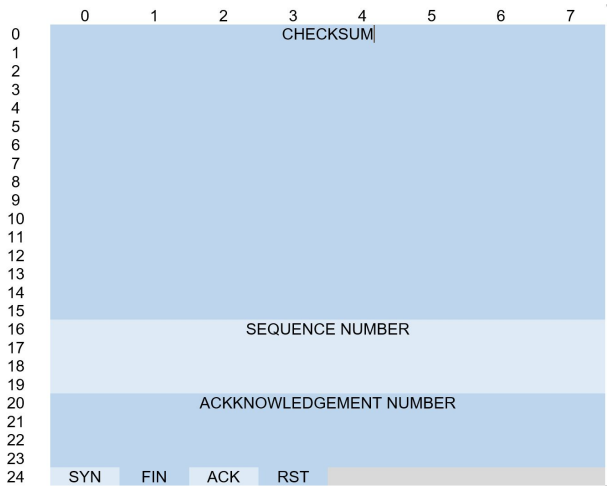
Joe Wang - joewang704@gatech.edu
Amy Liu- aliu66@gatech.edu
March 17, 2017

Project 2 - README

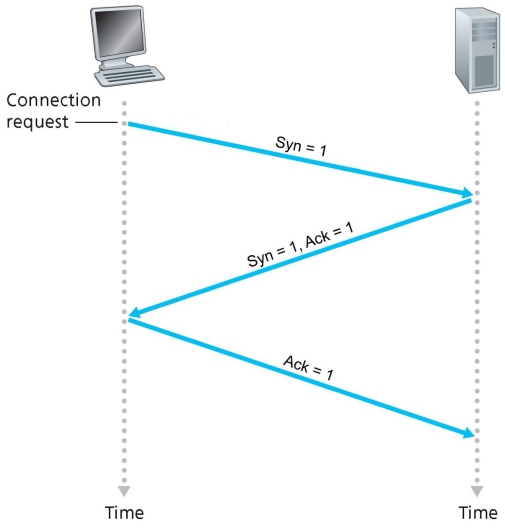
ReldatClient: File user calls to run the client application
ReldatClientHelper: Contains the client's finite state machine
ReldatClientState: Contains enums of the client's states
ReldatConstants: Contains constants used throughout the server and client application
ReldatFileReceiver: Called on by the ReldatServerHelper to receive packets, capitalize the payload, and send the transformed packet back to the client
ReldatFileSender: Called on by ReldatSRSender to send packet from the client to the server and create timers for all packets sent.
ReldatHelper: Contains methods used throughout the server and client application
ReldatPacketTimers: Called on by ReldatFileSender to schedule timers for packets and called on by ReldatSRSender to cancel timers. Contains global static map from packet numbers to timers.
ReldatSendTimerTask: Used by ReldatPacketTimers to execute sending packets as task in timers.
ReldatServer: File user calls to run the server application
ReldatServerHelper: Contains the server's finite state machine
ReldatServerState: Contains enum of the server states
ReldatSRSender: Called on by ReldatClientHelper. Contains the multipurpose buffer and the sliding window, determines which packets to send to the server, saves transformed payload to the multipurpose buffer, and writes to the transformed file.

To run the application

1. Compile using
   ```
   javac -d ./ *.java
   ```
2. Start the server application using
   ```
   java reldat.ReldatServer PortNumber WindowSize
   ```
3. Start the client application using
   ```
   java reldat.ReldatClient Host:PortNumber WindowSize
   ```
4. To transform a file
   ```
   transform F
   ```
5. To disconnect with the server
   ```
   disconnect
   ```



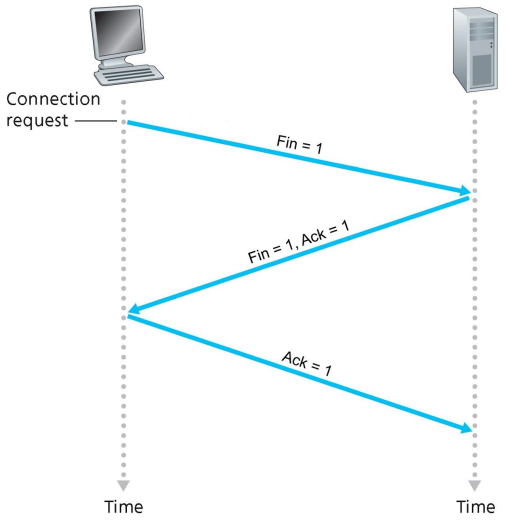
The Java DatagramPacket class provides a method to get the sender address and sender port. We created our own header to provide more information about the packet. Our header contains the packet checksum, sequence number, and acknowledgement number. It also has the syn, fin, ack, and reset flag bits. This 24 byte header is placed before the data onto the UDP payload. Our RELDAT payload size is 1000 bytes. Therefore, the UDP payload size we send will be 1024 bytes.



