Workshop in Communication Networks

Exercise 3 – Layer 4 – Simple TCP

General Guidelines:

Submission deadline is **Sunday, June 8, 23:55 moodle server time**
Submit your answers as TXT and Python files, packed into a single ZIP file.
Pack your files in a ZIP file named ex3-YourName1-YourName2.zip,
for example: ex3-LoisLane-ClarkKent.zip
Post the ZIP file in the submission page in course website

- No late submission will be accepted!
- Document your code. Place your names at the top of your source file, as well as in the TXT file.
- Make sure your files have EXACTLY the requested file names.
- We may give a bonus for exceptional works. However, do not make anything more complex than required.

The grading for this exercise will be mainly automatic. If your code does not work as expected, the grade will be very low.

Your submission ZIP file should contain the following files:

- Code – named **stcp.py** (and also **mysock.py** if you change it, see Appendix B)
- A text file named **description.txt** that *briefly* describes what you implemented, and how.

Your code must include log messages for a small set of important events. See the complete list in Appendix A. This will help you track your socket, and will help us grading it.

Useful links for this exercise:
TCP | RFC 793 | UDP | Python Structs | Python Socket | Python Threading

Introduction

In this exercise we will implement Simple TCP – a simplified version of the Transmission Control Protocol (TCP). TCP is a relatively complex protocol that promises reliable data transfer over unreliable channels, in addition to flow control and congestion control mechanisms.

TCP works over the IP protocol. However, Simple TCP works on top of the UDP protocol. Specifically, we will use a modified version of the UDP socket implementation of Python to send and receive messages. This modified version of the socket is simple to use and can be tuned to simulate a real unreliable channel with packet loss and packet reordering.

The boundaries of this exercise are weakly defined: you must complete some basic requirements in order to get 90. Then, the rest of the features are optional. The exercise will be graded relatively to others in class. The more you implement – the higher the grade you are likely to get. A grade higher than 100 is possible for exceptional works.

Steps:
1. Implement a Simple TCP socket with the TCP state machine (required)
2. Support multiple connections using demultiplexing (required)
3. Implement acknowledgement mechanism (optional)
4. Implement retransmission mechanism (optional)
5. Implement packet reordering (optional)
6. Add any additional feature of TCP to your Simple TCP implementation (optional)

<u>Attached Files</u>

You are provided with the following files:

1. mysock.py – A modified UDP socket implementation on which you should build your own Simple TCP socket.
2. Logger.py – A logger utility for your use.
3. Server.py, Client.py – Two test applications.
4. stcp.py – A skeleton file for your work with some already-implemented parts.

<u>Assumptions</u>

You may assume that:

1. The only control bits that are used are: SYN, ACK, FIN.
2. Control messages do not contain data (i.e. if one of the control bits is on, packet contains no data)
3. No checksum computation or validation is required.

It is recommended that you read Section 3 of RFC 793 before starting to work on this exercise.

<u>Part 1 – Implement a Simple TCP socket with the TCP state machine</u> (70 points)

Open stcp.py. It contains some predefined constants (specifically, see the STCP_STATE_* constants for your use when implementing the state machine), a class that represents a Simple TCP packet, a class that represents a socket exception (for your use in case of an error), and a class named stcp_socket that you should implement.

The stcp_socket class defines the following methods:

- Constructor
  Initializes the socket. Sets socket state to STCP_STATE_CLOSED. Initializes the underlying mysock object if no base socket is given (otherwise, uses the mysock object from the base socket).

- bind(ip, port)
  Binds the port to the given local IP and port. Also binds the underlying mysock socket to the same IP and port.

- listen()
  Sets socket state to STCP_STATE_LISTEN (if possible).

- accept()
  Accepts an incoming connection. This method blocks until an incoming SYN message arrives, handles the SYN message and returns the socket.

- connect(ip, port)
  Connects to a remote Simple TCP listening socket at the given IP and port.
  This method should bind the socket to some local port and send start the handshake process with the remote socket by sending a SYN message.

- send(data)
  Sends the given data to the remote socket.
  A Simple TCP message should not be longer than STCP_MSS as defined in stcp.py. Thus, if the data is longer, it should be broken into multiple consecutive Simple TCP packets.
  Note: If the socket is not ready (is in the process of handshaking) this method should stall shortly and retry.

- recv(length)
  Receives data from remote socket. The length of returned data is given in the length parameter. This method should fetch data until it has enough data to return. If the amount of fetched data is larger than length, it should remember the leftovers for future invocations. If this method fetches a control message, it should make sure it is handled.
  Note: If the socket is not ready (is in the process of handshaking) this method should stall shortly and retry.

