COMPX204-20B - Trivial File Transfer

7 September, 2020

1 Introduction

In this assignment, you will implement a client/server system to reliably transfer files using UDP sockets. We have talked about UDP sockets (Java DatagramSocket and DatagramPacket). There are demo code on Week five's slides for you to brush up memory. You will implement a variant of the Trivial File Transfer Protocol (TFTP) which is an open Internet standard published in RFC 1350. The protocol you implement will be slightly simplified from the TFTP protocol. It took me about 200 lines of Java code to implement the ability to send a file from the server, and about the same number for the client.

This assignment will introduce you using the DatagramSocket and DatagramPacket Java classes. Please take some time to look through the Java documentation provided for these classes. You can find the documentation at https://devdocs.io/openjdk~8/

2 Academic Integrity

The files you submit must be your own work. You may discuss the assignment with others but the actual code you submit must be your own. You must fully also understand your code and be capable of reproducing and modifying it. If there is anything in your code that you don't understand, seek help until you do.

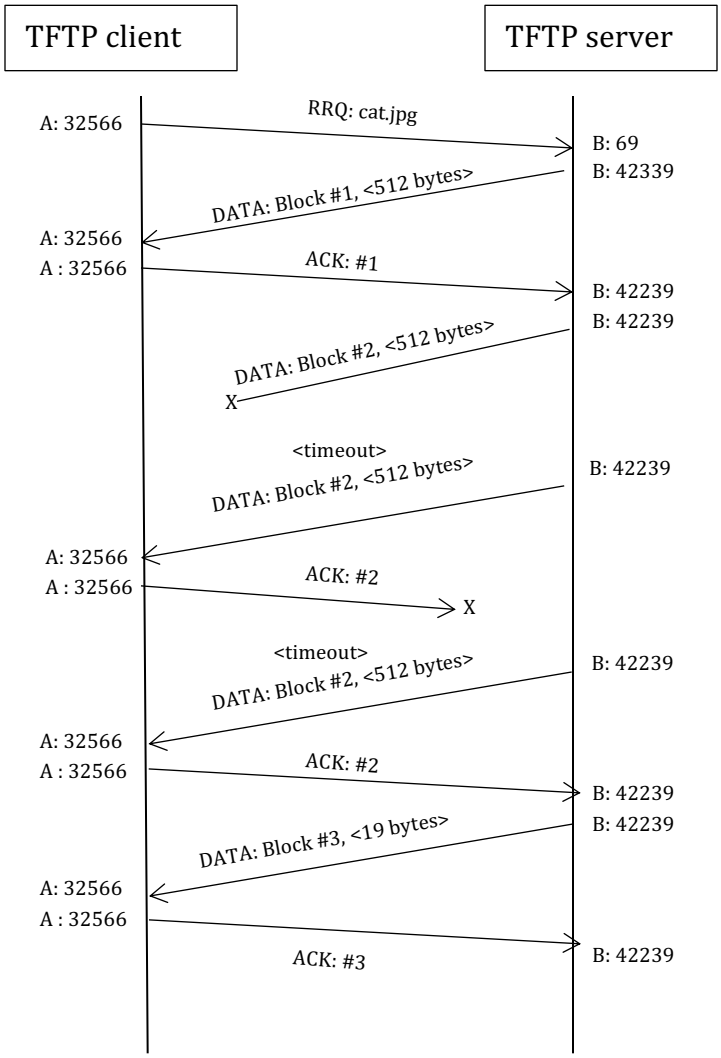
You must submit your files to Moodle before receiving any marks recorded on your verification page.

This assignment is due **7th September by 11am** and worth 8% of your final grade.

3 TFTP

TFTP servers usually listen on port 69, using the User Datagram Protocol (UDP). TFTP servers are widely used as the first stage in network booting to fetch the operating system kernel for a system to boot. In this case, the client opens a UDP socket and sends a read request (RRQ) with the name of the file to fetch. If the file does not exist on the server, the server sends the client an ERROR packet. Otherwise, the server opens a new DatagramSocket bound to a new port, and then sends one DATA packet at a time to the client, each numbered to distinguish this packet from the last. The server waits for the client to ACK the received packet, before sending the next DATA packet. If the server does not receive an ACK from the client after a timeout, the server will re-transmit the DATA packet using the same block number.

The server sends DATA packets containing 512 bytes of data at a time. The client infers that end of file (EOF) has been reached when it receives a DATA packet containing less than 512 bytes of data. If the file is a multiple of 512 bytes in size, the server signals EOF with a packet with zero bytes of data.



