Assignment 4

Computer Science 441

Due: 23:55, Friday November 18, 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 UDP-based program for reliable file transfer based on the Stop-and-Wait protocol.

2 Specification

2.1 Overview

In this assignment, you will implement a simplified FTP client based on UDP called `StopWaitFtp`. Since UDP does not provide any data reliability, you will implement your own reliability mechanism based on the stop and wait protocol. The simplified client only supports *sending* a file to the server. 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.

2.2 Handshake

The handshake takes place over a TCP connection and initiated by the client. The host name and port number of the server are provided to your program. Upon start, your program should open a TCP socket (with the provided server name and port number) as well as a UDP socket (using the no-argument constructor `DatagramSocket()`).

Use the TCP socket to exchange information about the file name to be sent, its length, the initial sequence number and UDP port numbers on the client and server. All control messages during the handshake are in binary format. You can use the Java binary stream classes `DataInputStream` and `DataOutputStream` to read from and write to the TCP socket in binary format. Specifically, follow the message sequence presented in Program 1 (in the exact same order) to complete the handshake process:

Program 1 `Handshake Message Sequence`

1. Send the local UDP port number used for file transfer as an `int` value
2. Send the name of the file as a `UTF` encoded string
3. Send the length (in bytes) of the file as a `long` value
4. Receive the server UDP port number used for file transfer as an `int` value
5. Receive the initial sequence number used by the server as an `int` value

Do not forget to `flush()` the output stream to force TCP to send your data to the server at the end of the send section of the handshake.

3 Implementation

3.1 High Level Structure

A high-level description of the internal operation of `StopWaitFtp` is presented in Program 2.

Program 2 `StopWaitFtp`

```
Open a TCP connection to the server
Open a UDP socket
Complete the handshake over TCP
while not end of file do
    Read from the file and create a segment
    Send the segment and start the timer
    Wait for ACK, when correct ACK arrives stop the timer
end while
Close sockets and clean up
```

3.2 Sending File Content

A file name is given to your program as input. Your program should read the file chunk-by-chunk and then encapsulate each chunk in a *segment* for transmission to the server. The class `FtpSegment` is provided to you on D2L. Refer to the Javadoc documentation provided in the source file for more information on how to use this class. Specifically, class `FtpSegment` provides methods for creating segments with a given sequence number and payload, creating a UDP packet that encapsulates a segment, and creating a segment from a UDP packet, as demonstrated in method `FtpSegment.main()`.

The maximum size of a segment payload is given by the constant `FtpSegment.MAX_PAYLOAD_SIZE`. The sequence number for segments starts at the *initial sequence number* received from the server during the handshake process, and is incremented per every segment transmitted. Once a segment is transmitted, you should wait to received an ACK for that segment. Your program should listen for arriving ACK segments from the server using the same UDP socket that is used for sending data segments to the server. The ACK segments received from the server carry the sequence number of the *next expected segment* at the server. That is, if the server receives a segment with sequence number n, then it sends an ACK segment with sequence number $n+1$, indicating to the client to send segment $n+1$ next. The ACK segments are regular segment objects that have no payload.

3.3 Retransmission Timer

As soon as a segment is transmitted, the client should start a retransmission timer. The duration of the timeout interval is given to the client program as an input parameter. Class `Timer` can be used to schedule a recurring timer using method `Timer.scheduleAtFixedRate()`. To use the `Timer` class, you need to define a timer task class as well. A timer task is similar to a `Thread` class. The only difference is that it extends class `TimerTask`.

In your program, you should create one `Timer` object when the program starts. Then use the timer object to start and stop recurring timer tasks. If an ACK arrives for the latest transmitted segment, you should cancel the recurring timer task by calling method `TimerTask.cancel()`. Note that calling the cancel method does not cancel a timer task that is currently executing. While this may result in an unnecessary retransmission of the segment that has been just ACKed, it is an acceptable behavior in this assignment.

You may find it convenient to define two helper methods to start and stop the timer task. When the entire file transmission is complete, make sure to shutdown the timer instance itself. This is achieved by calling `Timer.cancel()` and `Timer.purge()` methods to cancel the timer object and any timer tasks attached to it.

3.4 Handling Concurrency

Since the timer task runs concurrently with the main program, you should be careful when accessing any shared data. In particular, the last transmitted segment may be accessed concurrently by the main program and the timer task (for retransmission), which could lead to memory inconsistencies, *e.g.*, retransmitting a segment by the timer task before it has been fully initialized by the main program. In general, such race conditions can be avoided by protecting access to the shared data using Java `synchronized` blocks. In `StopWaitFtp`, since the shared data is only one segment, a simple solution is to store a copy of the segment (to be transmitted when a timeout happens) in the timer task object itself when creating a timer task for the segment.

3.5 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 by of maximum size.
- Retransmission Timer: Using socket timeout option to handle retransmissions is not allowed, as it could lead to incorrect behavior with duplicate ACKs. Your implementation must use the `Timer` class, as described above. Using a `ScheduledExecutorService` to implement retransmissions is also acceptable.

4 Software Interfaces

4.1 Method Signatures

The required method signatures for class `StopWaitFtp` are provided to you in the source file `StopWaitFtp.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 Testing Your Code

A driver class named `StopWaitDriver` is provided on D2L to demonstrate how we are going to run your code. Read the inline documentation in the source file `StopWaitDriver.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 following aspects of your program:

- How did you implement retransmissions? Explain how you setup the timer and when you start and stop the transmission timer.
- What happens in your implementation if a timeout occurs at the same time that an ACK arrives from the server. Is there going to be a race condition in your program in that case? Explain your answer.

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. Any changes in `StopWaitDriver` and `FtpSegment` classes will be overwritten during marking.
- 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.