ECSE 4670: Computer Communication Networks
Fall 2021

Programming Assignment

Due: *Mon, Nov 22*

In this multi-part assignment, we will ultimately build a simple reliable file transfer application **over UDP**. However, as a buildup towards that, we will first implement a 'ping' application over UDP, in Part A of the assignment. This *UDP Pinger* application will help you familiarize yourself with UDP sockets and develop a simple request-response protocol with timeouts. In Part B of the assignment, we will use this knowledge to develop and implement a **unidirectional stop-and-wait** file transfer protocol over UDP.

NOTE: *We strongly prefer that you write your code in either Java or Python. Accordingly, we have provided some baseline code (PingServer.java and PingServer.py – more on those later in this document), adopted from textbook (Kurose and Ross).*

In the description below, it is assumed that you will be programming in Java. If you are coding in Python, all the discussion still applies to you – you only have to replace the commands for compiling and running Java code with the equivalent Python commands.

Before starting, read the section on "Socket Programming with UDP" from Chapter 2 of the textbook. The last few editions of the textbook describes socket programming using Python; the older editions describes socket programming using Java. You should be able to find many good resources on UDP socket programming in either of these languages on the web.

Part A: UDP Pinger

In this part, we will do the *UDP Pinger Lab* assignment (altered slightly) of the textbook (Kurose and Ross), which is one of the socket programming assignments in Chapter 2 of the textbook. The lab is accessible on the textbook Companion Website, using the access code that you will find in the textbook. In this assignment, you will study Java code for a simplified Internet ping server, and write the corresponding client.

The ping application:

The ping application allows a client machine to send a packet of data to a remote machine, and have the remote machine return the data back to the client unchanged (an action referred to as echoing). Among other uses, the ping protocol allows hosts to determine round-trip times to other machines. We will discuss more on ping when we study ICMP messages (ping uses ICMP ECHO request and ECHO response messages) in Chapter 4 (network layer). You should try out

ping <server-name>

where <*server-name*> is the name of out favorite server, say www.google.com. Note that some servers may not respond to 'ping' request. For example, try pinging www.ecse.rpi.edu (from outside RPI) and see what happens.

The simple ping server-client that you will implement provides a functionality that is similar to the standard ping programs available in modern operating systems, except that you will use UDP rather than Internet Control Message Protocol (ICMP) to communicate with each other.

Server code:
The server code is available as *PingServer.java* (a python version, *PingServer.py*, is also provided, in case you prefer to code in python), which provides a full implementation of the ping server. You need to compile and run this code. You should study this code carefully, as it will help you write your ping client. As you study the server code, you will notice that the server sits in an infinite loop listening for incoming UDP packets. When a packet comes in, the server simply sends the encapsulated data back to the client.

Note that UDP provides applications with an unreliable transport service, because messages may get lost in the network due to router queue overflows or other reasons. However, since packet loss is rare or even non-existent in typical campus networks, this server code injects artificial loss to simulate the effects of network packet loss. The server has a parameter LOSS_RATE that determines which percentage of packets should be lost. The server also has another parameter AVERAGE_DELAY that is used to simulate transmission delay from sending a packet across the Internet. You should set AVERAGE_DELAY to a positive value when testing your client and server on the same machine, or when machines are close by on the network. You can set AVERAGE_DELAY to 0 to find out the true round-trip times of your packets.

Compiling and Running the Server:
To compile the server, do the following:
javac PingServer.java

To run the server, do the following:
java PingServer <port-number>
where <*port-number*> is the port number the server listens on. Remember that you have to pick a port number greater than 1024, because only processes running with root (administrator) privilege can bind to ports less than 1024.

Note: if you get a "class not found" error when running the above command, then you may need to tell Java to look in the current directory in order to resolve class references. In this case, the command will be as follows:
java -classpath PingServer <port-number>

What to do:
In part A of the assignment, your job is to write the ping client code, *PingClient.java*, so that the client sends 5 ping requests to the server, separated by approximately one second. Each ping message consists of 56 bytes of data. As in the actual ping application, these 56 data bytes can be anything, and can be the same for all ping messages. After sending

each packet, the client starts a timer and waits up to one second to receive a reply. You will notice in the server code that the ping message (the 56 bytes of data in each UDP packet) is simply copied into the reply message. Once the reply is received, the client stops the timer and calculates the round trip time (rtt). If one second goes by without a reply from the server, then the client assumes that its packet or the server's reply packet has been lost in the network. For this purpose, you will need to research the API for DatagramSocket to find out how to set the timeout value on a datagram socket.

Your client should start with the following command:
java PingClient <host-name> <port-number>
where <*host-name*> is the name of the computer the server is running on, and <*port-number*> is the port number it is listening to. Note that you can run the client and server either on different machines or on the same machine.

When developing your code, you should run the ping server on your machine, and test your client by sending packets to localhost (or, 127.0.0.1). After you have fully debugged your code, you should see how your application communicates between two machines across the network.

Message Formatting:
The ping messages in this part are to be formatted in a simple way. The client prints on the screen a one-line message corresponding to each ping request. If the client receives a response for a ping request, it prints the following line:
PING <server-ip-address> <sequence-number> <rtt-estimate>
where <*server-ip-adress*> is the IP address of the server in dotted decimal format, <*sequence-number*> starts at 0 and progresses to 4 for each successive ping message sent by the client, <*rtt-estimate*> is an estimate of the round-trip time between the client and the server, calculated as the difference between the time a ping response is received and the time the request was sent (the timestamp in the response packet). The <*rtt-estimate*> is expressed in millisecs, showing up to 3 decimal places (microsec accuracy). Note that this RTT estimate includes the AVERAGE_DELAY value that is introduced artificially, and in general may not be a good estimate of the actual RTT.

