# Design Document

## Contributors:

Ameya Salankar – 2017A7PS0182H  Atharva Sune - 2017A7PS0183H

Harsh Kumar – 2017A7PS1584H  Lovekush Sharma - 2017A7PS0125H

Arnav Buch - 2017A7PS1722H

## Overview

The designed protocol is for a file transfer application. This protocol assumes that file transfer occurs between two entities – the *server* and the *client*. The server has access to all the files. The client 'requests' the file, i.e. it provides the path for the file. This protocol assumes the availability of a UDP socket service.

## Packet Format

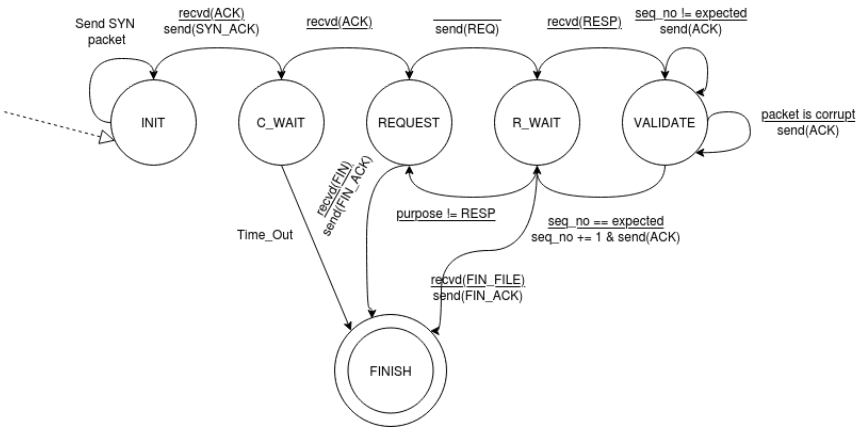
The protocol defines a format for the data carried inside the UDP packet. The data consists of two parts – the *header* and the rest of the packet. The header has a fixed format. The rest of the packet is the data to be delivered. A packet may contain only the header without any data.

The header is defined to be a string of text between the markers **`$*`** and **`*$`**. This string of text consists of key value pairs in the format **`key:value`**, separated by a comma '`,`'. The header has to contain three compulsory keys – *checksum*, *seqno* and *purpose*. The first key has to be *checksum*. This is a 64 bit checksum of the rest of the packet without the checksum value, generated by the SHA-256 algorithm. The *seqno* is the sequence number of the packet. The accepted values for *seqno* are non-negative integers. The *purpose* is the role of the packet. The accepted values of the purpose are `SYN`, `SYN_ACK`, `ACK`, `REQ`, `INVFILE`, `RESP`, `FIN_FILE` and `FIN`. The header may contain any number and any type of keys besides the three defined earlier. An example packet is shown below.

```
$*checksum:014a1176e9a374ced548edde5bd08ef2ee7876b73d0ba6cc43da6309cc2d3bd4,seqno:29,purpose:ACK*$
```

## The Client

The client is defined to have 6 states. It operates as shown in the state diagram.



(text above the horizontal line is an event and the text below it is the response)

In the INIT state, the client is to send *SYN* packets till it gets an ACK packet from the server. The duration of this activity is implementation dependent, but a timeout of 90 seconds is recommended. If there is no ACK packet till the timeout, the client may assume that the server is offline. This shows the client that the server is online. The client notes down the sequence number here and acknowledges with the same sequence number. It expects a sequence number which is 1 + sequence number received. It should then reply with a SYN_ACK packet and wait till an ACK packet is received from the server. At this stage, the client knows that a connection is established.

