LLP Protocol

By Yamilex Avila-Stanley and Se Yeon Kim
(yjas3 and skim870)

November 4, 2016

Answers

- Is your protocol non-pipelined (such as Stop-and-Wait) or pipelined (such as Selective Repeat)?

- Pipelined **(Go-Back-N)**

All of lost can be detected by comparing the sequence number of received packet to the expected sequence number. Corrupted packet can be detected through checksum.

- How does your protocol handle lost packets?

- Receiver:
    - If a packet is lost, the subsequent packets will be considered as out-of-order packets and therefore will be discarded. Just like in an out-of-order packet scenario, an ACK will be resent with the acknowledgement number, which would be the sequence number of the last successfully received packet+1. This acknowledgement number should be equal to the sequence number of the lost packet. ACK is sent to notify the sender to stop sending the subsequent packets and retransmit starting from the lost packet.
- Sender:
    - A timer will be used to detect when a packet (ACK) is lost. The timer keeps track of the oldest transmitted packet that has not been acknowledged. Since a lost ACK will never be received, the timer will never be restarted or stopped, which will lead to a timeout event.
    - When a timeout event occurs, all the packets that were previously sent but not acknowledged will be resent unless the ACK with acknowledgement number that is equal to the sequence number of any of the subsequent packets is received.

- How does your protocol handle corrupted packets?

- Receiver:
    - When a packet is received, we perform the checksum algorithm as specified in our documentation. If the checksum fails (does not equal 0), then we know that the packet is corrupted. If a received packet is corrupted, the receiver will discard the packet and resend an ACK for the most recently received in-order packet.
- Sender:
    - When a corrupted ACK is received, the sender ignores that packet. Sender would continue to send normally and also wait for the ACK from the receiver. On the

receiver's end, it doesn't know that the sender didn't receive the last acknowledgement, so it sends the acknowledgement for the packet received after the packet that has been acknowledged with corrupted ACK. Meanwhile, the sender might time out for the packet where the ACK has been lost, then it would simply retransmit that packet and subsequent packets. When the sender receives the acknowledgement that is for the packet in the buffer put in after the current packet it is waiting for, since this protocol guarantees delivery in order, the sender can safely assume that the receiver has actually received the packet that the sender was waiting for an acknowledgement of and can slide the windows to exclude the packet and the packet that it just received acknowledgement for from the send buffer. Those packets are, then, removed from the buffer to free up a space in the buffer. If the sender retransmitted the packet to the receiver again, receiver ignores the duplicate packet(s) and proceeds to the next non-duplicate packet received.

- How does your protocol handle duplicate packets?

  - Receiver: A duplicate packet will be ignored and discarded since the sequence number of the duplicate packet will not equal the sequence number of the expected packet. It is okay to ignore the duplicate packet since if it is received already, the sender should have received the ACK for it at some point after sending the duplicate packet, and after receiving the duplicate packet the receiver will resend the ACK for the last properly received packet.
  - Sender: When a duplicate ACK is received, it means that a packet was lost or corrupted during the transmission. Therefore, the sender would retransmit the packet that has the sequence number equal to the acknowledgement number of ACK received and all of the subsequent packets.

- How does your protocol handle out-of-order packets?

  - Receiver:
    - If the receiver receives an out-of-order packet, it will be discarded. The receiver will then re-send an ACK for the last received in-order packet.
  - Sender:
    - In the case that out-of-order acknowledgement is received and the acknowledgement number is greater than the one the sender is waiting on currently, the sender can safely assume that the acknowledgement for the current packet was either corrupted or lost during the transmission. This doesn't change the fact that the receiver did receive the packet the sender is waiting

acknowledgement for, and thus, the sender would update necessary parameters (acknowledgement number it is waiting on and either re-setting or stopping the timer) and slide the window past both of the packets. Then the packets are removed from the send buffer. If the acknowledgement number for ACK is somehow less than the ACK number the sender is waiting on, then the sender ignores that ACK.