To establish a TCP connection,

1) The client initiates a three-way handshake by sending a packet to the server with the syn bit set to 1.
2) The server will reply back with a packet with the syn bit and ack bit flagged.
3) Once the client receives the syn-ack from the server, it will reply with an ack and a connection is established.

Note: We also implement the RESET bit, so that if the server never gets a third ack. It will send a RESET packet to the client, which might have already assumed the connection is established, in order to tell it to reset, in our case we have defined resetting for the client as gracefully closing.



To close a TCP connection,

1) The client sends a fin to the server.
2) The server will reply with a fin-ack.
3) The client will send an ack to the client and wait 5 seconds before closing the connection.

To implement file transfer, we have a FILE_TRANSFER state that we enter into for both the client and server (seen from the state machine below).

The Client:

To implement file transfer, we take inspiration from the Selective Repeat model. Our client has a multipurpose buffer. This buffer is a String array with size equal to the number of packets needed to send the file. This buffer is first utilized similar to the send window in Selective Repeat. We have a set window size that we send from the buffer. Each packet sent has a timeout of 3 seconds, in which it will be resent. For each ACK we receive back, we store the capitalized data inside the index corresponding to the ACK number (since our array's size allows this to work). We then attempt to move the window from the left, moving until the leftmost window slot finds a NULL slot. If the leftmost window slot is currently NULL (hasn't been ACKed back yet) then we don't move the window at all. In this way, we both handle out of order packets and make sure every index left of the window has been ACKed. In the later sections, we go in depth on how this architecture allows us to handle out-of-order, corrupted, lost, and duplicate packets.

The Server:

Because our client handles most of the complexity, our server can simply be a simple server that just responds to every received packet with an ACK equal its SYN number. It also takes the data from the received packet, capitalizes it, and then piggybacks the capitalized data onto the ACK. If any SYN, ACK, or FIN is sent to the server during the FILE_TRANSFER state, the server will simply ignore it. In this case,

the client will not receive any response and will eventually timeout and close. Then the server will not receive any response as well and timeout and start listening for new clients.

For a detailed look at how the client and server move between the beginning handshake, file transfer, and closing handshake, refer to the FSM diagrams below.

An example of a non-trivial RELDAT function we implemented is creating an individual timer for each packet. We utilize the Java Timer and TimerTask classes in order to do this. These classes allow us to repeatedly execute tasks in background threads over a period of time. We set this time to 3 seconds and set the task to resend a packet. In order to keep track of all these timers, we create a static map from each packet's sequence number to its timer. We can then pass in this sequence number and other data in order to ensure that the correct packet is resent for each timer. When we receive an ACK for a certain packet, we simply index that ACK into the map (since the ACK corresponds to the packet's client side sequence number), call the cancel function on the timer, and remove it from the map. The timer is set to repeatedly send a packet every 3 seconds until the cancel function is called (cases of infinite sending are handled by our global receive timeout of 15 seconds).

How does RELDAT detect and deal with duplicate packets?
There is a buffer in the client that stores all capitalized data received from the server. Each index of the buffer corresponds to a ACK number. The server will send an ACK to the client with the capitalized data regardless of whether it is a duplicate or not. The client will only store the capitalized data from the server if the index in the buffer corresponding to the ACK number does not contain data and the ACK number of a packet is within the window.

How does RELDAT detect and deal with corrupted packets?
When the client creates a packet, it will include the checksum of the payload in the header. When the server receives a packet, it will recalculate the checksum of the payload. The server will only transform and send back a packet if the calculated checksum is equal to the checksum in the header of the packet. If they are not equal, the server will ignore the packet. The client will eventually get a timeout for that packet and resend the un-acked packet to the server.

When the server sends the client a packet, it will include in the packet header the checksum of the payload. When the client receives a packet, it will calculate the checksum of the payload and compare it to the checksum in the packet header. If a

packet from the server to the client is corrupted, the client will ignore the packet and refrain from clearing the timeout on the packet. The client will again eventually time out for that packet and resend the un-acked packet to the server.

How does RELDAT detect and deal with lost packets?
A timer is created for each packet sent from the client to the server. The timer will only be canceled once the client receives an ACK for a packet.

If a packet containing a payload is lost in transit from the client to the server, the timer associated with that packet will timeout in 3 seconds and resend the lost packet. It will continue this process for at least 15 seconds. If the client does not receive an ACK for the packet after 15 seconds, the server is assumed to have crashed and the client will gracefully exit the program.

If a packet is containing a payload is load in transit from the server to the client, the same thing will happen as the previous scenario.
If a SYN, FIN, ACK, FIN-ACK, SYN-ACK, or RST is lost, the client will timeout in 15 seconds and assume that the server has crashed and gracefully exit the system. The server would assume that the client has crashed and return back to the LISTEN state.