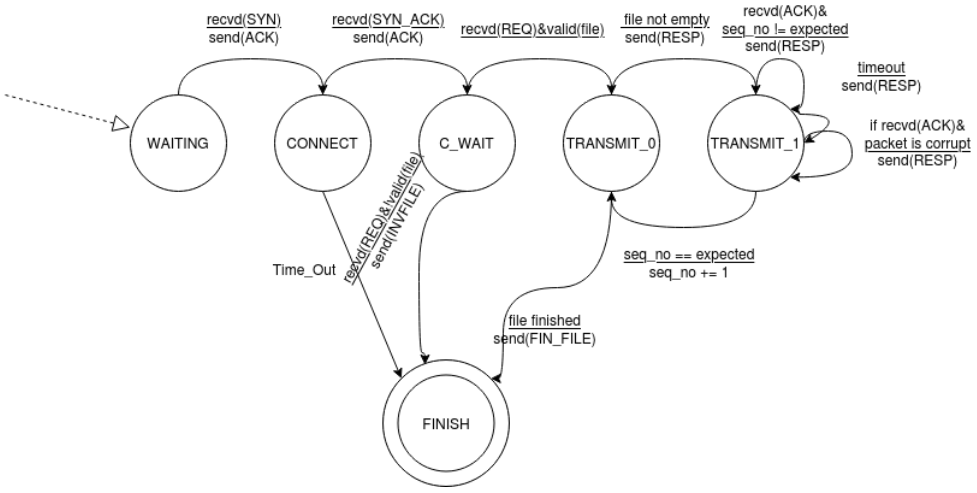
The client now requests the server with a file by giving the path of the file on the server in the *file* field of the header and setting the purpose of the packet as REQ. If the server responds with INVFILE, then the requested file does not exist on the server. If the server responds with RESP, then the file exists and the file transfer has started. If any other packet is received, the client must send the last sent packet again, in this case, the REQ packet.

Once the responses start coming in, the client stores the data into a local file and replies with an ACK packet having the correct sequence number. It waits for more RESP packets until a FIN_FILE or a FIN packet is received. If any other packet is received, or if the received sequence number is incorrect, or if the packet is corrupt, the client must send the last sent packet again, in this case, the last sent ACK packet.

If a packet with purpose FIN_FILE is received, it means that the file is sent completely and the file transfer is successful. If a FIN packet is received, this means that file transfer has stopped abruptly due to large network delays or some other reason. The file is not received fully here.

## The Server

The server has 6 states as shown.




In the WAITING state, the server waits for SYN packets from clients. Once any SYN packet is received, the server replies with an ACK packet with *seqno* as 0. In the CONNECT stage, if a response is not received within a timeout, the server is expected to end the connection. If a SYN_ACK packet is received, the connection is taken to be as established. The server now waits for REQ packets.

When a REQ is received, the server looks for the file specified by the path in the *file* field of the header. If the file is not found, an INVFILE packet is sent, denoting that the file was not found. The connection then terminates. If the file is found, the server must start sending the file in RESP packets. After sending out a RESP packet, the server must start a timer and wait for an ACK packet of the correct sequence number. If a packet is received with an incorrect sequence number, or if the packet received is corrupted, or if the timeout

occurs, then the last sent RESP packet should be sent again. If there is no response after 30 timeouts, the server should send a FIN packet for 30 more timeouts and end the connection.

Finally, if the no data remains to be sent, the server should free all the resources and send a FIN_FILE packet. This tells the client of the successful file transfer. The server then waits for a FIN_ACK packet for a defined number of timeouts, and then exits.

## Implementation Details

The implementation implements the above protocol and takes some additional design decisions.

In the client's implementation:

- The timeout duration for the program is set to be 90 seconds, i.e. if there is no response at all from the server during these 90 seconds, the program terminates.
- The timeout duration for retransmission of the control packets (`SYN, SYN_ACK, REQ`) is 3 seconds.

In the server's implementation:

- The default port of the server is kept as 8080
- The max number of bytes of data in a packet is set to be 450 bytes.
- The timeout duration for the program is set to be 150 seconds.
- The default timeout duration for any retransmission is set to be 3 seconds.
- The server runs on a main thread which handles the incoming data. Each client gets its own server thread. The main thread receives input and routes it appropriately to the correct server thread. A queue is shared between a main thread and a server thread. The server thread continuously checks for data in the shared queue. Every server thread gets a different queue.
- When the file transfer ends (if file transfer is complete or an error occurs), the server sends out `FIN_FILE` or `FIN` packets while waiting for a `FIN_ACK` packet. If a timeout occurs before that, the server exits without waiting for the `FIN_ACK` packet.