Application Assumptions

We are using UDP sockets to transfer data.Hence reliability is a major concern.We are building an application which consists of a client and server.Here is our basic application:

- The client sends a message to the server and the server writes the message to a file .The name of the file is *<clientip>_<client_port>.txt*. This file is stored in a folder called *client_messages.*
- Multiple clients can communicate to the server simultaneously.This is achieved by creating a thread for each of the clients.Each time a client file is run,a new thread is spawned.
- The server sends an ack,once it receives a message from the client.The client waits for 2 seconds before retransmitting the same message.
- The client cannot send a message if an acknowledgement for a previous message has not been received.
- On the server side,if server receives a packet having sequence number less than expected sequence number(i.e. the case when an ack is lost),the server resends the ack .Although this is not included in the stop and wait protocol given in textbook,but we customised it as per our requirements.
- The server runs on port 9999 and hence it has been hard coded in the client file.This can be modified but for simplicity sake we have fixed it.
- Sequence numbers are generated randomly for server and each client.Server has a different sequence number for each client as well.The sequence numbers are generated randomly between (100000,999999) for the possibility of multiple clients connecting to the server.
- Since UDP is a connectionless protocol, server does not have to maintain the state of each client.It just has to listen at its socket for any incoming data packet.It is a barebone no frills plain vanilla protocol.It is as good as talking to IP layer.Hence,there is no need for handshaking n UDP sockets.
- The timeout is fixed and assumed to be 2 seconds here,for depiction purposes.
- One major assumption is the client keeps on resending messages until an ack is received.Although,this consumes a lot of time in hypothetical networks having higher loss,it still is beneficial.Proper conditions can be put in for the same.
- Since there is no handshaking,similarly there is no quitting protocol.The client simply leaves by exiting the file,and the server keeps listening until the server file is closed.

Applications of the protocol

Our application keeps sending packets until an ack is received for that message.Although,it significantly affects throughput and the client has to wait to send the next message, it still has useful applications.

- This can be used in an **email client** like Thunderbird or Gmail Suite,who send an email to a server.The Thunderbird can act as a client and keep sending messages to the client server which then forwards it to the receiving server.

- The same protocol can be used when the receiving server picks up an email and sends it to the receiver client.Here the receiving server would act as a client and the receiver client would act as the server.Our implementation is basic but can be scaled and modified accordingly as per convenience.

- One more interesting application for this protocol is in the DNS system.The client types a website name, the packet goes to the DNS server, and then the DNS server searches in the cache for the mapping.If no matching is found,it creates an entry for the website.Here,the user acts as a client and the DNS server as the server.If by any chance the user’s request gets corrupted or is lost, the protocol retransmits the message.hence,it can be used in the DNS server systems with appropriate modifications.

Network assumptions

- We have used *localhost* for development and testing purposes ,so the application works well on localhost.Since,we have used *localhost* we were unable to simulate real network conditions.
- Using a tool *netem* we were able to test our protocol for packet loss and constant packet delays.
- **Packet delays** are handled by retransmissions.The client waits for a couple of seconds before sending the message again.
- **Packet losses** can occur on both server and client part.These losses are handled by retransmitting the packet by the client after a fixed timeout.
- **Packet corruption** was tested but UDP by default drops corrupted packets.UDP has a checksum feature built into it,so in cases of packet corruption the corresponding packet was not delivered.Packets were retransmitted here ,as usual.
- **Packer reordering** has not ben tested.Since each client sends one message at a time, there is no scenario for reordering here.If there existed a file transfer application where a server sends a file to a client,which has to be broken into packets reordering packets becomes valid there.
- **Jitter** in IP networks is the variation in the latency on a packet flow between two systems, when some packets take longer to travel from one system to the other.This is a subcase of adding delay to every packet.in either case,the protocol retransmits the packet.