Project 3: TCP Data

Due: May 6

You will complete a fully-functional implementation of TCP.
You will build on your implementation of the TCP Finite State Machine from the previous project and add the capability to exchange data.
Your implementation will function as a SocketImpl for java, allowing java applications to open and close connections using your code.
Your implementation will tolerate packet loss. The actual packets are sent between machines using UDP packets generated by the provided project software.

Requirements

Your implementation may not rely on modifying any of the provided files except for StudentSocketImpl.java, which is where your implementation will be built. You may modify the other provided files for debugging purposes, and we expect you to modify or write new client and server programs, but your project will be graded using our versions of all provided files except for StudentSocketImpl.java. You may, however, add new files if you feel it necessary.

Your implementation must be capable of opening the TCP connection and closing it. Both ends of the connection must clean up and exit cleanly. Your implementation is also required to support multiple simultaneous open sockets (Although we did not provide application code to test this).

Your implementation must print state transitions when they occur. It must also print a line indicating a packet being retransmitted when that occurs. Actual packet information is printed with the code we provide.

You are required to deal with lost packets, which includes both retransmitting packets when appropriate and responding properly to a retransmitted packet when the ACK to that packet was previously lost. However, you do not have to deal with broken implementations or hostile packets, such as receiving a FIN when you are waiting for a SYN. Feel free to ignore such packets.

We suggest you should use the **go-back-n protocol** (or some variation of GBN or even mix of GBN and SR) as discussed in 3.4.3 of the book, EXCEPT that you will use cumulative ACKs and sequence numbers as used with TCP (1byte=1sequence number), rather than the 1 sequence number per packet used in 3.4.3. Otherwise, the single timer for the oldest packet is fine, or you can create a timer for each packet that hasn't been ACKed (This is a design decision we leave up to you). You will use a fixed window of 8 packets. Also note that the fixed timeout from the "TCP State Machine" project should also be used for this project.

Implementation plan

This is the procedure we recommend for implementing this project. You are free to implement it in whatever order you believe appropriate, but following this order will probably help...

- Familiarize yourself with the new portions of the code. InfiniteBuffer is the major new class you will have to use, but there are some other new classes and methods you may want to be familiar with.

- Write a method to send an array of data (bytes) with a particular sequence number. Then add onto receivePacket to receive this data, copy it into the recvBuffer, and send an ACK. Test the send data method by calling it before you return from connect. Just put simple wait() loops in getData and dataFromApp

- InfiniteBuffer is a very simple class with no parameter checking whatsoever. You might find it very useful to either wrap it in another class or modify it for debugging. You WILL make mistakes with the sequence numbers you pass it, and doing that will help track them down.

- Now write getData and dataFromApp. Remember that these will be called by other threads and will wind up with loops like:

  ```
  while(I can't fit more data in sendBuffer)
      wait();
  ```

- Make sure that any calls you make to notify() are notifyAll()

- Probably the easiest restriction to use on the amount of data you send is the receiver window size (essentially, you're implementing a very primitive TCP without congestion control). You need a method to advertise the available space in recvBuffer whenever you send a packet.

- Remember that whenever you receive an ACK or receive more data into the sendBuffer, you will have to try to see if you can send more data, according to the window size.

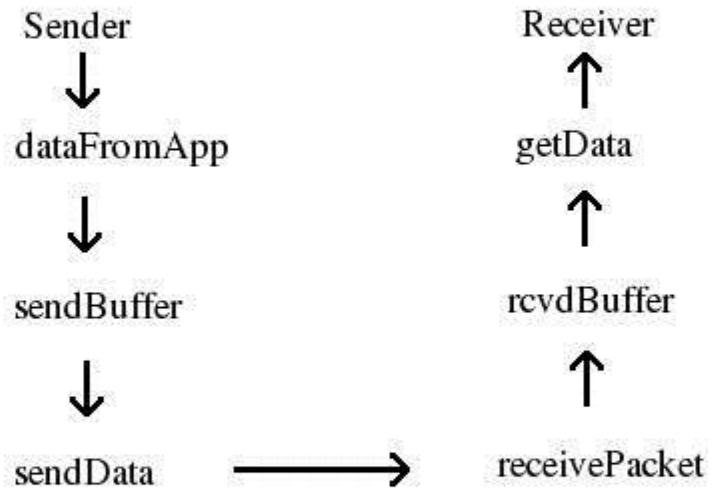
- There is a problem when the available space in the receiver window gets down to zero. The easiest way to handle this is to have the sender ignore the window size when it's less than one packet, and send one packet anyway. The receiver will have to drop this packet because it won't fit in the buffer, so it will ACK the highest byte it received successfully. The sender will eventually timeout on the extra packet and send it again. Eventually, the receiver window will move and more data will get through. Retransmitting the extra packet will ensure that the sender knows about it.

