Exercise Session 8: Reliability over an Unreliable Channel

COM-208: Computer Networks

The goal of this lab is to undestand what it means to provide reliability over an unreliable channel (which is what TCP does).

Please make sure you have completed Lab5 before you try this one. To do the lab, you will need some initial code that you can get from Moodle.

You will develop a distributed application that **reliably transfers a large file** from one computer (we will call it the Client) to another (called the Server) and prints the file's contents on the Server. But there's a catch: this application must use **UDP** as the transport-layer protocol; given that UDP does not provide reliability, your application will need to provide it itself.

More specifically, your application will perform the following tasks:

- At the Client, break the file into packets.
- Transfer each packet from the Client to the Server, as many times as needed to transfer it successfully.
- At the Server, assemble the file from the packets (ensuring the packets are put in the correct order).

You will build a **Stop-and-Wait** protocol, very similar to the rdt3.0 protocol from the book.

The Stop-and-Wait Protocol

Stop-and-Wait Sender

1. *Data received from above.*

   When data is received from above, the Stop-and-Wait sender puts it in a packet and assigns it the next available sequence number.

   If the sender is not currently waiting for an ACK, the packet is sent and a timer is started; otherwise it is buffered.

2. *Timeouts.*

   Timers are used to protect against lost packets. If a timeout occurs, the packet is re-transmitted.

3. *ACK received.*

   If an ACK is received for the packet currently pending an acknowledgment, the Stop-and-Wait sender marks the packet as having been received.

   If there are any packets in the buffer, the first of these packets is sent and a timer is started.

Stop-and-Wait Receiver

1. *Packet with expected | sequence number received | for the first time.* | | If the sequence number of | the received packet is | equal to the sequence | number of the packet the | receiver is expecting to | receive (`1 + #last received packet`), the packet's payload is | delivered to the upper | layer and an ACK packet is | returned to the sender. | | | |
2. *Packet with sequence | number smaller than the | expected sequence number | is received by 1.* | | In this case, an ACK must | be generated for this | packet, even though this | is a packet the receiver | previously acknowledged. | |
3. *Otherwise.* | | Ignore packet. | | | | |

There are the following basic differences with respect to the book version:

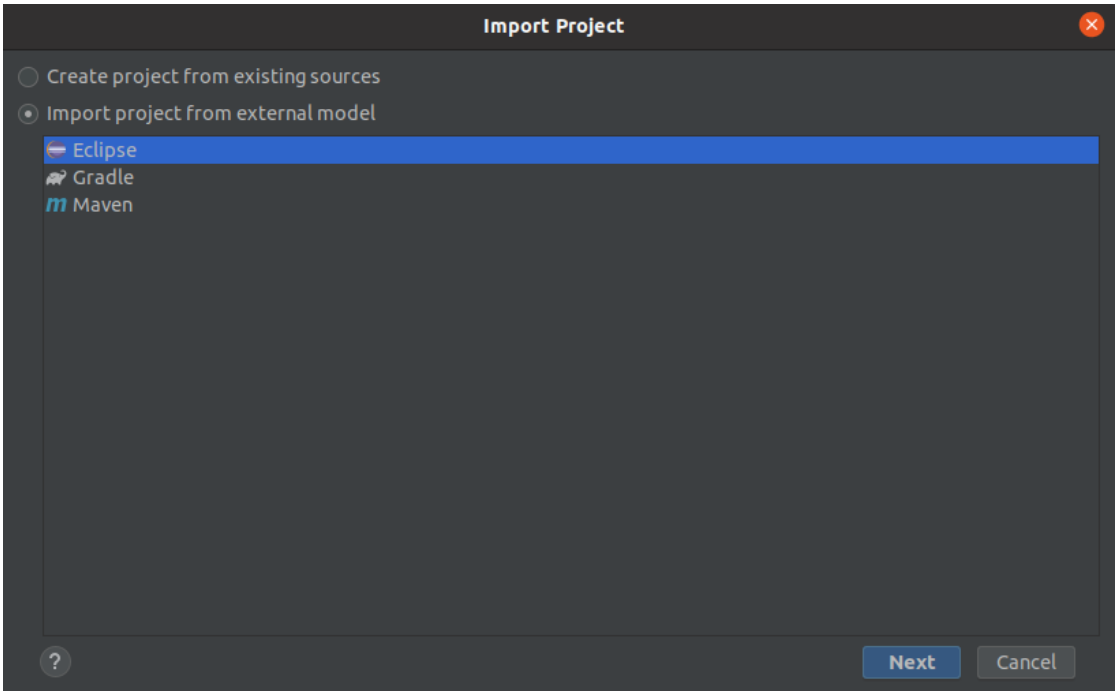
- We are using `int` sequence numbers, in place of only `0` and `1`.
- We are not including packet length or checksum in the packets of our protocol, because UDP provides these.