If the client does not receive a response to a ping request within one second of sending it, it prints the following line:
PING <server-ip-address> <sequence-number> LOST

Part B: Stop-and-Wait File Transfer

In this part, we will develop a simple unidirectional file transfer application over UDP. Our goal would be to make it as simple as possible, while ensuring correctness/reliability of the file transfer. It will be built along the lines of the Stop-and-Wait reliable transfer protocol (Rdt 3.0) that we studied in Chapter 3. The requirements that the application (which we will call *sftp*) must satisfy are described below.

The sftp application requirements:
The data transfer will be unidirectional – from the client to the server, although the acknowledgements would need to be sent from the server to the client. To transfer a file,

the user (client) should call *sftpClient <server_ipaddress>* where *server_ipaddress* is the IP address of the server in dotted decimal format. The stop-and-wait version of the protocol that you will implement has some similarities with the Trivial FTP (or TFTP) protocol, that also runs over UDP.[1] In a way, the *sftp* application that you will implement can be viewed as a simpler version of TFTP.

- When called, *sftp* will read a file called *inputfile* from the local directory, and transfer file in packets (**each of which should contain 512 bytes of data, except possibly the last packet**) using UDP sockets.
- The server will reassemble the data in the packets into the file and write it as *outputfile* in the current working directory (an existing file with the same name will be rewritten).
- There is no explicit connection setup and closing (unlike TCP). The first packet containing file data implicitly "sets up the connection", so to speak. The end of the file is indicated by the transfer of a packet with less than 512 bytes of data; this also implicitly "closes the connection", so to speak. So, no SYN or FIN flags are needed. (What if the file size is an integral multiple of 512 bytes? Well, then the client sends an additional packet with just 0 bytes of data to indicate the end of file!) Note that this is similar to the way TFTP implicitly sets up/closes the "connection", and indicates the end of the file.
- Like Rdt3.0, you can use only **one-bit (0-1) sequence and acknowledgment numbers**. You can allocate a full byte for it, for convenience, but the value of that byte can only be 0 or 1. Thus the data part of each UDP packet sent from the client to the server can be 513 bytes (1 byte header just indicating the 0-1 sequence number, plus the 512 bytes of file data), except possibly for the last packet. The data part of each UDP packet sent from the server to the client can just be the 1 byte header just indicating the 0-1 acknowledgment number. You will assume that no bit errors happen in either forward or backward direction, so no checksum needs to be included in any packet.
- The client can use any ephemeral port number, but the server must listen on port number 9093. The client's *retransmission time*r should be set to 1 sec. The *retransmission limit*, of the maximum number of times that the client will attempt retransmitting a packet on timeout, should be set to 5, i.e., a total of six transmission attempts per packet.
- Note that for *sftp*, you need to write both client and server (no code will be provided), *sftpClient.java* and *sftpServer.java*. However, feel free to borrow inspiration and code that you wrote for the ping application in Part A of this

[1] RFC 1350: The TFTP Protocol, https://tools.ietf.org/html/rfc1350. TFTP is meant for file transfer is systems which are too resource constrained to run TCP. TFTP can however be inefficient for large file transfers, due to its stop-and-wait nature.

assignment! You server should implement a LOSS_RATE and AVERAGE_DELAY, as in the ping application in part A.

- You should test your code for different file sizes, but we ask you to report the results (see "Deliverables" below) for a fixed file size of 50 Kbytes.
- You need to time the file transfer, and provide that in the output. For that the client can start a timer just before it starts sending the file data, and stop it when the entire file is transferred. If the file is transferred successfully, the client prints the following line:

  sFTP: file sent successfully to <server-ip-address> in <time in secs> secs

  If the file transfer fails due to retransmission limit being reached for some packet, the client prints the following line:

  sFTP: file transfer unsuccessful: packet retransmission limit reached
- Make sure you compare *inputfile* and *outputfile* (bitwise comparison, not just the size) to make sure the file is transferred correctly from the client to the server.

Deliverables:

You are required to upload on LMS the following:

Part A:

- Client code for the ping application, *pingClient.java*
- A README file (Ascii), *pingREADME*, that discusses the tests you have conducted, and some sample outputs to demonstrate that it is working correctly, both under no packet losses and delays, as well as with packet losses (say 20%) and delays (say 100 msec).

Part B:

- Client and server codes for the sftp application, *sftpClient.java* and *sftpServer.java*
- A README file (Ascii), *sftpREADME*, that discusses the tests you have conducted, and some sample outputs to demonstrate that it is working correctly.
- In a single Word or PDF document, provide two figures (plots) of the following experiments (with necessary explanation), as follows:
  - Fix the AVERGAGE_DELAY to 100 msec, and plot the *file transfer time* vs LOSS_RATE, by varying the LOSS_RATE as 0%, 5%, 10%, 15%. Compute each data point (that you will plot) as the average of 10 runs.
  - Fix the LOSS_RATE to 5%, and plot the *file transfer time* vs AVERAGE_DELAY, by varying the AVERAGE_DELAY as 50 msec, 100 msec, 200 msec, 400 msec.

Grading:

The assignment will be graded out of 10; there will be 4 points for Part A, and 6 points for Part B.

Approximately 80% of the points will be on correctness, 10% on compactness of the code, and 10% on proper commenting.