CS 352 Spring 2017
Programming Project Part 1

1. Overview:

For part 1 of the project, your team will implement a simple go-back-N protocol similar to TCP. This protocol is called the 352 Reliable Data Protocol (RDP) version 1 (352 RDP v1). You will realize it as a Python (version 2) module that uses UDP as the underlying transport protocol. Later versions will add security, port spaces, and concurrency.

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

1. **`sock352.py`** : This is a skeleton definition of the class and methods you need to write. You should modify this file with your implementation. That is, fill in the methods with your own code.

2. **`client1.py`** : A Python client program that uses CS 352 sockets. You may not alter the source code for this file.

3. **`server1.py:`** A Python server program that uses CS 352 sockets. You may not alter the source code for this file.

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

```
init(udp_port1, udp_port2)
socket()
bind(address)
connect(address)
listen(backlog)
accept()
close()
send(buffer)
recv(numBytes)
```

These function map to the existing Python methods for sockets. See this link: https://docs.python.org/2/howto/sockets.html for the definitions of these functions. The one exception `init()` call. This call takes a single parameter, which is the UDP port that the rest of the CS 352 RDP library will use for communication between hosts. Setting the udp_port to zero should use the default port of 27182.

For part 1 of the project, you will only need to make a *single connection* work over a *single port* for a *single thread.* The goal is to correctly implement a go-back-N protocol for one connection, for example, when sending a single file between a client and server. Later versions of the project will build on part to add port-spaces and handle multiple simultaneous connections.

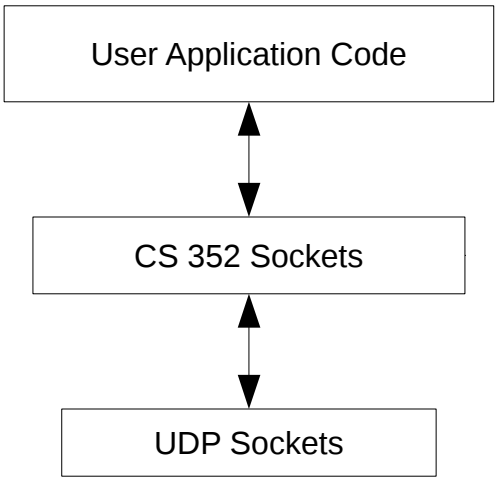

Figure 1: Layering in CS 352 Sockets

2. Architecture

Figure 1 shows the architecture of the 352 socket layer. All communications go through a single UDP socket. A set of ports which are managed by 352 sockets exists on top of the UDP socket.

To allow 2 instances of 352 sockets to communicate on the same machine, 2 UDP sockets must be used, one for each instance. As these can not have the same UDP port number, they must have different port numbers. In this case, each instance must use 2 UDP ports, one for sending and one for receiving.

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 a C structure, which must be translated into a Python **`struct`**:

```
/* a CS 352 RDP protocol packet header */
struct __attribute__ ((__packed__)) sock352_pkt_hdr {
      uint8_t version;        /* version number */
      uint8_t flags;          /* for connection set up, tear-down, control */
      uint8_t opt_ptr;        /* option type between the header and payload */
      uint8_t protocol;       /* higher-level protocol */
      uint16_t header_len;    /* length of the header */
      uint16_t checksum;      /* checksum of the packet */
      uint32_t source_port;   /* source port */
      uint32_t dest_port;     /* destination port */
      uint64_t sequence_no;   /* sequence number */
      uint64_t ack_no;        /* acknowledgement number */
      uint32_t window;        /* receiver advertised window in bytes*/
      uint32_t payload_len;   /* length of the payload */
};
typedef struct sock352_pkt_hdr sock352_pkt_hdr_t; /* typedef shortcut */
```

Fields in **Red** (**`version,flags,header_len,sequence_no,ack_no,payload_len`**) must be filled in correctly in part 1, while the **Blue** fields can be ignored for this part of the project.

Note that uintX_t is an X-bit unsigned integer., as defined in `<sys/types.h>`. At the packet level, all these fields are defined to be in network byte-order (big-endian, most significant byte first).