Milestones

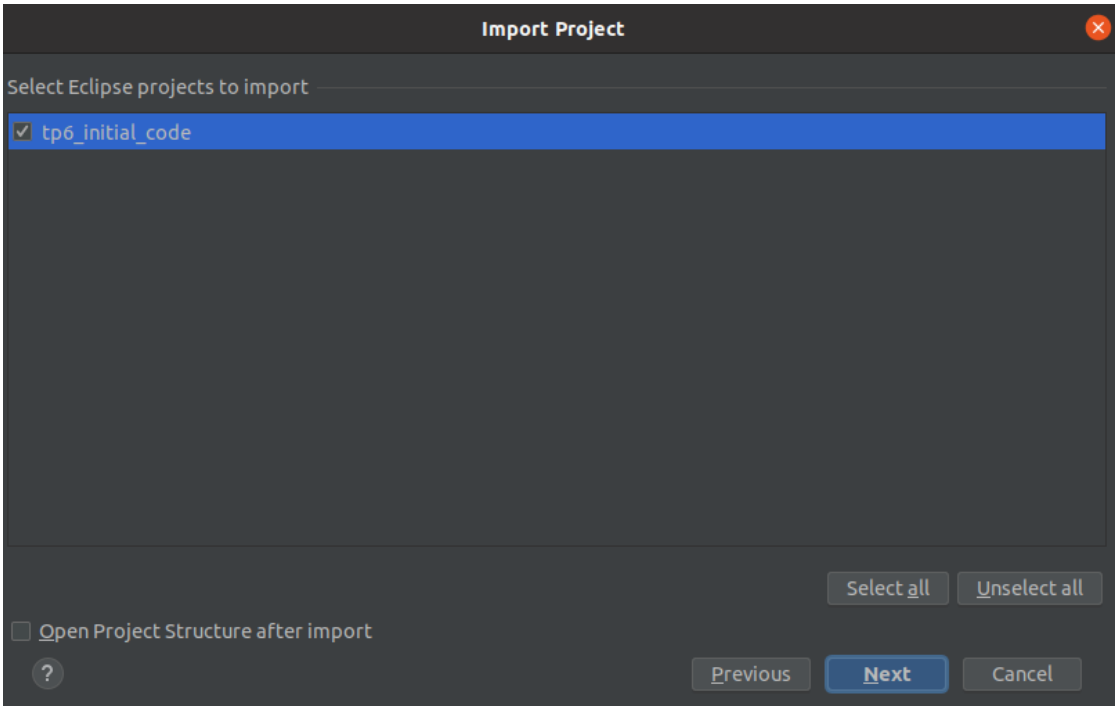
0. Running the base code

To start working, you can use the base implementation available on Moodle (initial_code.zip). You can import this .zip file into Eclipse as a Java project or into IntelliJ following the next steps:

1. Select File > New > "Project from Existing Sources" in the upper part of the window.
2. Choose the directory containing the code.
3. Select "Import project from external model" with "Eclipse" as an option in the bottom window (see figure below).



4. Press "Next" until prompted to select the project to import (see figure below). You should see "tp6_initial_code" already selected.



5. Select the project SDK, we recomend Java 11 or higher.

1. Transfer a single UDP packet from Client to Server.

```
//ReliableClient.java

public ReliableClient(String address, int port, int timeout)   {
  try {
      // Open the UDP socket.
      //@TODO

      // Initialize the filestream we will be using to read data from
      this.fis = new FileInputStream("MonaLisa.txt");
  } catch (IOException e) {
      e.printStackTrace();
  }
}
```

2. Read a small file and break it up into 10-byte packets.

Don't worry yet about reliability, e.g., acknowledgments etc.

```
//ReliableClient.java
public void SendFile() {
  boolean successFlag = true;
```

```
    int length;

    try {
        // Read a fixed length of data
        int bufferSize = ...
        byte[] data = new byte[bufferSize];

        // Until we finish sending the file...
        while ((length = fis.read(data)) != -1) {
            // Create a UDP packet
            //@TODO

            // Send it
            //@TODO

            // And wait for an ACK
            //@TODO
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

3. Make the Server send an ACK for every packet it receives.

