Reliable data transfer using UDP

FINAL REPROT - FOR THE FULFILMENT OF NETWORK AND INFORATION ASSURANCE PROECT

THE UNIVERSITY OF MEMPHIS

SAURAB DULAL

UIID: U00658143

Table of Contents

List of Figures

List of Acronyms

UDP	UDP file download system architecture
TCP	Flow Diagram - UPD file download process using client-server model
SRP	State diagram, client server - request response process
ABP	State Diagram - Alternate bit protocol
SR	Class Diagram UDP Client Program
AB	Class Diagram UDP Server Program
SHA	Secure Hash Algorithm
HTTP	Hyper Text Transfer Protocol
ACK	Acknowledgment
SOCK	Socket
DGRAM	Datagram

1. Introduction

"Reliable data transfer using UDP (User Datagram Protocol – RFC 768) for file transfer" is a client-server application. The reliability over UDP is achieved using Alternating Bit(AB) or Selective Repeat(SR) protocol. UDP, unlike TCP (Transmission Control Protocol), is an unreliable, connectionless data transfer protocol existing in the transport layer of TCP/IP architecture. The primary purpose of this project is to download a big file from a server residing in different IP location using one of these protocols i.e. either SRP or ABP.

Alternating Bit Protocol(ABP) is a connection-less protocol used to transfer one-directional message between client and server. It is, in fact, a simplest form of a sliding window protocol where sequence number alternates between 1 and 0, and the window size is of one. In ABP, a request or ACK message is sent from a client to the server with AB 1 or 0, and upon receiving a response from the server, AB is flipped (1-0 or 0-1) and again a new request is sent with new AB info. If a desired response is not received from the server, a request is again sent by the client without flipping the AB. Thus, looking at the AB sent by the client, the server decides whether to send a new packet or to retransmitted the old one.

Selective Repeat Protocol(SRP) is also connectionless protocol. But unlike ABP, server parallelly sends a certain number of packets (default 10 in our case) for each new request from the client. Once receiving the packets from the server, clients somehow verifies sequence of the received packets. Upon verifying, client can know about the missing packets – if there are any. Client sends a request for those missing packets and repeats the process until all the packets for a particular request are received. Otherwise, it requests for the fresh set of packets from the server.

This project has implemented both Selective Repeat protocol and Alternating Bit protocol along with UDP. In regard to the project, a client makes an HTTP request to the server requesting for a file to download. Server checks if the requested file exists or not. If the file exists, server splits the file into many segments of specific UDP length (say, 1024bytes). A packet is created for each split, which contains checksum as SHA1(20-byte hash of data), data length, sequence number, and/or alternating bit. Client, upon receiving the packet, verifies it, re-computing the hash, and performs a necessary action which may be saving the packet or requesting for retransmission. Retransmission can occur in various scenarios. Such as, if a packet is lost, or if a checksum is not validated, or if some sequence is missed –selective repeat case. All the action performed during this process are saved on a file and also are displayed in the console – i.e. both of client and server.

2. Problem Statement

The application should consist a client program and a server program. The server program hosts files and responds to requests for files. It breaks the requested file into segments and sends them to the client over UDP. The client program takes a file name as input and requests the file from the server. The programs should handle UDP packet losses to make sure that the entire file is correctly received by the client program. Code needs to include implementation of the **Alternating Bit** and **Selective Repeat protocols**. Refer chapter 3.4 (Principles of Reliable Data Transfer) - Computer Networking, Kurose to understand these two protocols.
Client program should print detailed information about any requests sent and data received at byte level (e.g., time X received byte Y to Z) during the file download. Server program should print detailed information about any requests received and data sent at byte level. Client program and server program should be run on different machines in different networks.

3. Architecture Description

The top-level architecture of the project is shown below in figure 3.2. Basically, it contains a client process and a server process. The client process implements various methods via different classes such as client connection, transport, and file handler. Client connection creates a socket, handle requests and responses. It also holds some parameters, such as IP address, port number, message, error bit, alternative bit handler and so on. Transport creates the packets, implements selective repeat and alternating bit, verifies the received packets and so on. File handler performs necessary action to the packet sent by transport, it saves the packet, or sends back the error message to the transport – if the file is not saved. In addition, the client process also contains methods such as exception handler, alternating bit and selective repeat handler, and so on.

