Polytech'Grenoble

INFO4, année 2018-2019

F. Boyer, O. Gruber

TP2 - NIO-based Distributed Applications

Organisation	Binome
Evaluation	Code-based
Advised time	6-8h per binome
Tools	JDK ==1.8, Eclipse
Return	**Eclipse Projet**, named **LastName1.LastName2**. In compressed format, either zip or .tar (**nothing else**)
Contact	Fabienne.Boyer@univ-grenoble-alpes.fr Olivier.Gruber@univ-grenoble-alpes.fr

1 Baby steps

There are two baby steps. The first baby step is there to help you understand the general flow of a client/server interaction when using NIO. The first step is however overly simple and therefore needs improvements. Making such improvements is the purpose of the second and third baby steps.

1.1 First Baby Step

Look at the given code in the package *ricm.nio.babystep1*. The two classes NioClient and NioServer implements an overly simplified echo system, using Java NIO.

The client-server interaction can be sketched as follows:

- Client side:
  - The client connects to the server.
  - The client sends a small message (e.g., "Hello") to the server.
  - The client prints any message received and echoes it back to the server.
- Server side:
  - The server waits on a given port number for a connection from the client.
  - The server prints any message received from the client and echoes it back to the client.

Note that the NioClient and NioServer classes contain redundant code. This is a deliberate choice. It is to help your understanding, with each class being self-contained.

We ask that you do the following:

1. Read and understand the code in detail. In particular, pay particular attention to the inverted nature of the execution flow. The main thread is captured in the selector loop, waking up when something happens, an event that has been registered of interest (accept, connect, available bytes to read, room to write bytes). So the overall execution reacts to occurring events rather than following a given algorithm.

2. Single step the execution with the debugger, on both the client and server side, to understand the overall execution flow.

3. Make sure you understand why the given code does not ensure that messages are sent or read entirely. Ask questions if it is not clear for you.

4. We ask that you force the given code to fail to read or write full messages. The way to do so is to send larger and larger messages until you see it happen. To achieve this, make the client grow the received message before echoing it back to the server.

   WARNING: for this last item, you need to no longer print the message, just print the number of bytes read and in how many steps. If you keep printing longer and longer messages, **Eclipse will become unresponsive** and the console output will be all garbled. Make sure to remove the printing on both the client and server sides.

   WARNING: Make sure to stop the client from echoing messages when it takes more than 5 steps to read an entire message. Otherwise, your machine may become unresponsive as messages become extremely large.

1.2 Second Baby Step

We now want to ensure that clients and server receive full messages. As discussed in the lecture, this means implementing automata for reading or sending entire messages. You will design two classes:

1. A class Reader to ensure that a message is received in its entirety.

2. A class Writer to ensure that a message is sent in its entirety.

The designs of these two classes are straightforward. The design of the class Reader is to keep reading bytes until it has received a complete message. The design of the class Writer is to keep writing bytes until it has sent a complete message. To do so, you need to setup READ and WRITE interests. Think however about when these READ and WRITE interests should be setup or when they should be cleared.

You will therefore have one reader and one writer on either side. The server will have one reader object and one writer object to receive and to send messages to the client. The client will also have one reader object to receive messages and one writer object to send messages. **But the classes Reader and Writer are shared, they are the same for the client and server.**

We ask that you test your code, run again your server and client, checking that they send and receive full messages, even when growing messages. You will reuse the test code you developed earlier that showed the problem, growing messages before echoing them back.

- **Make sure that your automata are at work.**

- **Make sure that when you client stops echoing messages that the CPU load on your machine goes back to normal**, that is, a low CPU load. If not, check you NIO READ/WRITE interest management.

1.3 Third Baby Step

In the previous steps, the code is overly simple, the design allows one client to connect to one server, that is it. This is of course not realistic. A client may connect to several servers and a server may accept connections from multiple clients. The design is missing the concept of a communication channel, between one client and one server.

