03APR2017

CS 3251 Programming Assignment 2: Reliable Data Transfer

Group members:

Connor Lindquist (connorlindquist@gatech.edu)
Charlie McKnight (cmcknight9@gatech.edu)
Stephen Zolnik (szolnik3@gatech.edu)

Submitted files:

Packet.java: Server side Java implementation for interpreting received packets and creating outgoing packets.
packet.py: Client side python implementation for interpreting received packets and creating outgoing packets.
PacketStats.java: Java 'struct' to maintain status about sent out packets.
reldat-client.py: Client side python file that handles file deconstruction and reconstruction as well as order of execution and terminal inputs.
ReldatServer.java: Main server side file to handle receiving and sending out packets. Maintains the server side.
selective_repeat.py: Client side selective repeat implementation to handle all things selective repeat including resending packets, acks, and bad checksums.
SelectiveRepeat.java: Server side selective repeat implementation to handle all things selective repeat including resending packets, acks, and bad checksums.

Compiling and Running:

Server Side (Java):

Compile: `javac *.java`
Run: `java ReldatServer port_number window_size`
Example: `java ReldatServer 4096 5`

Client Side (Python):

Run: `python reldat-client.py serverIP:port_number window_size`
Example: `python reldat-client.py 127.0.0.1:4096 10`

Bugs and Limitations:

We do not have any bugs or limitations that we are aware of.

Description and Features:

3-Way Handshake: The server is always in a state of listening for a connection request if it is not connected to another client. Once it receives a connection request from a client, the server sends an acknowledgement to the client to confirm it received the connection request. The server will then expect an acknowledgement from the client to confirm the connection. At this stage the connection has been officially made between client and server. This process is commonly known as the 3-way handshake.

Selective Repeat:

Client: The client begins by sending out the packets filled with the data from the file. These packets are received by the server, and stored for transformation. While this is happening, the server is replying to the packets received with acknowledgement packets to tell the client it received uncorrupted packets. If the packets have been corrupted or timed out, the server will discard the packets and the client will need to resend the unreceived packet. If the packets have been received out of order, the server will sort the packets based on the sequence number the packets were sent out with.

Server: After the server has transformed the file to upper case letter. The server begins by sending the client a "start" packet. This tells the client the server is about to send the transformed file back. After this packet, the server begins sending the data packets to the client, while simultaneously sending acknowledgements to the server. If a packet had been corrupted in any way, the server resends the packet. This is done until the entire packet is received by the client. After the client has received the entire file, the server sends an "end" packet. This tells the client, "Server is done", the client then acknowledges the end packet. After if the client wishes to close the connection it sends a close packet back to the server, and the server acknowledges the "close" packet. At this point, the connection has been ended by the client, and the server goes back into a state of listening for a new connection request.

Header Structure:

Checksum	Type	Seq/Ack	Payload Length	Payload
8 bytes	1 byte	8 bytes	3 bytes	1024 bytes

Header Field:

Checksum: Used to validate if a packet had been corrupted during the transaction. If a packet had been corrupted then the sender would resend the packet after a certain length of time.

Type: The type was used as a way to identify what kind of packet was in transaction between the client and server. There were 6 different types of packets that could be used. The table below shows how we identify what type of packet is in transaction.

Packet Type	Start Packet	Start Ack Packet	Ack Packet	Data Packet	End Packet	Close Packet	Send Packet
String	"1"	"2"	"3"	"4"	"5"	"6"	"7"