The server facilitates the file transfer process via various methods existing in different classes such as server connection, server packet, and file handler. Server connection creates a socket, send and receive requests and responses. Like the client connection, it also holds parameters such as IP address and port number of the client to sends back the response. File handler class checks if the requested file exists or not, if exist, it reads the file and divide it into no of chunk of a specific size. Each chunk is sent to the server packet class, where the give chunk(data) is serialized, checksum, length, sequence number is calculated. And finally, packet is made ready to be sent to the client. Other various functions are also created on the server side such as selective repeat, alternating bit, exception handler etc. to support the file transfer process. Generally, these methods perform same action as that of corresponding client functions.

In addition to these, all the intermediate actions are printed on the client and on the server console. These actions include, the requested file name, file status, sequence number sent and received and so on.

3. 2 Architecture Diagram

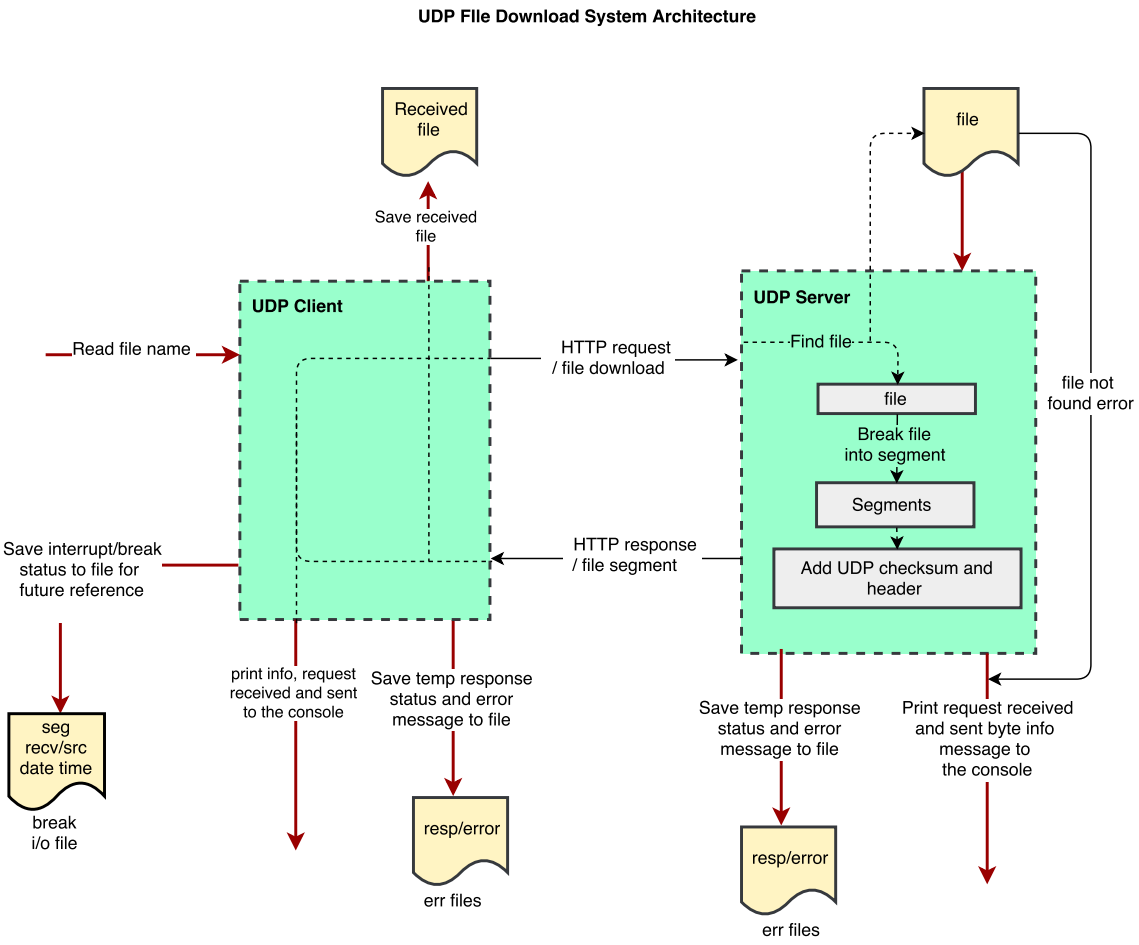

Figure 3.2: UDP file download system architecture

3. 3 Proposed but not implemented functionalities.