We ask you to design and code the corresponding class Channel. A channel allows to send and receive full messages, so it encapsulates a reader and a writer. Of course, the class Channel would be instantiated on both sides, representing both end points of the communication channel. In other words, you will instantiate the class Channel on the client side to represent the communication channel towards a server, from the client perspective. You will also instantiate the same class Channel on the server side to represent the communication channel towards a client, from the server perspective.

We ask that you test your code with multiple clients connecting to a server. All clients will send first messages of different lengths and then keep echoing them over and over, keeping the original size (not growing messages). This way, your server will have to accept and serve multiple clients concurrently.

Note: to simplify your testing, you may design a test client that establishes multiple connections to a server, keeping the number of processes down to two at first. One process for the server. One process for the client. But multiple channels between them.

2 The Real Thing

It is time to leave our baby steps behind and consider a real design for a message-oriented middleware allowing to establish communication channels between processes, with channels enabling to send and receive variable-size messages. The intent is to define an abstract interface for that message-oriented middleware, one that would be independent of any underlying technology used to implement it. Of course, we will use our knowledge of NIO to implement it in the end, but other technologies could also be used.

Think of it this way. In Java, you have the abstract concept of a stream, represented by two interfaces: InputStream and OutputStream. These interfaces may be implemented using different technologies, for different purposes. They may be file input and output streams, or they may be socket input and output streams. They may be even implemented using in-memory byte arrays (class ByteArrayInputStream and class ByteArrayOutputStream).

But whatever the technology, you have three distinct sets of classes, allowing to speak of three distinct parts: the interface, the interface implementation, and the application using that implementation through its interface. We will also have those three parts.

- Look at the interface we designed in the Eclipse project "ricm.channels". This is the interface of our message-oriented middleware. It has the concept of channels to send and receive messages. It has the concept of brokers to establish channels.

- Look at the Eclipse project "ricm.channels.samples". It provides an *echo* application and a *file-server* application, both using our interface from the Java package *ricm.channels*.

- You will create an Eclipse project "ricm.channels.nio" for the implementation using NIO.

  To help you understand the flow, we provided you with a local and quite simple implementation of the middleware, implementing our interface from the Java package *ricm.channels*.

To help you understand our message-oriented interface and the overall event-oriented execution flow, we provided you with a local implementation. The local implementation allows to run multiple applications within the same Java Runtime Environment. This is how our sample applications (echo and fileserver) can be run. Look at the class *LocalMain* in both applications (package *ricm.channels.echo* and package *ricm.channels.fileserver*).

You may look at the local implementation, but it is not provided for that. It is provided so that you can look at how an application uses our message-oriented middleware and to get you accustomed to an event-oriented execution flow. So go ahead, put breakpoints in the methods of our sample application classes (*EchoClient*, *EchoServer*), launch the execution, and follow the execution with the debugger. Then, do the same with the fileserver application, with the classes *FileServer* and *FileDownloader*.

2.1 First Task

Now the idea is that you will code your own implementation of our message-oriented middleware above NIO. You will do so in a new Eclipse Java project "*ricm.channels.nio*". That project will of course depend on the interfaces, that is, the Eclipse Java project "*ricm.channels*". You can do so through editing the *build path* in Eclipse.

2.2 Second Task

Then, you will be able to run our sample applications on your implementation, in a distributed setting rather than a local one, using one Java Runtime Environment per application. To achieve that, you will only need to write your own setup

classes, let's say class *NioMain*, to replace the local setup classes (classes *LocalMain*). Don't forget to update accordingly the build path in Eclipse for the Java project *ricm.channels.applications.*

2.3 Third Task

You will improve our sample file-server application; the problem to fix is the following:

> *Each file is loaded entirely in memory before it is sent as one single message.*

It is not hard to realize that this is not the best possible design:

- A design that requires a lot of memory, loading entire files in memory, files that may be large
- A design with high latency, while loading a large file, no other file download may be initiated or progress.

The solution is simple, you already implemented it with regular sockets. You need to load files by chunks, sending each chunk as a message. In other words, sending a file will become a multi-step automaton. Each step reads a chunk and sends it. You will read the next chunk once you know the previous one has been sent.

You may extend our message-oriented middleware interface, if needed. But you need to explain and justify any extension you make.

3

1