Computer Networks Project 2

Prof. Prasad

Due date: Oct 24, 2025 11:59pm
Learning goals:

1. Students will be able to describe the setup and teardown mechanisms of a reliable transport layer.
2. Students will be able to identify, design and develop the data structures and methods required for setup and teardown.
3. Students will be able to write multithreaded Java code

Java knowledge required: Read up on the following Java classes: Thread, Timer, TimerTask

Starting with this project, we will build our very own simplified Internet stack called **SkidNet**. Since our protocols are simplified, we cannot run our code on routers in the Internet. So instead, SkidNet will use its own transport and network layers that you will build, called the Simple Reliable Transport (SRT) and Simple Network Protocol (SNP) layers respectively, which will run on Java sockets that you already wrote for Project 1. By doing so, you will focus on building a simple transport and network protocol whose functionality are similar to the TCP/IP protocols that drive the Internet. We will write the protocols in Java, since you are already familiar with Java programming.

In this project, you will implement the Simple Reliable Transport protocol. First, you will need to update the Server and Client classes that you wrote for Project 1 to include the fields and methods required for implementing the reliable protocol. We only consider methods associated with set up and tearing down connections between the client and server. SRT is capable of supporting multiple simultaneous connections like TCP. We will discuss the data structures and Finite State Machines (FSMs) need to support this later in this document.

The SRT sits directly on an overlay layer. The overlay in this case simply implements a TCP connection between the client and server SRT transport protocols. The overlay just contains a direct TCP connection between the client and the server, which you already implemented for Project 1. The `startOverlay` and `stopOverlay` methods create the direct TCP connection between the client and the server. `startOverlay` should return the TCP socket descriptor. `stopOverlay` closes the TCP connection. You already wrote most of the code needed to do Project 1.

The methods you need to implement for this project are shown in Fig **??**.

Methods to implement

The flow of the methods are as follows.

Figure 1: Simple Reliable Transport API

- *Overlay setup:* Start the overlay by creating a direct TCP link between the client and the server in `startOverlay`. In this project, you will implement `startOverlay` in the `Client.java` and `Server.java`.
- *Initialization:* The server initializes the SRT server by calling `initSRTServer` and the client initializes the SRT client by calling `initSRTClient`.
- *Connection setup:* Then the server calls `createSockSRTServer` to create a server side socket and `acceptSRTServer` to accept a connection request from the client (i.e., control message SYN at the SRT layer). The client creates a socket and connects to the server socket by calling `createSockSRTClient` and `connectSRTClient`, respectively. In this project, the client and server establish two connections on different ports to prove that the SRT can cope with multiple connections: the server creates another socket and waits for incoming connection, and the client creates another socket and connects to the new server socket. By doing this, we show that SRT can have multiple connections at the same time.
- *Connection teardown:* In this project, the client disconnects from the server after a wait period by calling `disconnSRTClient` for each connection. Finally, the server closes the socket by calling `closeSRTServer` and the client closes the socket by calling `closeSRTClient`.
- *Overlay teardown:* The `Client.java` and `Server.java` stop the overlay before terminating their processes by calling `stopOverlay` at the client and server.

You will implement the send and receive in Project 3.

Data structures

Segment: The `Segment` object should contain the following fields, to contain both the header and the data portion.

```
int src_port; //currently not used
int dest_port; //currently not used
int seq_num; //currently not used
int length; //currently not used
int type; //segment type
int rcv_win;  //currently not used
int checksum;  //currently not used
char data[MAX_SEG_LEN]; //currently not used
```

The type is defined using the following static final values: `SYN 0, SYNACK 1, FIN 2, FINACK 3, DATA 4, DATAACK 5`.

Transport Control Block: The SRT maintains all the state associated with a connection using a *Transport Control Block (TCB)*. For each connection the client and server side initialize and maintain a TCB. The figure below shows a server side TCB maintained for each connection. A TCB table is initialized at the client and server on startup when `initSRTServer` and `initSRTClient` are called by the `Server` and `Client` objects, respectively. For each connection, the SRT client maintains a client and server TCB.