How does RELDAT detect and deal with re-ordered packets?
The server accepts and transforms the payload of all packets it receives, regardless of the order in which it receives it.

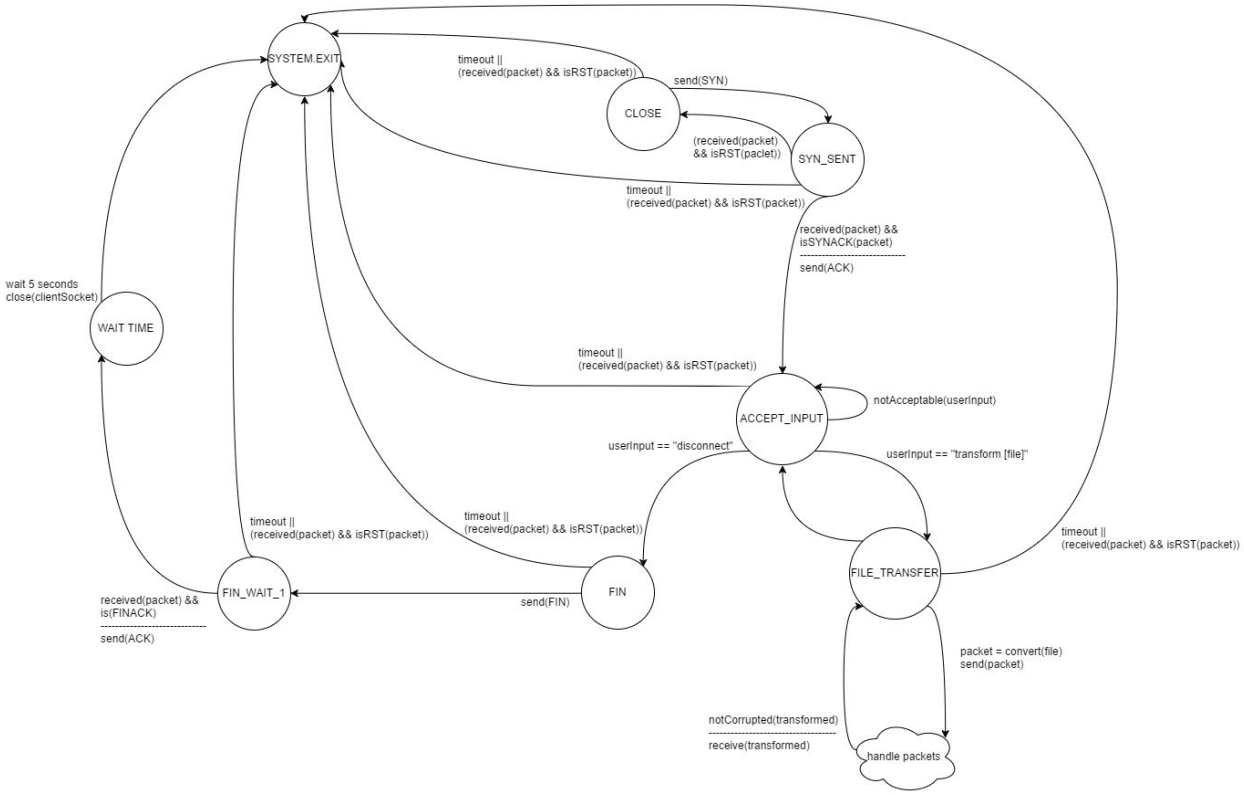
There is a buffer in the client that stores all capitalized data received from the server. Each index of the buffer corresponds to a ACK number. As long as the index in the buffer corresponding to the ACK number does not contain data and the ACK number of a packet is within the window, the transformed data will be stored into the buffer. Once the buffer is full, signifying that the whole file has been successfully transformed, the program will iterate through the buffer and write the contents into a new file.

How does RELDAT support bi-directional data transfers?
RELDAT supports bi-directional data transfer through piggybacking. The server piggybacks the capitalized data for each packet it receives to the ACK sent back. In this way, we have bi-directional data transfer, essentially data being transmitted by both the server and the client at the same time.

