<?> ctor = clazz.getConstructor();
       abstractServlet = (AbstractServlet)ctor.newInstance();
    } catch (InvocationTargetException | NoSuchMethodException | IllegalAccessException |
InstantiationException | ClassNotFoundException e) {
       LOGGER.error("", e);
    }
```

```
    return abstractServlet;
}
```

8. The *StudentServlet* is created
9. The *StudentServlet* is returned to the *Session*
10. Each Servlet class has 3 methods:

```
1. public String doRequest(RequestMessage message) {

   try {
      switch (message.getMethod()) {
         case GET:
            return doGet(message);
         case POST:
            return doPost(message);
         default:
            return ResponseMessageEncoder.encode(StatusCode.BAD_REQUEST,null);
      }
   } catch (UnsupportedOperationException e) {
      LOGGER.error("", e);
      return ResponseMessageEncoder.encode(StatusCode.NOT_ALLOWED,null);
   }
}


2. public abstract String doPost(RequestMessage message);
3. public abstract String doGet(RequestMessage message);
```

The methods that will be called when a GET or POST message is received will be overridden by subclasses in order to process the particular messages.
In this case, the *doRequest* method will further delegate the request to the *doPost* method from the *StudentServlet* class.

11. The object is de-serialized by the *RequestMessage* class.
12. The actual message is decoded by reflection techniques by the *POJOSerializer* class.
13. A *Student* object is returned
14. A *Student* object is returned
15. The *StudentServlet* class calls the corresponding *addStudent* method from the *StudentDAO* class. This class is implemented using **Hibernate [2].**

```
studentDao.addStudent(student);
```

16. The inserted entity is returned
17. If the operation succeeded, a return message with the status code OK and the ID of the student is created by the *ResponseMessageEncoder* class

```
response = ResponseMessageEncoder.encode(StatusCode.OK,String.valueOf(student.getId()));
```

18. The string containing the encoded message is returned
19. The response is returned to the servlet
20. The response is send back to the client through the sockets. The session closes.

```
sendMessageToClient(response);
closeAll();
```

6. Running the Solution

1. Setup GIT and download the project from https://bitbucket.org/utcn_dsrl/ds.handson.addignment-1/
    - Create an empty local folder in the workspace on your computer
    - Right-click in the folder and select **Git Bash**
    - Write commands:
        - git init
        - git remote add origin https://bitbucket.org/utcn_dsrl/ds.handson.addignment-1.git
        - git pull origin master
2. Import the DB script assignment-one-db.sql in MySQL. You can use MySQL WorkBench, go to Server -> Data Import -> Import from self-contained file->browse for file->click Start Import.
3. Import the project into Eclipse: FILE-> Import->Maven-> Existing Maven Projects-> Browse for project in the folder created at step 1
4. Modify the hibernate.cfg.xml file from the *server* module (main/src/resources/):
    - Change the hibernate connection url to localhost (the IP from line 16 to *localhost)*
    - Set the username and password from your local MySQL server (line 19 and 22)
5. Run the project:
    - Run the ServerStart class from the server module, package start
    - Run the ClientStart class from the client module, package start

7. Reinforcement Learning

1. Run and study the project and identify the following features:
    - Message encoding / decoding
    - Accessing resources on the server
    - Parameter sending techniques
2. Add a new method DELETE that sends to the server the ID of a student to be deleted. The ID will be sent in the message body. The request will return the student that has been deleted and the corresponding status code.

8. Bibliography

1. Java Sockets: https://docs.oracle.com/javase/tutorial/networking/sockets/
2. Hibernate: http://hibernate.org/orm/