Reliable Data Transfer : 'Stop and go' protocol

`STOP AND GO` Protocol

THE ALGORITHM: the sender and receiver are both sending one message at a time. The sender sends a piece of data and waits for an acknowledgement from the receiver that the data was received successfully before sending the next piece of data.

Assumptions

Transfer of files in other formats should be first converted to text data. The Timeout, RTT, Sender port number, Receiver port number, Packet size for network and transport layer can be changed in config.py

Both the sender and receiver code are contained within the `sw.py` file. In order to distinguish between sender and receiver actions, the following observations are made:

- Messages with type `ACK` will only be received by the sender and sent by the receiver
- Messages with type `DATA` will only be received by the receiver and send by the sender
- The method `send()` will only be called for the sender

In a world where the underlying network has reliable data transfer properties, the sender and receiver will take turns, sending messages and acks to each other alternatively until all the data has been transmitted. However, that is not the case. Here are the possible scenarios that may occur and how we handled them:

Special cases for the receiver

1. *Packet received is corrupt:* When the packet received is corrupt, we cannot tell what the sequence number is or if it is duplicate data. Resend the last ACK. The sender will ignore ACKs with sequence numbers that do not match the current sequence number. The sender will timeout eventually and resend the last data packet it sent.
2. *Packet received is a duplicate:* This occurs when the sender times out due to either receiving corrupt ACKs, duplicate ACKs, or no ACKs at all. Resend the last ACK. Sender will eventually resend the next DATA packet.

Special cases for the sender

1. *ACK received is corrupt:* Do nothing. The timer will eventually go off, resending the last DATA packet. This will trigger the receiver to send

another ACK packet with that sequence number. If we resent the last data packet every time we received a corrupt ACK, we would be flooding the channel with duplicate messages.

2. *Timeout:* When there is a timeout, this indicates that the sender did not receive the ACK it was waiting for. Resend the last data packet sent and restart the timer.

3. *`Send` calls from above when waiting for ACKs:* In order to handle this, we spawned a thread for each call made to `send()`. The thread waits for the sender state to switch to `WAIT_FOR_APP_DATA`, grabs the lock, sends the data, and restarts the timer.

4. *Waiting for the last ack:* There is a case where the sender may send the final message and since `send()` returns True on success, the sender may close the connection if not careful. The last message may be corrupt or dropped by the network so the sender must wait to receive the ACK from the receiver indicating successful transfer. The method to sit and wait for this is called in the `shutdown()` process.

Advantages:

- Stop and wait requires minimal state/memory. It only needs to hold a couple flags to record state, the last packet sent, a timer, and a lock.
- In a very unreliable network, stop and wait may reduce the number of messages in the channel since it individually asserts each message is successfully received before sending the next one.
- In a network with very small propagation delay and small RTT, this algorithm will transmit the data quickly while minimizing the number of packets in the channel to do it.

Disadvantages:

- This is a slow, serial algorithm. If you want to send large pieces of data, this protocol will take a long time simply due to the RTT between each send.
- Stop and wait has very low throughput. It takes 1RTT in between each message sending in the best case. In the worst case where there is corruption or data lost, it will be at least a couple RTTs before the subsequent message is sent.
- If the propagation delay on the network is large, the RTT will be large and this will have a direct negative impact on speed/efficiency.

Calculating the checksum

We chose to implement the checksum algorithm the way its implemented in actual protocols instead of the simplified version. We find bit manipulation cool and the fact that *the checksum of a message containing the ones complement of the checksum of its data will equal 0 if not corrupt* fascinating. The basic steps are as follows: - Break the data into chunks of 16 bits and add them altogether - Carry the overflow if the sum is now greater than 16 bits - Take the ones complement of the checksum and return this value

How to detect corruption

When calculating the checksum for a newly created packet, the value for the checksum was 0. The ones complement of the checksum is then put into the checksum field. Upon packet arrival, take the checksum and if the checksum is 0, you can be relatively sure that no bit corruption occurred. You can only be relatively sure because reordering 16-bit chunks in your message can result in the same checksum if you split the chunks the same way. However, in a network where no malicious activity is assumed, bit corruption will typically be random and will be caught by using the checksum.

Group memeber Details

Arun Inani(2017A7PS0085H)

Abhishek Bhardwaj(2017A7PS1497H)

Akshat Gupta(2017A7PS1699H)