# Assignment #2 – Selective Repeat Protocol Implementation

## Fall 2020


### 이름: 완

### 학생번호:2019007901


**Build Environment: Windows 10**

**Java Version: Java v1.8.0 or later**

- **Compiling Procedure**

Begin by saving the source files for both client application and server application (server.java and client.java respectively) to the desired location. That location will be the starting path for server application and default path for client application.

```
PS D:\TCPServer> ls


    Directory: D:\TCPServer


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/11/2020   7:19 PM          10377 server.java
```

```
PS D:\TCPClient> ls


    Directory: D:\TCPClient


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/11/2020   6:48 PM           5530 client.java
```

*Example: Server.java in "D:\TCPServer" and client.java in "D:\TCPClient".*

Start compiling the both java files in their respective directories by using the command "*javac <java filename>*" in the command prompt. If the compiling stage is succeeded, the CLASS file for both of java file will be created in the same directory.

```
PS D:\TCPServer> javac server.java
PS D:\TCPServer> ls


    Directory: D:\TCPServer


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/11/2020   9:17 PM           6961 server.class
-a----         11/11/2020   7:19 PM          10377 server.java
```

```
PS D:\TCPClient> javac client.java
PS D:\TCPClient> ls


    Directory: D:\TCPClient


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/11/2020   9:17 PM           4607 client.class
-a----         11/11/2020   6:48 PM           5530 client.java
```

*.CLASS files are created after compiling using javac.*

After successfully compiling the java files. You can run both programs using "*java <filename>*" command in the command prompt. You will need to **run the server application first** before executing client application. Starting client application first will cause error as there is no server available at the moment.

- **Running the Program**

1. Server Program

To run the server application, simply type "*java server*" in the command prompt that contains the compiled *server.java* file. After running the server program, the program will tell the user that the server is on and displays the command port number and data port number.

```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121
```

By default, the server command port number is set to "2020" and data port number is "2121". You can provide your own designated port numbers by giving the command line arguments. Type "*java server <command port number> <data port number>*" when executing the server program.

```
PS D:\TCPServer> java server 4000 6780
Server is ready!
Command port number: 4000
Data port number: 6780
```

*Executing with command line argument to change the default port number*

Now your server program is ready to receive any connection from the client!

2. Client Program

You can run the client program after executing the server program. The method for running the client program is similar as before, by typing "*java client*" into the command prompt of client directory. If the execution successful, the program will display the IP Address of FTP server host, command line port number, and data port number onto the screen.

```
PS D:\TCPClient> java client
Program Running..
IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121
```

By default, the server FTP host will be designated to "127.0.0.1" which is the localhost, command port "2020" and data port "2121". To change these default arguments, type "*java client <server host> <command port number> <data port number>*" before executing the program.

```
PS D:\TCPClient> java client 121.161.196.193 4000 6789
Program Running..
IP address: 121.161.196.193
Command port number: 4000
Data port number: 6789
```

*Executing with command line argument to change the default port number*

The client's port numbers **must be similar to server program's port numbers** for the program to run successfully. Make sure the server IP address is valid to the server program too.

- # Changelog for Assignment #2

1. Size of server-to-client chunk has been changed from 1000 bytes each to 1003 each. The structure of the chunk has been changed to { SeqNo(1 byte) | Chksum (2bytes) | Data (1000 bytes) } . This is done by first creating 3 different byte arrays for each sections. Then, using `ByteArrayOutputStream`, we will write each byte arrays into a single stream. Finally, using `.toByteArray()` method, the stream will converted into a single byte array which is the "chunk" and ready to send to client in GET command.

2. Size of client-to-server chunk has been changed from 1000 bytes each to 1005 bytes each. The structure of the chunk has been changed to { SeqNo(1 byte) | Chksum (2bytes) | Size(2 bytes) | Data (1000 bytes) } . This is done by first creating 4 different byte arrays for each sections. Then, using `ByteArrayOutputStream`, we will write each byte arrays into a single stream. Finally, using `.toByteArray()` method, the stream will converted into a single byte array which is the "chunk"and ready to be sent to server in PUT command.

