Polytechnic University of Cartagena

School of Telecommunications Engineering

DISTRIBUTED SYSTEMS AND SERVICES

PRACTICE 4: PROGRAMMING DISTRIBUTED SYSTEMS (3)

MESSAGES INTERFACE AND SERIALIZATION

Teachers:

Antonio Guillén Pérez

Esteban Egea Lopez

INDEX

1. Goals.

- Describe alternatives for implementing application protocols.
- Application programming of binary protocols.

2. Message passing.

In previous labs we saw how to establish connections through Sockets between two processes. The connection establishment is only a first step for the exchange of messages between processes. A Socket manages a TCP connection between two processes. On that connection messages are sent. It is therefore necessary to define the interface between processes, ie which messages are sent between them and how to read those messages. This is precisely what makes an application level protocol: to define the messages that can be exchanged between two applications.

3. Application level protocols.

The main problem in defining the application level protocols is that the data handled by applications can be arbitrarily complex, unlike what happens in the lower layers. Imagine an application that manages a list of medical records, where each record contains very diverse patient information: personal information, medical test results, associated images. Sending this information to a server requires encode it in a suitable format to be transmitted efficiently over the network and the server to be able to rebuild destination. This functionality is what is usually called serialization.

Most application-level protocols are text-based. That is, applications are exchanged text messages. Strings are sent over the TCP connection. The task of the server, or the remote end, is to analyze the text and extract the information. The advantage of this approach is that it is very flexible, there is no rigidity in the interface: simply changing the text to change the type of message being sent.

Examples of application level protocols have text-based HTTP (Web), SMTP and IMAP (mail) or DNS among others.

The main problem is its inefficiency: the encoded data as characters generally use more bandwidth and require more processing time.

The alternative is to use binary formats. But in this case a careful definition of the types of messages and serialization format is needed.

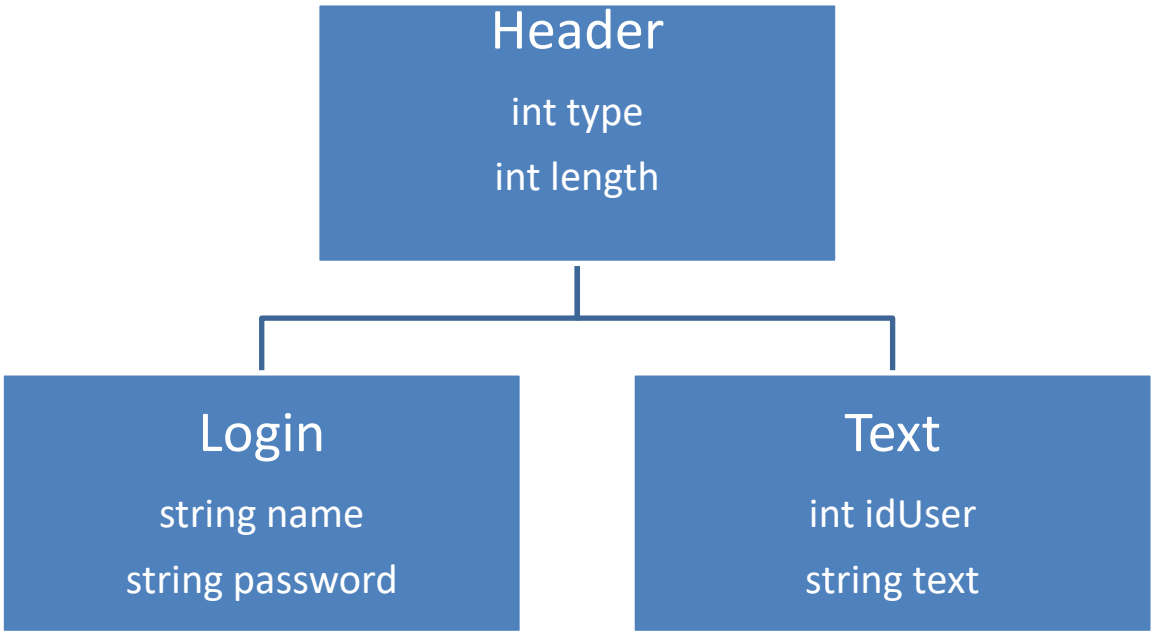
In this lab briefly we study these two issues, developing a binary interface for passing messages between two processes on a Socket.

4. Message Passing in Java.

Java provides an internal serialization mechanism that greatly simplifies application development. The main drawback is that both the client and the server must be implemented in Java, which is not usual.

When this is the case, we can just leave Java the serialization process and focus on the definition of the interface or protocol. A common approach is to define a common header for all messages to report the type of message being sent and the length thereof and other essential information. So that the server, after examining the header can process the rest of the message properly.

To implement this approach in Java, usually it is employed a class hierarchy like that of the figure:



A class Header with a type and a length field is declared. This header is of constant size. Then, for each message type (Login, Text, etc.) a corresponding class with the necessary information is declared.

