Code Assignment 05 - Implementation of a reliable connected transport protocol over UDP

Exercise 1 : sending and receiving datagrams

For this exercise, you have to write a class which implements the basic functions to send and receive datagrams. For convenience, it is build over UDP, but will be considered as our ground layer for further developments.

Then, this class is named `GroundLayer` and, as only one instance of this layer will run from a considered Java program, it has only `static` methods. At above layers, there may be multiple instances and they will be objects implementing the given `Layer` interface. Program `Talk_1` illustrates the use of the `GroundLayer` and allows to test. The idea is to launch two executions of this program, either both on your machine, or better with a classmate's and to make them communicate. Remember that you need to write interoperable code.

The **annex web page** gives you files `Layer.java` and `Talk_1.java` and a skeleton for `GroundLayer.java`.

In `GroundLayer` class:

- method `start` makes things operational on the specified UDP port and returns `true` if it's okay, or `false` in case of problem; to follow the `Layer` contract, method `start` must launch a thread which waits for incoming packets on the port and passes the corresponding payload to method `receive` of the layer above (for now, the `source` parameter of `receive` may be any string which makes sense for you). This thread will run until an exception occurs or it is interrupted.
- method `deliverTo` stores a reference to the `Layer` to which payload of incoming packets is passed. A `null` argument makes that incoming packets are not pushed upwards.
- method `send` sends an UDP packet with the given payload to the specified destination. To simulate the loss of packets in the network, the value of `RELIABILITY` is used as the probability that `send` really sends a datagram. So, this method must send nothing with probability `1-RELIABILITY`.
- method `close` must interrupt the receiving thread and close the UDP socket.

Note that:

1. every `IOException` must be caught and reported by a message on `System.err`,
2. methods of `GroundLayer` must not throw any exception,
3. for convenience, payloads are represented as `String` objects, but the char encoding in these `String` objects may differ from one system to another.

So, for interoperability, one must convert payloads to a common encoding, say the UTF-8 charset, before sending them and, upon receiving, convert payloads back to its own default encoding.

Exercise 2 : toward a Connected Transport Protocol

For this exercise, you have to write a class named `ConnectedLayer` which implements the basics of a connected transport protocol over our `GroundLayer`.

Program `Talk_2` illustrates the use of this class and allows to test. As you see, an applicative layer now allocates an instance of `ConnectedLayer` and the former is stacked above the later. The `ConnectedLayer` will be itself stacked above the `GroundLayer`. For now, the `GroundLayer` can deliver to only one `Layer` and we will assume that it is still the case, as for `Talk_2`.

The `ConnectedLayer` constructor takes three parameters, the `host` and `port` which identify the destination for all packets send through this `ConnectedLayer`, and an `id` for the connection, here picked as random. This connection `id` will be inserted in every packet transmitted to the `GroundLayer`, as well as an increasing packet number. So, for interoperability, let us specify the packet format: for any `payload` given by the applicative layer, the wrapped payload passed to the `GroundLayer` must be of the form:

```
connectionId;packetNumber;payload
```

where the three fields are separated by semi-colons. Moreover, the `ConnectedLayer` constructor sends an initial packet:

```
connectionId;0;--HELLO--
```

so the first payload sent from above will have the packet number 1.

The requested methods of the `ConnectedLayer` are those implementing the `Layer` interface, notably:

- method `send` which makes the call to `GroundLayer.send` with suitable parameters. It must also increment the packet number for each new payload.
- method `receive` must extract the payload destinated to the layer above. It must also send an ACK for each received packet (but it should not ACK an ACK, you see why). For any received non-ACK packet:

  ```
  connectionId;packetNumber;payload
  ```

  the answer must be

  ```
  connectionId;packetNumber;--ACK--
  ```

  so using the `connectionId` and `packetNumber` from the remote sender.

Exercise 3 : implementing a Reliable Connected Transport Protocol

For this exercise, you have to add reliability to the `ConnectedLayer` by implementing the *Stop and Wait* protocol. The principle is really simple: when the `send` method has to send a packet, it periodically repeats the call to `GroundLayer.send`, until a corresponding ACK is received and, during this time, the (thread running the) `send` method must block. The implementation requires a little work, however. It is advisable to use a single `Timer` and schedule a new `TimerTask` for each packet. Then, `send` enters a `wait` state, until the receipt of a suitable ACK triggers by a `notify`. Finally `send` can resume and stop the sending task.

This also applies for the HELLO packet, so the receiver is now considered to have received an HELLO, it knows the connectionId from the sender and, then, it must ack a packet only when its connectionId matches the HELLO's connectionId.