In Python, you will need to use the **`struct`** module to pack and unpack binary data (see this link for a description of the struct library: https://docs.python.org/2/library/struct.html) A struct object takes a formating string and allows you to create a binary object based on the format and Python variables. This is called a *pack* operation. The inverse, or *unpack*, takes a binary object and unpacks it into a set of Python variables. Recall for the format string, B = Byte, H = Half-Word (16 bits) , L=Long (32 bits) Q=Quad (64bits), and the "!" means use network byte order. Some example code to pack a CS 352 packet header might look like this:

```
sock352PktHdrData = '!BBBBHHLLQQLL'

udpPkt_hdr_data = struct.Struct(sock352PktHdrData)

header = udpPkt_header_data.pack(version, flags, opt_ptr, protocol, checksum,
source_port, dest_port, sequence_no, ack_no, window, payload_len)
```

For part 1, in the packet, the `version` field should be set of 0x1. The `protocol, opt_ptr, source_port` and `dest_port` fields should all be set to zero. Future versions of the protocol will add port spaces and options. The `header_len` field will always be set to the size of the header, in bytes.

An address for the CS 352 RDP is slightly different from the normal socket address, as found in the sock352.h file. The main difference is the addition of a port layer on top of the UDP port space, as seen in the `cs352_port` field. This will be used in later versions of the protocol.

Connection Set-up:
352 RDP follows the same connection management protocol as TCP. See Chapter 3.5.6, pages 252-258 of Kurose and Ross for a more detailed description. The **`version`** field of the header should always be set to 1.

Flags:

The bit flags needed are set in the `flags` field of the packet header. The exact bit definitions of the flags are:

Flag Name	Byte Value (Hex)	Byte Value (Binary)	Meaning
SOCK352_SYN	0x01	00000001	Connection initiation
SOCK352_FIN	0x02	00000010	Connection end
SOCK352_ACK	0x04	00000100	Acknowledgement #
SOCK352_RESET	0x08	00001000	Reset the connection
SOCK352_HAS_OPT	0xA0	00010000	Option field is valid

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 RESET flag set.

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), it is first broken up into segments, that is, parts of the application byte-stream, 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 in-order based on the `recv()` calls made. If insufficient data exists for a `recv()` call, partial data can be returned and the number of bytes set in the call's return value.

For CS 352 RDP version 1, the window size can be ignored.

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 simplified. In this model, each side closes it's send side separately, 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 conditions.

3. Invoking the client and server:

The client and server take a number of arguments :

-f <the file name to send or to save>
-d <the destination host to send to>
-u <the UDP port to use for receiving>
-v <the UDP port to use for sending> (optional)

If you are running both the client and server on the same machine, you would, for example, in one terminal window, first run the server:

```
server1.py  -f savedFile.pdf  -u 8888 -v 9999
```

In a second terminal window, run the client:

```
client1.py  -d localhost -f sendFile.pdf  -u 9999 -v 8888
```

Notice how the send and receive ports are swapped. If we were running on different machines, a single UDP socket (port) could be used for both sending and receiving.

4. Grading:

Functionality: 80%
Style: 20%

Functionality:
We will run the **client.py** program using our module (called the 'course client)' against the **server.py** program using your module (the 'student server'), and the **client.py** using your module (the 'student client') against the **server.py** linked to our module ('course server'). We will send a file and see if the checksum on the client and server match the correct checksums. The **client.py** program opens a single file, sends to the server, and then both close the socket and exit. See the source code for more details. The size of the file may range from a few bytes to many megabytes. There will be a total of 4 tests, as below, and each test is worth 20% of the total grade:

(1) student client, course server, in-order packets.
(2) course client, student server, in-order packets,
(3) student client, course server, random 20% packets dropped by the course module.
(4) course client, student server, random 20% packets dropped by the course module.

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) the client1.py source code, (2) the server1.py source code, (3) your sock352.py library file, and (4) any other files for your library source code.