Code Assignment 06 - Implementation of a reliable file transfer protocol over UDP

Exercise 4 : Implementation of a single-connection file receiver

The guideline for today is the implementation of a simple kind of File Transfer Protocol, at the applicative layer on top of the already implemented layers. To make it simple, let's first consider the case of the transfer of a single file, from a identified client to the server. Moreover, we only consider text files and they will be sent line by line.

For this exercise, you have to write the applicative layer of the server, so as to be compatible with an existing client. The client is given as file `ClientFileSender.java` and you should not have to modify this program, but only changing the `RELIABILITY` parameter for your own tests. As you see, `ClientFileSender` presents many similarities with `Talk_2`. You may yet run one instance of `ClientFileSender` against one instance of `Talk_2.java` to observe what it does.

A `Server_4.java` is also given as the top level for the server. Complete class `FileReceiver`, so as to store the file sent by a connected `ClientFileSender`. As a safety, to avoid overwriting some important file during the tests (oh sure you can!), the transmitted file will be renamed. So, if the client sends, as an example the name `foo.txt`, the server will write the received content in file `_received_foo.txt` and you can then easily check by the unix command:

```
diff foo.txt _received_foo.txt
```

After some convincing tests, submit your file `FileReceiver.java`, through the usual upload form.

Exercise 5 : Implementation of a multi-connection file receiver

The previous implementation suffers from some awkward problems for a server:

- we need to know the location (host and port) of the client at the time of launching the server,
- it can process only one file and then we must restart it.

Also, for this exercise, you have to make changes toward a server that:

- runs continuously and accepts all incoming connections from clients,
- can handle multiple clients at the same time (assuming they send different files).

We keep the overall architecture, with a `GroundLayer` that we absolutely won't change and a `ConnectedLayer` which will be very slightly modified. Most of the work for this exercise concerns a new `DispatchLayer` that will be interleaved between

the `GroundLayer` and the `ConnectedLayer`, without affecting the applicative layer on top of the stack. It is at this new level that will take place both:

- the detection of a new connection, which triggers the launch of a new `FileReceiver`,
- the dispatching of every incoming packet to the `ConnectedLayer` corresponding to its `connectionId, and to it only.`

The server `main` function is now given in file `Server_5.java` and, as we see, it is very simple. Now, after the start of the `GroundLayer`, one starts the `DispatchLayer`. A skeleton for `DispatchLayer.java` is also given. The overall principle is the following:

- Once the `DispatchLayer` is started, it will receive all the incoming packets from the `GroundLayer`,
- then, you have to complete the method `receive` of the `DispatchLayer` to handle these packets,
- this method `receive` must use the `table` which maps the known connection ids to their respective `ConnectedLayer`,
- so, for an already known `connectionId`, a lookup to the table is enough to deliver the packet to the concerned `ConnectedLayer`.
- In case of a new `connectionId`, discovered in an HELLO packet, one must build a new `FileReceiver`, and properly register its `ConnectedLayer`.

This last operation is a little subtle:

1. The `ConnectedLayer` constructor is going to wait for an ACK. Then, to be able to fetch this expected ACK, `receive` must return (to the `GroundLayer`) without waiting for the end of the construction, which is blocked until the ACK. So, we have to launch the construction in an independent thread.
2. Furthermore, to correctly propagate this ACK, the new `ConnectedLayer` should be already registered during its construction. Then, the `ConnectedLayer` will register itself. This is achieved by putting in its constructor the instruction:

```
3.      DispatchLayer.register(this, sessionId);
```

   in place of:

```
    GroundLayer.deliverTo(this);
```

   On the client side, as the `DispatchLayer` is not started, `DispatchLayer.register` simply asks the `GroundLayer` to directly forwards to the only running `ConnectedLayer`. On server side, as the `DispatchLayer` is started, each new `ConnectedLayer` is properly registered and will receive through the `DispatchLayer`.

4. Note that one should now retrieve the location (host and port) of the client. They should be passed by the `GroundLayer` as the `source` argument to `receive`. As a specification, consider that this argument is given as the raw result of `toString()` for the source `SocketAddress`, hence one can retrieve the missing parameters in this `String`.

When all is done and has been seriously tested, including the launch of simultaneous clients and disrupting `RELIABILITY` settings, submit your `DispatchLayer.java` through the usual upload form.

Exercise 6 : Enhanced independence between layers

So, our server works as expected, but it can be argued a bad engineering because the `DispatchLayer` needs to know how to start an applicative `FileReceiver`. For this exercise, you have to solve this issue. The idea is to write a new class `QueueingDispatchLayer` which takes the place of `DispatchLayer.java`, such that:

1. Upon receipt of a **new** HELLO, method `receive` of `QueueingDispatchLayer` only pushes the connection parameters into a queue of pending connections. As `receive` must return fast, enqueuing should not block. As for a TCP server, consider a fixed size queue, which drops out new entries when it is full. You are allowed to use any standard implementation for this queue.
2. As you did for the web server, the `main` function of the server, say in a class `Server_6` now, as a loop of the form (in pseudo code):

```
3.     while (true) {

4.       ConnectionParameters parameters = QueueingDispatchLayer.accept();

5.       start a new FileReceiver with these parameters

6.     }
```

   where `QueueingDispatchLayer.accept()` dequeues any pending connection or waits for one.

A `ConnectionParameters` class is given in the **additional web page**. You are not expected to change it.

For this exercise, you are expected to upload your new `Server_6.java` and `QueueingDispatchLayer.java` files. If you had chosen to use your own implementation for a queue, its definition must be included in `QueueingDispatchLayer.java`.