The propose to handle unexpected network problem i.e. client to save the download segment information, and segment as well if the network gets broken is not implemented at this point. It had aimed to facilitate user to start download from the place it was broken earlier - if desired. Also, saving each intermediated state to error files is not implemented till date of this report preparation, but is underway towards the development.

4. Flow, State and Class Diagram

The high-level flow, state and the class diagram of the project are shown below. Some the functionalities may not exactly depict the implementation. Thus, the diagrams are intended to give a general upper level overview of the project only.

4. 1 Flow Diagram

The flow diagram of the UPD file transfer is shown in the image below.

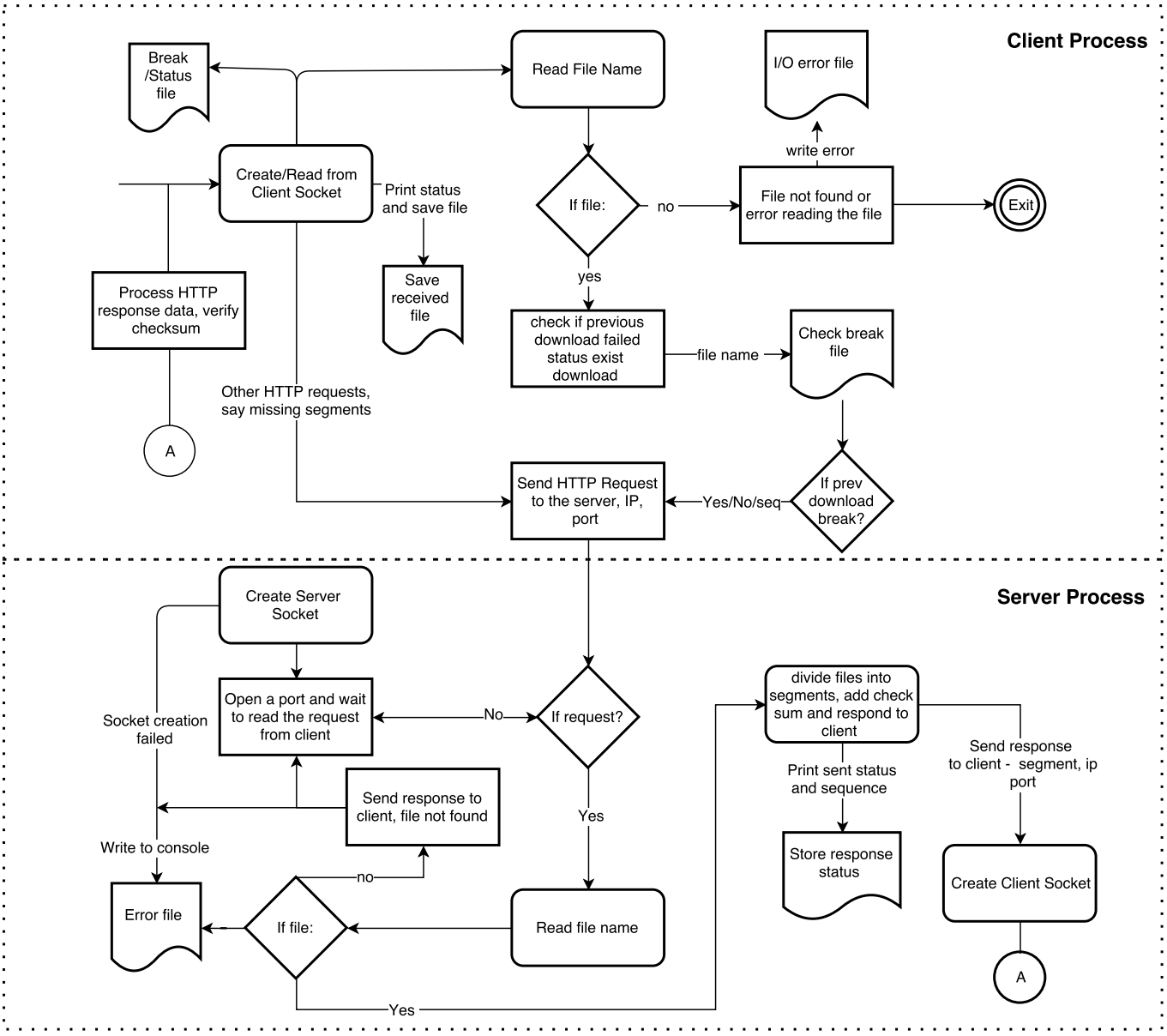

Figure 4.1: Flow Diagram - UPD file download process using client-server model

Note: The symbols used in the flow diagram are adopted from the standard flow diagram design.