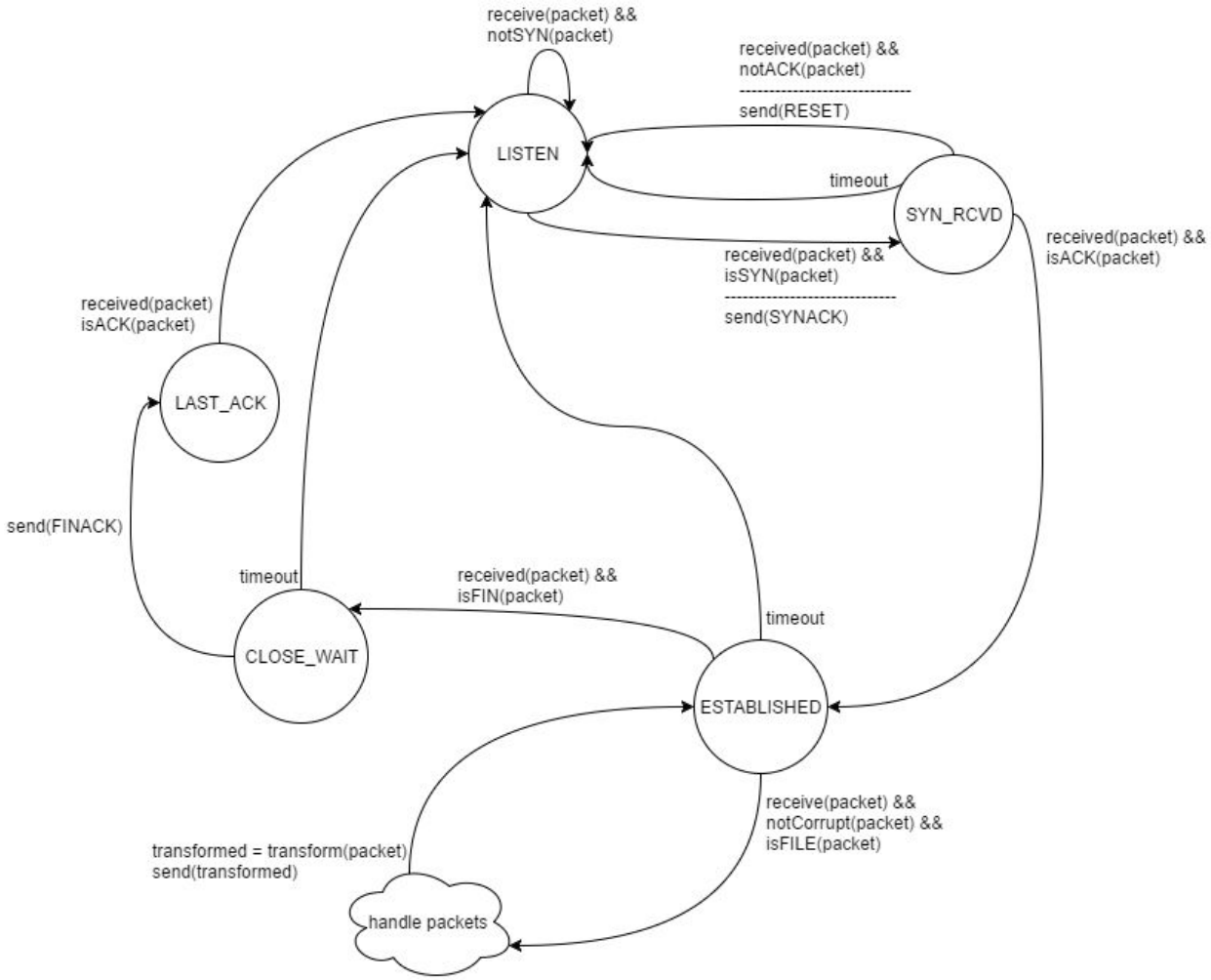
How does RELDAT provide byte-stream semantics?
The client breaks up the file into packets and continuously sends packets to the server based on the window logic. The client reads off the file using a BufferedInputStream and a specified offset.

Are there any special values or parameters in your design (such as a minimum packet size)?
The sequence number and ack number are given four bytes in the header. As such, we can only send packets up to those with a sequence number that can be represented with four bytes. Our payload size is always 1000 bytes. Our header size is 24 bytes. We either send packets of size 1024 bytes when we want to send data or send packets with only size 24 bytes when we only send a header with no payload. We do not send a payload in the case of SYN, SYNACK, FINACK, and certain ACKs (the ones we do not piggyback data onto). We have a global timeout duration of 15 seconds. After 15 seconds if the client does not receive a packet from the server, the client will gracefully close. After 15 seconds if the server does not receive a packet from the client, the server will begin listening for a new client. During file transfer, the timeout on our data packets is 3 seconds. So if the client has sent a data packet and that packet has been ACKed in 3 seconds, it will resend the packet.

Client Finite State Machine Diagram



Server Finite State Machine Diagram



Known Bugs or Limitations

- If the client application starts before the server application and 15 seconds have passed, the client will timeout and exit the application. User needs to restart the client application to attempt at establishing a connection with the server again.
- After establishing a connection between the client and server, the user only has 15 seconds to type in a command on the client terminal or the connection will be automatically closed gracefully.