- How does your protocol provide bi-directional data transfers? (lookup TCP full duplex)

- Each end of the connection will have:
    - A max window size which represents how many packets each end is willing to receive at a time.
    - A send buffer to store data to be transmitted and where the "window" will slide over.
    - Sequence number
    - With both ends having a window size and send buffer, they will both be able to send and receive data, and both sides will know which data they received and are sending since they know if they are receiving and sending and through the **sequence numbers**.

- Does your protocol use any non-trivial checksum algorithm (i.e., anything more sophisticated than the IP checksum)?

- No, we are using the commonly used checksum algorithm for this protocol

- Does your protocol have any other special features for which you request extra credit? Explain.

- Full duplex: we are planning to implement full duplex feature through piggybacking ACKs so that both the client and the sender can receive and send *simultaneously*.
- Multi-client handling
- Post

Description of LLP

Overview:

1. Establish Connection: 3 way handshake between client and server
    a. Initialize the sequence number as zero for both ends of the connection
    b. Window size is passed in from the application for receiving data.

c. The server and the client would have send buffer. The use of a receive buffer is not necessary in this case since UDP will buffer some packets for you -- which we deemed enough for the purpose of this assignment.

2. Both the client and server can send/receive data (Go-Back-N, bi-directional communication)
    a. Sender would keep sending the packets until the receive window is full. When a packet has successfully reached to the receiver, the receiver would send an ACK with acknowledgement number (sequence number of the packet received + packet size). When an ACK is received from the receiver, the sender will slide the window over one packet/segment size in the send buffer and remove the packet from the send buffer, since it means that the packet has successfully reached the other side and there is no need for a retransmission.
    b. **If the window is full**: the sender knows the size of the receiver's window, so the sender will ensure that it does not send more packets than the receiver is willing to receive by keeping track of the window size and how many packets it has sent which have not received acknowledgments. When the window is full the sender will stop sending packets. When the window opens up, the receiver will notify the sender via acknowledgments that the sender can start sending packets again.
    c. **If the packet gets lost:** the receiver would not increment the acknowledge number for the acknowledgement. The receiver will re-send the ACK for the last in-order packet received, when unexpected packet is received. When the sender detects duplicate ACKs, it will retransmit the packets starting with the packet with the sequence number same as the acknowledgement number of the duplicate ACKs. The sender knows that the packet got lost when a duplicate ACK is received because with GBN, out-of-order packets gets dropped in the receiver side, meaning ACKs must be sequential (in-order). (Lost ACK case is described in the question-answer section)
    d. If the packet is corrupted: the receiver sums up the received packets and header and performs rest of the checksum algorithm described below. With the result from this algorithm, the receiver verifies if it matches the checksum in the header of the packet received. When they don't match, meaning the packet is corrupted, the receiver drops the packet and sends duplicate ACK (same as the one sent before), and the sender would respond same as if the packet got lost. (With corrupted ACK, there might be a case where timer might be necessary in sender side. See Timer)
    e. Timer:
        i. When sending a packet, if there is not already a timer running (in other words there are no currently unacknowledged packets), start the timer. Otherwise do not modify the current timer.

ii. The length of time of this timer would theoretically be double the RTT; however, since there is no way to compute this value, we will approximate by trying to use an initial value of 1 second and then we will slowly adjust this value based on whether it seems too long or short until we find a value that works appropriately. (When making the project, we found 500ms to be sufficient.)

iii. When an ACK is received, if there are additional unacknowledged packets, then the timer is restarted. If there are no more unacknowledged packets the timer is stopped.

3. Close the connection
    a. The close() function in our LLP protocol will follow the state diagram provided
    b. If close() is properly called, whether it be on server or a client, the connection should be closed gracefully.
    c. If force closed, it would be handled in application level.

Algorithmic Descriptions:

- Checksum:
    a. Before sending a packet: separate packet header and data into 16 bit words and sum them together (excluding the checksum field)
    b. Compute the one's complement of the sum and put this in the checksum field of the header
    c. When the packet is received, perform step a) again on the received packet.
    d. Then add the checksum field to the sum computed in part c). This should result in FFFF (all 1's).
    e. Take the one's complement of the result in part d), if it is evaluated to 0 then the packet is not corrupted.