The figure shows an example transfer of cat.jpg of 1043 bytes in size from the server B to the client A. A sends the request to port 69 on B; B allocates a new DatagramSocket on an unused port (42239) and sends it back to A who is listening on port 32566 for the DATA. Block #2 is transferred three times, as the first Block #2 is lost on the path to A and the first ACK #2 is lost on the path to B. The server B re-transmits the block until it receives an ACK for it. Note that if the ACK (e.g. the first ACK #2) is lost, the client B must re-send the ACK when receiving a duplicated block (e.g. the third Block #2), but skip over the block (e.g. the third Block #2) which it has already written to disk. For this assignment, the server gives up if it transmits the same block five times and receives no acknowledgment. Block #3 contains less than 512 bytes and signals end of transmission.

3.1 Skeleton code and Server code

I have provided the basic skeleton of a TftpServer implementation to get you started. Or **you can use my implementation of TftpServer** and write your own TftpClient to talk to it. I have also implemented some useful fields/methods in TftpUtil.java. Have a look at how they are used in my TftpServer.java. I have left comments that describe roughly the process you should use in the skeleton of a TftpServer. My TftpServer responds to each TftpClient request using a different thread and implements a retransmission method, in case a data or acknowledgment packet is lost using DatagramSocket::setSoTimeout method.

3.2 Packet Formats

Every packet begins with a 1 byte type field. The defined types are:

- RRQ: 1
- DATA: 2
- ACK: 3
- ERROR: 4

To start with, implement the “Read Request” (RRQ) exchange in both the client and the server. Have the server listen on a port it can bind to (not 69). The RRQ packet (Figure 1) contains a String of the requested filename after the type field:

```
 1 byte       <variable bytes>
+------+------------------------------+
| Type |      Filename                |
+------+------------------------------+
```

Figure 1. Format of RRQ packet.

To obtain a byte array from a String, use the getBytes() method on the String. To generate a String from a byte array, use the appropriate String constructor. Use the DatagramPacket::getLength field to determine the size of the packet, and thus be able to compute the size of the filename, in bytes.

If the file does not exist on the server, send an error packet back. The ERROR packet

(Figure 2.) contains a String that could explain why the request could not be fulfilled, such as because the file does not exist. Print the error message in the client.

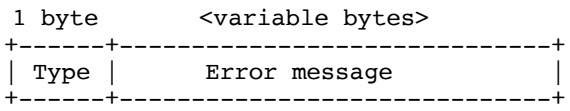


Figure 2. Format of ERROR packet.

If the file does exist, create a new thread in the server which will send the DATA (Figure 3.) back using a new DatagramSocket. Send the first block using block number #1. Put up to 512 bytes of data in each packet: a full size DatagramPacket, including the type and block number, contains 514 bytes of data. Send the DatagramPackets to the IP and port number the client made the request from.

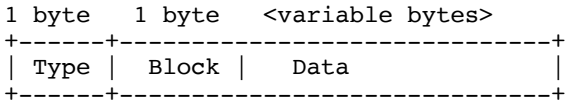


Figure 3. Format of DATA packet.

The client sends an ACK packet (Figure 4.) when it receives a DATA packet: the block number has to be either:

1. the block it is expecting, or
2. the block it received last time.

The second condition occurs if the ACK is lost in the network being carried back to the server and the server re-transmits the DATA packet that has not been acknowledged when time-out.

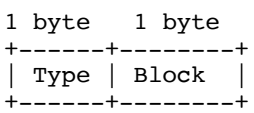


Figure 4. Format of ACT packet.

To test your implementation, transfer a file at least 131,072 bytes in size. Verify that the received file is correct using the *shasum -a 256* command to compare the hash values of the sent and received files.

In the server, use setSoTimeout on the DatagramSocket object to set a 1 second timeout before calling the recv method to wait for an ACK. The DatagramSocket will throw a SocketTimeoutException if it does not receive an ACK in a second.

VERIFICATION PAGE

Name: _____

Id: _____

Date: _____

Note: this assignment is due **7 September by 11am** and worth 8% of your final grade. **You must submit your code to the Moodle assignment page before you have had the assignment verified.** Do not zip your source code. Submit all the individual *.java files.

1. Explain to the demo the structure of your TftpClient programs and show that your TftpClient sends an ACK packet when it receives a DATA packet whose the block number is the block it is expecting, or the block it received last time.

2. Demonstrate that your TftpServer and TftpClient programs transfer randomly generated large data files that are a multiple of 512 bytes in size correctly:

   ```
   dd if=/dev/urandom bs=512 count=400 of=test.file
   ```

   Transfer `test.file`, and then ensure the file was received correctly using:

   ```
   shasum -a 256 test.file
   ```

3. Demonstrate that your TftpClient programs interoperate with my implementation of TftpServer.