```
//ReliableServer.java
public void listen() {
  try {
      while (true) {
          // Allocate enough space for the biggest possible packet
          //@TODO

          // Receive the new packet
          //@TODO

          // If we expect this packet, send data to upper layer.
          // In this case, we simply print the bytes
          //@TODO

          // ACK packet if it is the one we have been expecting.
          // Otherwise, do nothing (we drop the packet silently)
          //@TODO
      }
  }
  catch (Exception e) {
      e.printStackTrace();
  }
}
```

4. Make the Client wait for the ACK before transmitting the next packet.

So, if a packet gets lost, the Client gets permanently stuck (which is OK for now).

Tip: To simulate packet loss, you can make the Server ignore a particular packet.

```
//ReliableClient.java

public void SendFile() {
  boolean successFlag = true;
  int length;

  try {
      ...

      // Until we finish sending the file...
      while ((length = fis.read(data)) != -1) {
          // Create a UDP packet
          //@TODO

          do {
              // Send it
              //@TODO
```

```
            // And wait for an ACK
            //@TODO
        } while (successFlag == false);
    }
  } catch (IOException e) {
      e.printStackTrace();
  }
}
```

5. Add timeouts and retransmissions.

Use the `setSoTimeout` option on a socket object. This will make the Client timeout if it waits too long for an ACK.

Remember to catch the exception that sockets with this option throw; you can use it to implement the retransmission functionality.

```
//ReliableClient.java
public ReliableClient(String address, int port, int timeout)   {
  ...
  // Set the timeout duration for the socket
  ...
}

private boolean WaitForAck() {
  try {
      while (true) {
          // We wait until we receive the right ACK
          DatagramPacket dp = new DatagramPacket(new byte[MyAckPacket.MAX_SIZE
    ↪ ],
                                                 MyAckPacket.MAX_SIZE);
          //@TODO

          // If this is the right ACK, we return "Success"
          //@TODO

          // If an ACK wakes us up, check the time again
          //@TODO
      }
  } catch (... e) {
      // If we timed out waiting for an ACK we return "Failure"
      return false;
  } catch (IOException e) {
      e.printStackTrace();
      return false;
  }
}
```

6. Extend your protocol to transfer packets of variable size.

Instead of reading the input file 10 bytes at a time, make the Client read a random number of bytes (say, between 10 and 30). This will simulate what happens when the application layer does not always have enough data ready to transmit.

You can use the following code snippet to generate a random number between min and max:

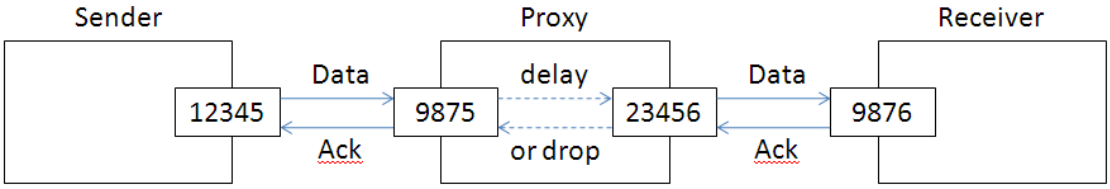
```
private static int getRandomNumberInRange(int min, int max) {
  if (min >= max) {
  throw new IllegalArgumentException("max must be greater than min");
}

  Random r = new Random();
  return r.nextInt((max - min) + 1) + min;
}
```

7. Test your protocol against an unreliable channel.

In the campus network, delays are very small, and packet loss is almost non-existent. Since you need both to test your protocol, you will to introduce delays and packet loss artificially.

For this purpose, we provide you with a Java application (included in the source code) that creates a lossy channel. This application works like a UDP proxy:

1. The sender sends the data packet to the proxy.
2. The proxy delays the data packet, and retransmits it to the receiver, or drops it.
3. The receiver responds with an ack packet.
4. The proxy delays the ack packet, and retransmits it to the sender, or drops it.



Start the lossy channel by running `StartLossyChannel`. You can configure the following:

- **sender port** - should be the same as the port to which the sender sends the data
- **receiver host** - the machine at which the receiver is executed (e.g., `localhost`)
- **receiver port** - should be the same as the port receiver listens on
- **packet delay** in milliseconds and **loss ratio** as a float value between 0.0 (no loss) and 1.0 (100% loss)

Note that you need to press ENTER after you input a new value; the bottom window displays a brief confirmation message. **Remember that if you are running the proxy and the receiver at the same machine, the receiver port needs to be different from the sender port.**

To confirm that the proxy is running, you can check the debug messages shown on the bottom window.

8. Review your implementation.

What is the maximum amount of time between when a packet is lost and when it gets retransmitted?

If you only relied on the `setSoTimeout` option mentioned above, your protocol might take longer than the advertised time to retransmit a lost packet.

- Which sequence of actions will prevent your program from retransmitting the packet in time?
- Think of a way to reduce/eliminate this additional delay!