Semantics of Connection

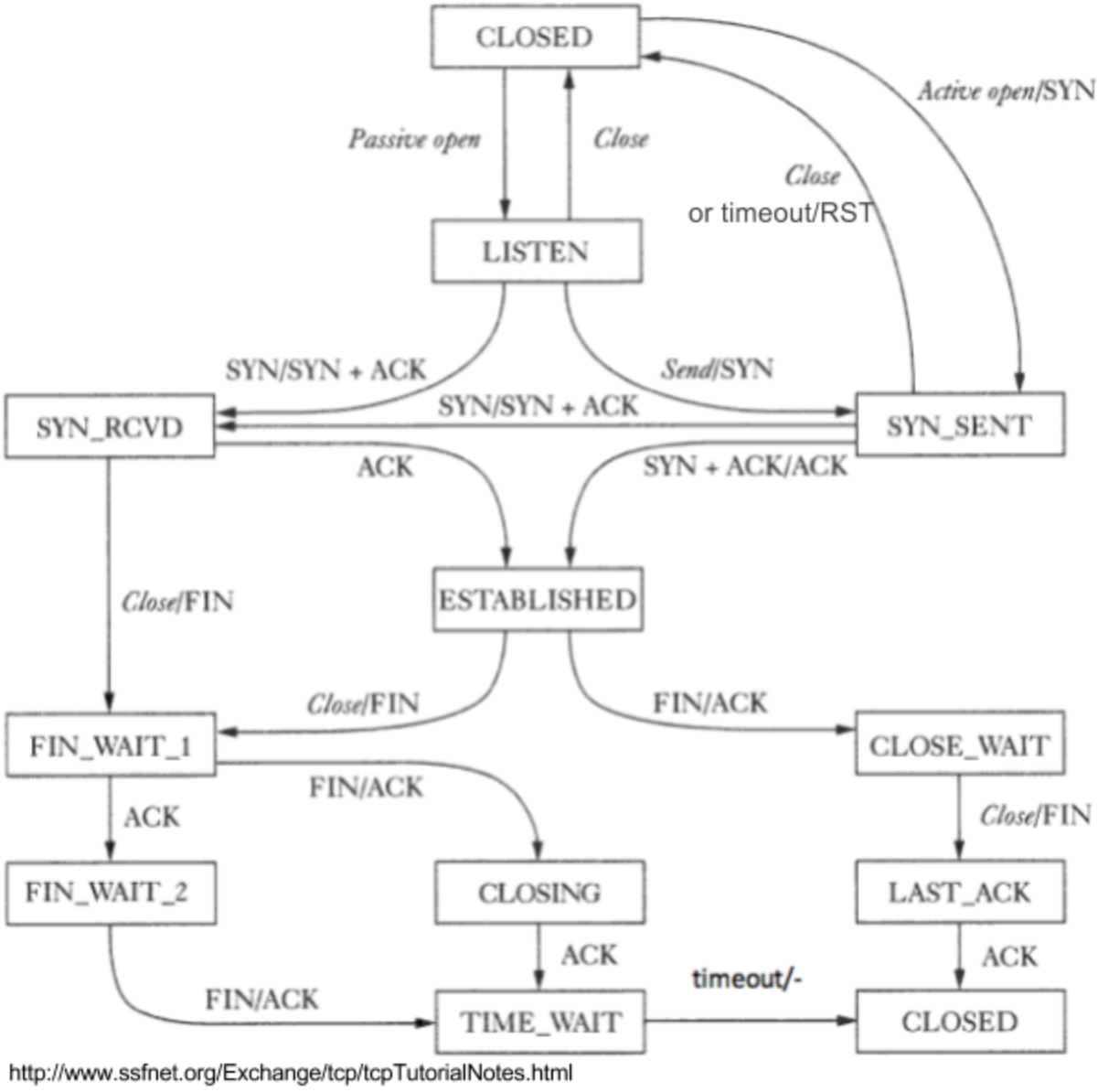
Establishment of Connection:

Client			Server		
Start State	Action	Next State	Start State	Action	Next State

CLOSED	Cannot move to next state. Must wait for server to be passive open.	N/A	CLOSED	Server becomes passive open and is now ready to receive connection SYN from a client. LLP provides the functions socket(), and listen() which the application layer will call to transition to this state.	LISTEN
CLOSED	Client performs active OPEN via our LLP function connect(). Our connect function will send a SYN message to the server with an initial sequence number of 0..	SYN-SENT	LISTEN	Server waits for client connection. Our exported LLP function listen() will wait for the client's SYN message.	N/A
SYN-SENT	Client waits to receive an ACK from the server for the SYN it sent. Also waiting for the server's SYN. Cannot transition until SYN+ACK received.	N/A	LISTEN	Received SYN from the client. Sends a packet to client with SYN+ACK flags enabled; representing an ACK for the client's SYN, and its own SYN. Server will also send its initial sequence number of 0 to the client. The accept() function in our protocol will handle the sending of this packet.	SYN-RECEIVED
SYN-SENT	Receives SYN+ACK from the server. Sends a server an ACK to respond to the server's SYN. Client allocates space for a send window and receive window. Connection establishment for client is done.	ESTABLISHED	SYN-RECEIVED	Server waiting for an ACK for its previous SYN. Does not transition state until then.	N/A
ESTABLISHED	Waiting for server to finish	N/A	SYN-RECEIVED	Server receives ACK from client. Server is done with connection	ESTABLISHED

	establishing a connection.			establishment. (Three-way handshake finished)	
ESTABLISHED	Now that both client and server are done, establishing connection, both endpoints can send and receive data to each other.	N/A	ESTABLISHED	Now that both client and server are done, establishing connection, both endpoints can send and receive data to each other.	N/A

Connection State Diagram:



http://www.ssfnet.org/Exchange/tcp/tcpTutorialNotes.html

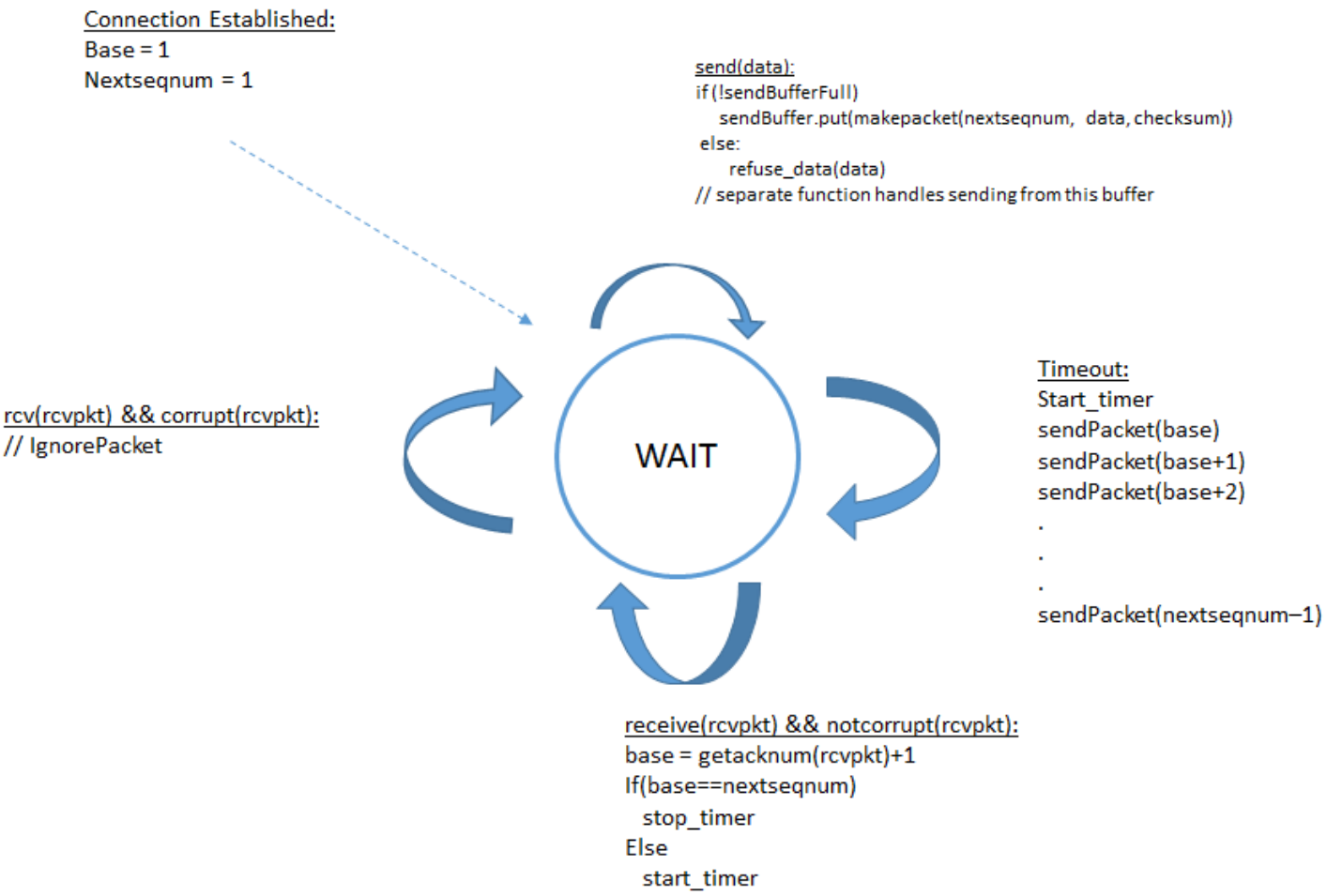
Closing a Connection:

Client			**Server**		
Start State	**Action**	**Next State**	**Start State**	**Action**	**Next State**
ESTABLISHED	Send a packet with the FIN flag set, thereby requesting to close the connection.	FIN-WAIT-1	ESTABLISHED	N/A; Server proceeds the same.	N/A
FIN-WAIT-1	Client is waiting to receive an ACK and a FIN from the server. Client can still receive data from server at this time.	N/A	ESTABLISHED	Server receives the FIN from client. Server then sends an ACK to the client acknowledges it's want to close; however, server now needs to let the application know that it's closing and wait for the application.	CLOSE-WAIT
FIN-WAIT-1	Client receives the ACK, but is still waiting for the FIN indicating that the server is ready to close.	FIN-WAIT-2	CLOSE-WAIT	Waiting for application to be ready to close.	N/A

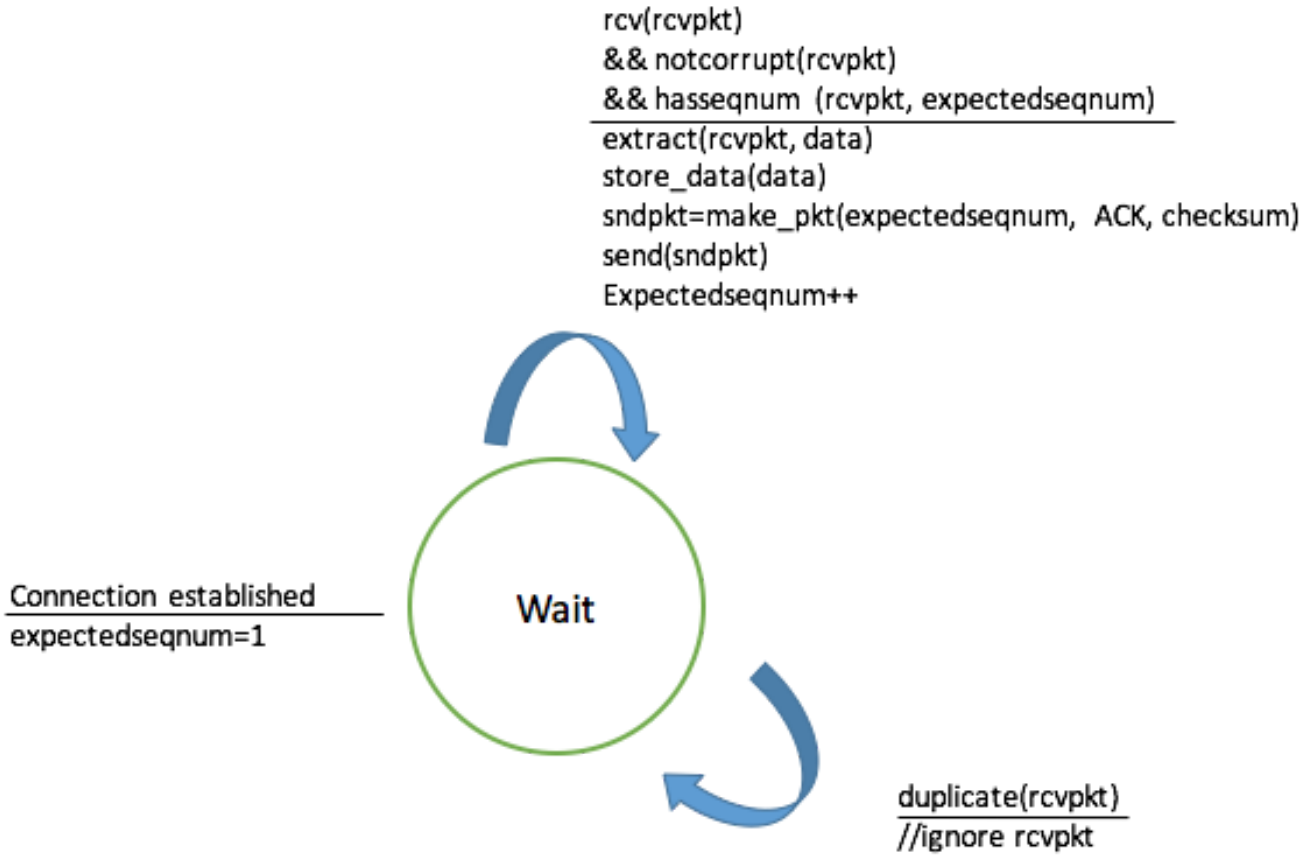
FIN-WAIT-2	Waiting for FIN from server.	N/A	CLOSE-WAIT	Application tells server it's ready; server then sends FIN to client to let it know it's ready to close the connection.	LAST-ACK
FIN-WAIT 2	Receives FIN from server; sends ACK in response.	TIME-WAIT	LAST-ACK	Waiting for ACK from client for confirmation that the FIN was received.	N/A
TIME-WAIT	Client waits for a certain period of time.	N/A	LAST-ACK	Server receives ACK and closes the connection.	CLOSED
TIME-WAIT	After the period of time expires, client closes.	CLOSED	CLOSED	Connection is closed for server.	
CLOSED	Connection is closed.		CLOSED	Connection is closed.	

Post-Connection Establishment

FSM at Sender Endpoint:



FSM at Receiver Endpoint:



LLP API

- socket(int localport, InetAddress localAddress, SocketAddress remoteAddress)
    - Create an LLP socket.
    - In our protocol implementation this socket will be instantiated as a UDP socket.