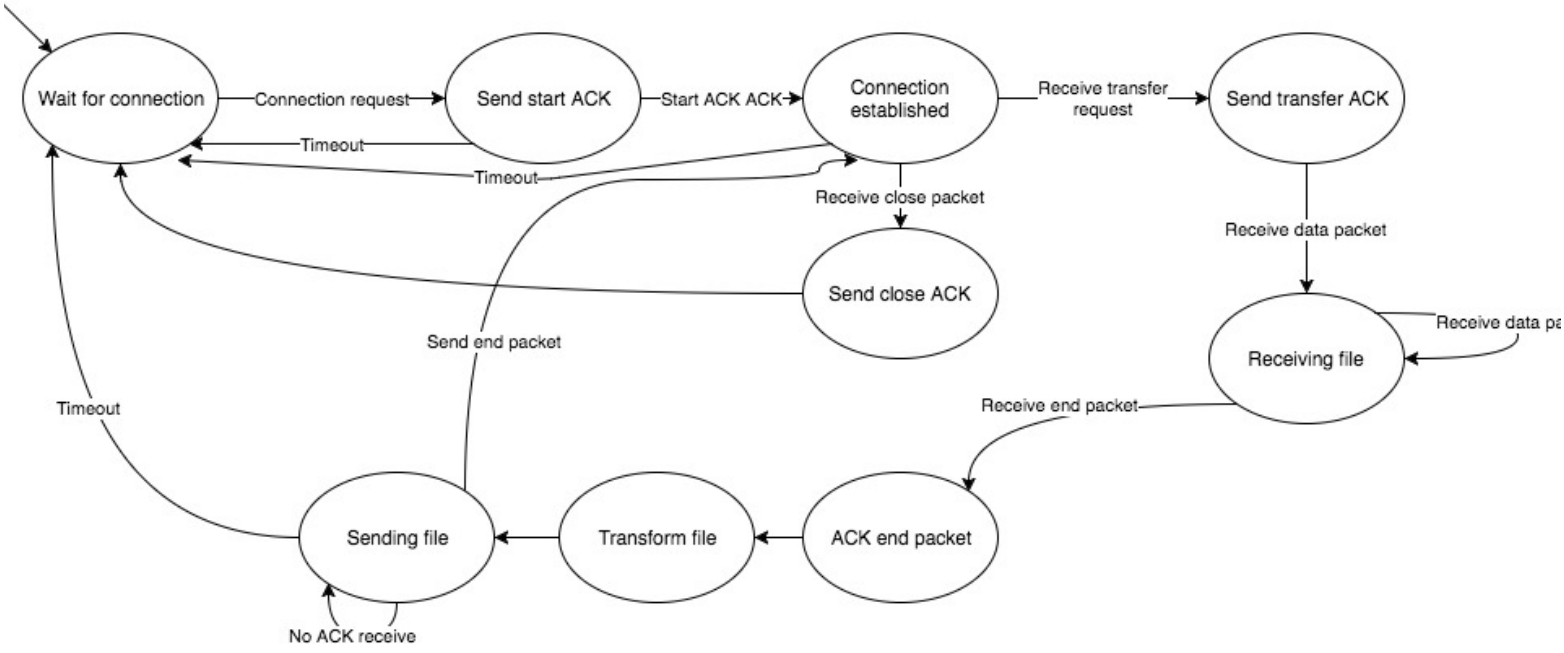
Seq/Ack: Seq helps keep track of the order of the packet being sent. The seq number can be used to help make sure the packets are received in order. The ACK helps tell the sender the packet has been received correctly.

Payload Length: This tells the receiver the length of the message in the packet.