An advantage of this approach is that is works whether Java serialization is used or not, as we shall see in the next section.

The next step is simply to use the ObjectInputStream and ObjectOutputStream class to send / read messages on the socket, allowing Java to handle the serialization process. In this case, the server

need only use the type field to instantiate a class corresponding message, you do not really need the length field.

5. Pass arbitrary binary messages.

When the client or the server are not implemented in Java, you need to perform a binary serialization of objects to be exchanged. In this case, you can use the above class hierarchy, but you need to serialize the data before sending. Since over TCP only bytes are sent, we must encode the data in binary form to send and receive. This process is sometimes called pack / unpack. This process can be complex.

Fortunately, Java has the ByteBuffer class that allows adding primitives to a buffer of bytes.

For example, the putDouble (double c) to insert a double function in the array of bytes.

The idea is to convert a class to an array of bytes and write directly on the socket. To this end, we add the methods byte [] pack () and unpack (byte []) to the Header class. Each derived class will handle redeclare these methods, which will return a byte array with the data of the class or class reconstructed from an array of bytes.

The process is now as follows: the client will always send a header, as bytes, indicating the type and longitude of subsequent message; then it will write the array of bytes for the message. The server, after accepting a connection, reads in an array of bytes the size of the header, the header will be rebuilt and it will use the length field to read the following array of bytes, which is used to reconstruct the message.

6. Lab development

1. For this lab you will use the client and server developed in previous sessions. You will have to modify it accordingly. As a first step declare a hierarchy of messages as indicated in thefigure. In addition to the Header, use a message for a User, whose data are the name and an ID number and a class Text containing an ID and a string. These messages could be used to make the chat of the previous sessions for example. The first to register a new user and the other for sending text messages. In any case, the server will only display the data on screen.

- Define a *Header* class, with a constructor and two parameters: int type and int length, that are going to indicate the type of message that is going to be sent and the length of that message. This length is up to you, you can indicate the total size of the message, or you can indicate the total size of the message that comes after the *Header* (*login*, *text*, etc.).
- Define a *UserMessage* class, extending by inheritance to *Header*, with a constructor and two internal parameters: String name and int dni. The constructor is of free choice, you can have as input parameters only those of the *Header* case, and then indicate them explicitly, or have as parameters of the constructor all the necessary variables. This message, for example, could be used to register a new user.

- Do the same with a *TextMessage* class, which extends *Header*, constructor and parameters: int dni and String text. Indicating, for example, the message to be sent and the identifier of the user to whom the message should be sent.
- Additionally, create a *CloseMessage* class, extending *Header*, which only serves to close communication on the server.

2. Use Java serialization (ObjectOutputStream / ObjectInputStream) to send messages to the server.

- To do this, the server must have a *ServerTask* class that will be in charge of communicating with the clients, through threads, as we saw in the previous practice, with a *run()* method that will be called when a communication request arrives. This *run() method* will read from the socket through the *ObjectInputStream* the message received. After receiving it, it converts it into a Header type message, and reads the type of message received. After decoding the type of message received, it converts the message to the type of message received and prints out on the screen, all the fields included in the message.
- The client, just create the message, indicating all the attributes of the message to be sent, and pass it to the server through an *ObjectOutputStream.* It is recommended to create a *UserMessage* first, then a *TextMessage and* finally a *CloseMessage,* to test the whole process.

3. Now, implement a binary version. To do this, implement the pack / unpack methods of all the classes.

- All previously developed classes (*Header, UserMessage, TextMessage* and *CloseMessage*) must have a pack method and an unpack method. The idea of these methods is going to be to store all the variables in an array of bytes (*byte []),* and for this, we will help us from the *ByteBuffer* class, that provides methods to add primitives in a byte buffer, and then copy the *ByteBuffer in the byte* array.
- For each class, in the *pack* method, define a *ByteBuffer of the* size of the message to send, add the variables in a certain order (putInt, putDouble, etc), create the byte array of the same size, copy the *ByteBuffer* into the byte array. Note that, if you are packing a class that inherits from *Header,* it must also be packed and returned with the result of packing the message.
- For each class, in the *unpack* method, using the *ByteBuffer* class, we are obtaining the primitive data in the same order in which they were packed (*getInt*, *getDouble*, etc).

4. Modify the server and client to use the binary version and display the data on screen.

- To do this, on the client, when you create the messages you must *pack* them before sending them to the server through the *socket*.
- On the server, when it receives a connection, it will launch a task to communicate with the client. In this task, when a message arrives from the client, the first thing it will do is unpack the header and identify the type of message received. When it knows, it will transform the message to its type and unpack it, being able to print on screen all the variables of that message.

5. Implement the Cristian synchronization algorithm between client and server.

Bibliography

1. Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006

2. David Reilly, Java Network Programming and Distributed Computing, Addison-Wesley, 2002.

3. Al Williams, Java 2 network protocols black book, Coriolis Groups Books, 2001.