- receive(int maxsize)
    - Maxsize - the max amount of bytes the application is willing to receive at a time
    - Receive data. Application specifies how many bytes it wants to read as a parameter.
    - When the receive function is called, our protocol will retrieve that many bytes from the buffer (if available) and return it to the upper layer.
- send(byte[] data)
    - Send data passed in.
    - When the application calls send we will add the data to the send buffer.
    - When the receive window is not full on the other end, we will form packets from the in the send buffer (based on our maximum segment size), and send these packets to the other end.
    - Data is removed from the send buffer only when an ACK is received from the other end after the sent data has been successfully received. Otherwise, we keep it in the send buffer in case we need to re-transmit. In order to make it even easier, we created a hashmap from sequence numbers to packets, so if we missed a packet we can This way, if we have to retransmit data, we will not have to request it from the application again.
- accept()
    - Listens and accepts a new connection.
    - Handles the semantics of the three-way handshake for the server.
- connect(InetAddress address, int port)
    - InetAddress handles both IPv4 and IPv6 connections
    - Connect to a remote address at specified IP address.
    - Initiates a three way handshake with a server.
- bind()
    - Bind LLP socket to a port.
- close()
    - Close the connection.
    - Our protocol implements the states to close a connection specified by the LL Connection State diagram. This function, close(), will initiate the conversation between the two endpoints.
- setMyWindowSize(int windowSize)