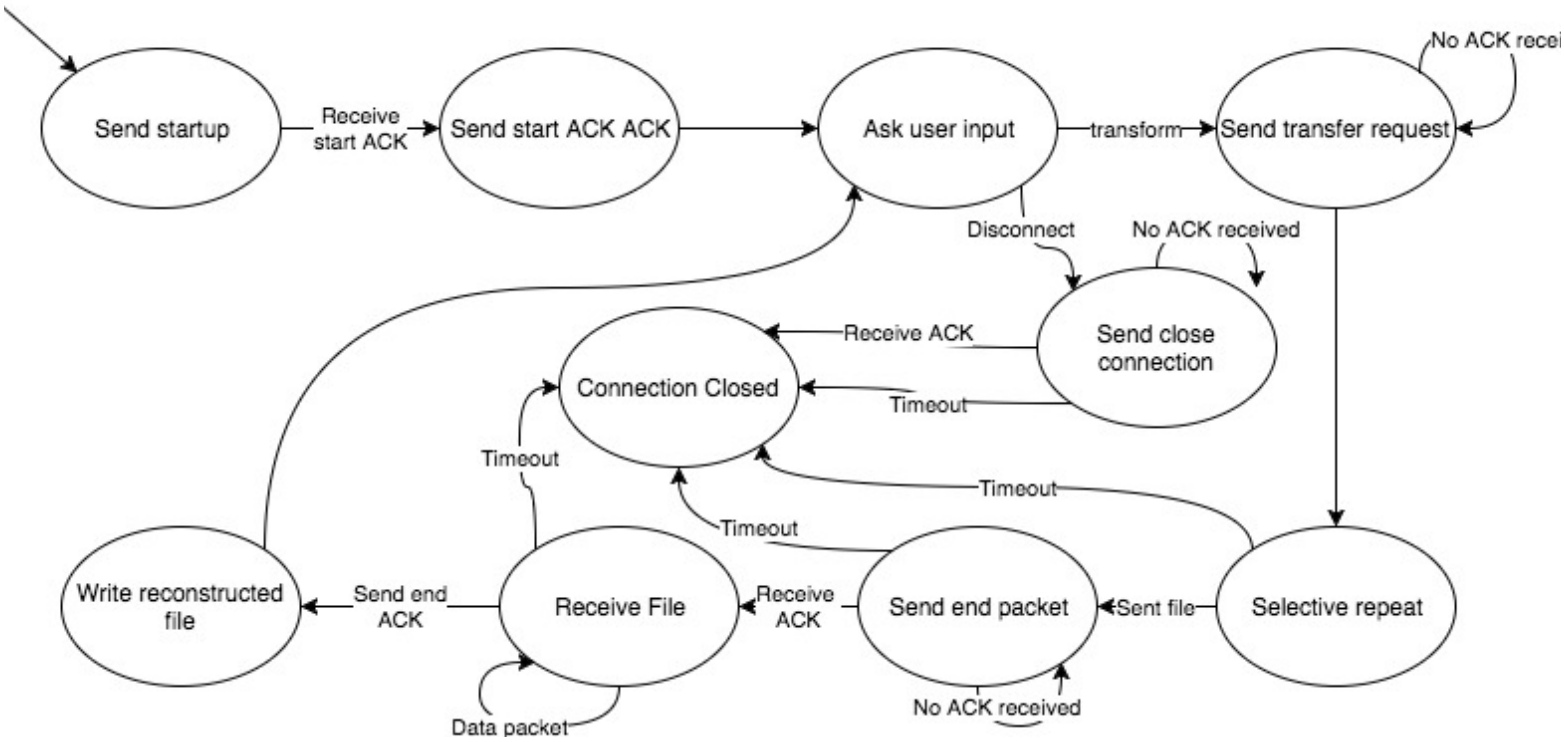
Payload: The actual data in the packet. This section is also used for miscellaneous data such as the windows size in "Start" and "Start Ack" as well as the ack number in "Start Ack".

Note: Data Transfer is bidirectional, meaning the client sends the file to the server. The server changes the file, and sends the file back to the client.

Server Finite State Machine:



Client Finite State Machine:



Algorithm:

On Startup: The client first sends a request to connect, this request is then acknowledged by the server, and once the client receives the server's ack, the client then sends an acknowledgment back to the server to confirm the client received the ack. This approach is commonly known as a 3-way handshake.

On Data Transfer: After a connection has been established, the client begins sending a file to transform into upper case letters. The file is broken up into packets that are represented as Strings. The packet structure can be seen above.

Non-Trivial Functions:

Client:

`main():` parses through the input values, and sets the appropriate variables based on the instructions given by the user.