The `TCBServer` object should contain the following fields. The state can only take the following static final values: `CLOSED 1, LISTENING 2, CONNECTED 3, CLOSEWAIT 4`.

```
int nodeIDServer; //node ID of server, similar as IP address
int portNumServer; //port number of server
int nodeIDClient; //node ID of client, similar as IP address
int portNumClient; //port number of client
int stateServer; //state of server
```

Figure 2: SRT connection setup and teardown

Figure 3: SRT Client FSM

The `TCBClient` object should contain the following fields. The state can only take the following static final values: `CLOSED 1, SYNSENT 2, CONNECTED 3, FINWAIT 4`.

```
int nodeIDServer; //node ID of server, similar as IP address
int portNumServer; //port number of server
int nodeIDClient; //node ID of client, similar as IP address
int portNumClient; //port number of client
int  stateClient; //state of client
```

Connection setup and teardown

SRT has a similar setup and teardown of connections as TCP. However SRT simply uses a two-way handshake. A connection is established when the client sends a SYN and the server responds with a SYNACK as shown in shown in Fig **??**. What happens if the SYN is lost or corrupt; client knows it sent the SYN, server has no knowledge. So how is this operation issue solved? Furthermore, what happens if the server received the SYN, sends the SYNACK but the SYNACK is lost. The server knows it sent an SYNACK and does not know that the client did not get it.

Solutions to these problems are as follows. In the case of the lost SYN, the `Client.java` needs to set a `SYN_TIMEOUT` and if it does not receive a `SYNACK` in that period it sends another `SYN` and sets the timer again. It repeats these steps until it either gets a `SYNACK` or the `SYN_MAX_RETRY` is exceeded. For example, the client sends a `SYN` and moves into a `CONNECT` state. In that state it is likely to receive `SYN`s. What should it do? Send a `SYNACK` and let the client move to CONNECT. We discuss the correct state transitions later on client and server FSMs.

A connection is taken (tear) down when the client sends a FIN and the server responds accordingly with a `FINACK` as shown in Fig **??**. But we can lose packets here. What happens in a `FIN` is lost? The client needs to send another `FIN` after `FIN_TIMEOUT` up to `FIN_MAX_RETRY`. What happens if the `FINACK` is lost? The client side has to wait for retransmissions of `FIN_MAX_RETRY` in an attempt to get a `FINACK`. If it does not get the `FINACK` it transitions to CLOSE state after a wait period and assumes the connection had time to close in an orderly manner.

Client and Server

The Client class should have the following in the `main` method.

```
main() {
  //call the startOverlay method, throw error if it returns -1
  //call the initSRTClient method, throw error if it returns -1
  //create a srt client sock on port 87 using the createSockSRTClient(87)
     and assign to socksr, throw error if it returns -1
  //connect to srt server at port 88 using connectSRTClient(socksr,88),
     throw error if it returns -1
  //for now, just use a Thread.sleep(10000) here
  //disconnect using disconnSRTClient(socksr), throw error if it returns -1
  //close using closeSRTClient(socksr), throw error if it returns -1
  //finally, call stopOverlay(), throw error if it returns -1
```

Figure 4: SRT Server FSM

```
}
```

See Figure **??** while developing the Client class. You will implement all the methods in Fig **??** except the `sendSRTClient`, which you will implement for the next project.

startOverlay: This method finds the ip address of the client machine using `InetAddress.getByName("localhost");` and establish connection with the server by creating the socket `s = new Socket(ip, ServerPort);`

initSRTClient: This method initializes the TCB table marking all entries NULL. The method starts the ListenThread thread to handle incoming segments.

createSockSRTClient (int client_port): This method looks up the client TCB table to find the first NULL entry, and creates a new TCB entry. All fields in the TCB are initialized e.g., TCB state is set to CLOSED and the client port set to the call parameter client port. The TCB table entry index should be returned as the new socket ID to the client and be used to identify the connection on the client side. If no entry in the TC table is available the method returns -1.