- Sets the window for this end of the connection. When a packet is transmitted to the other end, it will communicate this window size.

LLP Header

Sequence Number						
Acknowledgement Number						
Reserved Bits	Checksum	ACK	RST	SYN	FIN	Window Size

Sequence Number (32 bits):

- If SYN flag is set, this is the initial sequence number. Sequence number of first data byte and the acknowledgement number in corresponding ACK are this sequence number + 1.
- If SYN flag is **not** set, this is the sequence number so far for the current session.

Acknowledgment Number (32 bits):

- If ACK flag set, value in field is the next sequence number the receiver is expecting.
- Acknowledges receipt of prior packets.
- Initial ACK sent by each end acknowledges the other's initial sequence number.

Reserved Bits (2 bits):

- Reserved bits of data in the header. Will always be two 0 bits.

Flags (4 bits):

- ACK (1 bit)
  - If the acknowledgment field is relevant, ACK bit set to 1.
  - All packets after the initial SYN packet sent by the client should have ACK flag set.
- RST (1 bit)
  - Indicates to reset the connection
- SYN (1 bit)
  - Synchronize sequence numbers.
  - The first packet sent from each end should have this flag set.
- FIN (1 bit)
  - Indicates there is no more data from sender

Window size (10 bits):

- The size of the receive window.
- Number of packets that the sender of this segment is willing to receive.

Checksum (16 bits):

- Used to check for errors in header and data.

Max Data Size (1024 bytes):

- We queried MTU from the docker image which was 1500 bytes. Since we should choose a max data size somewhat smaller than the MTU in order to fit the headers, we chose 1024 bytes which seems reasonable.

TODO:

- Add more specific details about the send buffer and when data in the buffer will be freed?
- Add more specific details about what if the application tries to read from an empty buffer
- How are sequence numbers generated, when?
- When is buffer space allocated?

Questions

1. Client and server have receive buffer? yes
2. Sender Buffer? Can we not have one and constantly request the packets from the upper layer?

   (if window is full) have send buffer;
3. Full duplex: what to change from bi-directional?
4. Should take care of push, urgent? (EC)
    a. Urgent is for transport layer
    b. Push is for application layer; if want the packets to go to application right away
5. Message boundary: UDP should take care of it
    a. 3 way handshake
    b. Sequence number to ensure in-order and guaranteed delivery
6. Top of udp, so don't need to know source and destination port?
7. Timeout amount?

   Double RTT; might want to test with 1 second, and see if it's slow enough for a packet→ continue reducing
8. Max data size?
    a. Depends on buffer;
    b. Limitation on IP layer: MTU?
9. Multiple clients for a server
    a. Equal division of the server's buffer for multiple clients

Sender buffer is full: block application from sending

- Multithreading? (application tries to read but if buffer is empty, one way is to return empty buffer)

WAN emulation, see the script → will grade it based on network emulations

- Write some simple pseudo code to see if we have everything

Server needs to stop gracefully → close()

Bidirectional: Can't do get() at same time you're doing a put()

Full-Duplex: Piggybacking acks, two separate buffers, two separate states

- Wouldn't have data & acknowledgement (so you don't need two sequence numbers in header) → state diagram might look different

Protocol should have port numbers in it

Max Data Size: MTU - (IP Header + UDP Header)

You can also query the socket interface will give the MTU

Don’t deal with fragmentation (could be an extra)