4. 2 State Diagram

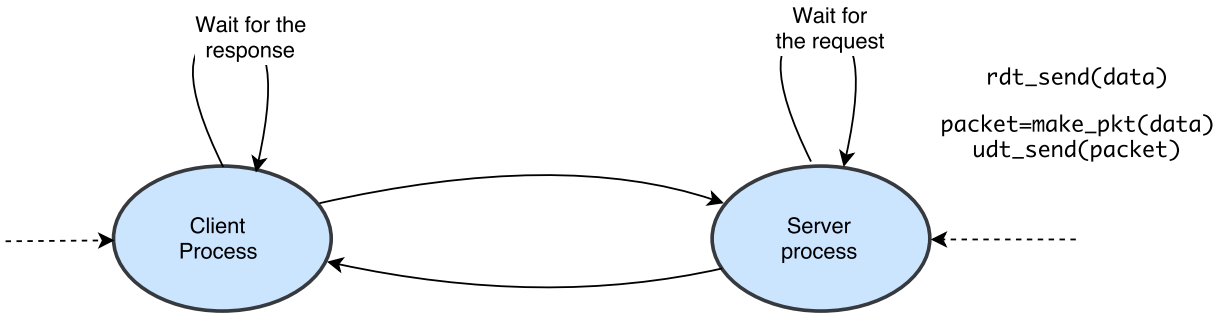

Fig 4.2.1: State diagram, client server - request response process

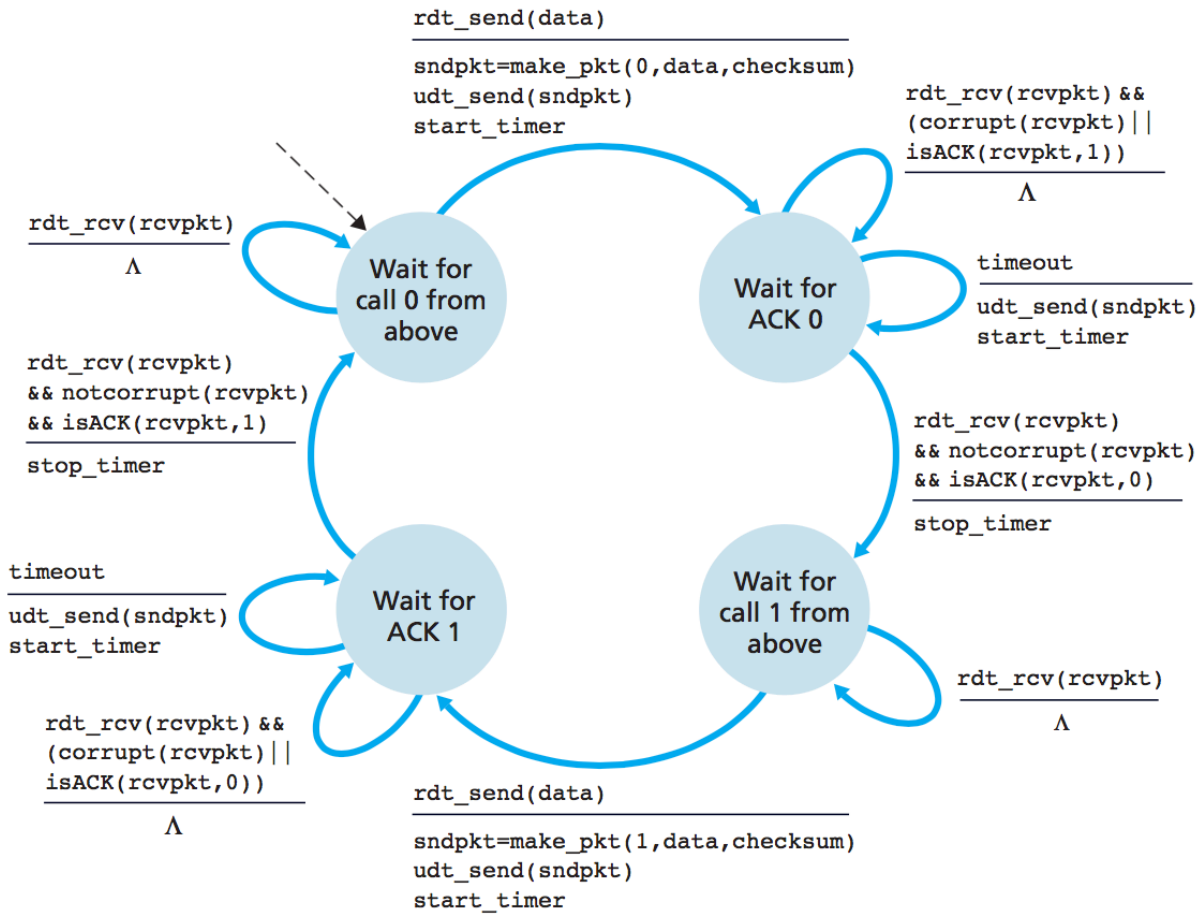

Fig 4.2.2: State Diagram - Alternate bit protocol

Reference: Computer Networking – A top down approach 6th edition, Kuros, Ross

4. 3 Class Diagrams

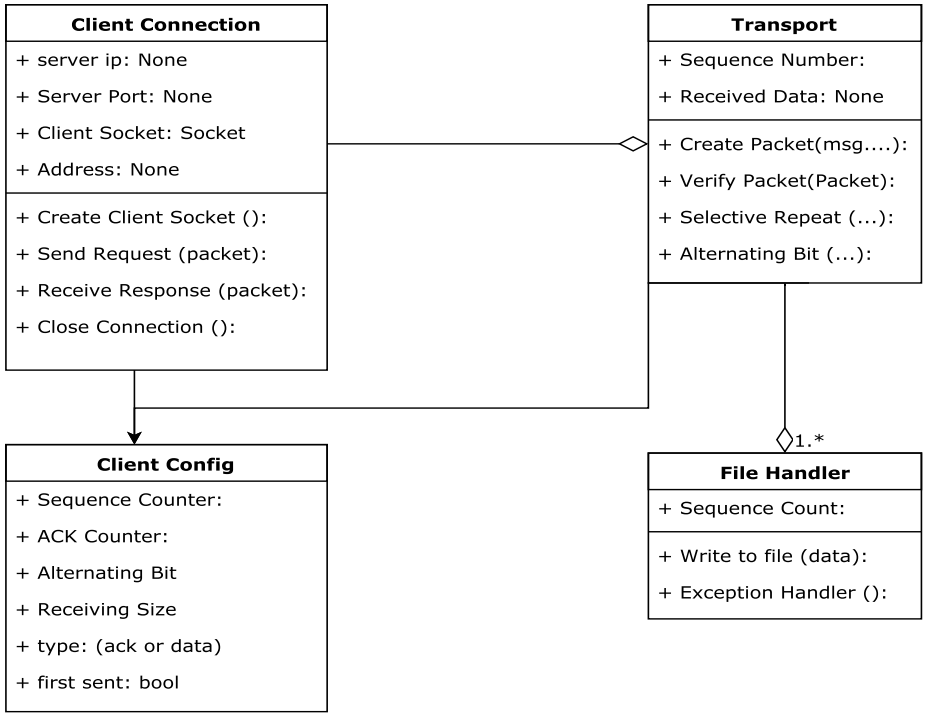

Figure 4.3.1: Class Diagram UDP Client Program

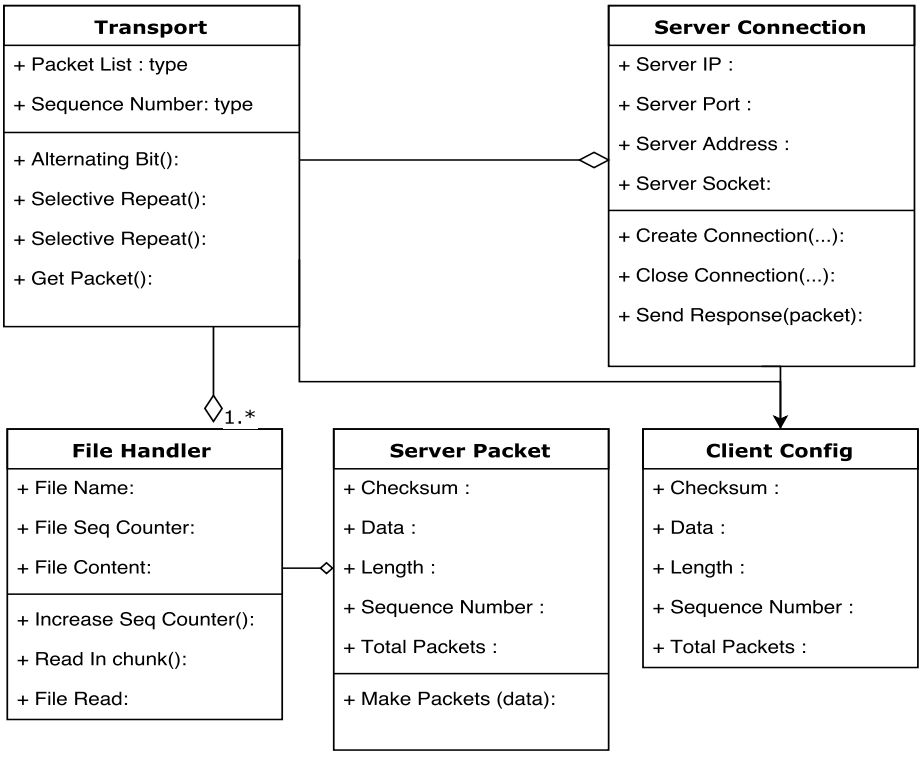

Figure 4.3.2: Class Diagram UDP Server Program

5. Implementation

"Reliable data transfer using UDP" implements a file transfer mechanism to send a big data file from the server to the client as a response to a client request. As UDP is an unreliable protocol, the reliability is added to the transfer process either by alternating bit or selective repeat protocol. Since these protocols [Section 1] are contrasting to each other, on them can be selected for the file transfer process.

5.1 Description

Before the execution of client/server programs, it is very important to choose either selective repeat or alternating bit at first, and this can only be done by going to the code itself. It is very "important" to choose same method on the both side, otherwise program execution terminates or may continue on loop.

Once the selection is done, the server program is executed first, and upon successful execution it waits for the client request.

- execution of client/server program: `< python UDPClient/UDPServer.py >`