3. New commands DROP, TIMEOUT, and BITERROR has been implemented to simulate the error occurring during file transfer.

4. Sliding window has been implemented for sending chunks in PUT command. Window size is set to 5.

5. For PUT command, the client will not send all chunks at once. Instead, it only sends the chunks that are in the sending window. After received the ACKs from the server, send the next set of chunks.

6. In server, implement 2 `ArrayList<byte[]>` types to store the chunks that are arrived from client and to store buffered chunks.

7. Error handling for `Socket Exception` has been added to both client and server.

8. Set timer in client side every time all chunks in window has been sent for 100ms to simulate the sliding window process.

9. In this assignment, this program only supports DROP/BITERROR/TIMEOUT for a single packet number. Multiple packets cannot be corrupted at the same time.

- **DROP , TIMEOUT, BITERROR <packetNum>**

These commands are used to corrupt any chunk corresponding to the packet number given by the user. If any of these commands are entered right before PUT command, the chunk where it's `seqNo` is same as `<packetnum>` will be corrupted accordingly to those 3 error types above during the chunk transmission from client to server. Remember that this program on support a single chunk corruption. If these commands receive multiple input, no chunk will get corrupted.

- **Update on PUT command (Client side)**

Unlike before, in the client size, the PUT command starts by calculating the total packets to send. This amount is calculated according to the size of file to be transmit. Then an additional byte array `corrupted[]` is initialized to store the corrupted packet (if any) which will be retransmitted later. With window size of 5, we begin out window by sending "`Window start`" to the server. Then we start reading the file, create a chunk, check if the `seqNo` is equal to corrupt target (if any).

If equal, next we will check what type of error does that chunk has (dropped/timed out/bit error). If it was timed out, the thread is delayed for 1 second before sending the chunk. If it was `chkSum` error, we change the `chkSum` of the chunk to `0xFFFF` just before sending it to the server. If it was dropped, we skip the transmission of that chunk (ie. not sending the chunk). Then, we will copy the chunk to the byte array `corrupted[]` and send it on the next window.

Finally, we send a message "`Window end`" to server to notify that all chunks in the client window has been transmitted. Then, we will wait any message from the server if there are any chunks corrupted at the server side. If there is, the variable `corruptExist` is set to true and this will trigger the retransmission of corrupted chunk on the next window.

All these processes will be repeated until the amount of sent chunks is equal to the total packets. Every time the client receives ACKs from server, the packets which are ACKed are shown to the client monitor first before begin retransmitting the next window.

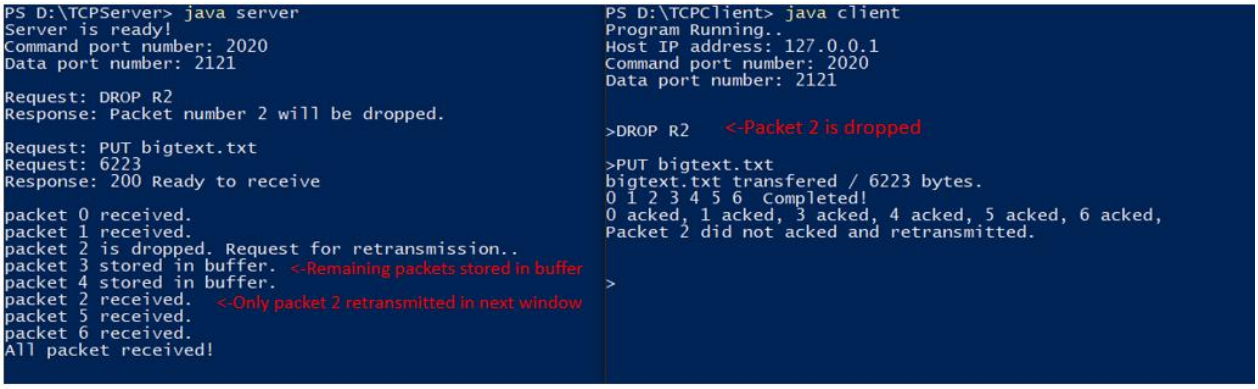
- **Update on PUT command (Server side)**

