CSEE 4119: Computer Networks
Spring 2016

Assignment 5: Simple TCP-like transport layer protocol (updated 04/11/2016)

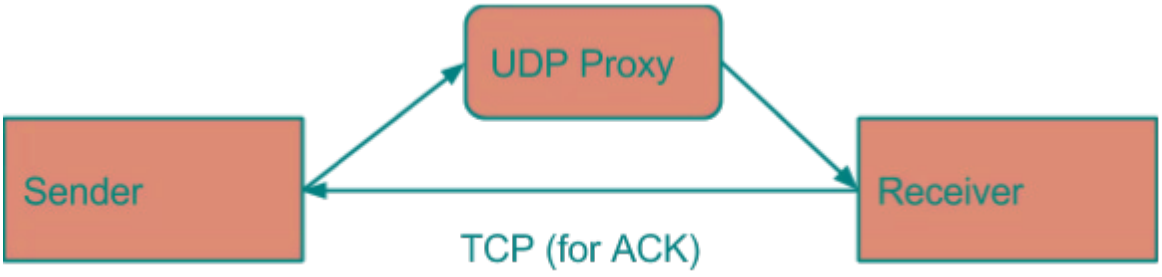
Academic Honesty Policy

You are permitted and encouraged to help each other through Piazza's web board. This only means that you can discuss and understand concepts learnt in class. However, you may NOT share source code or hard copies of source code. Refrain from sharing any material that could cause your source code to APPEAR TO BE similar to another student's source code enrolled in this or previous years. Refrain from getting any code off the Internet. Source code should be yours and yours only. Do not cheat.

Introduction

In this programming assignment, you will implement a simplified TCP-like transport layer protocol that supports both IPv4 and IPv6. Your protocol should provide reliable, in order delivery of a stream of bytes. It should recover from in-network packet loss, packet corruption, packet duplication and packet reordering and should be able cope with dynamic network delays. However, it doesn't need to support congestion or flow control.

To test your implementation, you will use a link emulator provided by us: see http://www.cs.columbia.edu/~hgs/research/projects/newudpl/newudpl-1.4/. (We will provide Ubuntu binaries shortly.) Data is exchanged via UDP, i.e., you will be running your TCP version "on top of" UDP. In the real world, a network can drop, corrupt, reorder, duplicate and delay packets. To mimic this behavior, you will use the link emulator. Specifically, you will invoke the emulator so that it intercepts packets on the forward path, whereas the acknowledgements (i.e., the packets on the path from receiver to sender) will be sent directly from the receiver to the sender, i.e., you can assume that packets arrive without a loss on this path.



You can run your sender, receiver and link emulator either on one, two or three machines. The link emulator acts like a "proxy", i.e., the sender sends packets towards the emulator and the emulator should forward the packets to the receiver. The receiver process should send its acknowledgements directly to the data sender.

You can use either C, C++, Java or Python your implementation, using standard UNIX or Java datagram socket calls (for brevity we refer only to UNIX network APIs in the assignment description).

Usage scenario and functionality requirements

You will implement a one way ("simplex") version of TCP. Your program has to handle only one set of packets, i.e., the delivery of a single file. Your implementation will be composed of two separate programs, sender and receiver. The sender reads data from a file and uses the sending services of the TCP-like protocol to deliver it to the remote host. The receiver uses the receiving services of the TCP-like protocol to reconstruct the file.

The TCP receiver should be invoked as follows:

`receiver <filename> <listening_port> <sender_IP> <sender_port> <log_filename>`

```
% receiver file.txt 20000 128.59.15.37 20001 logfile.txt

% Delivery completed successfully
```

The receiver receives data on the `listening_port`, writes it to the specified file (`filename`) and sends ACKs to the remote host at IP address (`sender_IP`) and port number (`sender_port`). In the above example the ACKs are sent to the sender (`128.59.15.37`) with an ACK port 20001. The IP address is given in dotted-decimal notation, the equivalent IPv6 or a host name and the port number is an integer value. The receiver will log the headers of all the received and sent packets to a log file (`log_filename`) specified on the command line. The log entries should be ordered according to the timestamps of the packets, from lowest to highest, and should be displayed to the standard output if the specified log filename is "stdout". The output format of a log entry should be as follows.

```
timestamp, source, destination, Sequence #, ACK #, and the flags
```

The receiver should indicate whether the reception was successful, and report file I/O errors (e.g., 'unable to create file').

The data sender should be invoked as follows:

```
% sender <filename> <remote_IP> <remote_port> <ack_port_num> <log_filename>
<window_size>

% sender file.txt 128.59.15.38 20000 20001 logfile.txt 1152

Delivery completed successfully
Total bytes sent = 1152
Segments sent = 2
Segments retransmitted = 0 %
```

The sender sends the data in the specified file (`filename`) to the remote host at the specified IP address (`remote_IP`) and port number (remote_port). In the above example the remote host (which can be either the receiver or the link emulator proxy) is located at 128.59.15.38 and port 20000. The command line parameter `ack_port_num` specifies the local port for the received acknowledgements.

