Polytech/INFO4
F. Boyer, O. Gruber

PROJET - Socket-based Distributed Applications

Organisation	Binome
Evaluation	Involvement and Code
Advised time	3-4h per binome
Tools	JDK >=1.7, IDE (eclipse or equivalent)
Return	Compressed source files, in a file named Name1_Name2_Socket.zip (or .tar)
Contact	Fabienne.Boyer@imag.dfr, Olivier.Gruber@imag.fr

This practical work aims at learning how to design and implement simple application servers on top of TCP/IP. Once done, essential design points, such as stream-based communication and lossless properties, should be fully understood.

Important point : to achieve this practical work, you should look at the slides on Sockets (TCP part). Explanations are not reminded in this document.

1 Baby step

Let's start with programming a very very basic client-server application, involving a *Hello* server over TCP/IP. The client-server interaction can be sketched as follows:

- Client side:
  - The client connects to the server.
  - The client sends a request containing its name.
  - The client prints the server response.
- Server side:
  - The server waits on a given port number (e.g., 9999) for connections from remote clients.
  - When receiving a client request, the server returns the string "Hello <clientName>".

Of course, the client and server parts refer to two distinct programs, started as independent processes. One simple way to achieve this is to define a *Client* class with its own *main* method, and a *Server* class with its own *main* method as well.

To check your application, you will start your server first, then start several clients and check that all behave as expected. Notice that despite your server and clients run on the same host, they are all running in their own separate address space (their own Java Runtime Environments in fact), forming a real distributed system.

Regarding the server implementation, do not forget that a typical application server goes into an infinite loop as follow (*listenSoc* referring to the *ServerSocket* instance created for receiving connection requests from clients):

```
<initializations (among others, create listenSoc from ServerSocket class) >
while (true) {
        Socket clientSoc = listenSoc.accept();
        if (clientSoc != null) process(clientSoc);
}
```

Regarding both your client and server implementations, we advise you to use the **readUTF()** and **writeUTF()** methods provided by DataInputStream/DataOutputStream Java classes to exchange strings using the classical UTF-8 encoding. Such methods manage by themselves the sending/receiving of the String length such as to receive the entire String that has been sent, so you do not have to manage it in your programs. This is a very important point that should be clear.

Of course we advise you to use readUTF() and writeUTF() but you can use other methods, such as read() and write(). With streams, you can also write bytes with a given granularity (e.g., write a line) and read them later with a different granularity (e.g., read byte per byte).

2 Basic file server

We now consider a more complex client-server application over TCP/IP: a file server allowing remote clients to download files. The client-server interaction can be sketched as follows:

- Client side:
  - The client connects to the server.
  - Sends a request which contains (at least) the name of the requested file.
  - Stores the received file locally on disk.
- Server side:
  - The server waits, on a given port number, for connections from remote clients.
  - When receiving a client request, the server returns the corresponding file to the client.

Follow these steps:

1. Define your file-transfer protocol over TCP/IP that includes the following:
   - Format of the request and format of the response.
   - Remember to handle errors, such as file not found.
2. Write the server program that takes as arguments its port number and the folder from which files to download can be found locally.
3. Write the client program that takes as arguments the server hostname and port and the path of the file to download.

   Advice: for both the client and the server, think of using Java classes such as ***FileInputStream/FileOutputStream*** *for manipulating files, and DataInputStream / DataOutputStream for reading and writing data of primitive types.*

4. Run the Server and Client programs on two different Java Runtime Environments.

3 Multi-threaded file server

The previous version of the server implements a purely sequential server, that is, the server is mono-threaded so it can only handle one request at a time. We will now implement a multi-threaded server, able to serve multiple requests in parallel.

3.1 Basic Multi-threaded Design

In this step, you will implement a server that creates one thread per client request. For each client request, the server creates a dedicated worker thread; each worker thread processes one client request and dies.

Notice that this design allows client requests to be processed in parallel. This means that different threads will possibly read and send the same file to clients. You have to ensure to ensure that your code is thread-safe regarding this aspect, so check that the libraries you use to read files are convenient.

3.2 Pool-based Multi-threaded Design

The previous design has an important drawback: thread creation is a time-consuming operation, meaning that the performances of the server may be degraded by the time spent in creating threads and garbaging thread objects.

A commonly used technique to address this issue is using a pool of (worker) threads. Worker threads are created when the server is launched. Each time a client sends a request to the server, the server delegates the processing of that request to worker threads, using a bounded shared buffer (typically, a *Producer/Consumer* buffer as the one you developed in the Concurrent Programming lecture during the first semester).

We ask you to implement a pool-based multithreaded server, using a fixed-size pool of threads. If designing such a server is not clear, go back to the lecture slides and you'll get the necessary information to properly implement such a server.

4 Considering large files

Is your implementation relevant for large files? There are two aspects to consider: performances and failures.

4.1 Performances

How did you read and send the file?

- Did you read the entire file in memory before sending it back to the client?
- Did you read the file one byte at a time or did you use a byte array?

Look again at the *InputStream* interface:

`abstract int`	**`read`**`()` Reads the next byte of data from the input stream.
`int`	**`read`**`(byte[] b)` Reads some number of bytes from the input stream and stores them into the buffer array `b`.
`int`	**`read`**`(byte[] b, int off, int len)` Reads up to `len` bytes of data from the input stream into an array of bytes.
`void`	

Make sure that your implementation uses a byte array to read the file by chunks of 512 bytes at a time. Also make sure that your program reads one chunk and sends it immediately to the socket channel. Explain why this approach provides better performances.

4.2 Failures [OPTIONAL]

Long downloads increase the risk of failures. Propose a solution based on splitting the downloads of large files into smaller downloads that are more likely to succeed. In case of failure, ensure that you restart from the last small download that failed.

Let's now try to make failures transparent client side. We will assume that when a server crashes, it is restarted rapidly. Adapt your Client program such that when a download is interrupted, the client tries to reconnect to the server during a given period. If it manages to reconnect, it continues the download. It does not manage to reconnect, it simply prints a dedicated error message.

5 Performance evaluation [OPTIONAL]

We ask you to compare the performances of the three designs for your server: mono-threaded, multi-threaded, pool-based multi-threaded. To this end, you will have to stress your server. For instance, you may create a dedicated client program that launch many threads requesting to download different files in parallel.

6 Lessons learned and open questions

We expect that you learned about the following key concerns:

- Stream-based communication pattern: do not forget to either send the length or a dedicated mark (e.g., empty line) to allow a receiver to determine when it has received all the data that was sent.

- FIFO loss-less communication: be aware that you did not manage any ordering or loss of bytes when transferring files, due to the use of TCP/IP. What about using UDP? If you have time, think of how you would implement file download over UDP.

- Multi-threaded server: multithreading is the way to exploit parallelism with the intent of providing a better throughput to clients. Using a pool-based server is more efficient than creating a thread per request, you have certainly observed this fact with your file server. However, pre-determining the number of thread to create may be a difficult task. Moreover, in some cases, we have to program servers that may face very variable loads, which increases the complexity of determining the pool size. What could you propose to address this issue?