`start_up(self):` Initializes the connection by performing the 3-way handshake.
`close_conn(self):` After the file transformation is completed, this function terminates the connection.
`check_response(self):` Tells the server, data is about to be sent and repeats until the server acknowledges the response.

In client selective-repeat.py

`receiveFile(self):` receives and validates the packets received from the server, sends acknowledgements for valid .
`sendMode(self, msgList):` starts and waits for the the multiThreaded selective repeat to finish.
`ReceiveAcksThread(threading.Thread):` This thread listens for ACKs from the server, and updates the list of unacknowledged packets.
`SendingDataThread(threading.Thread):` This thread sends the packets to the server, while checking the unACKed packets to see if it needs to resend any packets that have been lost.

Server:

`waitForConnection():` Sets the server in a state of always listening for a connection request. All the logics for the 3-way handshake is performed here.
`mainLoop():` This function the server checks the type field in the received packets, and based on the specific packet received performs various actions. The actions would be tearing down the connection, calling runListener() where we listen to the client sending data, after receiving the data correctly from the server the mainloop then calls the class selective repeat to send the data back after file transformation.
`receiveInstruction():` This function is called from the mainLoop, and basically evaluate the packets received from the client, and based on the packet type, different functions are called.

In Server SelectiveRepeat.java:

There are two modes in this class. The Send Mode, and the Listen Mode. The send Mode is multi-threaded, and the listen Mode simply receives packets, and send ACK packets to the sender.

`mainSendModeLoop():` This method starts the multi threaded send mode and returns when it has completed. This is called after the file has been received, and transformed, and returns a boolean true if the file has been sent and received correctly.
`runListener():` Receives the file from the client, and returns a correctly sorted file of strings.
`sendEndPacket(int seq):` After the file has been transformed and sent back to the client, the server begins to send an end packet. This occurs in this method, and expects an endAckPacket back from the client to confirm the connection has been officially closed.
`SendMode(String threadName):` The run method of this thread takes in the transformed file, and breaks the file up into appropriate number of packets. These packets are stored, and sent out to the client in a certain window size. The window of packets being sent advances once the listen thread receives an acknowledgement. This thread also checks the timer to see if a packet was lost, and needs to be resent.
`ListenAckMode(String threadName):` The run method in this thread is focused on receiving ACKs. Once it receives an ACK, this thread lets the other sending thread know which

packet was correctly received, which allows the sending thread to advance the window of packets. If an endACK packet has been received, then the entire packet has successfully been received by the client. A boolean then returns back to the mainLoop() to notify the server the file has been returned to the client.

On Termination:

Termination occurs after the client receives an end acknowledgement packet, meaning the file has been fully received and acknowledged. The client then closes the socket connection, and the server goes into a state of listening for a connection.

Further Questions:

How does RELDAT perform connection establishment and connection termination?

We establish the connection using the 3-way handshake algorithm. We close the connection by sending an endACK packet to tell the server the client is officially done, and therefore the connection is closed.

How does RELDAT detect and deal with duplicate packets?

The receiver does acknowledges duplicated packets, but does not do anything with them. The packet is acknowledges because it could be that the previous ACK was lost.

How does RELDAT detect and deal with corrupted packets?

The corrupted packets are not acknowledged, and after a certain period of time the sender resends the packet.

How does RELDAT detect and deal with lost packets?

If a packet has been timed out, the sender resend the packet.

How does RELDAT detect and deal with re-ordered packets?

The receiver sorts the packets based on the sequence number of the packet sent from the sender.

How does RELDAT support bi-directional data transfers?

The client, and server are multi threaded, and receives acknowledgements, and sends packets simultaneously.

How does RELDAT provide byte-stream semantics?

We send packets of a 1044 bytes which are represented as strings. Once the receiver gets the packets, we parse through the string to extract the appropriate data.

Are there any special values or parameters in your design (such as a minimum packet size)?

The smallest valid packet is 20 bytes and consists of just the header with no data. These are common for special packets such as Acknowledgements.