The total chunks to be received are calculated the same way how it was calculated in client side. After the server send the response message "`200 Ready to receive`", the server will wait until it receive the status from client ("Window start" /"In window"/ "Window end" / "Stop" ). If the status is "`Window start`", then the server starts reading the chunks arrived and save it into `byte[] chunk`. The chunk will then added into `ArrayList<byte[]> currentWindow`. This procedure is repeated until the server receive "`Window end`" from client.

Into the next step, the server will start processing all the chunks in the `currentWindow` for error checking. First, the `seqNo` and `chkSum` of every chunk are extracted and check either `seqNo` is equal to corrupt target or if `chkSum==0xFFFF`. If the current chunk is clean, then it is written to file. Else, `corruptExist` variable will set to true and it will send the corrupted `seqNo` to the client to request retransmission.

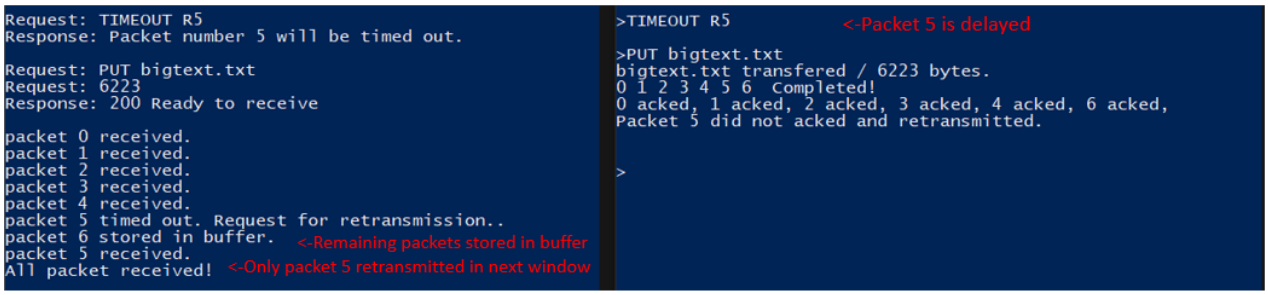
From here, all the remaining chunks in the window will be added to `ArrayList<byte[]> buffer`. After the corrupted chunk has been retransmitted, it will be written first to file. Then, we will check if there's any chunks in buffer. If it is, write all the chunks in buffer and empty the buffer before start receiving the next window.

- **DROP (window size = 5)**



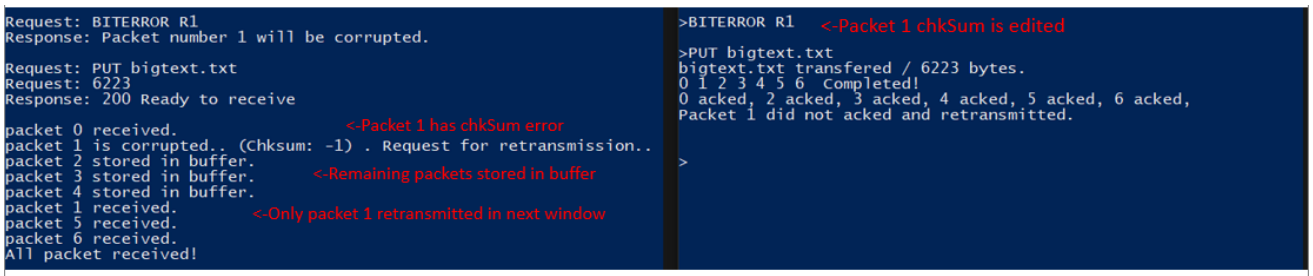

Server Client

- **TIMEOUT (window size = 5)**




Server Client

- **BITERROR (window size = 5)**




Server Client