Similarly, client program is also executed, and prompts user to input the file name to download from the server. Now the client makes all the necessary setup up before sending request to the server. Setup such as, initialize server IP and port, bind IP and port to create address, and finally create socket.

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

It also imports various global variables (*type, sequence counter, alternating bit etc.*) from the *client config*. The type variable tells sever if the request is of type acknowledgement, or data. Sequence counter stores the information about all the received sequence number and alternating bit to assist the same process. In case of SR, AB is always set to 1 and it acts as ACK message only. Upon completion of these setup, client creates a UDP packets and sends request to the server for the initiation of file transfer.

Server receives the request, verifies if the file exists or not. If the file exists, it creates small chunk of default size 1024 and stores it in a dictionary. Packets are created out of these chunks and are also stored in a dictionary. Each UDP packets created by server contains following information

- *checksum* : Check sum of the data
- *length* : Length of the data
- *seqNo* : Sequence number of the packet
- *msg* : Data or message to be send
- *total_packets* : Collection of all the packets
- *total_number_of_packets* : Total no of packets available

Now on the basis of selected method, server start sending the packets, either 1 at a time in case of ABP, or 10 packets in case of SRP. Server response can be a new transmission of data, or re-transmission in case of packet loss. Moreover, if file is not found at the first hand, it sends a "file not found" exception to the client.

Client waiting for server response, receives the packet (in some case it may not receive as well), send it to the transport for the verification of the packet. Transport, dissect the packet, calculate the checksum and length, verify it against the received parameters and upon verification, it sends the data to the file handler. And so, file handler writes the data to the file. During this process, client may undergo time out, or may receive distorted packets, in both of these cases it requests the server for the retransmission. The basic rule for retransmission in case of ABP is, if the received AB from client is different than the one existing in the server, server performs retransmission. Similarly, in case of SRP, client notifies server about the specific missing packets, and server retransmits them. This process continues until all the packets from server is received by the client. Client upon receiving all the packets from the server, sends a ACK with type = 'c' (c for complete) to the server. Receiving ACK 'c' server closes the connection and terminates the transmission process with transmission complete message. There also exists some other helper functions such as exception handler – to track/handle, and print exceptions, missing sequence verifier - client process, to verify missing sequence in case of SRP and so on. All the class and function names are presented in the Appendix.