As before, a log filename is specified. The log entry output format should be similar to the one used by the receiver, however, it should have one additional output field (appended at the end), which is the estimated RTT. At the end of the delivery the sender should indicate whether the transmission was successful, and print the number of sent and retransmitted segments. The sender should report file I/O errors (e.g., 'file not found').

The `window_size` is a parameter and window_size is measured in terms of the number of packets. Your sender should support variable window size, and it will be specified as the last parameter on the command line. If no parameter is specified, the default window size value should be 1.

Requirements

- You will implement a one-way version of TCP without the initial connection establishment, but with a FIN request to signal the end of the transmission.
- Sequence numbers should start from zero.
- You do not have to worry about congestion or flow control You should adjust your retransmission timer as per the TCP standard, although it may be advisable to use a fixed value for initial experimentation.
- Both the data and the ACK should implement the full TCP header. The checksum should be validated for both data and ACK segments.
- All segments except the last part of the file can be the same size. (Supporting varying-length segment below the MSS is obviously acceptable.)
- You need to implement both IPv4 and IPv6. However, the link emulator may only support IPv4, so you can test IPv6 by directly connecting sender and receiver.
- You must support binary files containing any byte value, including zero.
- The total bytes sent should include headers and retransmissions, i.e., file size plus retransmissions plus all packet headers sent (without ACKs). The segments should also include retransmissions.
- You need to implement the 20-byte TCP header format, without options. You do not have to implement push (PSH flag), urgent data (URG), reset (RST) or TCP options. You should set the port numbers in the packet to the right values, but can otherwise ignore them. The TCP checksum is computed over the TCP header (with the checksum field assumed to be all zero) and data; this does not quite correspond to the correct way of doing it (which includes parts of the IP header), but is close enough.
- **Do not change the sequence/syntax of the input** parameters on the command line.
- The IAs will compile and test the code on a CLIC-lab machines (Ubuntu 14.04). It is your responsibility to ensure that the code compiles and runs. (Note that there are two kinds of CLIC machines, with different operating system versions.)

Tips

- You should test your programs with and without the link emulator. When using the link emulator, specify its IP address and port number in the command line of the data sender.
- The link emulator can be run on Linux, either a host or a VM.
- Test your program over a wide range of network settings.
- Use file diff on the file sent and received to make sure they are the same.
- You can choose a reasonable value for the maximum segment size, e.g., 576 bytes.

Submission guidelines

Please submit your assignment to Courseworks. Please submit a single file, named <UNI>_<Language>.zip (e.g., zz1111_java.zip). The zip file should include the following:

- README.txt. This file contains the project documentation; program features and usage scenarios; a brief description of (a) the TCP segment structure used (b) the states typically visited by a sender and receiver (c) the loss recovery mechanism; and a description of whatever is unusual about your implementation, such as additional features or a list of any known bugs.
- Your source files, with ample comments. The sender and receiver “main” files should be called something like `sender.java` and `receiver.java`.

- For C/C++ and Java: A `make` file that compiles your code.
- The data file you used for testing.
- The log file of at least one run.

Evaluation

	Issue	Deductions
1	**Compilation and execution related**	
	Compilation fails	-100
	Core dump or similar fatal error	-80
	Sender not invoked in given format	-5
	Receiver not invoked in given format	-5
	No easy to read documentation / make file	-3
2	**Functionality**	
	No IPv6	-10
	No domain names	-10
	In order delivery not handled	-10
	Packet loss not handled	-10
	Corrupted packets not handled	-10
	Packet delays not handled	-10
	Duplicate packets not handled	-10
	Log file not maintained on sender side	-5
	Log file not maintained on receiver side	-5
	Log file on sender does not follow given format	-5
	Log file on receiver does not follow given format	-5
	Statistics on sender not printed properly	-5
	File on receiver is different than what sender sent	-10

	Variable window size not supported	-10
	Any errors in above functions	-1 to -15 per error (TA discretion)
3	**Extra features**	Max +20
	Up to 10 points per extra feature (at TA discretion)	

Additional references

TCP/IP Illustrated, Vol 1, by W. Richard Stevens. An excellent guide to the TCP/IP protocol.

'Beej's Guide' which can be found at http://www.beej.us/guide/bgnet/. An online tutorial for socket programming.

Unix man pages for socket, bind, sendto, recvfrom.

Proxy references

http://www.cs.columbia.edu/~hgs/research/projects/newudpl/newudpl-1.4/newudpl.html ○
http://www.cs.columbia.edu/~hgs/research/projects/newudpl/