- close()
  Closes the socket.
  See the CLOSE steps in the TCP state machine in page 23 of RFC 793.

Note that you should probably add more methods and logic to handle the three-way handshake initiated by the SYN message sent upon connecting. You may use multithreading for that if you want to (this is not necessary though).
You should always verify that the state of your socket allows the requested operation and if not, raise an exception.

Part 2 – Support multiple connections using demultiplexing (20 points)
A TCP server allows multiple connections on the same incoming TCP port. However, an incoming message is being directed to the socket of the correct application and is not handled by the socket that is actually bound to the port the message was sent to.
For example, a web server listens on port 80. Once a client connects, a new socket is created, with some random local port, which is not 80. However, the client will continue to send messages to port 80, and these should arrive at the new socket.
This is done by *demultiplexing*: server uses the four-tuple (source IP, destination IP, source port, destination port) to determine the destination socket of an incoming packet. That is, a packet to port 80 may not go to a socket that is bound to local port 80.

Change the implementation of accept() as follows:
Upon arrival of a SYN message, create a new socket based on the current one (that is, uses the same underlying socket), assign it a new port number, handle the SYN message and return the new socket.

Add a mechanism that all sockets with the same underlying socket use, that is responsible to fetch messages and add them to the queue of the correct socket, based on the four-tuple (source IP, destination IP, source port, destination port). If a four-tuple is unknown, then it means that the message should go to the base socket (actually, only if it's a SYN message).

By the end of this part the attached test code should work. You should be able to run multiple clients that connect to the same server.

Part 3 – Implement acknowledgement mechanism (Optional)
TCP sends an ACK message to acknowledge the successful receiving of a packet (or a sequence of packets). In this part you are expected to send an ACK message for each received packet. You may want to send cumulative ACK for more than one packet, if those arrive together or within a short while.

Part 4 – Implement retransmission mechanism (Optional)
Given that you implemented part 3, you can now track the arrival of packets you sent. Implement a mechanism that retransmits packets if they were not ACKed after more than 3 seconds.
Tip: Store a window of recently sent packets, in the order you sent them. Once you receive an ACK, remove all packets in the window that this ACK fully acknowledges. Use Python's

threading.Timer class (or take the timer from utils.py of previous exercises) to create a timer that resends packets in the window.

Part 5 – Implement packet reordering (Optional)
Another aspect of reliable data transport is the order in which packets arrive. Packets may take different paths from source host to destination, and thus may arrive in a different order than they were sent in.
Implement a mechanism to reorder arriving packets, according to their sequence numbers. Specifically, your code should:

- Store a window of incoming packets (this window may be limited in size of at least 4*STCP_MSS, or unlimited)
- Once a packet arrives, store it in the window of the correct socket (according to the demultiplexing decision from Part 2)
- Release packets from the window only if all preceding packets were already processed or if they are being released at the same time. That is, the recv() operation should now interact with the window and should not see a packet if a preceding packet is yet to arrive. You can know the first sequence number you should expect from the SYN message and from this point on, track sequence numbers to avoid skipping delayed packets.
- You may improve your ACKing mechanism from Part 3 by ACKing multiple packets at once more easily now that you have a window for received packets.

Part 6 – Implement additional features (Optional)
TCP has more features that are not listed in this exercise for Simple TCP. However, you may implement some of them to increase your grade.
The most important features are of course flow control and congestion control, but any additional feature will get some points, depending on how important it is and how complex its implementation is (and how good you implement it).

Appendix A – Required Logging

The following events must print some organized log messages in the DEBUG log level (using `log.debug(`*`message`*`)`):

Part 1:
- Every TCP state change (print previous and new states, can be in numbers)
- Every SYN/FIN received
- Every ACK sent

Part 3:
- Every ACK sent

Part 4:
- Every retransmission – print the number of packets retransmitted

Appendix B – mysock

mysock is a simple UDP socket implementation that is based on Python's original UDP socket. In addition to regular UDP socket behavior it can simulate packet loss and packet reordering.

Take a look at the code in mysock.py to understand it. You may change the code only in some parts as described below. A short explanation of each method is provided below:

- Constructor
  Initializes the UDP socket (with timeout of 3 seconds. You may change it and submit the updated mysock.py if necessary).

- bind(local_addr, local_port)
  Binds the socket to the given local address and port.

- sendto(data, remote_addr, remote_port)
  Sends the given data over UDP to the given address and port.

- recv()
  Receives the next UDP packet. Packet may get lost with a probability of MYSOCK_LOSS_PROB, and if not, it may be out of order with a probability of MYSOCK_REORDER_PROB.
  Packet is returned as a mysock_msg object as defined in mysock.py.

- close()
  Closes the UDP socket.