Assignment 3

Computer Science 441

Due: 23:55, Wednesday November 23, 2016

Instructor: Majid Ghaderi

1 Objective

The objective of this assignment is to practice UDP socket programming and reliable data transfer. Specifically, you will implement a file transfer utility that employs the Go-Back-N protocol for reliability.

2 Overview

In this assignment, you will implement a simplified FTP client based on UDP called `FastFtp`. Since UDP does not provide any data reliability, you will implement your own reliability mechanism based on Go-Back-N. The simplified client only supports *sending* a file to the server.

The client and server use a TCP connection to exchange control information about the file transfer. After an initial handshake process over the TCP connection, the client uses a UDP socket to transmit the actual file content to the server. The server uses the same port number for both its TCP and UDP sockets. Similarly, at the client side, TCP and UDP sockets should use the same port number, as depicted in Figure 1.

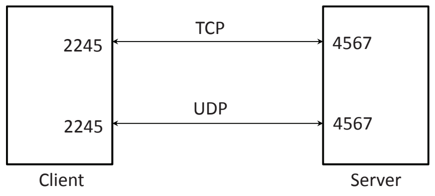

Figure 1: TCP and UDP sockets use the same port number.

A high-level description of the internal operation of `FastFtp` is presented in Algorithm 1.

Algorithm 1 `FastFtp`

```
1: Open a TCP connection to the server
2: Send file name to the server over TCP
3: Wait for server response over TCP connection
4: Open a UDP socket at the same port number as your local TCP port number
5: Send the file content to the server segment-by-segment over UDP (handle ACKs, time-outs,
   and retransmissions)
6: Send end of transmission message to the server over TCP
7: Clean up and close TCP and UDP sockets
```

All control messages over the TCP connection are in binary format (enough with text based protocols!). Use the Java stream classes `DataInputStream` and `DataOutputStream` to read from and write to the TCP socket. Specifically, use `writeUTF()` method to send the file name to the server. To read the server response during the handshake use method `readByte()`. Reading a value of 0 from the server means that the server is ready for file transmission. Any non-zero value indicates an error. To send the end of transmission message to the server, use method `writeByte(0)`.

To implement Step (5) of Algorithm 1, you need to implement the following operations in your program:

- **Sending data:** A file name is given to your program as input. Read the file chunk by chunk, encapsulate each chunk in one segment and send the segments to the server. The maximum size of chunks is given by the constant `MAX_PAYLOAD_SIZE` in class `Segment`. The sequence number for segments starts at 0 and is incremented per every segment transmitted. Once a segment is transmitted, you should add it to a *transmission queue* until the segment is acknowledged by the server. The capacity of the transmission queue is determined by the *window size*, which is given to you as an input parameter.
- **Receving ACKs:** Your program should be listening for arriving ACK segments from the server. Recall that in Go-Back-N, ACKs are cumulative, meaning that an ACK segment with sequence number n indicates that all segments with sequence numbers $0, \ldots, n-1$ are received by the server, and the next expected segment is the segment with sequence number n. If an ACK with sequence number n is received, then check the transmission queue and remove all segments with sequence numbers *smaller* than n, as they have been received by the server.
- **Time-outs:** Go-Back-N has a single timer which is set for the oldest unacknowledged segment in the window. In your program, the timer is set for the *first* segment in the transmission queue (*i.e.*, the segment at the head of the queue). If the timer goes off then all pending segments in the transmission queue should be retransmitted to the server and the timer restarted, if the queue is not empty.

Important: *The above three operations should be implemented in parallel. They are asynchronous operations that should not be serialized. For example, while the program is sending segments, it should be able to receive ACKs and handle time-outs simultaneously.*

3 Software Interfaces

Define a Java class named `FastFtp`, which includes the following public methods:

- `FastFtp(int window, int timeout)`
  This is the constructor to initialize the program. The parameter `window` specifies the win-

dow size (in segments) of the Go-Back-N protocol. The parameter `timeout` specifies the time-out interval (in milli-seconds) at the sender side.

- `void send(String serverName, int serverPort, String fileName)`
  Transmits the file specified by `fileName` to the server `serverName` reliably. The specified server is listening on port `serverPort`. Once this method returns, the file has been fully transmitted and all transmitted segments have been ACKed.

Your implementation should include appropriate exception handling code to deal with various exceptions that could be thrown in the program. A skeleton class, named `FastFtp`, is provided on D2L which includes a simple test driver so that you can see how we are going to test your code. Notice the import statement

```
import cpsc441.a3.*
```

at the top of the file. The package `cpsc441.a3` contains classes `Segment` and `TxQueue` to be used in your program. A jar file named `a3.jar` containing the package `cpsc441.a3` is provided to you on D2L. Make sure to add this jar file to your class path when compiling and running your program. For example, if you are using Java command line tools, use the option `-cp` to include `a3.jar` in your class path.

The source code for classes `Segment` and `TxQueue` is also provided to you as a source of documentation for using these classes:

- `Segment`: This class defines the structure of a segment that is transmitted between the sender (*i.e.*, client) and receiver (*i.e.*, server). Read the Javadoc documentation of the class for how to use it. Note that both data packets and ACKs are of type `Segment`. Segments that go from the sender to receiver carry data, while segments that come from the receiver are ACKs that do not carry any data.
- `TxQueue`: This class implements a bounded and fully synchronized queue data structure to use for your transmission queue. The internal implementation is based on a circular array for improved efficiency. Look at the code and Javadocs for the usage. This queue is synchronized across its various operations to avoid data integrity problems when multiple threads access the same structure for both reading and writing. Curious students could learn a bit about synchronization mechanisms in Java by reading the source code.

A server program that implements the receiving side of the protocol is provided to you in a Java jar file called `ffserver.jar`. The server comes with a `readme` file that explains how to run it.

Restrictions

- You are not allowed to change the signature of the methods provided to you. You can however define other methods or classes in order to complete the assignment.
- Your program should use the library file `a3.jar`. Do not use `Segment.java` or `TxQueue.java` in your code. The source code is provided to you only for documentation purposes.
- You have to write your own code for sending and receiving UDP packets. Ask the instructor if you are in doubt about any specific Java classes that you want to use in your program.