CS 352 Spring 2019 Programming
Project Part 3

1. Overview:

For part 3 of the project, your team will continue to improve the socket library you have developed in part 1. For this part, you will add windowing to the messages sent by your socket library. The client and server have been modified to use random packet sizes for sending and receiving.

As part of the project part 1, you will be given a number of files. You can also find them in the sakai site under "Resources" -> "Project resources" -> "Part 3" .

1. **`client3.c`** : This is the original client source code. file. You may not alter the code for this file. It must run using your `sock352.py` file.

2. **`server3.c`** : This is the original server file You may not alter the code for this file. It must run using your your sock352.py file.

3. **`sock352.py:`** This is a new library for part 3. You must fill in the methods defined in this file, as below.

Your library must implement the following methods as defined in the `sock352.py` file:

```
def init(UDPportTx,UDPportRx):
def readKeyChain(filename):
def __init__(self):    def
bind(self,address):    def
connect(self,*args):    def
listen(self,backlog):    def
accept(self,*args):
    def close(self):
def send(self,buffer):
def recv(self,nbytes):
    def readKeyChain(filename):
```

3. The 352 RDP v1 protocol:

Recall as in TCP, 352 RDP v1 maps the abstraction of a logical byte stream onto a model of an unreliable packet network. 352 RDP v1 thus closely follows TCP for the underlying packet protocol. A connection has 3 phases: Set-up, data transfer, and termination. 352 RDP v1 uses a much simpler timeout strategy than TCP for handling lost packets.

Packet structure:

The CS 352 RDP v1 packet as defined as:

< ----------32 Bits ---------- >			
Version	Flags	Option	Protocol
Header Length		Packet Checksum	
Source Port			
Destination Port			
Sequence Number			
Acknowledgement Number			
Receiver's Window			
Payload Length			

The flags field is defined as:

< ----------8 Bits ---------- >							
			Has Option	RESET	ACK	FIN	SYN

Connection Set Up:

The client initiates a connection by sending a packet with the SYN bit set in the `flags` field, picking a random sequence number, and setting the `sequence_no` field to this number. If no connection is currently open, the server responds with both the SYN and ACK bits set, picks a random number for it's `sequence_no` field and sets the ack_no field to the client's incoming `sequence_no+1`. If there is an existing connection, the server responds with the `sequence_no+1`, but the RST flag set.

Connection additions for part 3: Window sizes

Since we are doing flow control, the receiver must keep a buffer of received data, every time recv is called, the specified amount of data is returned and removed from the buffer. The buffer is of size W, W=32kb for this project. If the buffer is empty and recv is called, wait until there is something to return. Alternately, if there is no more room in the buffer, send() calls by the sender should not return until there is room. This is kept track of by the widow size field as in real TCP. Whenever you send an ack, include the room left in your buffer at the moment.

Additionally, you should use a simplified congestion window. Start at two packets and increase

exponentially (double each time). If a packet is dropped (timeout) go back to two. Remember that you may not send more than the receive window no matter the congestion window.

Data exchange:
352 RDP follows a simplified Go-Back-N protocol for data exchange, as described in section Kurose and Ross., Chapter 3.4.3, pages 218-223 and extended to TCP style byte streams as described in Chapter 3.5.2, pages 233-238.

When the client sends data, if it is larger than the maximum UDP packet size (64K bytes, minus the size of the sock352 header), it is first broken up into segments, that is, parts of the application bytestream, of up to 64K. If the client makes a call smaller than 64K, then the data is sent in a single UDP packet of that size, with the `payload_len` field set appropriately. Segments are acknowledged as the last segment received in-order (that is, go-back-N). Data is delivered to the higher level application inorder based on the `read()` calls made. If insufficient data exists for a `read()` call, partial data can be returned and the number of bytes set in the call's return value.

Not that just like TCP, the ACK field is set for each data packet.

Timeouts and retransmissions:
352 RDP v1 uses a single timer model of timeouts and re-transmission, similar to TCP in that there should be a *single timer per connection*, although each segment has a logical *timeout*. The timeout for a segment is 0.2 seconds. That is, if a packet has not been acknowledged after 0.2 seconds it should be re-transmitted, and the logical timeout would be set again set to 0.2 seconds in the future for that segment. The timeout used for a connection should be the timeout of the oldest segment.

There are two strategies for implementing timeouts. One approaches uses Unix signals and other uses a separate thread. These will be covered in class and recitation.

Connection termination:
Connection termination will follow a similar algorithm as TCP, although simpified. In this model, each side closes it's send side separtately, see pages 255-256 of Kurose and Ross and pages 39-40 of Stevens. In version 1, it is OK for the client to end the connection with a FIN bit set when it both gets the last ACK and `close` has been called. That is, `close` cannot terminate until the last ACK is received from the server. The sever can terminate the connection under the same confitions.

If the socket receives an FIN from the other side, and it's data buffer is empty, the socket can be closed after a timeout of 5 seconds.

3. Grading:

Functionality: 80% (windowing with random packet sizes – client3.py and server3.py)
Style: 20%
NOTE: I will not be testing encryption just regular send and recv as in the attatched files.

Style:
Style points are given by the instructor and TA after reading the code. Style is subjective, but will be

graded on a scale from 1-5 where 1 is incomprehensible code and 5 means it is perfectly clear what the programmer intended.

4. What to hand in

You must hand in a single archived file, either zip, tar, gzipped tar, bzipped tar or WinRAR (.zip, .tar, .tgz, .rar) that contains: (1) README.TXT file with your team members, (2) the cllient1.py, client2.py and client3.py source code, (3) the server1.py, server2.py and server3.py source code, (4) **a single sock352.py which works for all clients and servers** and (5) any other files your library needs to work.

Your archive file must include a file called "README.TXT" that includes the names of the project partners for the project!