- Remember that the trick to implementing this project is to break up the functionality into appropriate modules. receivePacket has to send the appropriate ACK when a data packet comes in. If the data packet contains new data, it has to copy it into recvBuffer. Let getData handle the rest. Sending data is similarly broken up between dataFromApp and another routine you should write that sends data in the sendBuffer.

- All told, this project is only about 300 more lines of code than the "TCP State Machine" project. But it's hard to develop and debug, so don't wait until the last minute.

Data Path

Heres a picture of the data flowing through the functions and structures of the program. Sender and Receiver are the applications using the socket and sendData is the function you create to send the array of data. The rest are already in the code.



Notes and FYIs

We STRONGLY encourage you to only run your version of the project on two machines you are physically in front of (or at least near). If you are testing your code and make a mistake, and the network goes down, kill your programs. If you can't kill them (or don't know how), reboot the machines (hold in the power button). You have official techie approval to hard reboot machines rather than taking down the network. (note that this should not be a problem with the current TCPWrapper, but just in case...)

`dataFromApp` should trigger whatever code you have to process and send packets (if there is room in the window). Whether you do this by calling something directly or by having a separate thread that you notify is up to you.
You should not need to do anything with appIS, appOS, pipeAppToSocket, or pipeSocketToApp in your project. These are used by SocketReader and SocketWriter. You only need to implement getData and dataFromApp as commented to successfully transfer data to the application.

Please note that the maximum amount of data you should put into a packet is 1000 bytes. This is known as the maximum segment size (MSS), and your book uses this value when discussing congestion control.

getData and dataFromApp both should block. It is important to note that they block differently. getData blocks until there is any data in recvBuffer. As soon as there is data in recvBuffer, it should return with as much data as possible in buffer. There is no need to fill up buffer. dataFromApp must block until it copies ALL data in buffer into sendBuffer. By returning, it indicates it has copied all data. Note that there is no requirement that the data has been ACKed (or even sent, although you probably want to try to send as much as you can before this method returns). The method should return after the data is copied into sendBuffer.

The other issue is what to do with data when close() is called. You must deliver any data that an application wrote to the socket before calling close before you send the FIN. Typically a web server dumps a file to its socket, then immediately calls close()---it expects all the data it wrote will be delivered. Note that the close() method in SocketReader helps you with this by flushing the internal stream with the application. It is up to you to make sure you send all data in your sendBuffer. Then you send the FIN after the last byte is sent. The FIN can be sent immediately (before the data is ACKed), but if you want to wait until all outstanding data is ACKed that's OK too. Also note that it is still possible for the other side of the connection to be sending data after the local side has called close(). TCP has a standard way of handling this, but my suggestion to you is just to send ACKs to any data that arrives at that point and dump it on the floor. We won't be testing this behavior, so it's not important how you handle it. It might help you in debugging if you note that it's happening. My code happily copies it into the recvBuffer like normal.

Note that StudentSocketImpl.java is called StudentSocketImpl-data.java. Because you will continue to need the code you already wrote for the "TCP State Machine" project for this project, you should study the new features in StudentSocketImpl-data.java and copy the relevant portions to your current StudentSocketImpl.java. There are other minor change to the other files, so you should simply copy your old StudentSocketImpl.java into the new porject directory, then update it with the new portions.

The Code

Data Code Version 1