Assignment 5
Computer Science 441
Due: 23:55, Friday December 2, 2022
Instructor: M. Ghaderi

1 Objective

The objective of this assignment is to practice UDP socket programming and reliable data transfer. Specifically, you will implement a pipelined UDP-based program for reliable file transfer based on the Go-Back-N protocol, which is more efficient that the Stop and Wait protocol in Assignment 4.

2 Specification

In this assignment, you will implement a simplified FTP client based on UDP called `GoBackFtp`. While this assignment is implemented independent of Assignment 4, **it uses the same TCP/UDP design, handshake process and segment structure**. Specifically, the description in this assignment is focused on the process of sending the file using Go-Back-N protocol. This requires implementing processes for sending segments and receiving acknowledgments to run *concurrently*, which is somewhat similar to the design of `GzipClient` in Assignment 1.

Recall that, before the actual data transmission, the client and server go through an initial handshake process to exchange control information about the file transfer. The handshake takes place over TCP, while the actual file transfer is carried out over UDP. The handshake process is exactly the same as in `StopWaitFtp` program of Assignment 4, and thus is not repeated here.

3 Implementation

3.1 High Level Structure

A high-level design of `GoBackFtp` is presented in Program 1.

Program 1 `GoBackFtp`

```
1: Handshake with server
2: Create the re-transmission timer
3: Start the ACK receiving thread
4: Start the segment sending thread
5: Wait for the segment sending thread to finish
6: Wait for the ACK receiving thread to finish
7: Shutdown the re-transmission timer
```

In addition to the handshake process, there are three main operations in `GoBackFtp`:

1. Creating and sending data segments,
2. Receiving and processing ACKs,
3. Re-transmission timeouts.

A critical aspect of `GoBackFtp` design is that these three operations should be executed concurrently. They are asynchronous operations that *should not be serialized*. It means that, while the program is sending segments, it should be able to receive ACKs and handle timeouts simultaneously. As such, your design should rely on two separate threads for sending segments and receiving ACKs, similar to Assignment 1, which run in parallel with a *timer task* that is in charge of timeouts. More details are provided on each of these design components in the following subsections.

3.2 Sending Thread

The high-level design of the sending thread is described in Program 2. In `GoBackFtp`, the sender can have multiple in-flight segments. Once a segment is transmitted, it should be added to a *Transmission Queue* until the segment is acknowledged by the server. In other words, the transmission queue contains all those segments that have been transmitted but not acknowledged yet. The capacity of the transmission queue is determined by the *window size* of the Go-Back-N protocol, which is given to your program as an input parameter. The transmission queue is *full* if the number of segments in the queue is equal to the window size.

Program 2 Sending Thread

```
1: while not end of file do
2:     Read from the file and create a segment
3:     Wait while transmission queue is full
4:     Send the segment and add it to the transmission queue
5:     Start timer if the segment is the first in the transmission queue
6: end while
```

3.3 Receiving Thread

Your program should be listening for arriving ACK segments from the server using the same UDP socket that is used for sending data segments to the server. Recall that in Go-Back-N, ACKs are cumulative, meaning that an ACK with sequence number n indicates that all segments with sequence numbers *smaller than n* 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.

The high-level design of the sending thread is described in Program 3. Notice that if the transmission queue is empty, it does not necessarily mean that the file transfer is complete. Only when the queue is empty and the sending thread has finished, the receiving thread can finish. Also, as in Assignment 3, to give the receiving thread a chance to check the conditions of the while loop, you have to set the socket timeout options for the UDP socket using method

Program 3 Receiving Thread

```
1: while sending thread not finished or transmission queue not empty do
2:     Receive ACK
3:     if ACK is valid then
4:         Stop re-transmission timer
5:         Update the transmission queue based on the received ACK
6:         Start re-transmission timer if the transmission queue is not empty
7:     end if
8: end while
```

`DatagramSocket.setSoTimeout()`. A received ACK is valid if it acknowledges any pending segments, *i.e.*, its sequence number is greater than the sequence number of the first segment in the transmission queue.

3.4 Timeout Task

Go-Back-N uses a single retransmission timeout which is set for the oldest unacknowledged segment in the window. In your program, a recurring timeout should be 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 are retransmitted to the server, as described in Program 4.

Program 4 Timeout Task

```
1: for all segments in transmission queue do
2:     Send the segment via the UDP socket
3: end for
```

Note that, as explained in Assignment 4, the timeout task is scheduled to happen at regular intervals. This was achieved by using the method `scheduleAtFixedRate()` of class `Timer`. Thus, when a timeout occurs, there is no need to manually reschedule the next timeout task. Whenever the first segment in the queue is ACKed, then the receiving thread cancels the timeout that was set for that segment.

