**Polytech/Info4**

**F. Boyer, O. Gruber**

# PROJET - NIO-based Distributed Applications

| | |
|---|---|
| Organisation | Binôme |
| Evaluation | Code-based |
| Advised time | 9-10h per binome |
| Tools | JDK ==1.8 or later, Eclipse |
| Return | **Eclipse Projet**, named **LastName1.LastName2**.<br>In compressed format, **only source files**, either zip or .tar (**nothing else**) |



| | |
|---|---|
| Contact | Fabienne.Boyer@univ-grenoble-alpes.fr<br>Olivier.Gruber@univ-grenoble-alpes.fr |



## 1 Baby steps

There are several baby steps. The first baby step is there to help you understand the general flow of a client/server system built on top of Java NIO. This first step is however overly simple and therefore needs improvements. Making such improvements is the purpose of the second and third baby steps.

### *Baby Step 1*

We consider an overly simplified *echo* application over Java NIO, where 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 from the server 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.

We ask that you do the following:

1. Read and understand the code in the package *ricm.nio.babystep1*. In particular, pay particular attention to the event-based nature of the execution flow. The main thread is captured in the selector loop, waking up when an event that has been registered of interest happens (accept, connect, available bytes to read, room to write bytes). So the overall execution reacts to occurring events rather than following a given algorithm.

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

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

3. Understand why the given code does not ensure that messages are always sent or received entirely. Try to send larger and larger messages until you see that messages are not received entirely. To achieve this, make the client grow a received message before echoing it back to the server (see below). Ask questions if it is not clear for you.

```
byte[] tmp = new byte[2*data.length];
System.arraycopy(data, 0, tmp, 0, data.length);
System.arraycopy(data, 0, tmp, data.length, data.length);
digest = md5(tmp);
send(tmp, 0, tmp.length);
```

**WARNING:** for this last item, you need to no longer print the message, just print the number of bytes read. 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.

## *Baby Step 2*

We now want to ensure that clients and server receive full messages. As discussed in the lecture, this means implementing an **automata** for reading (receiving) and for writing (sending) entire messages. So we ask you to do the following:

1. Design & implement two classes:

   - A class ***ReaderAutomata*** implementing an automata to receive entire messages. This class should keep reading bytes until it has received a complete message.

   - A class ***WriterAutomata*** implementing an automata to send entire messages. This class should setup a WRITE interest when some message has to be sent, then it should keep writing bytes until all has been sent. Finally it should clear the WRITE interest

   Your server will use one reader to receive and one writer to send messages to a given client. The client will also use one reader to receive messages and one writer to send messages. **But the classes *ReaderAutomata* and *WriterAutomata* are shared, they are the same for the client and server.**

   *To achieve this part of work, read the TP-NIO-HELP-AUTOMATA note that gives detailed explanations.*

2. 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 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.

## *Baby Step 3*

In the previous steps, the server may interact with a single client only. We ask you to do the following:

1. Adapt your server code to support multiple simultaneous clients. This means that your server should be able to invoke the right reader or writer automata when receiving a READ or WRITE event. To achieve this, you may use attachment objects. More precisely, for any client "accepted" at server side (in *handleAccept()*), assume the client is associated to a channel key K. Make the server create a reader and writer automata for that client, alias them in an object called *automata*, and then attach the reference of this object to the channel key K):

```
key.attach(new Automata(readerAutomata, writerAutomata))
```

Later, when at server side some bytes can be read or written from/to that client, just get back the reader and writer automata through the attachment :

```
Automata automata =  (Automata) key.attachment();
```

2. To test your new 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.

# 2 The Real Thing

We consider here designing a more generic design. Our goal is to providing a message-oriented middleware allowing to create communication channels between processes, with channels enabling to send and receive variable-size messages. The intent is to define an abstract interface for that 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 the InputStream and *OutputStream* interfaces:. 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 parts: the interface, the implementation, and the application using that implementation through its interface. We will also have those three parts.

- Look at the interface 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 an application part (package ricm.channels.fileserver), given to help you understand our message-oriented interface and the overall event-oriented execution flow.

We ask you to do the following.

1. Create an Eclipse project "ricm.channels.nio" for the implementation part that you will program over 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. Test your implementation with our file server application.

3. Open issue. A last point, quite open, will be to improve the file-server application. The problem to fix is that each file is loaded entirely in memory before it is sent as one single message. This design may require a lot of memory and introduce high latency (while loading a large file, no other file download may be initiated or progress. The principle of the solution is simple, you already implemented it with regular sockets: load files by chunks, sending each chunk as a message. In other words, sending a file will become a multi-step automaton, where a next chunk is read once the previous one has been sent. To achieve this, feel free to extend our message-oriented middleware interface, if needed.