6. Technical Specifications

- Python 3.x+ for client and server programming
- Python socket library for the network socket
- *Hashlib* for the computation of hash
- *Pickle* for the serialization of data
- **Note:** Whole project is developed under a virtual environment so the project contains all the necessary libraries and the python environment as well.

7. Limitations

a. Selection of method ABP or SRP should be done by going to the code itself
b. The program cannot handle unexpected connection loss cases, so it needs a manual restart of the process in case of connection loss.
c. Only data file can be transferred, it doesn't support transfer of media or any other types of files
d. The program may contain various bugs due to the lack of proper code review

8. Future Enhancement

a. To support various file type such as audio, video, image etc.
b. Functionalities to handle unexpected connection losses, and start re-transmission from where it was interrupted
c. The process is not parallel enough, such as when the client is in the process of packet/s verification at some intermediate point, server remains idle in such case. Thus, additional parallelism and functionalities can be introduced to maximize the efficiency of server and client.
d. Logging of all the process in the file for error tracking and handling
e. Addition of some security during the transfer process

9. Conclusion

Thus, a UDP file download application is developed using python. Along with the basic UDP implementation, the application has also implemented UDP packet loss handling, Alternating Bit and Selective Repeat protocols to add reliability features on the UDP.

10. References

[1] https://www.macs.hw.ac.uk/~pjbk/nets/protocolsimulations/abp.html, Alternating Bit Protocol Simulator

[2] *https://github.com/mamgad/RUDPy,* implementation of UDP client/server

[3] *Kuros, Ross 2017, Computer Networking – A top down approach 6th edition, Boston Massachusetts, and New York*

[4] *https://www.draw.io/,* Creation of flow chart, class diagrams, and state diagrams

[5] *https://staff.fnwi.uva.nl/b.diertens/psf/specifications/abp.html,* The Alternating Bit Protocol

[6] *https://www.ietf.org/rfc/rfc1216.txt,* Alternating Bit Protocol

[7]*http://www.geeksforgeeks.org/sliding-window-protocol-set-3-selective-repeat/,*Sliding Window Protocol | Set 3 (Selective Repeat)

Appendix :

Client functions

- *class _transport()*
  - *selective_repeate(self, packet_buffer,transport_object):*
  - *method_alternating_bit(self, packet, writeObject, chunk_received, address)*
  - *create_packet(self, alternating_bit, protocol, message, file_name='', type='a',):*
  - *verify_packet(self, packet):*
- *class file_handler():*
  - *write_to_file(self, filename, data): #data {seq1:data, seq2:data }*
  - *exception_handler(e):*
- *class _client_connection():*
  - *create_client_socket(self):*
  - *send_request_to_server(self, message, address, client_socket):*
  - *receive_response_from_server(self,client_socket):*
  - *close_connection(self,client_socket):*
- *connection_handler_selective_repeate(transport, writeObject, connection_object, request_message=[], file_name = "some_test_file", protocol='SR'):*
- *connection_handler_alternating_bit(chunk_received,writeObject,connection_object, request_message, file_name='some_test_file', protocol='AB'):*
- *class client_globals*

Server Functions

- *class Transport()*
  - *connection_handler_selective_repeat(file_object, connection_object)*
  - *connection_handler_alternating_bit(file_object, connection_object)*
  - *get_packets(file_object, seq_list)*
  - *method_alternating_bit(message, connection_object, file_object, counter, address)*
  - *selective_repeate(message, file_name, file_object)*
- *class file_handler(server_packet)*
  - *increase_sequence_counter(self)*
  - *read_in_chunks(self, file_object, chunk_size=1024)*
  - *file_read(self,file_name,type='s')*
- *class server_packet()*
  - *increase_seqNumber(self)*
  - *make_packet(self, data):*
- *class server_connection()*
  - *create_connection(self)*
  - *close_connection(self,server_socket)*
  - *send_response_to_client(self,server_socket,message,address):*
- *class server_globals:*