3.5 Handling Concurrency

3.5.1 Transmission Queue

The transmission queue is a shared resource among the three concurrent process in `GoBackFtp`. Thus, access to the queue has to be synchronized to avoid race conditions. While this can be achieved using `synchronized` blocks in Java, the `ConcurrentLinkedQueue` class already provides this functionality using a highly efficient implementation. This queue has internal

locking mechanisms to prevent memory inconsistencies while allowing simultaneous access to the queue elements. It implements the generic `Queue` interface with all the familiar operations of a FIFO queue. It is recommended that you implement the transmission queue in your program as an instance of the `ConcurrentLinkedQueue` class.

3.5.2 Timeout Task

Recall from Assignment 4 that to start and stop timeout, you need to schedule and cancel the corresponding timer task. There is only one active timer task at any given time, which could be accessed simultaneously by both the sending and receiving threads when starting or canceling timeouts. To prevent this situation, you have to synchronize access to the timer task. If you have followed my recommendation from Assignment 4 to define two helper methods to start and stop the timer, then all you have to do is to define these two methods as being `synchronized` methods:

```
synchronized startTimer();
synchronized stopTimer();
```

Then Java ensures that only one of them can be executed at any point in time, effectively preventing them from being executed concurrently. You are of course not limited to this approach and can use other mechanisms (such as locks) to synchronize access to the timer task in your program.

3.6 Design Requirements

Your client design should be based on the following principles:

- Segment Payload: Create segments with the maximum payload size for good performance. Only the last segment from the client to the server may not carry sufficient data to be of maximum size.
- Parallel Design: The three processes in charge of sending data segments, receiving ACKs, and re-transmissions must be implemented to run concurrently, *i.e.*, in separate threads.

4 Software Interfaces

The class `Segment` and the server program `ftpserver` are reused from Assignment 4. The server is implemented so that it can work with Stop-and-Wait as well as Go-Back-N protocol.

4.1 Method Signatures

The required method signatures for class `GoBackFtp` are provided to you in the source file `GoBackFtp.java`. There are two methods that you need to implement, namely a constructor and method `send`. Refer to the Javadoc documentation provided in the source file for more

information on these methods.

4.2 FtpSegment Class

This class defines the structure of the segments exchanged 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 `FtpSegment`. 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.

4.3 Exception Handling

Your implementation should include exception handling code to deal with all checked exceptions in your program. This includes catching the exceptions and printing exception messages (or the stack trace) to the standard system output. After that, clean up, close all streams and sockets and return from the `send` method.

4.4 Running Your Code

A driver class named `GoBackDriver` is provided on D2L to demonstrate how we are going to run your code. Read the inline documentation in the source file `GoBackDriver.java` for detailed information on how to use the driver class. Moreover, you can download the server program that implements the service side of the protocol from D2L. The server program is provided in a jar file named `ftpserver.jar`. The server comes with a `README` which includes instructions on how to run it. Although you can check the transferred file in the working directory of the server for correctness, you should also correlate the output of your program with that of the server to verify correct protocol implementation. For example, you can correlate packet drops at the server with timeouts at the client, which should match.

4.5 Console Output

Add print statements (using `System.out.println`) in your code to print the following information on the console:

1. `send <seqNum>` – every time the client sends a segment to the server.
2. `ack <seqNum>` – every time the client receives an ACK.
3. `retx <seqNum>` – every time a segment is re-transmitted due to timeout.
4. `timeout` – every time a timeout happens at the client.

In the above, `<seqNum>` refers to the sequence number of the relevant data or ACK segment. Do not directly print anything else to the console beyond exception messages and the above information. For debugging purposes, you can use the global `logger` object defined in the driver class, whose level can be set using the command line option `-v`. Refer to the driver class

source file for more information. The logger can be used to write messages to console during code development.

4.6 Design Document

Prepare and submit a design document to describe the effect of window size on the performance of your program. Performance is defined as the time it takes to upload a given file to the server. Use file `large.jpg` and set server parameters as: `Loss = 0` and `Delay = 0`. Notice that the server prints the amount of time it takes to complete the file transfer. Include a plot to show the relation between the performance of your FTP client and its window size. Experiment with a range of window sizes and explain your results.

Follow the design document formatting requirements on D2L.

Restrictions

- You are not allowed to modify the class and method signatures provided to you. However, you can (and should) implement additional methods and classes as needed in your implementation.
- 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.