connectSRTClient(int socksr, int server_port): This method is used to connect to the server. It takes the socket ID and the server's port number as input parameters. The socket ID is used to find the TCB entry. This method sets up the TCB's server port number and a SYN segment to sent to the server. After the SYN segment is sent, a timer is started. If no SYNACK is received after SYN_TIMEOUT timeout, then the SYN is retransmitted. If SYNACK is received, return 1. Otherwise, if the number of SYNs sent > SYN_MAX_RETRY, transition to CLOSED state and return -1.

disconnSRTClient(int socksr): This method is used to disconnect from the server. It takes the socket ID as an input parameter. The socket ID is used to find the TCB entry in the TCB table. This method sends a FIN segment to the server. After the FIN segment is sent the state should transition to FINWAIT and a timer started. If the state == CLOSED after the timeout the FINACK was successfully received. Else, if after a number of retries FIN_MAX_RETRY the state is still FINWAIT then the state transitions to CLOSED and -1 is returned.

closeSRTClient(int socksr): This method removes item from the TCB table and returns 1 if succeeded (i.e., was in the right state to complete a close) and -1 if fails (i.e., in the wrong state).

class ListenThread extends Thread: The run() method handles incoming segments and cancels the thread on receiving SYNACKs and FINACKs.

The values to be used for the static final variables are as follows:

SYN_TIMEOUT: the number of milliseconds to wait for SYNACK before retransmitting SYN should be 100 milliseconds

SYN_MAX_RETRY: the max number of SYN retransmissions in srt_client_connect() should be 5

FIN_TIMEOUT: the number of milliseconds to wait for FINACK before retransmitting FIN should be 100 milliseconds

FIN_MAX_RETRY: the max number of FIN retransmissions in srt_client_disconnect() should be 5

The Server class should have the following in the `main` method.

```
main(){
  //call the startOverlay method, throw error if it returns -1
  //call the initSRTServer() method, throw error if it returns -1
  //create a srt server sock at port 88 using the createSockSRTServer(88)
     and assign to socksr, throw error if it returns -1
  //connect to srt client using acceptSRTServer(socksr), throw error if it
     returns -1
  //for now, just use a Thread.sleep(10000) here
  //disconnect using closeSRTServer(sockfd), throw error if it returns -1
  //finally, call stopOverlay(), throw error if it returns -1
```

See Figure **??** while developing the Server class. You will implement all the methods in Fig **??** except the `rcvSRTServer`, which you will implement for the next project.

startOverlay: create a server socket using `ss = new ServerSocket(59090);` and accept the client connection using `s = ss.accept();`

initSRTServer: This method initializes a TCB table containing TCBServer objects.

acceptSRTServer(int sockfd): This method gets the TCBServer entry using the `socksr` and changes the state of the connection to LISTENING. It starts the ListenThread to wait for SYNs, and changes the state to CONNECTED on receiving the SYN segment.

closeSRTServer(int socksr): This method removes the TCB entry, obtained using socksr. It returns 1 if succeeded (i.e., was in the right state to complete a close) and -1 if fails (i.e., in the wrong state).

public class ListenThread extends Thread: The run() method handles incoming segments and cancels the thread on receiving SYNs and FINs. The thread must also start a CLOSE_WAIT_TIMEOUT timer on receiving FIN before changing state to CLOSED.

The static final variable CLOSE_WAIT_TIMEOUT should be set to 1 second.

Grading Rubric

It is intentional that I have not given any starter code. This is an advanced class and I want you to develop skills to design and write software on your own. Of course, it is okay to ask the professor for help if you do not think you are ready to design your own software. Start early so you have time to ask for help, if needed!

- A score of 3 will be given to students whose code works correctly, their test code demonstrates their understanding of the concepts, and their code is well-commented and uses meaningful variables. Minor errors in the implementation are okay; for example, threads do not close, clients throw an error if server closes first.
- A score of 2 will be assigned if code demonstrates a lack of understanding of concepts; for example, you made a good attempt at implementing the extensions but code does not work.
- Code that is incomplete or major compilation errors will get a score of 1